Merlin brings nonblocking I/O to the Java platform **[[[1]](#footnote-1)]**

A server's ability to handle numerous client requests within a reasonable time is dependent on how effectively it uses I/O streams. A server that caters to hundreds of clients simultaneously must be able to use I/O services concurrently. Until JDK 1.4 (aka Merlin), the Java platform did not support nonblocking I/O calls. With an almost one-to-one ratio of threads to clients, servers written in the Java language were susceptible to enormous thread overhead, which resulted in both performance problems and lack of scalability.

To address this issue, a new set of classes have been introduced to the Java platform with the latest release. Merlin's java.nio package is chock full of tricks for resolving thread overhead, the most important being the new SelectableChannel and Selector classes. A channel represents a means of communication between a client and a server. A selector is analogous to a Windows message loop, in which the selector captures the various events from different clients and dispatches them to their respective event handlers. In this article, we'll show you how these two classes function together to create a nonblocking I/O mechanism for the Java platform.

**I/O programming before Merlin**

We'll start with a look at a basic, pre-Merlin server-socket program. In the lifetime of a ServerSocket class, the important functions are as follows:

Accept incoming connections

Read requests from clients

Service those requests

Let's take a look at each of these steps using code snippets to illustrate. First, we create a new ServerSocket:

ServerSocket s = new ServerSocket();

Next, we want to accept an incoming call. A call to accept() should do the trick here, but there's a little trap you have to watch for:

Socket conn = s.accept( );

The call to accept() blocks until the server socket accepts a client request for connection. Once a connection is established, the server reads the client requests, using LineNumberReader. Because LineNumberReader reads data in chunks until the buffer is full, the call blocks on a read. The following snippet shows LineNumberReader in action (blocks and all).

InputStream in = conn.getInputStream();

InputStreamReader rdr = new InputStreamReader(in);

LineNumberReader lnr = new LineNumberReader(rdr);

Request req = new Request();

while (!req.isComplete() )

{

String s = lnr.readLine();

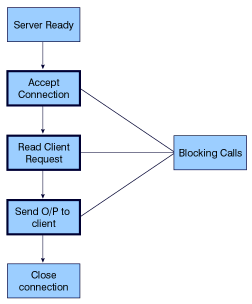
req.addLine(s);

}

InputStream.read() is another way to read data. Unfortunately, the read method also blocks until data is available, as does the write method.

Figure 1 depicts the typical workings of a server. The bold lines represent blocking operations.

Figure 1. A typical server in action



Prior to JDK 1.4, liberal use of threads was the most typical way of getting around blocking. But this solution created its own problem -- namely thread overhead, which impacts both performance and scalability. With the arrival of Merlin and the java.nio package, however, everything has changed.

In the sections that follow, we'll look at the foundations of java.nio, and then apply some of what we've learned to revising the server-socket example described above.

**The Reactor pattern**

The principal force behind the design of NIO is the Reactor design pattern. Server applications in a distributed system must handle multiple clients that send them service requests. Before invoking a specific service, however, the server application must demultiplex and dispatch each incoming request to its corresponding service provider. The Reactor pattern serves precisely this function. It allows event-driven applications to demultiplex and dispatch service requests, which are then delivered concurrently to an application from one or more clients.

Core functions of the Reactor pattern

* Demultiplexing events
* Dispatching events to their corresponding event handlers

The Reactor pattern is closely related to the Observer pattern in this aspect: all dependents are informed when a single subject changes. The Observer pattern is associated with a single source of events, however, whereas the Reactor pattern is associated with multiple sources of events.

See [Resources](http://www.ibm.com/developerworks/library/j-javaio/index.html?S_TACT=105AGX52&S_CMP=cn-a-j#resources) to learn more about the Reactor pattern.

**Channels and Selectors**

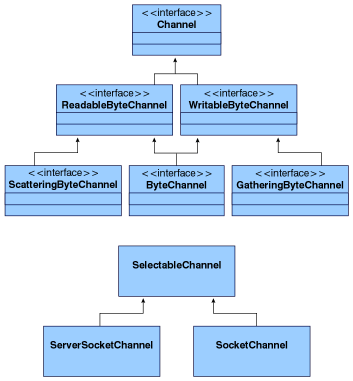
NIO's nonblocking I/O mechanism is built around selectors and channels. A Channel class represents a communication mechanism between a server and a client. In keeping with the Reactor pattern, a Selector class is a multiplexor of Channels. It demultiplexes incoming client requests and dispatches them to their respective request handlers.

We'll look closely at the respective functions of the Channel class and the Selector class, and at how the two work together to create a nonblocking I/O implementation.

**What the Channel does**

A channel represents an open connection to an entity such as a hardware device, a file, a network socket, or a program component that is capable of performing one or more distinct I/O operations, such as reading or writing. NIO channels can be asynchronously closed and interrupted. So, if a thread is blocked in an I/O operation on a channel, another thread can close that channel. Similarly, if a thread is blocked in an I/O operation on a channel, another thread can interrupt that blocked thread.

Figure 2. Class hierarchy for java.nio.channels



As Figure 2 shows, there are quite a few channel interfaces in the java.nio.channels package. We're mainly concerned with the java.nio.channels.SocketChannel and java.nio.channels.ServerSocketChannel interfaces. These channels can be treated as replacements for java.net.Socket and java.net.ServerSocket, respectively. Channels can be used in a blocking or a nonblocking mode, though of course we will focus on using channels in nonblocking mode.

**Creating a nonblocking channel**

We have two new classes to deal with in order to implement basic nonblocking socket read and write operations. These are the InetSocketAddress class from the java.net package, which specifies where to connect to, and the SocketChannel class from the java.nio.channels package, which does the actual reading and writing operations.

The code snippets in this section show a revised, nonblocking approach to creating a basic server-socket program. Note the changes between these code samples and those used in the first example, starting with the addition of our two new classes:

String host = ......;

InetSocketAddress socketAddress = new InetSocketAddress(host, 80);

SocketChannel channel = SocketChannel.open();

channel.connect(socketAddress);

**The role of the buffer**

A is an abstract class that contains data of a specific primitive data type. It is basically a wrapper around a fixed-size array with getter/setter methods that make its contents accessible. The Buffer class has a number of subclasses, as follows:

ByteBuffer

CharBuffer

DoubleBuffer

FloatBuffer

IntBuffer

LongBuffer

ShortBuffer

ByteBuffer is the only class that supports reading and writing from and to the other types, since the other classes are type specific. Once connected, data can be read from or written to the channel with a ByteBuffer object. See Resources to learn more about the ByteBuffer.

To make the channel nonblocking, we call configureBlockingMethod(false) on the channel, as shown here:

channel.configureBlockingMethod(false);

In blocking mode, a thread will block on a read or a write until the operation is completely finished. If during a read, data has not completely arrived at the socket, the thread will block on the read operation until all the data is available.

In nonblocking mode, the thread will read whatever amount of data is available and return to perform other tasks. If configureBlockingMethod() is passed true, the channel's behavior will be exactly the same as that of a blocking read or write on a Socket. The one major difference, mentioned above, is that these blocking reads and writes can be interrupted by other threads.

The Channel alone isn't enough to create a nonblocking I/O implementation. A Channel class must work in conjunction with the Selector class to achieve nonblocking I/O.

**What the Selector does**

The Selector class plays the role of a Reactor in the Reactor pattern scenario. The Selector multiplexes events on several SelectableChannels. Each Channel registers events with the Selector. When events arrive from clients, the Selector demutliplexes them and dispatches the events to the corresponding Channels.

The simplest way to create a Selector is to use the open() method, as shown below:

Selector selector = Selector.open();

**Channel meets Selector**

Each Channel that has to service client requests must first create a connection. The code below creates a ServerSocketChannel called Server and binds it to a local port:

ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.configureBlocking(false);

InetAddress ia = InetAddress.getLocalHost();

InetSocketAddress isa = new InetSocketAddress(ia, port );

serverChannel.socket().bind(isa);

Each Channel that has to service client requests must next register itself with the Selector. A Channel should be registered according to the events it will handle. For instance, a Channel that accepts incoming connections should be registered as shown here:

SelectionKey acceptKey = channel.register( selector,SelectionKey.OP\_ACCEPT);

A Channel's registration with the Selector is represented by a SelectionKey object. A Key is valid until one of these three conditions is met:

The Channel is closed.

The Selector is closed.

The Key itself is cancelled by invoking its cancel() method.

The Selector blocks on the select() call. It then waits until a new connection is made, another thread wakes it up, or another thread interrupts the original blocked thread.

**Registering the Server**

Server is the ServerSocketChannel that registers itself with the Selector to accept all incoming connections, as shown here:

SelectionKey acceptKey = serverChannel.register(sel, SelectionKey.OP\_ACCEPT);

while (acceptKey.selector().select() > 0 ){

......

After the Server is registered, we iterate through the set of keys and handle each one based on its type. After a key is processed, it is removed from the list of ready keys, as shown here:

Set readyKeys = sel.selectedKeys();

Iterator it = readyKeys.iterator();

while (it.hasNext())

{

SelectionKey key = (SelectionKey)it.next();

it.remove();

....

....

....

}

If the key is acceptable, the connection is accepted and the channel is registered for further events such as read or write operations. If the key is readable or writable, the server indicates it is ready to read or write data on its end:

SocketChannel socket;

if (key.isAcceptable()) {

System.out.println("Acceptable Key");

ServerSocketChannel ssc = (ServerSocketChannel) key.channel();

socket = (SocketChannel) ssc.accept();

socket.configureBlocking(false);

SelectionKey another =

socket.register(sel,SelectionKey.OP\_READ|SelectionKey.OP\_WRITE);

}

if (key.isReadable()) {

System.out.println("Readable Key");

String ret = readMessage(key);

if (ret.length() > 0) {

writeMessage(socket,ret);

}

}

if (key.isWritable()) {

System.out.println("Writable Key");

String ret = readMessage(key);

socket = (SocketChannel)key.channel();

if (result.length() > 0 ) {

writeMessage(socket,ret);

}

}

**Abracadabra -- nonblocking server socket appear!**

The final part of this introduction to nonblocking I/O in JDK 1.4 is left to you: running the example.

In this simple nonblocking server-socket example, the server reads a file name sent from the client, displays the file contents, and writes the contents back to the client.

Here's what you need to do to run the example:

Install JDK 1.4 (see Resources).

Copy both source files onto your directory.

Compile and run the server as java NonBlockingServer.

Compile and run the client as java Client.

Input the name of a text or java file in the directory where the class files are present.

The server will read the file and send its contents to the client.

The client will print out the data received from the server. (Only 1024 bytes will be read since that is the limit of the ByteBuffer used.)

Close the client by entering the command to quit or shutdown.

**Conclusion**

The new I/O packages from Merlin cover a broad scope. The major advantage of Merlin's new nonblocking I/O implementation is twofold: threads no longer block on reads or writes and the Selector is able to handle multiple connections, greatly reducing server application overhead.

We have highlighted these two primary advantages of the new java.nio package. We hope that you will apply what you've learned here to your real-time application development efforts.

1. Author: Aruna Kalagnanam, Balu G

   Original link: <http://www.ibm.com/developerworks/library/j-javaio/index.html?S_TACT=105AGX52&S_CMP=cn-a-j> Visit time: 2011-03-10 [↑](#footnote-ref-1)