Practical Session #11 - Selector, Flyweight Pattern, Reactor Pattern

Tom Mahler
In This Recitation We’ll Cover

- Why and What is the Reactor Pattern?
- The Reactor
Why and What is the Reactor Pattern?
Most servers need to handle many clients simultaneously.

One way of achieving this is using a Multi-Threaded server, as seen in Tirgul 10.

In this design, one thread is in charge of accepting new connections, and opens a new thread for each new client.

Thus, handling 30 clients requires 31 threads.
Why and What is the Reactor Pattern?

This kind of a Multi-Threaded server is problematic for several reasons:

1. It's wasteful
   - Creating a new Thread is relatively expensive
   - Each thread requires a fair amount of memory
   - Threads are blocked most of the time waiting for network IO
Why and What is the Reactor Pattern?

- This kind of a Multi-Threaded server is problematic for several reasons:
  2. It's not scalable
    - A Multi-Threaded server can't grow to accommodate tens of thousands of concurrent requests
    - It's also extremely vulnerable to Denial Of Service attacks
Why and What is the Reactor Pattern?

This kind of a Multi-Threaded server is problematic for several reasons:

3. Poor availability
   - It takes a long time to create a new thread for each new client
   - Moreover, the response time degrades as the number of clients rises
Why and What is the Reactor Pattern?

- The Reactor Pattern is another (better) design for handling several concurrent clients.

- It's based on the observation that if a thread does not need to wait for Network IO, a single thread could easily handle tens of client requests.
Why and What is the Reactor Pattern?

Based on this observation, the Reactor Pattern:

1. Use Non-Blocking IO, so Threads don't waste time waiting for Network
2. Have one thread in charge of the Network:
   - Accepting new connections and handling IO
   - As the Network is non-blocking, read, write and accept operations "take no time", and a single thread is enough for all the clients
Why and What is the Reactor Pattern?

Based on this observation, the Reactor Pattern:

3. Have a fixed number of threads (for example in a thread pool), which are in charge of the protocol
   - These threads perform the message framing (tokenization) and message processing (what to do with each message)
   - Note that unlike the Multi-Threaded server, in this design a single thread may handle many clients
Selector

- A non-blocking IO such as `read()`, reads the numbers of currently available bytes, and returns
  - This means we can read 0 bytes, and this is what will probably happen most of the time, when a socket is empty

- We would like a way for our thread to wait until any of our channels is ready, and only then perform the `read()`, `write()` or `accept()`, only on the ready channel

- This is what the Selector class does
The Selector **registers** itself to a Channel

Registration means that the channel informs the Selector when some event happens

This event can be

"the channel is ready to be read"

"the channel can now be written to"

"the channel can now accept()"
Selector

- During registration, the channel is told which events to monitor.

- In addition, when registering a Selector to a channel, one can add an “attachment“:
  - An attachment is an object that we will have access to when the event occurs.
  - Usually, this object is associated with a task that should be performed.
Selector

Registration:

Selector selector = Selector.open(); // a new Selector
Object anAttachment = new Object();
socketChannel.register(selector, SelectionKey.OP_READ, anAttachment);
Bitmasks

- SelectionKey.OP_READ is a number representing the READ event

- Similarly, SelectionKey.OP_WRITE is a number representing the WRITE event

- How would we represent a "READ or WRITE" event?
One option would be to define another constant `SelectionKey.OP_READ_OR_WRITE`

But this is cumbersome

When checking for a value we would need to write something like:

```java
if (e == SelectionKey.OP_READ || e == SelectionKey.OP_READ_OR_WRITE)
```

Now think about an `OP_READ_OR_ACCEPT` event, etc
Bitmasks

- A better solution is to use a Bitmask

- We'll define the values of the flags so that their 1-bits won't overlap
  - For example:
    - OP_READ=1
    - OP_WRITE=2
    - OP_ACCEPT=4
Bitmasks

- We could then use bitwise operations
  - The "read or write" event will be:
    \[
    \text{OP\_READ} \mid \text{OP\_WRITE}
    \]
    - and its value will be 3
    - Then, when checking for a read event, it's enough to ask:
      \[
      \text{if } (e \& \text{OP\_READ} \neq 0)
      \]
Once a Selector is registered to some channels, one can use the “select()“ method

- This method blocks until at least one of the channels is ready for the registered event
- Then, a list of SelectionKeys is returned
- Each Selectionkey is associated with one event, and holds the attachment registered with the event
The Reactor
The Reactor

Now that we know how Java's non-blocking IO works, we'll take a closer look at the Reactor implementation which employs it.

It's the same implementation given in class.
The Reactor

We will concentrate on the "core" reactor classes:

- Reactor
- ConnectionHandler
- ConnectionAcceptor
- ProtocolTask
The Reactor

- The Reactor is the main thread, which is in charge of:
  - Reading
  - Writing
  - Accepting new connections

- Reading and Writing are encapsulated in the ConnectionHandler class, and accepting is encapsulated in the ConnectionAcceptor class
The Reactor

- The reactor thread also holds a thread-pool of workers (an Executor)

- These worker-threads are in charge of the protocol handling: they receive the bytes read from the network, and let the protocol do something with them

- The class which represents tasks submitted to the executor is ProtocolTask
The Reactor

- The ProtocolTask object is created along with the ConnectionHandler, and the same task object is reused over and over again.

- This is a bit different than what you saw in the beginning of the course, where each task was run once and then discarded.
The Reactor

Once some bytes were read by the reactor thread, the ConnectionHandler adds these bytes to its ProtocolTask, and passes the task to the thread pool.

The ProtocolTask passes the bytes to the tokenizer, which looks for complete messages.

Then, the protocol task goes over all the messages which are available in the tokenizer, and asks the protocol to process them.
The Reactor

- After the protocol process a message and return a response, the tokenizer is used to add framing information to the response, and then convert it into bytes.

- Then, the protocol-task thread adds the response bytes back to the ConnectionHandler.
The Reactor

- Notice that we don't have one thread per connection
- What we do have is one ConnectionHandler object per connection
- This object holds the socket, the task, the protocol and Tokenizer instances associated with this connection
The Reactor

- Also notice that we are using an extended version of the ServerProtocol presented in the previous practical session
The Reactor

This extended version (AsyncServerProtocol) has two additional functions:

1. boolean shouldClose() returns true after the terminating message has been encountered

   This is used to signal the connection handler that the protocol has done its work and it's time to send all the remaining bytes and quit
This extended version (AsyncServerProtocol) has two additional functions:

1. void connectionTerminated() is used to notify the protocol the client closed the connection, or that the network connection was lost

   This is called by the ConnectionHandler when it identifies a network error
Examples

See code...
(See Reactor, NonBlockingConnectionHandler, ActorThreadPool, NonBlockingConnectionHandler, Server)