Published on ONJava.com (http://www.onjava.com/)
See this if you're having trouble printing code examples

Building Highly Scalable Servers with Java NIO

by Nuno Santos
09/01/2004

About one year ago, a client of the company where I work asked us to develop a router for telephony protocols (i.e., protocols used for communication between a SMS center and external applications). Among the requirements was that a single router should be able to support at least 3,000 simultaneous connections.

It was clear that we could not use the traditional thread-pooling approach. Most thread libraries do not scale well, because the time required for context switching increases significantly with the number of active threads. With a few hundred active threads, most CPU time is wasted in context switching, with very little time remaining for doing real work. As an alternative to thread pooling, we decided to use I/O multiplexing. In this approach, a single thread is used to handle an arbitrary number of sockets. This allows servers to keep their thread count low, even when operating on thousands of sockets, thereby improving scalability and performance significantly. Unfortunately, there is a price to pay: an architecture based on I/O multiplexing is significantly harder to understand and to implement correctly than one based on thread pooling.

The support for I/O multiplexing is a new feature of Java 1.4. It builds on two features of the Java NIO (New I/O) API: selectors and non-blocking I/O. The article "Introducing Nonblocking Sockets" provides a good introduction to these two features.

In this article, we describe the lessons we learned while designing and implementing our router, focusing on architectural issues such as I/O event dispatching, threading, management of client data, and protocol state. This is not an introductory article; the intended audience is developers that already have a basic knowledge of I/O multiplexing and Java NIO, but haven't yet used those technologies to develop a full-scale server.

The article includes the source code for a echo server and client based on the architecture described. Both the server and the client are functional and can be complied and executed without any modification. The source code can also be used as a starting point to develop a full server.

Related Reading

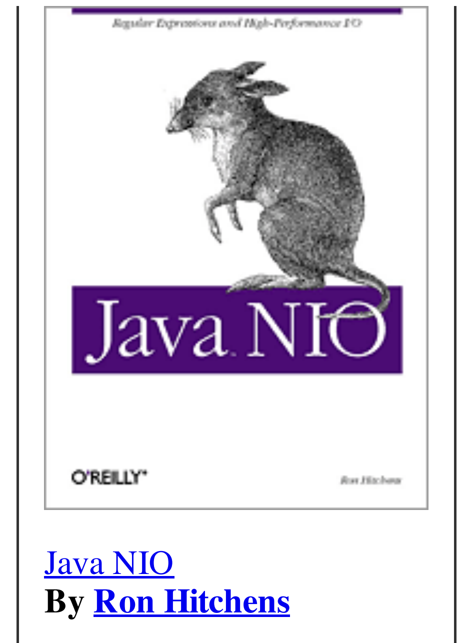
I/O Event Handling

The I/O architecture of our router was strongly inspired by the Swing event-dispatch model. In Swing, events generated by the user interface are received by the JVM and stored in an event queue. Inside of the

JVM, an event dispatch thread (implemented in the class `java.awt.EventQueue`) monitors this queue and dispatches incoming events to interested listeners. This is a typical example of the Observer pattern.

In our router, there is also an event dispatch thread, implemented in the class `SelectorThread`. As the name suggests, this class encapsulates a selector and a thread. The thread monitors the selector, waiting for incoming I/O events and dispatching them to the appropriate handlers.

The `SelectorThread` class generates four types of events, corresponding to the operations defined on `java.nio.channels.SelectionKey`: connect, accept, read, and write. Handlers register with the `SelectorThread` class to receive events. Depending on the type of events they are interested in, they must implement one of the following interfaces:



[Java NIO](#)
By [Ron Hitchens](#)

- `ConnectSelectorHandler`: For establishing outgoing connections.
- `AcceptSelectorHandler`: For receiving incoming connections.
- `ReadWriteSelectorHandler`: For reading or writing data to a connection.

Figure 1 describes the class hierarchy of these interfaces. We chose not to define a single interface for all of the possible events because a single handler will likely be only interested in some of the events. For instance, a handler that accepts connections will most likely not need to establish connections. This separation allows handlers to implement only the operations they really need.

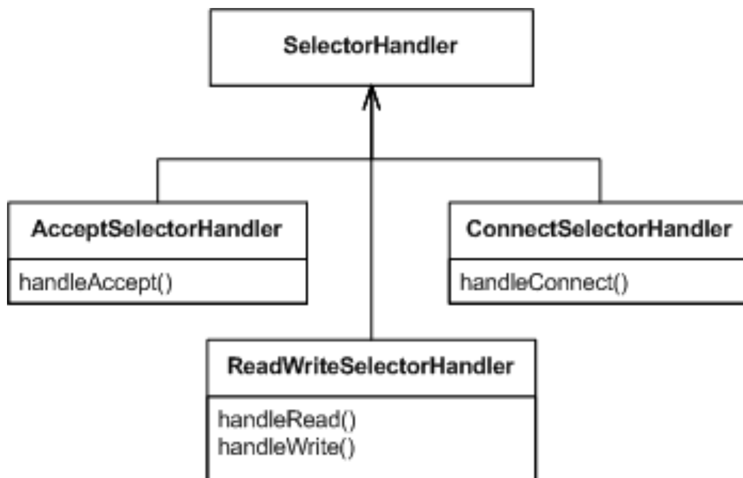


Figure 1. Class hierarchy for I/O event handlers

The Life of a Handler

One important difference from the thread-per-client model is that all read and write operations are non-blocking, forcing the programmer to deal with partial reads and writes. When the handler receives a read event, it means only that there are some bytes available in the socket's read buffer. This data may contain either part of a packet, a full packet, or more than one packet. All cases have to be considered while reading. A similar situation occurs when writing. It is only possible to write as much as the space available on the socket's write buffer. A call to write will return as soon as the buffer space is exhausted, regardless of

whether the data has been fully written or not. This has a direct impact on the lifecycle of a handler, which needs to deal with all of these situations.

A handler is basically a state machine reacting to I/O events. Its typical lifecycle is the following:

1. **Waiting for data**
   The handler is interested in reading but not in writing, since there is nothing to send to the client. Therefore, it activates read interest and waits for the read event.

2. **Reading**
   After receiving the read event, the handler retrieves the available data from the socket and starts reassembling the request. During this state, it is not interested in receiving any type of event. If a packet is fully reassembled, it starts processing it (state 3). Otherwise, it saves the partial packet, reactivates read interest, and continues waiting for data (state 1).

3. **Processing request**
   The handler enters this state whenever a request is fully reassembled. While here, the handler is not interested in either reading or writing (assuming that it only processes a request at a time). All interest in I/O events is disabled.

4. **Writing**
   When the reply is ready, the handler tries to send it immediately using a non-blocking write. If there is not enough space on the socket's write buffer to hold the entire packet, it will be necessary to send the rest later (step 5). Otherwise, the packet is sent and the handler can reactivate read interest, waiting for the next packet.

5. **Waiting to write**
   When a non-blocking write returns without having written all of the data, the handler activates interest in the write event. Later, when there is space available in the write buffer, the write event will be raised and the handler will continue writing the packet.

Figure 2 shows the state transition diagram of a handler.

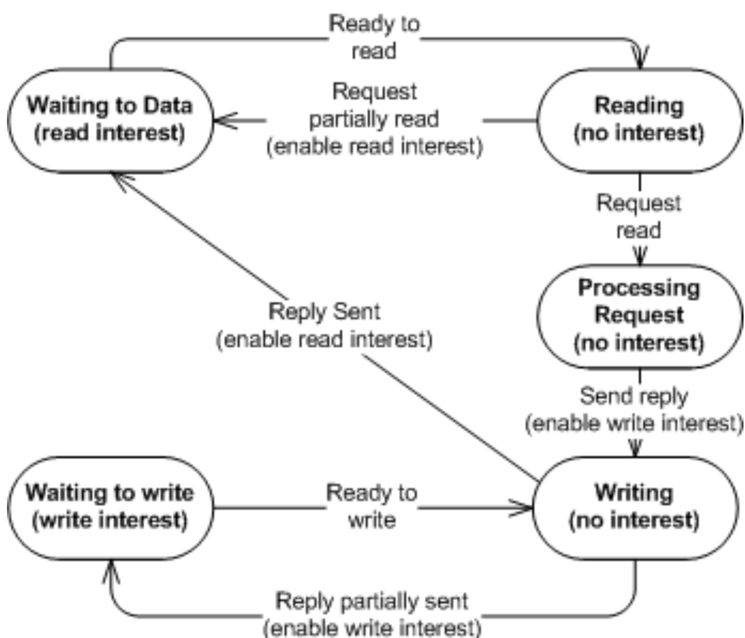


Figure 2. State transition diagram of an handler

Dispatching I/O Events

The `SelectorThread` class is responsible for supporting the lifecycle of the handlers. For that, it manages the following information for each handler:

- **A `SelectableChannel`**: The channel to be monitored for events.
- **The handler itself**: The object to be notified of events.
- **An interest set**: The set of operations to be monitored.

Handlers must provide these elements when they register themselves. The `SelectorThread` will then register the channel with the internal selector, using the interest set to activate monitoring of the corresponding I/O operations. The handler is stored as an attachment, which is a convenient way of associating application data with a registered channel. Internally, the following method call is performed to register a handler:

```
channel.register(selector, interestSet, handler);
```

After being registered, handlers can activate or deactivate interest in specific I/O events by updating their interest sets. Internally, the `SelectorThread` class updates the interest set of the corresponding `SelectionKey`. There is no support for de-registering a channel, since this can be easily accomplished by closing the socket.

Here is what the `SelectorThread` class looks like:

```
public class SelectorThread implements Runnable {

  /**
   * Graceful shutdown.
   */
  public void requestClose() {
    ...
  }

  /**
   * Adds a new interest to the list of events
   * where a channel is registered.
   */
  public void addChannelInterestNow(
      SelectableChannel channel,
      int interest) throws IOException {
    ...
  }

  /**
   * Like addChannelInterestNow(), but executed
   * asynchronously on the selector thread.
   */
  public void addChannelInterestLater(
      SelectableChannel channel,
      int interest,
```

```
        CallbackErrorHandler errorHandler) {
      ...
    }

    /**
     * Removes an interest from the list of events
     * where a channel is registered.
     */
    public void removeChannelInterestNow(
        SelectableChannel channel,
        int interest) throws IOException {
        ...
    }

    /**
     * Like removeChannelInterestNow(), but executed
     * asynchronously on the selector thread.
     */
    public void removeChannelInterestLater(
        SelectableChannel channel,
        int interest,
        CallbackErrorHandler errorHandler)  {
      ...
    }

    /**
     * Like registerChannelLater(), but executed
     * asynchronously on the selector thread.
     */
    public void registerChannelLater(
        SelectableChannel channel,
        int selectionKeys,
        SelectorHandler handlerInfo,
        CallbackErrorHandler errorHandler) {
      ...
    }

    /**
     * Registers a SelectableChannel with this
     * selector.
     */
    public void registerChannelNow(
        SelectableChannel channel,
        int selectionKeys,
        SelectorHandler handlerInfo)
        throws IOException {
      ...
    }

    /**
     * Executes the given task in the selector
     * thread. Does not wait for its execution.
     */
    public void invokeLater(Runnable run) {
      ...
    }

    /**
```

```
   * Executes the given task synchronously in the
   * selector thread, waiting for its execution.
   */
  public void invokeAndWait(final Runnable task)
    throws InterruptedException
  {
    ...
  }

  /**
   * Main cycle. This is where event processing
   * and dispatching happens.
   */
  public void run() {
    ...
  }
}
```

The purpose of the `invoke*()` methods and of the two variants (`*Now()` and `*Later()`) for most of the public methods will be explained shortly.

Threading in a Multiplexed World

In theory, with I/O multiplexing it is possible to have a single thread do all of the work in a server application. In practice, that is a very bad idea. When using a single thread, it is not possible to hide the latency of disk I/O (Java NIO does not support non-blocking file operations) or to take advantage of systems with multiple CPUs. As a rough guideline, a server application should have at least `2*n` threads, with `n` being the number of execution units available. Therefore, we had to implement a way of dividing the work among threads.

Getting the threading model right was the hardest part of the development. We considered the following architectures:

- **`M` dispatchers/no workers**
  Several event-dispatch threads doing all the work (dispatching events, reading, processing requests, and writing).

- **1 dispatcher/`N` workers**
  A single event-dispatch thread and multiple worker threads.

- **`M` dispatchers/`N` workers**
  Multiple event dispatch threads and multiple worker threads.

In all cases, incoming connections are assigned to an event-dispatch thread (a `SelectorThread`) for the duration of their lives. In the first architecture, I/O events are fully processed by the event-dispatch thread. In the other two, the processing is delegated to worker threads.

The Complex Solution

Our initial approach was based on the `M-N` architecture. This proved to be a bad option. The main problem

was keeping the whole system thread-safe. There were many interaction points between dispatcher and worker threads, all of them requiring careful synchronization. An even worse problem is that the group formed by a selector and its associated selection keys is not safe for multithreaded access. If a selection key is changed in any way by a thread while another thread is calling its selector `select()` method, the typical result is the `select()` call aborting with an ugly exception. This happened often when worker threads closed channels and indirectly cancelled the corresponding selection keys. If `select()` is being called at that time, it will find a selection key that was unexpectedly cancelled and abort with a `CancelledKeyException`. This is perhaps the most important lesson we learned about I/O multiplexing and Java NIO: **A selector, its selection keys, and registered channels should never be accessed by more than one thread.**

We tried to enforce this rule by delegating to the selector thread all tasks related to selector structures. But after a while, the architecture was getting very complex, inefficient, and error-prone. We decided it was time to admit defeat and start over using a simpler architecture.

The Working Solution

The simplest way of avoiding the concurrency problems between dispatcher and worker threads was not to use worker threads at all and let the event-dispatch threads to do all of the work (the **`M` dispatchers/no workers** architecture option described above). What we lost in flexibility, we gained in simplicity. There was no more need to ensure thread safety on the handler or on the selector, and there was no need to delegate tasks between threads. The life of a selector thread is as simple as:

- Block on the select call. Return only when there are operations ready to be performed.
- Execute all operations that are ready. This includes accepting or establishing connections, reading, writing, registering new sockets with the selector, changing the set of operations monitored, and so on.
- Go back to the select call.

In the `M-N` architecture, steps 1 and 2 were happening simultaneously, creating many concurrency problems. Doing them in the same thread put an end to the concurrency bugs that plagued our first attempt.

However, this architecture has some issues that must be carefully considered.

It is necessary to ensure a good distribution of work between threads. We used a simple round-robin algorithm to assign incoming connections to event-dispatch threads. So far, this has been working fine. But in other situations, it may be necessary to take in consideration other factors such as the activity going on in each thread.

Another problem is preventing the event-dispatch threads from blocking for too long. This can happen if processing a request requires reading or writing to a disk or to a database. To minimize this problem, we increased the ratio of event-dispatch threads to CPUs (to four or five). This does not prevent a thread from blocking, but when that happens, it is more likely that another one will be ready to take over the idle processor time.

One final problem is that sometimes it is necessary for external threads to interact with objects managed by the event-dispatch threads. For instance, there may be a timer to close idle connections. But the timer's thread should not close the connection directly, for the reasons mentioned previously. The solution is to provide a way for external threads to schedule tasks for execution on the event-dispatch thread of a selector.

This is done by calling either `invokeLater()` or `invokeAndWait()` of the `SelectorThread` class. If these names seem familiar to you, it's because they exist in the `java.awt.EventQueue` class to serve a similar function in Swing.

The Main I/O Cycle

Now that all of the pieces are in place, we can go on to the main event-dispatch loop:

```
public void run() {
  while (true) {
    // Execute all the pending tasks.
    doInvocations();

    // Time to terminate?
    if (closeRequested) {
      return;
    }

    int selectedKeys = selector.select();
    if (selectedKeys == 0) {
      // Go back to the beginning of the loop
      continue;
    }

    // Dispatch all ready I/O events
    Iterator it = selector.selectedKeys().
      iterator();
    while (it.hasNext()) {
      SelectionKey sk = (SelectionKey)it.next();
      it.remove();
      // Obtain the interest of the key
      int readyOps = sk.readyOps();
      // Disable the interest for the operation
      // that is ready. This prevents the same
      // event from being raised multiple times.
      sk.interestOps(
        sk.interestOps() & ~readyOps);

      // Retrieve the handler associated with
      // this key
      SelectorHandler handler =
         (SelectorHandler) sk.attachment();

      // Check what are the interests that are
      // active and dispatch the event to the
      // appropriate method.
      if (sk.isAcceptable()) {
        // A connection is ready to be completed
        ((AcceptSelectorHandler)handler).
          handleAccept();

      } else if (sk.isConnectable()) {
        // A connection is ready to be accepted
        ((ConnectorSelectorHandler)handler).
          handleConnect();

      } else {
```

```
        ReadWriteSelectorHandler rwHandler =
          (ReadWriteSelectorHandler)handler;
        // Readable or writable
        if (sk.isReadable()) {
          // It is possible to read
          rwHandler.handleRead();
        }

        // Check if the key is still valid,
        // since it might have been invalidated
        // in the read handler (for instance,
        // the socket might have been closed)
        if (sk.isValid() && sk.isWritable()) {
          // It is read to write
          rwHandler.handleWrite();
        }
      }
    }
  }
}
```

This is what the loop does:

1. **Execute scheduled tasks**
   These are the tasks that were scheduled for execution on the event-dispatch thread by external threads (by calling `invokeLater()` or `invokeAndWait()`).

2. **Check if close was requested**
   Checks if it is time to close the selector. For graceful shutdowns.

3. **Call** `select()`
   Wait for I/O events.

4. **Dispatch I/O events**
   Obtains the set of selection keys with operations ready, iterates over them and dispatches the operations to the appropriate handlers.

When all ready selection keys are dispatched, it's time to go back to the beginning of the loop, repeating the process once more.

Conclusion

Developing a fully functional router based on I/O multiplexing was not simple. It took us three months to get the architecture right. Part of this time was spent learning an unfamiliar I/O model. The other part was spent dealing with its inherent complexity. In the end, we were quite happy with the results. In internal benchmarks, the router was able to handle up to 10,000 clients with no significant drop in throughput. For comparison, we implemented a version based on the thread-per-client model, which was only able to reach 3,000 clients, with a significant drop in throughput as the number of clients increased.

Should you use I/O multiplexing in your next server? It depends. If the number of simultaneous clients will never exceed more than one or two hundred, then go for a simpler threaded model. But if you plan to support hundreds or even thousands of simultaneous clients, you should definitely consider using I/O

multiplexing. If you do use it, we hope this article and the companion source code will help you avoid some of pitfalls of using I/O multiplexing with Java NIO.

Resources

- Sample code for this article
- G. Naccarato, "Introducing Nonblocking Sockets," O'Reilly Network, April 2002.
- R. Hitchens, "Top Ten New Things You Can Do with NIO," O'Reilly Network, October 2002.
- J. Zukowski, "New I/O Functionality for Java 2 Standard Edition 1.4," java.sun.com, December 2001.

Nuno Santos is a software engineer at WIT-Software, a software company in the mobile telecommunication networking field.

Return to ONJava.com.

Copyright © 2009 O'Reilly Media, Inc.