Sponsored by:

JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-04-2003/jw-0411-select.html

Use select for high-speed networking

New Input/Output libraries speed up your server

By Greg Travis, JavaWorld.com, 04/11/03

Java uses an extremely elegant input/output (I/O) model, based on the idea of a *stream*. A stream is an object that produces or consumes a string of bytes. Streams can be chained together in conjunction with filtering routines and extended to handle other kinds of data. The stream model is very flexible, but not too fast. It's fine for many applications, but some systems require just about as much speed as the hardware can handle. Sometimes the stream model won't cut it.

The New Input/Output (NIO) libraries introduced in Java 2 Platform, Standard Edition (J2SE) 1.4 address this problem. NIO uses a *buffer*-oriented model. That is, NIO deals with data primarily in large blocks. This eliminates the overhead caused by the stream model and even makes use of OS-level facilities, where possible, to maximize throughput.

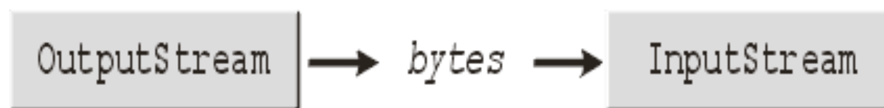
First I look at how NIO works, and then I put it to use in a high-speed server application.

Note: You can download this article's source code from Resources.

The NIO system

NIO is based on two concepts, *channels* and *buffers*. Channels are roughly analogous to streams used in the stream model. Buffers do not have a precise analog in the stream model.

The basic streams, `InputStream` and `OutputStream`, can read and write bytes; subclasses of these stream classes can read and write other kinds of data. In NIO, all data is read and written via buffers. See Figure 1 for a comparison of the two models.



The stream model

The NIO model

Figure 1. The stream model uses streams and bytes; the NIO model uses channels and buffers

Also notice that while the stream model distinguishes between `InputStreams` and `OutputStreams`, NIO uses one kind of object—a `Channel`—for both.

The main advantage of buffers is that they deal with data *in bulk*. You can read and write large blocks of data, and the size of the buffers you use is only limited by the amount of memory you are willing to allocate to them.

Another more subtle advantage of buffers is that they can represent system-level buffers. Some operating systems use a unified memory scheme that allows I/O without copying data from operating system memory into application memory. Some implementations provide `Buffer` objects that represent these system-level buffers directly, which means you can read and write data with minimal data copying (see Figure 2).

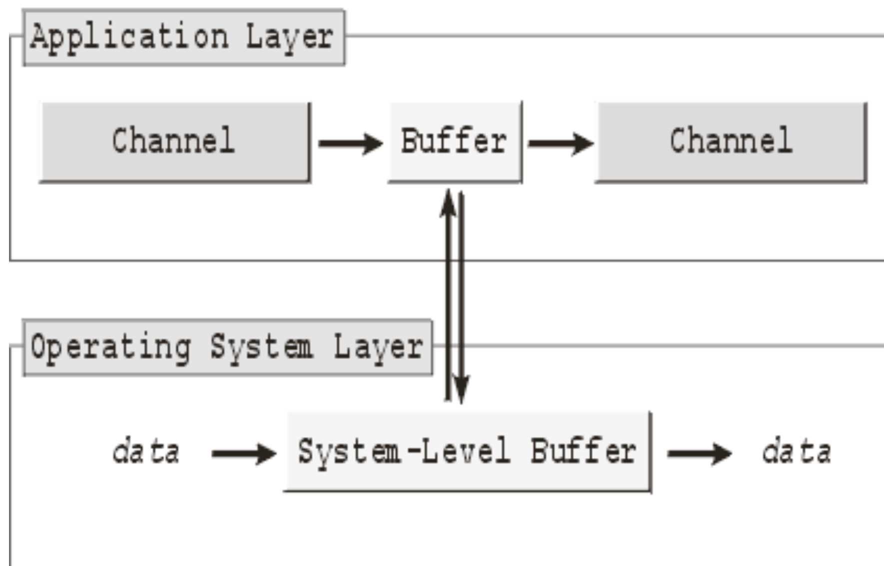


Figure 2. System buffers allow you to use system-level buffers directly, avoiding extraneous data copying

The select facility

The `select` facility provides an excellent way to deal with a large number of data sources simultaneously. It

takes its name from a Unix system call—`select()`—that provides the same kind of facility to a C program running on a Unix system.

Normally, I/O is done via *blocking system calls*. When you call a `read()` method on an input stream, the method blocks until some data is available. If you're reading from a local file, you don't have to wait long for the data to show up. On the other hand, if you are reading from a network file server or a network socket connection, you might wait longer. While you're waiting, your reading thread can't do anything else.

In Java, of course, it's easy to create multiple threads that read from multiple streams. However, threads can be resource-intensive. Under many implementations, each thread can occupy a sizeable amount of memory, even if it's not doing anything. And a performance hit can result from having too many threads.

The `select` facility works differently. With `select` you register many input streams with a `Selector` object. When I/O activity happens on any of the streams, the `Selector` tells you. This way, it's possible to read from a large number of data sources from a single thread. Note also that `Selectors` don't just help you read data; they can also listen for incoming network connections and write data across slow channels.

Sample application

To illustrate the use of the `select` facility, I will build a simple encryption server. This program takes data from a client, encrypts it, and sends back the encrypted data. I use the `select` facility to accept incoming connections and to read incoming data from existing connections.

Since this article isn't about encryption, the encryption used in this program is trivial. I just apply a `rot13` filter to the data and send it back. `rot13` is a simple filter that cycles each alphabetic character forward 13 steps, turning an *a* into an *n,* a *b* into an *o,* and so on. Letters towards the end of the alphabet cycle back around, so that *w* becomes *j*.

I won't look too closely at a client to go along with this server. You can you use telnet on your system to connect to the server. I provide a test program that will hit the server with numerous connections to see how it performs under stress.

Using Selectors

Let's examine how `Selectors` are used. In the example below, I use a `Selector` for two tasks: to accept incoming connections and to accept data from existing connections.

Note: The code snippets shown below are abbreviated for space; download the full source code from Resources.

Listen for incoming connections

First, I create a `Selector` object. The `Selector` is the central object in this process; every connection I listen to, in any way, must be registered with this object. The static `Selector.open()` method creates a `Selector`:

```
Selector selector = Selector.open();
```

Since I'm creating a client/server system, I will listen to a `ServerSocketChannel`. I must configure it as

nonblocking, so it can be used with the `Selector`:

```
ServerSocketChannel ssc = ...
ssc.configureBlocking( false );
ssc.register( selector, SelectionKey.OP_ACCEPT );
```

The `SelectionKey.OP_ACCEPT` parameter tells the `Selector` that I want to listen only for incoming connections, not for regular data. Since server sockets don't receive regular data, this is fine.

The main loop

Now that I've registered something with our `Selector`, let's fire it up. I use the `Selector`'s `select()` method and put it inside an infinite loop, since I will wait repeatedly for new activity:

```
while (true) {
   // See if you've had any activity -- either
   // an incoming connection or incoming data on an
   // existing connection.
   int num = selector.select();
   // If you don't have any activity, loop around and wait
   // again.
   if (num == 0) {
      continue;
   }
   // Get the keys corresponding to the activity
   // that have been detected and process them
   // one by one.
   Set keys = selector.selectedKeys();
   Iterator it = keys.iterator();
   while (it.hasNext()) {
      // Get a key representing one of bits of I/O
      // activity.
      SelectionKey key = (SelectionKey)it.next();
      // ... deal with SelectionKey ...
   }
   // Remove the selected keys because you've dealt
   // with them.
   keys.clear();
}
```

Note this structure is similar to an *event loop* used in graphical user interfaces. This makes sense because event loops are used when input can come from different objects, and we don't know ahead of time which one will have input first.

Within the loop, the `select()` method returns the number of channels that have I/O activity. If it returns 0, I just go back to the top of the loop and wait again. If not, then I must deal with the I/O activity.

The activity is represented by one or more `SelectionKey` objects. A `SelectionKey` represents the registration of a single `Channel` with a single `Selector`. When a `Selector` determines that a particular `Channel` has some activity, it returns the `SelectionKey` corresponding to that `Channel`.

Receive a connection

Once a `SelectionKey` represents some I/O activity, you must find out what kind of activity it is. At this point in my code, I've registered only one `Channel`—the `ServerSocketChannel`—so I need to deal with incoming connections on this `Channel`.

Use the `SelectionKey`'s `readyOps()` method to determine the type of I/O activity. This method returns a bitmask that represents the type (or types) of input that have occurred on this `Channel`. Check it to see if it contains the `OP_ACCEPT` bit, which represents an incoming connection:

```
if ((key.readyOps() & SelectionKey.OP_ACCEPT) ==
   SelectionKey.OP_ACCEPT) {
   // Accept the incoming connection.
   Socket s = serverSocket.accept();
   // ... Deal with incoming connection...
}
```

If it is, in fact, an incoming connection, then use the traditional blocking call, `accept()`, to get the connection. `accept()` won't block because the `Selector` has already told me about an incoming connection waiting to be processed.

Now that a genuine connection is established, use it for incoming data.

Listen for incoming data

After I accept a connection, I want to listen for data coming into it. Just as I registered the server socket to listen for incoming connections, I register the newly connected socket to listen for incoming data. I configure the new socket to be nonblocking, as I did for the server socket:

```
// Make sure to make it nonblocking, so you can
// use a Selector on it.
SocketChannel sc = socket.getChannel();
sc.configureBlocking( false );
// Register it with the Selector, for reading.
sc.register( selector, SelectionKey.OP_READ );
```

I set the `Selector` to listen for `OP_READ` activity, rather than `OP_ACCEPT`, which means I listen for data input, rather than incoming connections.

Back to the top

Two sockets are registered with the `Selector`: a server socket and a regular socket. At the top of the loop, I call `select` again:

```
int num = selector.select();
```

Now I receive notification if either socket has activity; that is, if the server socket receives another connection, or if the regular socket receives data, or both. When more connections come in, they, too, will be registered with the Selector.

Incoming data

Because at least one regular socket is registered, I will receive some data on one of them. You can tell this has happened when you get a selection key with the OP_READ bit set in the socket's readyOps() bitmask:

```
} else if ((key.readyOps() & SelectionKey.OP_READ) ==
   SelectionKey.OP_READ) {
   SocketChannel sc = (SocketChannel)key.channel();
   processInput( sc );
   // ...
}
```

When this happens, I pass the socket, or rather the socket's SocketChannel, off to the encryption routine. The encryption routine, which you can find in the source code, just encrypts the incoming data and returns the encrypted data to the client.

End result

These are a lot of steps, but they all fit into the select paradigm. All the input sources are registered with a Selector. Inside an infinite loop, the Selector's select() method is called repeatedly. Each time it returns, some of the input sources will have activity. I deal with the input on each of the active sources and then loop. Figure 3 illustrates that process.

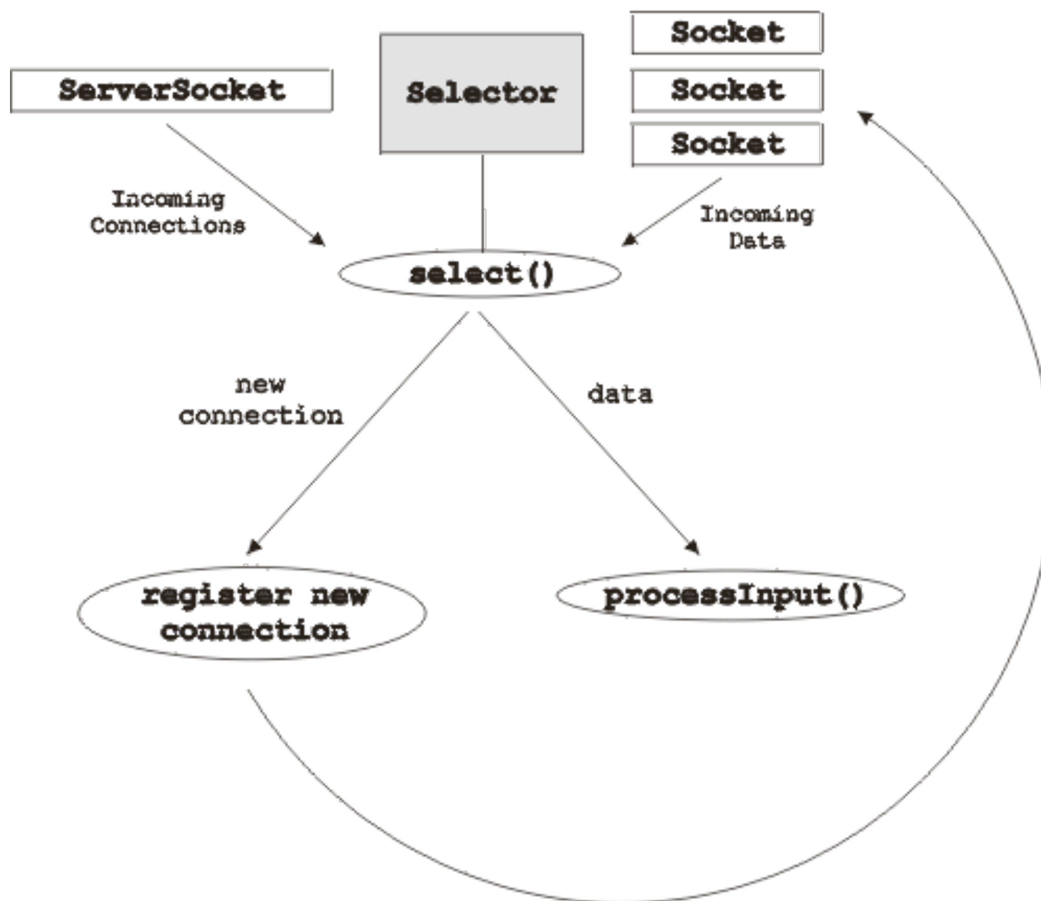


Figure 3. The complete select loop

The complete server

The code we examined earlier is only a part of the story. The full source code contains the complete main loop, as well as the code for `processInput`. To test the server, first run it on the command line:

```
java Server [port number]
```

Then, use telnet to connect to this server. If you run telnet on the same machine as the server, you can use `localhost` or `127.0.0.1` as the hostname when you telnet. Specify the port number in the command line above.

The source code also contains `Client.java`, which is a program you can use to test the server. It runs multiple threads, and each one sends data to the client and reads back the response. Each thread then pauses and sends data again, in an infinite loop. You run it like this:

```
java Client [hostname] [port] [number of threads]
```

Recap

As discussed earlier, the `select` I/O model is essentially event-driven. All of your input sources are

registered with a single `Selector` object, which waits for activity on any of the input sources. This model differs from the stream model, but it's still a solid model. In fact, in the larger scheme of things, it's generally more correct to think of the stream model as a layer that runs on top of the `select` model. At the hardware level, I/O is fundamentally event-driven, since peripherals such as network cards, keyboards, and some disk drives send their data without warning.

The stream model uses buffering to hide the complexity of event-driven I/O behind blocking I/O calls, which makes I/O programming much simpler. But when you need the extra speed, bypass the stream layer and go directly to the I/O events themselves.

The NIO library provides an elegant buffer-based interface to the `select` model. It's fully compliant with the older stream model; in fact, the classes in the traditional `java.io.*` packages are now based on the `java.nio.*`, so that the two function together seamlessly.

About the author

Greg Travis is a freelance Java programmer and technology writer living in New York City. After spending three years in the world of high-end PC games, he joined EarthWeb, where he developed new technologies with the then-new Java programming language. Since 1997, he has been a consultant in a variety of Web technologies, specializing in real-time graphics and sound. His interests include algorithm optimization, programming language design, signal processing (with emphasis on music) and real-time 3D graphics. Other articles he's written can be found at http://www.panix.com/~mito/articles. He is the author of JDK 1.4 Tutorial, published by Manning Publications.

All contents copyright 1995-2008 Java World, Inc. http://www.javaworld.com