The New Java I/O Package as an Enhanced Instance of the Reactor Pattern

Bogdan Dumitriu

November 14, 2005

Abstract

The Reactor pattern is a concurrency pattern which allows an application to cleanly separate its event demultiplexing code from its event handling code. By using this pattern, the application gains in performance by using demultiplexing instead of multithreading and it gains in reusability, since the event demultiplexing code can be easily factored out and reused across multiple application. The new Java I/O package (`java.nio`), which is in part a modified and somewhat extended implementation of the essential parts that define the Reactor pattern, can be seen as an example of how the latter gain can be achieved. This article explores the relation between the two, illustrates how Java's new I/O package can be used for non-blocking, demultiplexing I/O and finally discusses why the Java implementation can be regarded as an improvement of the original Reactor pattern.

1 Introduction

It is the case with many applications that events generated by multiple sources have to be handled simultaneously by an application. A common scenario is that of server which can have multiple clients connected to it at the same time and all of which have to be served fairly. A classical solution for this situation is to have the application spawn a new thread for each client (or even for each request) and thus let the concurrency be handled by the operating system's thread scheduler. However, this has a number of known disadvantages, among which the need to synchronize between threads, the performance decrease due to context switching and the operating systems limitations in creating more than a certain maximum number of threads.

Multiple threads are usually considered as a "necessary evil" in such cases because were the application to use a single thread, that would block on just one source of events (e.g., socket), introducing long or even infinite delays in handling the events coming from the other sources. Even if the usual solution, multiple threads are not the only solution, however, at least not if the operating system on which the application runs provides a way to at the same time wait for events from multiple sources. On of the alternatives is represented by the Reactor pattern, developed by Douglas C. Schimdt and described, among others, in [4, 5].

According to its author, *the Reactor pattern dispatches handles automatically when events occur from multiple sources [...], decoupling event demultiplexing*

and event handler dispatching from application services performed in response to events. The way the Reactor pattern achieves this is by having a central Reactor object which manages the demultiplexing of events and their dispatching to the application. The Reactor allows the application to register so called event handlers with it, each of them specifying a handle (or, in other words, a source of events) they are interested to monitor for events. Since most handles can produce more types of events, an event handler can also specify upon registration with the Reactor which of those types of events it is interested in. As soon as all the event handlers are registered with the Reactor, the application enters its event loop by asking the Reactor to start monitoring all the handles it has been asked to. This monitoring is most efficient if the underlying operating system offers a mechanism to demultiplex multiple events at the same time (e.g., the `select` function is such a mechanism on Unix operating systems). Even though this can also be simulated by the Reactor by using multiple threads, such a solution is not to be desired, since then most benefits of the pattern are defeated.

As soon as an event is detected by the Reactor, the table of registered event handlers is looked up to find out which event handler is registered to handle that event and its `handle_event` function is called, passing the type of event as an argument. After the handler finishes execution, the Reactor will return to the monitoring phase by calling the operating system's event demultiplexer.

As specified in [5], the Reactor pattern is useful when we have to deal with multiple events arriving concurrently from multiple sources, when handling of an event takes little time and incurs a limited amount of communication and when multi-threading is unavailable or to be avoided for either performance or complexity issues. The Reactor pattern brings a number of benefits, among which the clean separation of application code from the event demultiplexing and dispatching code and the avoidance of all problems that have to be dealt with when using multiple threads. One disadvantage of the pattern is that event handlers can take arbitrarily long to process an event, which might lead to apparent non-responsiveness of the application to other events.

This paper is going to focus on a particular implementation of the Reactor pattern, namely the one in the new Java I/O package. Although the Reactor pattern has only been briefly discussed in this introduction, familiarity with it is assumed in the rest of the paper, so you are referred to [4] for a more in-depth description. Section 2 of the paper discusses the principles behind the new Java I/O implementation. In section 3 we give an extended example to show how the mechanism described in section 2 can be applied in practice. The paper ends with a comparison of the Java implementation with the original pattern (section 4) and some concluding remarks (section 5).

2 The New Java I/O

The new Java I/O package (`java.nio`) is closely related to the Reactor pattern at least because of two reasons: it itself represents a nice (and arguably partial) implementation of the pattern and, at the same time, it can be used as a backend for implementing the pattern. In this section, we will describe the relevant parts of the Java new I/O system, explain how they work together and also relate them to the Reactor pattern so that the relation between the two becomes clear. The contents of this section, although unique to this paper, is based on several

sources [1, 3, 6], which will not be mentioned again throughout the section, to avoid clutter.

The old Java I/O system is well designed and easy to use, so the first question that comes to mind is why come up with a new one? The answer to this is quite simple: despite the ease of use, the old system is neither fast nor flexible enough for all purposes. Granted, it gets the job done, but not always as good as the developer would like. In what performance is concerned, the new Java I/O introduces a series of abstractions (e.g., buffers, channels) which allow the user to have much more control over, and thus get better performance from, the entire I/O process. One example is the new `MappedByteBuffer`, which piggybacks on the operating system's capabilities of mapping a file into a memory location and provide direct access to the file's contents in this way. This produces a considerable improvement in speed if compared to the traditional stream access to the file's contents. However, it is the second major aspect introduced by the improved I/O package, namely multiplexed, non-blocking I/O, that is more directly related to the Reactor pattern.

The novelty with non-blocking I/O is that programmers are no longer forced to create a new thread for each socket connection which has to be handled simultaneously. One thread per socket was necessary when blocking I/O was the only option because as soon as a thread would need to read or write data from/to a socket, it would have to block while doing so, thus rendering itself unavailable for processing other socket connections at the same time. Concurrency issues aside, the other big disadvantage of this approach is the lack of scalability. With a large enough number of simultaneous connections, either the maximum thread limit allowed by the operating system is reached or, if measures are taken against this, only so many clients can be served at the same time. This is clearly a scalability problem. With the non-blocking capabilities, a single thread can be used to manage a theoretically unlimited number of socket connections, thus easily getting rid of both synchronization problems and scalability problems. Moreover, it is even possible to awaken a thread which is blocked on multiple sockets by using the `wakeup()` method of the `Selector` object which the thread uses for waiting for events.

At the center of the new Java I/O multiplexing capabilities is the `Selector` class. It plays the same role that operating system calls like `select()` and `WaitForSingleEvent()` play in UNIX and Windows systems, respectively. Actually, its functionality goes beyond that of such system calls, since it is also capable of maintaining state from one call to the next. A `Selector` object is created upon request and means are provided to register `Channel`s (more precisely, `SelectableChannel`s) with it. `Channel`s will be discussed below, but for now it suffices to say that they can be regarded indirectly as event generators.[1] A `Channel` can be registered with any number of `Selector`s and a `Selector` can monitor any number of `Channel`s. When registering a `Channel` with a `Selector`, a set of *operations* has to be specified, essentially telling the `Selector` which types of *events* coming from this `Channel` to react to (also known as the *interest set*). The operations defined so far by the Java platform are `OP_ACCEPT`, `OP_CONNECT`, `OP_READ` and `OP_WRITE`. As soon as a `Selector` is set up in this way, one of the methods `select()`, `select(long timeout)` or

[1]For accuracy, it should be mentioned that it is not really the `Channel`s themselves that generate the events, but rather the resources connected to them.

`selectNow()` can be called on it to start monitoring for the occurrences of the specified events with the registered channels.

A `Selector` object plays the role of the Reactor from the Reactor pattern. Although the way it works is slightly different, the outcome is nevertheless the same. `Channel`s are registered with a `Selector` much in the same way event handlers are registered with a Reactor. Also, both `Channel`s and event handlers specify the event types they are interested in to the `Selector` and Reactor, respectively. However, `Channel`s do not quite serve the same purpose as event handlers, and this is basically the difference between the two approaches. If the Reactor pattern is based on a *push* mode, the Java new I/O system is based on a *pull* mode. This means that a `Selector` will not take any action when events of interest occur in one or more of the monitored `Channel`s, except return from the `select` call. It is then the responsibility of the caller to get the events out of the `Selector` and process them as it sees fit. This is why the concept of event handlers is really missing from the Java paradigm. It should be obvious at this point that although `Channel`s are registered with the `Selector` in the same way event handlers are with the Reactor, `Channel`s actually play the role of handles (in Reactor pattern nomenclature) and not that of event handlers. They are the indirect source of events, but they play no part in processing them.

To go further with the comparison between the Reactor pattern and the new Java I/O system, the `SelectoryKey` class has to be discussed next. As defined in [6], a `SelectionKey` is *a token representing the registration of a `SelectableChannel` with a `Selector`*. When a `Channel` is registered with a `Selector`, a value is returned to the caller. This value is the `SelectionKey` representing the registration. `SelectionKey`s are useful for a number of purposes. First of all, it represents one way in which a `Channel` can be deregistered from a `Selector`. This cannot happen automatically, though (i.e., there is no deregister method). Deregistration is achieved by *canceling* a `SelectionKey`. A canceled key is not automatically deregistered, however. Deregistration happens either at the end of a `select` call or at the beginning of one (but never while waiting for events). Secondly, a `SelectionKey` is the container of the types of events the selector monitors for a particular channel. Remember that when a `Channel` is registered with a `Selector`, a set of operations is also passed to the `register` method. These operations will be stored in the `SelectionKey` which encodes the registration. Thirdly, sets of `SelectionKey`s are maintained by `Selector`s and used to pass events to their callers. When an event of interest occurs on a channel, the `SelectionKey` representing the registration of that `Channel` with the `Selector` will be altered to indicate the occurrence of the event. As such, the `SelectionKey` can indicate the occurrence of one up to four different events on the channel (`OP_ACCEPT`, `OP_CONNECT`, etc.). At the end of a `select` call, the set of selected keys can be retrieved from the `Selector`, and by iterating through it one can find out what events have occurred and act upon them. Finally, each `SelectionKey` optionally allows an `Object` to be attached to it. It is up to the caller if and how (s)he wants to use this object. A multitude of cases can be imagined where this could come in handy. Aside from the attached object, a `SelectionKey` also "remembers" the `Channel` it is associated with, and offers a way to retrieve it at any time.

It is then by means of these `SelectionKey`s that the Java platform replaces the callback functionality of the Reactor pattern. The Reactor pattern notifies the user of the events that occur by either passing them as a parameter during

a callback, or by implicitly passing them by calling back different functions, depending on the type of event. Using `SelectionKey`s is closer to the former way of handling, since it is still the user that has to demultiplex the events once it has retrieved a selected `SelectionKey`. The distinction between the push mode used by the Reactor pattern (by calling functions of the event handlers) and the pull mode used with the new Java I/O (by retrieving a set of `SelectionKey`s from a `Selector` and iterating through them) should be clear at this point.

What is perhaps another interesting thing which distinguishes the Java new I/O system from the Reactor pattern is that events are not "thrown away" automatically as soon as they are retrieved from a `Selector`, but they have to be explicitly removed by the caller. Otherwise, if another `select` call is made, the resulting set of selected keys will also show events that have already been returned on previous calls as well (this could or could not be desired by the application, but in either way, care should be taken). Removal of `SelectionKey`s from the set returned by a `Selector` can be safely done by calling the `remove` method of the iterator used to iterate over the set. This will remove the `SelectionKey`s from the set with events, but not from the `Selector` altogether, so that on the next calls to `select`, the channels associated with the keys will still be monitored for events of interest.

This brings us to the last significant component in the new Java I/O configuration: the `Channel`. As already mentioned above, a `Channel` plays the role of a handle from the Reactor pattern. This is because a `Channel` is an open connection to either a hardware device or a process which communicates with this one. What really matters is that if we view the `Channel` as a connection between our program and an external entity, then the `Channel` will carry events generated by this entity to us. In the context of non-blocking, multiplexing I/O it is not really all `Channel`s that we take into consideration, but rather only `SelectableChannel`s. A `SelectableChannel` is simply a `Channel` which can be registered with a `Selector` and then multiplexed via this `Selector`, together with other `SelectableChannel`s. A `SelectableChannel` can be registered and (indirectly) deregistered with a `Selector` as discussed above. Aside from being multiplexable, `SelectableChannel`s can also be configured for working either in a *blocking* or *non-blocking* manner. Blocking/Non-blocking modes of operation have the usual semantics: in blocking mode, all operations performed on a `Channel` will only return when they are completed, while in non-blocking mode, they will do as much as possible without blocking and then return (even if the operation has not completed).

There are two `SelectableChannel`s defined which are related to network communication and which are commonly used with `Selector`s: `ServerSocketChannel` and `SocketChannel`. The first one encapsulates a `ServerSocket`, and is capable of producing only events of type `OP_ACCEPT`, while the second one encapsulates a `Socket` and is capable of producing events of type `OP_CONNECT`, `OP_READ` and `OP_WRITE`. Neither of them are complete encapsulations, though, which is why they also provide access to the underlying (server) socket, which can be manipulated directly.

3 An Example: Stock Quoter

In the previous section we have explained the part of the new Java I/O which directly relates to the Reactor pattern. In this section, we will show how everything that has been discussed in the earlier can be put together in the form of an as an example stock quoter server implemented with non-blocking, multiplexed I/O. A stock quoter server's clients want to use the server in order to request stock quotes. The clients pass a symbol of a company they want to get the stock quote for and the server returns several pieces of information about that company (in this example, the company's name, last trading price, opening price and volume of trades; of course, a real server would provide a lot more information than that). A real-life server would serve this information for a resource which is continuously updated with the most recent quotes. In this example, we will simply use a `HashMap` which maps stock symbols to stocks as the source of information. The information about stocks is thus static, but this is of little relevance to our example, since the interest here is to show how the server processes requests from clients, and not really to worry about what it passes back.

A stock quote server is an ideal candidate to be implemented by using non-blocking, multiplexed I/O because of two reasons. First of all, connections are (very) short-lived: a client simply connects to the server, gets its quote and the connection is closed. Short-lived connections are a lot more efficiently dealt with by means of multiplexed I/O as opposed to spawning new threads for each new connection, because the overhead in creating the threads is likely to be greater than the actual work it gets done. With multiplexed I/O, on the other hand, a single thread can simply manage a multitude of connections easily, ensuring a significant performance increase. This brings us to the second reason, which is the small amount of time needed to fetch the answer to send to the client. In this example, all it takes is a search for the stock symbol in the hash map. In a real example we could envision something more complex, but it is still to be expected that most of the requests would be served from an in-memory cache, so similar complexity (or rather non-complexity) is to be expected. This fits very nicely with a multiplexed model since client requests would suffer a minimal delay until they are processed (likely to be smaller than if multiple threads were created and context switching would have to be done all the time).

The code for stock quote server presented in this section is very remotely based on the code for the web server described in [3]. The only thing which was kept was the structure of processing, but even that only partially, since several changes have been made there as well. It is perhaps useful to begin the discussion directly by presenting this structure. The stock quote server runs two threads, one for accepting all new connections (the acceptor thread) and one for processing all established connections (the communication thread). These two threads interact by means of a synchronized queue. As new connections come in, the acceptor thread will put their `SocketChannel`s into the queue. The communication thread will later retrieve them from this queue and use them to process requests. Since the communication thread is usually asleep in a `select` call of a `Selector` (called henceforth the communication selector), a means has to be provided to wake it up when new connections come in (since even if they are included in the set of channels the communication selector checks, they will only be taken into consideration during the next `select` call, but not during the

one the thread is currently blocked on). Luckily, the `Selector` class provides the `wakeup()` method, which effectively ends a `select` call, event if no events appeared. The `wakeup` call has to be made by the acceptor thread every time it adds a newly connected `SocketChannel` to the queue, which is why it also needs to know, aside from the queue, the communication selector which the communication thread uses. How the queue and the communication selector are passed to both threads is shown in the following listing, which illustrates what happens when the stock quote server gets started.

```
1  public StockServer() throws IOException
2  {
3    // this creates and fills the hash map with stock data
4    initializeStocks();
5
6    // the synchronized queue used for communication between the two threads
7    BlockingQueue<SocketChannel> channelQueue;
8    channelQueue = new LinkedBlockingQueue<SocketChannel>();
9
10   // the communication selector (created using a factory method)
11   Selector commSelector = Selector.open();
12
13   // create the acceptor thread and pass it the queue and the selector
14   new AcceptorThread(PORT, commSelector, channelQueue).start();
15
16   // create the communication thread and pass it the queue,
17   // the selector and the stock data initialized by initializeStocks
18   new CommunicationThread(commSelector, channelQueue, stocks).start();
19 }
```

It is in the code of the acceptor thread where we first get to see how multiplexed, non-blocking I/O is done. First of all, we should discuss what happens when the acceptor thread gets created. The thread needs to open a new `ServerSocketChannel` which can listen for incoming connections and configure this channel to operate in non-blocking mode (this is required if we want to register the channel with a `Selector`). The following piece of code does that:

```
1  // server socket channels can only be created using a factory method
2  ServerSocketChannel ssc = ServerSocketChannel.open();
3
4  // configure the channel for non-blocking mode of operation
5  ssc.configureBlocking(false);
```

The newly created server socket channel is unbound, so in order to make it listen for incoming connections, it has to be first bound to a specific address:

```
1  InetSocketAddress address = new InetSocketAddress(port);
2  ssc.socket().bind(address);
```

Finally, before the thread is started, the acceptor selector has to be created as well and the server socket channel has to be registered with it for `OP_ACCEPT` operations. Configured like this, as soon as a `select` method of the acceptor selector is called, it will react if and only if there are connection requests on the server socket channel.

```
1 // create the acceptor selector
2 acceptSelector = Selector.open();
3
4 // register the server socket channel with the acceptor selector
5 // for OP_ACCEPT events
6 ssc.register(acceptSelector, SelectionKey.OP_ACCEPT);
```

As soon as this is set up, the acceptor thread can be started and it will immediately begin running the loop in which it waits for connection requests. This is not a busy loop, however, since the thread is blocked on the `select` call, and it only awakes if there is an incoming connection. This loop is straightforward (exception handling has been left out):

```
1 while (true)
2 {
3   acceptSelector.select();
4   process();
5 }
```

Needless to say, the bulk of the processing happens in the `process` method, which is provided in the listing below. When a selector (in this case, the accept selector) returns from a select call, it is usually because one or more events which have been registered as of interest to the caller have occurred. The way to find out which these events are is to iterate over the set of *selected keys*, i.e. those selection keys for which an event has occurred (please refer to section 2 for a thorough discussion about selection keys), and check them for events. This iterator is created in lines 2-3 below and iterated over in lines 5-19 below. For each key a number of actions are taken. First of all, if a key is in the selected key set, we can be sure that it is there because an `OP_ACCEPT` event has occurred (since we have not registered interest for anything else), which saves us from an extra check (something like `if (key.isAcceptable())`). As also discussed in section 2, selected keys have to be removed from the selected key set manually, if we do not want to process them again after the next call to `select`. This justifies the call to the iterator's `remove()` method in line 9. Once this is managed, we have to do something with the `OP_ACCEPT` event. An `OP_ACCEPT` event indicates that there is a pending connection request, which further means that a call to `accept` will be successful (since we have configured the server socket channel above to be non-blocking, calling `accept` would not block anyway, but it would return a null value if there was no pending connection; the `OP_ACCEPT` event simply guarantees that such a connection does indeed exist and then `accept` will return a valid socket channel). This is what happens then in lines 12-13: the server socket channel which is associated with the selection key (i.e., the one we created in the constructor of the acceptor thread) is retrieved and used to accept the call and thus generate a new socket channel. This new socket channel encapsulates the connection to the client, and it will be used by the communication thread in order to process the client's request. The way we send it to the communication thread is by putting it into the queue (line 16) and waking up the communication thread (line 18) to make sure that requests on this socket channel will be taken into consideration with no delay.

```
1 // get an iterator over the selected keys
2 Iterator<SelectionKey> keysIterator;
```

```
3  keysIterator = acceptSelector.selectedKeys().iterator();
4
5  while (keysIterator.hasNext())
6  {
7    // get the selection key
8    SelectionKey key = keysIterator.next();
9    keysIterator.remove();
10
11   // retrieve the channel associated with the key and accept the call
12   ServerSocketChannel acceptChannel = (ServerSocketChannel) key.channel();
13   SocketChannel commChannel = acceptChannel.accept();
14
15   // put the new socket channel in the queue
16   channelQueue.put(commChannel);
17   // and wake up the communication thread
18   commSelector.wakeup();
19 }
```

This concludes the description of the acceptor thread and brings the discussion to the more involved of the two threads of the stock quote server: the communication thread. Aside from saving the parameters passed to it into some field variables of the class, the constructor of the communication thread creates two other important structures: a `ByteBuffer` and a `Charset`. A detailed discussion about how buffers and character sets are used in the Java new I/O package is well beyond the scope of this article, but it should still be mentioned that a `ByteBuffer` is the only way to get the data out of a socket channel, while the character set is needed to encode and decode between Unicode and whatever character set the platform running the stock quote server uses. For more details about buffers and character sets, you are referred to any of [2, 6].

The loop run by the communication thread is very similar to the one run by the acceptor thread, with one extra ingredient: the communication thread also has to make sure it adds new socket channels to its selector as soon as the acceptor thread notifies it of their existence. This is done by the call to `registerNewChannels()` in the listing below.

```
1  while (true)
2  {
3    registerNewChannels();
4    commSelector.select();
5    process();
6  }
```

If you remember the `wakeup` call performed by the acceptor thread on the communication selector, it is here where it has its effect. It effectively awakens the communication thread which is blocked on the `select` call from line 4. Since there are probably no events at the time of the awakening, the call to `process` from line 5 will have no effect, the loop will be reentered and `registerNewChannels` will then be called, thus ensuring the addition of the newly connected socket channel to the set of channels the communication selector is monitoring. Here is how this is achieved in the `registerNewChannels()` method:

```
1  SocketChannel channel;
2  while ((channel = channelQueue.poll()) != null)
3  {
4    channel.configureBlocking(false);
5    channel.register(commSelector, SelectionKey.OP_READ,
6                     new StringBuilder());
7  }
```

The `poll` method of the queue returns the head of the queue and also removes it. The socket channel retrieved from the queue is then configured for non-blocking mode of operation (so that it can be multiplexed via a selector) and then registered with the communication selector (i.e., the selector the communication thread uses to wait for events). The channel is registered only for `OP_READ` events, which means that the communication thread expects the client to send its request, without taking any further action before that happens. What is also interesting to notice is the third argument of the `register` call. This argument is the object attached to the selection key representing this registration (see section 2 for more on this). The attached object used here is a `StringBuilder` which is nothing but an efficient way of constructing `String`s in Java (so it can logically be seen as a `String` object). This `StringBuilder` will be used across potentially multiple `OP_READ` events to "collect" the entire text of a client's request. Since each time the communication thread gets an `OP_READ` event it will read a fixed amount of bytes from the client and since the size of the entire client request is unknown beforehand, this trick has to be used in order to put a request back together. And the selection key's attachment is the best way to do this, since a selection key is connected to a particular socket channel. All the `OP_READ` events generated by that socket channel will determine this same selection key to be put in the selected keys set. The selection key has the attached `StringBuilder` object, so every time a new `OP_READ` appears on the channel, we are sure to append the text to the right `StringBuilder`, eventually reconstructing the request properly. How this is effectively done will be described shortly. But first let us have a look at what the `process` code of the communication thread does:

```
1  Iterator<SelectionKey> keysIterator;
2  keysIterator = commSelector.selectedKeys().iterator();
3
4  while (keysIterator.hasNext())
5  {
6    SelectionKey key = (SelectionKey) keysIterator.next();
7    keysIterator.remove();
8
9    if (key.isReadable())
10   {
11     processRequest(key);
12   }
13   if (key.isWritable())
14   {
15     processCompletedRequest(key);
16   }
17 }
```

What you see above is essentially the same mechanism based on iterating over the selected keys as the one described when discussing the acceptor's thread `process` method. There is one distinction, however. Here, we can have both `OP_READ` and `OP_WRITE` events associated with the selection keys. We have only showed registration for `OP_READ`s above, but you will also see at what point `OP_WRITE`s are registered further below. For now, let us assume that an interest exists for both `OP_READ` and `OP_WRITE` events. If this is the case, we have to distinguish between them and take appropriate actions, which is why we have the two cases in lines 9-12 and 13-16 above.

Before we advance, an overview of what happens for each connected socket channel can prove useful to better put things into context. First, only `OP_READ` events are monitored for. As it will be described below, in `processRequest` we will keep appending the data read from the client to the `StringBuilder` attached to the selection key until all the data has arrived. When this happens, we will have the full request reconstructed in the `StringBuilder` object, and we are ready to process it. In order to do this, we will register an interest for `OP_WRITE` events, thus preparing the sending of an answer to the client. An `OP_WRITE` event basically tells us that there is enough buffer space locally to send some data to the client. As soon as we get such an event, we finish processing the completed request and send the response to the client. Once the response is sent, the socket channel with the client is closed, which has as a side effect the cancellation of the selection key associated with that channel from the communication selector, thus preventing any more events being notified for that channel (specifically, `OP_WRITE` events would keep being notified indefinitely, if the channel was not closed or if an interest for such events was not removed).

Here is what happens in the code of `processRequest`, which is called to handle an `OP_READ` event. The code as given below is incomplete and not completely correct in order to prevent clutter. The correct code is given in the appendix.

```
1  // get the socket channel associated with the selection key
2  SocketChannel commChannel = (SocketChannel) key.channel();
3
4  // read (partial) data from the client into readBuffer and
5  // decode contents of readBuffer into the result variable (not shown)
6  commChannel.read(readBuffer);
7
8  // get the StringBuilder attached to the selection key
9  StringBuilder requestString = (StringBuilder) key.attachment();
10 // and append the new data to it
11 requestString.append(result);
12
13 // if the request is completed, register interest for OP_WRITE events
14 if (requestString.endsWith("\n\n"))
15 {
16   int oldInterestOps = key.interestOps();
17   int newInterestOps = oldInterestOps | SelectionKey.OP_WRITE;
18   key.interestOps(newInterestOps);
19 }
```

The data that is read from the socket channel is appended to the `StringBuilder`

object, as already discussed. A request is considered to be finished if two consecutive new lines are sent by the client. When this happens, the selection key is modified by adding an interest for `OP_WRITE` events to the key (lines 16-18). Since this key is the same as the one held by the selector (i.e., not a copy or a clone thereof), this will have immediate effect the next time a call to `select` is made, and `OP_WRITE` events will be notified. Let us see, then, what happens when a request is completed and an `OP_WRITE` event is notified, so the communication thread knows it can send data. This is the (brushed up) code of `processCompletedRequest`:

```
1  // get the complete request
2  String request = key.attachment().toString();
3  // get the socket channel associated with the selection key
4  SocketChannel commChannel = (SocketChannel) key.channel();
5
6  if (request is well formed)
7  {
8    Stock s = stocks.get(symbol specified by request);
9    if (s != null) // if symbol denotes a valid stock
10   {
11     // get a properly formatted byte buffer with the stock quote
12     ByteBuffer response = s.toByteBuffer(charset);
13     // and send it to the client
14     commChannel.write(response);
15   }
16   else // if symbol does not denote a valid stock
17   {
18     sendError(commChannel, "Stock symbol not found.");
19   }
20 }
21 else // if the request is not well formed
22 {
23   sendError(commChannel, "Badly formed request.");
24 }
25
26 // close the socket channel
27 commChannel.close();
```

Above, we first retrieve the complete request from the selection key in order to check it for well formedness and extract the request stock symbol out of it. A well formed client request looks like this:

```
<get-quote><symbol>YHOO</symbol></get-quote>
```

(to request the stock quote for Yahoo!). Regular expression matching is currently used to do this, although some form of XML processing could be employed for more flexibility (especially if other types of client requests were to be supported as well). Since this is not of interest for our discussion, it is not shown above. If the request is well formed, the local stocks hash map is searched for the stock symbol provided by the client and possibly a valid stock is retrieved. A `ByteBuffer` with the proper character set encoding is prepared and returned by the `toByteBuffer` method of the `Stock` class and this buffer is then written

to the socket channel associated with the selection key. If anything goes wrong, an error message is sent to the client instead of a stock quote. In either case, as soon as the response is sent, the socket channel with the client is closed, which also has the side effect of deregistering this channel from the communication selector.

The following `telnet` session illustrates how a client can get a stock quote from this server:

```
$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
<get-quote>
<symbol>MSFT</symbol>
</get-quote>

<quote>
        <symbol>MSFT</symbol>
        <company-name>MICROSOFT CP</company-name>
        <last-price>27.28</last-price>
        <opening-price>27.28</opening-price>
        <volume>51945506</volume>
</quote>

Connection closed by foreign host.
```

while this one shows the case where the stock is not found:

```
$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
<get-quote>
<symbol>MSF</symbol>
</get-quote>

<error>Stock symbol MSF not found.</error>

Connection closed by foreign host.
```

The code which has been presented in this section illustrates how the new Java I/O can be used to implement a server based on the Reactor pattern. Although some details of the concrete stock quote server implementation were given, we believe to have managed to keep the explanation general enough to show the basic application skeleton that is to be used. Of course, many variations can be imagined, but they would all have to build on a similar skeleton.

4 Discussion

The new Java I/O offers everything the Reactor pattern does and then some more. First of all, the possibility to have any number of selectors in use at any

time basically multiplies the reactor instances which can be used at the same time. Thus, Java allows either the use of the basic Reactor pattern (if only one selector is created) or a variation of the pattern in which multiple selectors are in use (this was also the case with our stock quote server, which used two selectors simultaneously).

Java also easily permits extending the Reactor pattern to the Active Object pattern by simply creating a new thread to handle each event and having the "main" thread return to blocking on the `select` call immediately. This could prove useful in cases when processing an event takes a lot of time and by doing this in the main thread we would be delaying other clients for unacceptably long periods of time.

Selection keys introduce a lot of extra flexibility by allowing a more direct control over the behavior of a selector. Selection keys contained by a selector can be easily manipulated to influence the types of events the selector is monitoring without having to resort to extra registrations. This is a quite welcome functionality, which we have shown in use in the stock quote server code. Another useful thing made available by Java are the attachments that can be associated with selection keys. They can be used to easily maintain state between the processing of events, without having to worry about creating external structures for that.

It is also interesting to point out that Java new I/O offers a mechanism to wake up a blocked selector programatically, which is again something additional to the basic Reactor pattern. We have seen this at work as a useful communication mechanism in the code of the stock quote server. On the other hand, a version of the `select` call which takes a timeout as a parameter is also available, adding to the flexibility of the platform. Even though encoded in a different way than in the Reactor pattern (which was aware of timeout events as normal events), its usefulness is not restricted in any way.

Ultimately, if one were to need this, the Reactor pattern as it is described in [4] can be implemented in Java, although this would take away much of the flexibility. As a sketch for implementation, the `Selector` would naturally fulfill the role of the Reactor, and a call to `register_handle` would be translated to a registration of that handle (which in Java would be a `SelectableChannel`) with the `Selector` playing the role of the Reactor. The handler itself could be stored as the object attached to the selection key, and upon return from a `select` call, it could be used to turn the default pull mode of the Java new I/O into the push mode requested by the "original" Reactor pattern. The code for the translation would look something like this:

```
1 Iterator<SelectionKey> keysIterator;
2 keysIterator = commSelector.selectedKeys().iterator();
3 while (keysIterator.hasNext())
4 {
5   SelectionKey key = (SelectionKey) keysIterator.next();
6   keysIterator.remove();
7   EventHandler eh = (EventHandler) key.attachment();
8   eh.handleEvent(key.readyOps());
9 }
```

Of course, further adaption of event types could be necessary, but this would merely be an issue of mapping an element of an enumeration to an element of

another enumeration. Naturally, the Reactor pattern backed by Java would only be able to handle the events supported by the Java implementation: `OP_ACCEPT`, `OP_CONNECT`, `OP_READ` and `OP_WRITE`. In addition to these, timeout events could also be simulated by calling the version of `select` which takes a timeout value as a parameter.

5 Conclusion

In light of the things discussed throughout this article, it is straightforward to conclude that the Java new I/O package is in essence a much improved implementation of the Reactor pattern. It provides virtually all the functionality of the pattern, even though in a somewhat different manner, and it also brings a number of new capabilities which make it even more flexible.

We have shown how the mechanisms provided by Java can be used to successfully implement a server based on non-blocking, multiplexed I/O and while doing so we have illustrated many of the basic concepts that have to be understood for making correct use of the platform. We have both theoretically and by example made the connection between the Java classes and the participants in the original Reactor pattern, thus further substituting our claim that the new Java I/O is indeed an instance of the pattern.

Finally, we have discussed how what Java offers supersedes the Reactor pattern in many ways, and we have demonstrated how all these extras can be put to practical use in our sample stock quote server.

References

[1] Eckel, B. Thinking in Enterprise Java. http://www.javareference.com/books/freebooks/TIE/TIEJv1.1.htm.

[2] Eckel, B. *Thinking in Java*. Prentice Hall PTR, Upper Saddle River, 1998.

[3] Michael T. Nygard. Master Merlin's new I/O classes. http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html.

[4] Schmidt, D. C. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. 529–545.

[5] Schmidt, D. C. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Commun. ACM 38*, 10 (1995), 65–74.

[6] Sun Microsystems, Inc. Java 2 Platform Standard Edition 5.0 API Specification. http://java.sun.com/j2se/1.5.0/docs/api/.

A Source Code of the Stock Quote Server

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SelectionKey;
```

```
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.charset.CharacterCodingException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.util.HashMap;
import java.util.Iterator;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StockServer
{
    public static final int PORT = 10000;

    private HashMap<String, Stock> stocks;

    public static void main(String args[]) throws IOException
    {
        new StockServer();
    }

    public StockServer() throws IOException
    {
        initializeStocks();

        BlockingQueue<SocketChannel> channelQueue;
        channelQueue = new LinkedBlockingQueue<SocketChannel>();

        Selector commSelector = Selector.open();

        new AcceptorThread(PORT, commSelector, channelQueue).start();
        new CommunicationThread(commSelector, channelQueue, stocks).start();
    }

    private void initializeStocks()
    {
        stocks = new HashMap<String, Stock>();

        stocks.put("CSCO", new Stock("CSCO", "CISCO SYS INC", 17.47, 17.47, 82869070));
        stocks.put("MSFT", new Stock("MSFT", "MICROSOFT CP", 27.28, 27.28, 51945506));
        stocks.put("DELL", new Stock("DELL", "DELL INC", 29.40, 29.40, 50636645));
        stocks.put("INTC", new Stock("INTC", "INTEL CP", 25.13, 25.13, 40317625));
        stocks.put("ORCL", new Stock("ORCL", "ORACLE CORP", 12.81, 12.81, 30479264));
        stocks.put("JDSU", new Stock("JDSU", "JDS UNIPHASE CP", 2.29, 2.29, 21820431));
        stocks.put("PTEN", new Stock("PTEN", "PATTERSON-UTI ENER", 30.00, 30.00, 21773211)
        stocks.put("SUNW", new Stock("SUNW", "SUN MICROSYS INC", 3.70, 3.70, 20790943));
        stocks.put("AMAT", new Stock("AMAT", "APPLIED MATERIALS", 17.96, 17.96, 17243503))
```

```
        stocks.put("CPST", new Stock("CPST", "CAPSTONE TURBINE C", 3.13, 3.13, 16493719));
        stocks.put("AAPL", new Stock("AAPL", "APPLE COMPUTER", 61.54, 61.54, 15196067));
        stocks.put("SIRI", new Stock("SIRI", "SIRIUS SATELLITE R", 7.00, 7.00, 14892773));
        stocks.put("EBAY", new Stock("EBAY", "EBAY INC", 43.89, 43.89, 14200480));
        stocks.put("QCOM", new Stock("QCOM", "QUALCOMM INC", 45.42, 45.42, 12494535));
        stocks.put("YHOO", new Stock("YHOO", "YAHOO INC", 38.49, 38.49, 12234337));
        stocks.put("PLAY", new Stock("PLAY", "PORTALPLAYER INC", 24.17, 24.17, 10873040));
        stocks.put("JNPR", new Stock("JNPR", "JUNIPER NETWORKS", 23.99, 23.99, 10363597));
        stocks.put("CMCSK", new Stock("CMCSK", "COMCAST CL A SPCL", 26.29, 26.29, 9818082)
        stocks.put("SYMC", new Stock("SYMC", "SYMANTEC CP", 19.61, 19.61, 9732226));
        stocks.put("CMCSA", new Stock("CMCSA", "COMCAST CP A", 26.93, 26.93, 9635642));
        stocks.put("SNDK", new Stock("SNDK", "SANDISK CP", 60.98, 60.98, 9088704));
        stocks.put("ATML", new Stock("ATML", "ATMEL CORP", 2.66, 2.66, 8068010));
        stocks.put("SRNA", new Stock("SRNA", "SERENA SOFTWARE IN", 23.50, 23.50, 7765246))
        stocks.put("RIMM", new Stock("RIMM", "RESEARCH IN MOTION", 66.79, 66.79, 7660668))
    }
}

class AcceptorThread extends Thread
{
    private Selector acceptSelector;
    private Selector commSelector;
    private BlockingQueue<SocketChannel> channelQueue;

    public AcceptorThread(int port, Selector commSelector,
            BlockingQueue<SocketChannel> channelQueue)
        throws IOException
    {
        super();

        this.commSelector = commSelector;
        this.channelQueue = channelQueue;

        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.configureBlocking(false);

        InetSocketAddress address = new InetSocketAddress(port);
        ssc.socket().bind(address);

        System.out.println("Bound to " + address);

        acceptSelector = Selector.open();
        ssc.register(acceptSelector, SelectionKey.OP_ACCEPT);
    }

    public void run()
    {
        while (true)
        {
            try
```

```
            {
                System.out.println("Acceptor thread: Selecting");

                acceptSelector.select();

                process();
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }

    private void process() throws Exception
    {
        Iterator<SelectionKey> keysIterator;
        keysIterator = acceptSelector.selectedKeys().iterator();

        while (keysIterator.hasNext())
        {
            SelectionKey key = keysIterator.next();
            keysIterator.remove();

            ServerSocketChannel acceptChannel = (ServerSocketChannel) key.channel();
            SocketChannel commChannel = acceptChannel.accept();

            System.out.println("Acceptor thread: Connection from "
                    + commChannel.socket().getInetAddress());

            channelQueue.put(commChannel);
            commSelector.wakeup();
        }
    }
}

class CommunicationThread extends Thread
{
    private static final int READ_BUFFER_SIZE = 16;
    private Selector commSelector;
    private BlockingQueue<SocketChannel> channelQueue;
    private HashMap<String, Stock> stocks;
    private ByteBuffer readBuffer;
    private Charset charset;
    private CharsetDecoder decoder;

    public CommunicationThread(Selector commSelector,
            BlockingQueue<SocketChannel> channelQueue, HashMap<String, Stock> stocks)
        throws IOException
    {
```

```
        super();

        this.commSelector = commSelector;
        this.channelQueue = channelQueue;
        this.stocks = stocks;

        this.readBuffer = ByteBuffer.allocateDirect(READ_BUFFER_SIZE);

        String encoding = System.getProperty("file.encoding");
        charset = Charset.forName(encoding);
        decoder = charset.newDecoder();
    }

    public void run()
    {
        while (true)
        {
            try
            {
                System.out.println("Communication thread: Selecting");

                registerNewChannels();

                commSelector.select();

                process();
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
    }

    private void registerNewChannels() throws Exception
    {
        SocketChannel channel;
        while ((channel = channelQueue.poll()) != null)
        {
            channel.configureBlocking(false);
            channel.register(commSelector, SelectionKey.OP_READ, new StringBuilder());
        }
    }

    private void process() throws Exception
    {
        Iterator<SelectionKey> keysIterator;
        keysIterator = commSelector.selectedKeys().iterator();

        while (keysIterator.hasNext())
```

```
    {
        SelectionKey key = (SelectionKey) keysIterator.next();
        keysIterator.remove();

        if (key.isReadable())
        {
            processRequest(key);
        }
        if (key.isWritable())
        {
            processCompletedRequest(key);
        }
    }
}

private void processRequest(SelectionKey key) throws Exception
{
    SocketChannel commChannel = (SocketChannel) key.channel();

    try
    {
        boolean eof = commChannel.read(readBuffer) == -1;

        readBuffer.flip();
        String result = decoder.decode(readBuffer).toString();
        readBuffer.clear();

        System.out.println("Communication thread: Processing partial request");
        System.out.println("---");
        System.out.println(result);
        System.out.println("---");

        StringBuilder requestString = (StringBuilder) key.attachment();
        requestString.append(result);

        String request = requestString.toString();

        if (request.endsWith("\n\n") || request.endsWith("\r\n\r\n") || eof)
        {
            int oldInterestOps = key.interestOps();
            int newInterestOps = oldInterestOps | SelectionKey.OP_WRITE;
            key.interestOps(newInterestOps);
        }
    }
    catch (IOException ioe)
    {
        sendError(commChannel, "An internal server error occured while processing your
    }
}
```

```
private void processCompletedRequest(SelectionKey key)
    throws IOException
{
    String request = key.attachment().toString();
    SocketChannel commChannel = (SocketChannel) key.channel();

    System.out.println("Communication thread: Processing full request");
    System.out.println("---");
    System.out.println(request);
    System.out.println("---");

    // looking for something like: <get-quote><symbol>CSCO</symbol></get-quote>
    // with random whitespace in between
    Pattern p =
        Pattern.compile("\\s*<get-quote>\\s*<symbol>\\s*([A-Z]+)\\s*</symbol>\\s*</get
    Matcher m = p.matcher(request);

    if (m.matches())
    {
        Stock s = stocks.get(m.group(1));
        if (s != null)
        {
            ByteBuffer response = s.toByteBuffer(charset);
            commChannel.write(response);
        }
        else
        {
            sendError(commChannel, "Stock symbol " + m.group(1) + " not found.");
        }
    }
    else
    {
        sendError(commChannel, "Badly formed request.");
    }

    commChannel.close();
}

private void sendError(SocketChannel channel, String message)
    throws IOException
{
    StringBuilder temp = new StringBuilder();
    temp.append("<error>");
    temp.append(message);
    temp.append("</error>\n\n");

    CharBuffer result = CharBuffer.allocate(temp.length());
    result.put(temp.toString());
    result.flip();
```

```
        ByteBuffer buffer = charset.newEncoder().encode(result);

        channel.write(buffer);
    }
}

class Stock
{
    private String symbol;
    private String companyName;
    private double lastPrice;
    private double openingPrice;
    private int volumeTraded;

    public Stock(String symbol, String companyName, double lastPrice, double openingPrice,
    {
        this.symbol = symbol;
        this.companyName = companyName;
        this.lastPrice = lastPrice;
        this.openingPrice = openingPrice;
        this.volumeTraded = volumeTraded;
    }

    public ByteBuffer toByteBuffer(Charset charset) throws CharacterCodingException
    {
        StringBuilder temp = new StringBuilder();
        temp.append("<quote>\n");
        temp.append("\t<symbol>");
        temp.append(symbol);
        temp.append("</symbol>\n");
        temp.append("\t<company-name>");
        temp.append(companyName);
        temp.append("</company-name>\n");
        temp.append("\t<last-price>");
        temp.append(lastPrice);
        temp.append("</last-price>\n");
        temp.append("\t<opening-price>");
        temp.append(openingPrice);
        temp.append("</opening-price>\n");
        temp.append("\t<volume>");
        temp.append(volumeTraded);
        temp.append("</volume>\n");
        temp.append("</quote>\n\n");

        CharBuffer result = CharBuffer.allocate(temp.length());
        result.put(temp.toString());
        result.flip();

        ByteBuffer buffer = charset.newEncoder().encode(result);
```

```
    return buffer;
}

public double getLastPrice()
{
    return lastPrice;
}

public void setLastPrice(double lastPrice)
{
    this.lastPrice = lastPrice;
}

public String getCompanyName()
{
    return companyName;
}

public void setCompanyName(String companyName)
{
    this.companyName = companyName;
}

public double getOpeningPrice()
{
    return openingPrice;
}

public void setOpeningPrice(double openingPrice)
{
    this.openingPrice = openingPrice;
}

public String getSymbol()
{
    return symbol;
}

public void setSymbol(String symbol)
{
    this.symbol = symbol;
}

public int getVolumeTraded()
{
    return volumeTraded;
}

public void setVolumeTraded(int volumeTraded)
{
```

```
        this.volumeTraded = volumeTraded;
    }
}
```