The Reactor Design Pattern

Objectives

We discuss the drawbacks of blocking sockets, and their impact on server scalability. We introduce non-blocking IO in Java, using the `java.nio` package, discuss the complications that arise because of the asynchronous nature of non-blocking IO and their impact on our message parsing algorithms. Finally, we present a generic server platform, following the Reactor design pattern, which is much more scalable than our earlier solutions (but can still be improved upon).

Non Blocking IO

In the previous lectures we have seen the Socket API, which defines the interaction with the RTE when a process wishes to use communication services. Namely, we employed three properties of Sockets:

- ServerSockets can accept incoming connections.
- Sockets can be used to send bytes (using their OutputStream)
- Sockets can be used to receive bytes (using their InputStream)

All of these operations were blocking, meaning that control did not return to the calling thread until the invocation of the operation has terminated. In more details:

- When calling the `accept()` method of a `ServerSocket`, the calling thread will be blocked until a new connection is established.
- When calling the `write(byte [] buffer)` method of a socket's `OutputStram`, the calling thread will be blocked until the entire content of `buffer` is sent over the network.
- When calling the `read(byte [] buffer)` method of a socket's `InputStram`, the calling thread will be blocked until the desired number of bytes (`buffer.length`) will be received.

The consequence of this blocking behavior is that a server that wishes to serve multiple clients needs at least one thread for each client, and one thread to accept (new) incoming connections. This poses obvious scalability problems, as the amount of resources used by the server process (threads, memory, CPU) will increase linearly as the number of connected clients increases.

We would like to design scalable and robust servers that are more scalable (have a better capability to grow in size while using a bounded amount of resources). A mechanism which would allow us to perform the following would go a long way:

- Check that a socket has some data available to read.
- Read the available data from the socket. Do not block! return immediately with an arbitrary amount of data.
- Check that a socket can send some data.
- Write some data through the socket. Do not block! write as much as you can, and return immediately.
- Check if a new connection has been requested. If so, accept it, without blocking!

We can partition such a solution into two logical parts: Readiness notification and Non-blocking input output. Modern RTEs supply both of these mechanisms; Given a set of sockets, the RTE is capable of telling us, for each socket, if there is data available for read, if the socket can send some data and, finally, if there is a new connection pending from this socket. In addition, the RTE offers us a non-blocking interface to `read` and `write` operations. To understand how non-blocking operations work, we first need to understand how the RTE internally manages IO for us.

When we ask the RTE to `write` some bytes through a socket (sending these bytes over the network), the RTE copies these bytes to a buffer internal to the RTE (there is such a buffer associated with each socket, called the output buffer). The RTE will then proceed to send bytes from this output buffer over the network. However, as memory is limited, so is this output buffer, and as the network is usually slower than a process, the process may fill its socket's output buffer more quickly than the RTE can send the bytes over the network. When the output buffer is full and the process tries to write more bytes, the RTE blocks the process until the output buffer will have enough free space to hold the new data. When using non-blocking write operations, the RTE simply copies as many bytes from the process as possible, filling the socket's output buffer, and notifying the process how many bytes have been copied. If some bytes need to be re-written, it is the responsibility of the process to invoke the write operation in a later time, when there is some more space at the socket's buffer.

On the other hand, `read`ing from a socket is slightly different. Here, we need to take into account that the RTE need to receive bytes over the network, and deliver these bytes to our process. We also need to take into account that the RTE needs to buffer these bytes until the process actually requests them using a `read` operation. This mechanism is implemented with the help of another buffer, allocated to each socket, called the input buffer. When the RTE receives incoming bytes destined to a specific socket, the RTE stores these bytes into the socket's input buffer. When the process requests to `read` from this socket, the RTE copies the bytes from the socket's input buffer into the process provided buffer. If the process requested more bytes than there are available, the RTE blocks the process until enough bytes have arrived from the network. When using non-blocking `read` operations, the RTE will copy as many bytes as are available in the socket's input buffer to the process, and notify the process with the number of bytes copied.

Input Buffer Overflow

As the input buffer has limited space, we need to take into account that the input buffer may be overflowed; this can happen when the process reads data more slowly than the data arrives from the network. When the input buffer is full, the RTE will discard any new data arriving from the network, with the effect that the sending side will retransmit the data later, possibly at a slower rate.

Java's NIO Package

Java's interface to non-blocking IO and readiness notification services of the RTE is encapsulated by the NIO package, which is part of every Java RTE starting from Java 1.4.

The NIO package provides convenient wrapper classes for readiness notification and non-blocking IO. We will start by giving only a high level overview, and leave more specific details to later.

Channels

Channels represent in NIO either a data source, a destination, or both. Examples of channels are `SocketChannel`, `ServerSocketChannel` and `FileChannel`, which represent sockets and files. Channels in java may either be blocking, or non-blocking. By default, newly created channels are set to blocking mode, and must be set manually to non-blocking mode.

Channels provide methods for writing and reading bytes. A `ServerSocketChannel` also provides an `accept()` method, which, in turn, returns a `SocketChannel` (in a similar fashion to the `accept()` method of the `ServerScoket` class).

Selectors

`Selector` is a class which implements readiness notification. Channels may be registered to a selector for specific readiness events; A channel may be registered for read readiness, write readiness or accept readiness. The selector can later be polled to get a list of ready channels, and the operation each channel is ready for.

A Channel ready for read guarantees that a read operation will return some bytes. A Channel ready for write guarantees that a write operation will write some bytes and a Channel ready for an accept guarantees that calling `accept()` will result in a new connection.

The `Selector` class abstracts a service given by the operating system under the system call `select` (or, on more modern operating systems, `epoll`). This function receives a collection of sockets and blocks until one of them is ready for reading or ready for writing. When the call returns, the caller is informed with the ready sockets.

Buffers

Buffers are wrapper classes used by NIO to represent data interchanged through a Channel. Buffers are usually backed by some king of an array. For example, we will use the `ByteBuffer` class extensively throughout our code, as `SocketChannel`s use `ByteBuffer`s for sending and receiving bytes.

Sample code to learn the behavior of the `java.nio` package can be found in Practical Session 11 .

Reactor Design Pattern

The complete code for this lecture can be download from here

The reactor design pattern comes to solve the scalability problems we encountered before. The reactor achieves this feat by employing non-blocking IO. In broad terms, the reactor maintains a set of sockets, which, using a selector, the reactor polls for readiness. For each such socket, the reactor attaches some state. Whenever there are bytes ready to be read from the socket, the reactor will read some bytes and transfer them to the specific protocol implementation used (as in the previous lecture, the reactor is protocol agnostic). In a similar fashion, if the socket is ready for writing, the reactor will see if the protocol requested to send some bytes, and if so, the reactor will send them. The final task implemented by the reactor is accepting new connections.

In the previous servers that we saw in the former lecture, a server handle each client using a ConnectionHandler which with the help of the `MessagingProtocol` and `MessageEncoderDecoder` classes is protocol agnostic.

The reactor server is not different, it defines a new `NonBlockingConnectionHandler` that will handle each client and will be protocol agnostic using the exact same `MessagingProtocol` and `MessageEncoderDecoder` that ware defined in the previous lecture. The NonBlockingConnectionHandler distinguish between IO processing which should take place on the selector thread and the protocol processing which should take place in a different thread.

The key difference between the reactor architecture and the one-thread-per-connection architecture we presented in the previous section is that the `NonBlockingConnectionHandler` becomes a passive object instead of being an active object. The `NonBlockingConnectionHandler` methods are executed by the main thread of the `Reactor` (a.k.a., the selector thread) in reaction to events relayed by the selector (hence the name of the reactor). This is possible because none of the `NonBlockingConnectionHandler` method blocks – they all execute very fast and consist only of copying bytes from one buffer to another. The task of actually parsing the bytes and processing the messages is delegated to different worker threads.

The reactor pattern attempt to fix the scalability issues discussed before by creating a pool of worker threads which handle protocol related tasks for the NonBlockingConnectionHandler. Unlike the thread-per-client or the fixed-thread-pool paradigms workers are not assigned each to a single connection only but instead shared with all the existing connections (i.e., whenever a ConnectionHandler have a protocol related task to do - one of the available worker threads will take and run it).

The main thread of the reactor performs the following:

1. Create a new thread pool
2. Create a new `ServerSocketChannel`, and bind it to a port.
3. Create a new `Selector`.
4. Register the `ServerSocketChannel` in the `Selector`, asking for `accept` readiness.
5. While(true)
   wait for notifications from the selector. For each notification arrived check:
   - **Accept notification** - the server socket is ready to accept a new connection so call `accept`. Now a new socket was created so register this socket in the `Selector`.
   - **Write notification** - For each socket which is ready for writing, check if the protocol asked to write some bytes. If so, try to write some bytes to the socket.
   - **Read notification** - For each socket which is ready for reading, read some bytes and pass them down to the protocol handler. The actual work done by the protocol will be achieved with the use of the thread pool; *e.g.*, protocol processing is assigned as a task for the pool.

Note that we must maintain direct mapping between socket channels and their associated handlers. As the `Selector` class allows us to attach an arbitrary object to a channel, which can later be retrieved, we just associate a `NonBlockingConnectionHandler` with each socket created when accepting a new connection.

The Reactor Class

`Reactor` is an active object. It is the heart of the architecture which connects the other components and triggers their operation. The key components of the `Reactor` are:

- The selector
- The thread pool executor

The `Reactor` thread listens to events from the selector. Initially, only a ServerSocketChannel is connected to the selector. The `Reactor` can, therefore, only react to accept events. The serve() method of the `Reactor` dispatches the events it receives from the selector, and reacts appropriately.

```
                                                101 lines ...
1. public class Reactor {
2.
3.     private final int port;
4.     private final Supplier<MessagingProtocol<T>> protocolFactory;
5.     private final Supplier<MessageEncoderDecoder<T>> readerFactory;
6.     private final ActorThreadPool<NonBlockingConnectionHandler<T>> pool;
7.     private Selector selector;
8.
9.     private Thread selectorThread;
10.    private final ConcurrentLinkedQueue<Runnable> selectorTasks = new ConcurrentLinkedQueue<>();
11.
12.    public Reactor(
13.            int numThreads,
14.            int port,
15.            Supplier<MessagingProtocol<T>> protocolFactory,
16.            Supplier<MessageEncoderDecoder<T>> readerFactory) {
17.
18.        this.pool = new ActorThreadPool<>(numThreads);
19.        this.port = port;
20.        this.protocolFactory = protocolFactory;
21.        this.readerFactory = readerFactory;
22.    }
23.
24.    @Override
25.    public void serve() {
26.        selectorThread = Thread.currentThread();
27.        try (Selector selector = Selector.open();
28.             ServerSocketChannel serverSock = ServerSocketChannel.open()) {
29.
30.            this.selector = selector; //just to be able to close
31.
32.            serverSock.bind(new InetSocketAddress(port));
33.            serverSock.configureBlocking(false);
34.            serverSock.register(selector, SelectionKey.OP_ACCEPT);
35.
36.            while (!selectorThread.isInterrupted()) {
37.                selector.select();
38.                runSelectionThreadTasks();
39.                for (SelectionKey key : selector.selectedKeys()) {
40.                    if (!key.isValid()) {
41.                        continue;
42.                    } else if (key.isAcceptable()) {
43.                        handleAccept(serverSock, selector);
44.                    } else {
45.                        handleReadWrite(key);
46.                    }
47.                }
48.                selector.selectedKeys().clear(); //clear the selected keys set so that we can know about new events
49.            }
50.        } catch (ClosedSelectorException ex) { //do nothing - server was requested to be closed
51.        } catch (IOException ex) {             //this is an error
```

```
            ex.printStackTrace();
        }
        System.out.println("server closed!!!");
        pool.shutdown();
    }

    void updateInterestedOps(SocketChannel chan, int ops) {
        final SelectionKey key = chan.keyFor(selector);
        if (Thread.currentThread() == selectorThread) {
            key.interestOps(ops);
        } else {
            selectorTasks.add(() -> {
                if(key.isValid())
                    key.interestOps(ops);
            });
            selector.wakeup();
        }
    }
    private void handleAccept(ServerSocketChannel serverChan, Selector selector) throws IOException {
        SocketChannel clientChan = serverChan.accept();
        clientChan.configureBlocking(false);
        final NonBlockingConnectionHandler handler = new NonBlockingConnectionHandler(
                readerFactory.get(),
                protocolFactory.get(),
                clientChan,
                this);
        clientChan.register(selector, SelectionKey.OP_READ, handler);
    }
    private void handleReadWrite(SelectionKey key) {
        NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
        if (key.isReadable()) {
            Runnable task = handler.continueRead();
            if (task != null) {
                pool.submit(task);
            }
        }
        if (key.isWriteable()) {
            handler.continueWrite();
        }
    }
    private void runSelectionThreadTasks() {
        while (!selectorTasks.isEmpty()) {
            selectorTasks.remove().run();
        }
    }
    @Override
    public void close() throws IOException {
        selector.close();
    }

}
```

Handling Accept Events

the handleAccept method is invoked by the `Reactor` main thread each time the ServerSocketChannel becomes acceptable. `accept()` obtains a `SocketChannel` from the ServerSocketChannel, and connects it to the selector. It then creates a `NonBlockingConnectionHandler` passive object to keep track of the state of the newly created connection. Finally, register the channel to the selector with OP_READ and the `NonBlockingConnectionHandler` attached to the selection key. This way, when the selector triggers an event on this channel, we will find easily the corresponding `NonBlockingConnectionHandler` object that keeps track of its current state.

```
                                                92 lines ...
public class NonBlockingConnectionHandler {
private static final int BUFFER_ALLOCATION_SIZE = 1 << 13; //8k
    private static final ConcurrentLinkedQueue<ByteBuffer> BUFFER_POOL = new ConcurrentLinkedQueue<>();

    private final MessagingProtocol<T> protocol;
    private final MessageEncoderDecoder<T> encdec;
```

```
    private final Queue<ByteBuffer> writeQueue = new ConcurrentLinkedQueue<>();
    private final SocketChannel chan;
    private final Reactor reactor;

    public NonBlockingConnectionHandler(
            MessageEncoderDecoder<T> reader,
            MessagingProtocol<T> protocol,
            SocketChannel chan,
            Reactor reactor) {
        this.chan = chan;
        this.encdec = reader;
        this.protocol = protocol;
        this.reactor = reactor;
    }
    public Runnable continueRead() {
        ByteBuffer buf = leaseBuffer();

        boolean success = false;
        try {
            success = chan.read(buf) != -1;
        } catch (ClosedByInterruptException ex) {
            Thread.currentThread().interrupt();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        if (success) {
            buf.flip();
            return () -> {
                try {
                    while (buf.hasRemaining()) {
                        T nextMessage = encdec.decodeNextByte(buf.get());
                        if (nextMessage != null) {
                            T response = protocol.process(nextMessage);
                            if (response != null) {
                                writeQueue.add(ByteBuffer.wrap(encdec.encode(response)));
                                reactor.updateInterestedOps(chan, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
                            }
                        }
                    }
                } finally {
                    releaseBuffer(buf);
                }
            };
        } else {
            releaseBuffer(buf);
            close();
            return null;
        }
    }
    public void close() {
        try {
            chan.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    public void continueWrite() {
        while (!writeQueue.isEmpty()) {
            try {
                ByteBuffer top = writeQueue.peek();
                chan.write(top);
                if (top.hasRemaining()) {
                    return;
                } else {
                    writeQueue.remove();
```

```
                }
            } catch (IOException ex) {
                ex.printStackTrace();
                close();
            }
        }
        if (writeQueue.isEmpty()) {
            if (protocol.shouldTerminate()) close();
            else reactor.updateInterestedOps(chan, SelectionKey.OP_READ);
        }
    }
    private static ByteBuffer leaseBuffer() {
        ByteBuffer buff = BUFFER_POOL.poll();
        if (buff == null) {
            return ByteBuffer.allocateDirect(BUFFER_ALLOCATION_SIZE);
        }
        buff.clear();
        return buff;
    }
    private static void releaseBuffer(ByteBuffer buff) {
        BUFFER_POOL.add(buff);
    }}
```

There are several important points that should be noted about the above code:

The first thing that we should notice in the `NonBlockingConnectionHandler` class is the `BUFFER_POOL` variable. When reading data from nio-channels it is recommended (for performance reasons) to use direct byte buffers (which reside outside of the garbage collector region and therefore one can simply pass pointer to them to the operation system to fill on read request). @NonBlockingConnectionHandler@s uses many of such buffers for a short period of time, since the creation/eviction cycle of a direct bytebuffer of the needed size is relatively costly operation, the `BUFFER_POOL` will cache the already created buffers for reuse. This concept follows the Flyweight design-pattern.

> A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory. Often some parts of the object state can be shared, and it is common practice to hold them in external data structures and pass them to the flyweight objects temporarily when they are used.

Next we should notice that both the `continueReading` and `continueWriting` are called from the selector thread (and therefore should only perform simple IO operations), the `continueRead` method can also return a Runnable that represents the protocol-related task that was created in response to the read event and should be executed in a worker thread.

Concurrency issues

The reactor as described above has some concurrency issues, lets take another look at the `handleReadWrite` method

```
                                            11 lines ...
private void handleReadWrite(SelectionKey key) {
        NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
        if (key.isReadable()) {
            Runnable task = handler.continueRead();
            if (task != null) {
                pool.submit(task);
            }
        } else {
            handler.continueWrite();
        }

    }
```

We can see that there may be a situation where two different threads are performing the tasks of the same connection - why this is a problem? Assume a client that send two messages M1 and M2 to the server. The server then, create two tasks T1 and T2 corresponding to the messages. Since two different threads may handle the task concurrently, it may happen that T2 will be completed before T1. This behavior will result in sending the response to M2 before the response to M1. This will most probably going to break our protocol.

How can we solve it? one possible solution is to create a queue of tasks for each connection handler and synchronized over it

```
                                            21 lines ...
class Reactor {
    ..
    ConcurrentMap<NonBlockingConnectionHandler, List<Runnable>> tasksQueue;
    ..

```

```
    private void handleReadWrite(SelectionKey key) {
        NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
        if (key.isReadable()) {
            Runnable task = handler.continueRead();
            if (task != null) {
                List<Runnable> tasks = tasksQueue.get(handler);
                tasks.add(task);
                pool.submit(() -> {
                     synchronized(handler){
                        tasks.remove(0).run();
                     }
                });
            }
        } else {
            handler.continueWrite();
        }
    }
```

This code, although working, has several problems.

1. The map `tasksQueue` must be maintained - when new connection is started one must add the corresponding task queue and when a connection ends one must delete it in order to avoid memory leaks.
2. The threads in the pool may block waiting for one another to handle tasks of the same connection instead of working on tasks of other connections in the meanwhile

In order to avoid the issues above we can design a new - more suitable for the task - thread pool, the Actor Thread pool.

you can think of the actor thread pool as if it run actions of actors in a play - one can submit new actions for actors and the pool will make sure that each actor will run its actions in the order they were received while not blocking other threads.

Lets first examine the code:

```
                                                59 lines ...
public class ActorThreadPool<T> {

    private final Map<T, Queue<Runnable>> acts;
    private final ReadWriteLock actsRWLock;
    private final Set<T> playingNow;
    private final ExecutorService threads;

    public ActorThreadPool(int threads) {
        this.threads = Executors.newFixedThreadPool(threads);
        acts = new WeakHashMap<>();
        playingNow = ConcurrentHashMap.newKeySet();
        actsRWLock = new ReentrantReadWriteLock();
    }
    public void submit(T act, Runnable r) {
        synchronized (act) {
            if (!playingNow.contains(act)) {
                playingNow.add(act);
                execute(r, act);
            } else {
                pendingRunnablesOf(act).add(r);
            }
        }
    }
    public void shutdown() {
        threads.shutdownNow();
    }
    private Queue<Runnable> pendingRunnablesOf(T act) {

        actsRWLock.readLock().lock();
        Queue<Runnable> pendingRunnables = acts.get(act);
        actsRWLock.readLock().unlock();

        if (pendingRunnables == null) {
            actsRWLock.writeLock().lock();
            acts.put(act, pendingRunnables = new LinkedList<>());
```

```
            actsRWLock.writeLock().unlock();
        }
        return pendingRunnables;
    }
    private void execute(Runnable r, T act) {
        threads.submit(() -> {
            try {
                r.run();
            } finally {
                complete(act);
            }
        });
    }
    private void complete(T act) {
        synchronized (act) {
            Queue<Runnable> pending = pendingRunnablesOf(act);
            if (pending.isEmpty()) {
                playingNow.remove(act);
            } else {
                execute(pending.poll(), act);
            }
        }
    }

}
```

The first thing to notice is that the ActorThreadPool uses WeakHashMap to hold the task queues of the actors, A weak hashmap is a special implementation of hashmap with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently from other Map implementations.

Like most collection classes, this class is not synchronized. And therefore we will guard access to it using the read-write lock: `actsRWLock`.

Internally, it uses a simple fixed executor service but in order to not add two task of the same act to the pool it maintain the `playingNow` set.

Using the ActorThreadPool in our reactor implementation will require the following modifications:

```
                                          15 lines ...
class Reactor {
    ..
    ActorThreadPool<NonBlockingConnectionHandler> pool;
    ..

    private void handleReadWrite(SelectionKey key) {
        NonBlockingConnectionHandler handler = (NonBlockingConnectionHandler) key.attachment();
        if (key.isReadable()) {
            Runnable task = handler.continueRead();
            if (task != null) {
                pool.submit(handler, task);
            }
        } else {
            handler.continueWrite();
        }
    }
```

Using this pool, we reduced the synchronization between the threads in the pool by a fair amount. the only method that is blocking and is executed by the pool threads is the `complete` method which acquire the `act` monitor but only for a very short amount of time. Can we remove the synchronization completely??