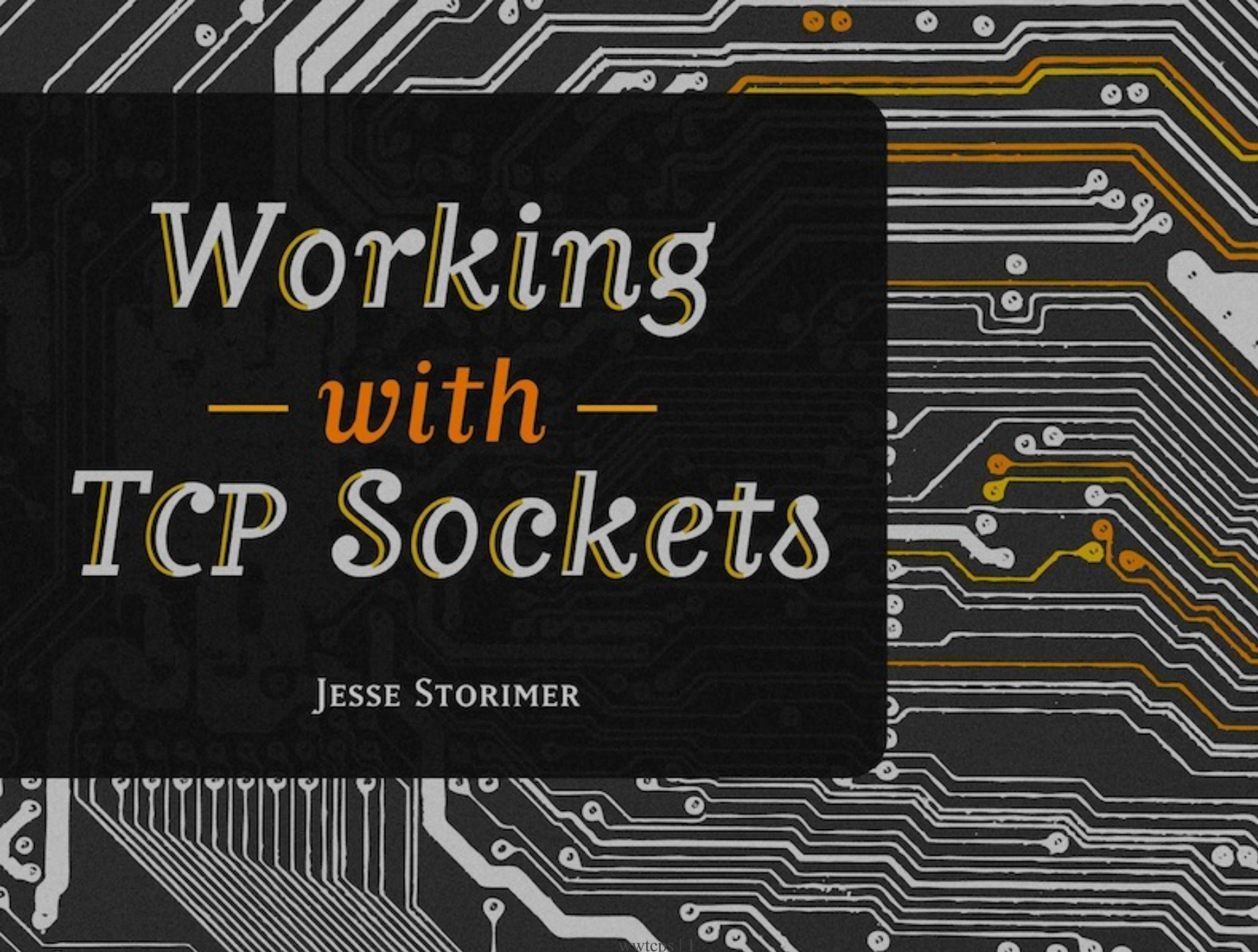
Working
— with —
TCP Sockets
Jesse Storimer

Chapter 21

Process per connection

This is the first network architecture we'll look at that allows parallel processing of requests.

Explanation

This particular architecture requires very few changes from the serial architecture in order to add concurrency. The code that accepts connections will remain the same, as will the code that consumes data from the socket.

The relevant change is that after accepting a connection, the server will `fork` a child process whose sole purpose will be the handling of that new connection. The child process handles the connection, then exits.

Forking Basics

Any time that you start up a program using `$ ruby myapp.rb`, for instance, a new Ruby process is spawned that loads and executes your code.

If you do a `fork` as part of your program you can actually create a new process *at runtime*. The effect of a `fork` is that you end up with two processes that are exact copies. The newly created process is considered the child; the original considered the parent. Once the `fork` is complete then you have two processes that can go their separate ways and do whatever they need to do.

This is tremendously useful because it means we can, for instance, `accept` a connection, `fork` a child process, and that child process automatically gets a copy of the client connection. Hence there's no extra setup, sharing of data, or locking required to start parallel processing.

Let's make the flow of events crystal clear:

1. A connection comes in to the server.
2. The main server process accepts the connection.
3. It forks a new child process which is an exact copy of the server process.
4. The child process continues to handle its connection in parallel while the server process goes back to step #1.

Thanks to kernel semantics these processes are running in parallel. While the new child process is handling the connection, the original parent process can continue to accept new connections and fork new child processes to handle them.

At any given time there will always be a single parent process waiting to accept connections. There may also be multiple child processes handling individual connections.

Implementation

```ruby
# ./code/ftp/arch/process_per_connection.rb
require 'socket'
require_relative '../command_handler'

module FTP
  class ProcessPerConnection
    CRLF = "\r\n"

    def initialize(port = 21)
      @control_socket = TCPServer.new(port)
      trap(:INT) { exit }
    end

    def gets
      @client.gets(CRLF)
    end

    def respond(message)
      @client.write(message)
      @client.write(CRLF)
    end

    def run
      loop do
        @client = @control_socket.accept

        pid = fork do
          respond "220 OHAI"

          handler = CommandHandler.new(self)

          loop do
            request = gets
```

```
            if request
              respond handler.handle(request)
            else
              @client.close
              break
            end
          end
        end

        Process.detach(pid)
      end
    end
  end
end

server = FTP::ProcessPerConnection.new(4481)
server.run
```

As you can see the majority of the code remains the same. The main difference is that the inner loop is wrapped in a call to `fork`.

```
                                                # ./code/ftp/arch/process_per_connection.rb
@client = @control_socket.accept

pid = fork do
  respond "220 OHAI"

  handler = CommandHandler.new(self)
```

Immediately after `accept`ing a connection the server process calls `fork` with a block. The new child process will evaluate that block and then exit.

This means that each incoming connection gets handled by a single, independent process. The parent process will not evaluate the code in the block; it just continues along the execution path.

```
                                                    # ./code/ftp/arch/process_per_connection.rb
  Process.detach(pid)
```

Notice the call to `Process.detach` at the end? After a process exits it isn't fully cleaned up until its parent asks for its exit status. In this case we don't care what the child exit status is, so we can detach from the process early on to ensure that its resources are fully cleaned up when it exits [2].

Considerations

This pattern has several advantages. The first is simplicity. Notice that very little extra code was required on top of the serial implementation in order to be able to service multiple clients in parallel.

A second advantage is that this kind of parallelism requires very little cognitive overhead. I mentioned earlier that `fork` effectively provides copies of everything a child process might need. There are no edge cases to look out for, no locks or race conditions, just simple separation.

An obvious disadvantage to this pattern is that there's no upper bound on the number of child processes it's willing to `fork`. For a small number of clients this won't be an issue, but if you're spawning dozens or hundreds of processes then your system will

2. If you want to learn more about process spawning and zombie processes then you should get my other book Working With Unix Processes.

quickly fall over. This concern can be solved using the *Preforking* pattern discussed a few chapters from now.

Depending on your operating environment, the very fact that it uses `fork` might be an issue. `fork` is only supported on Unix systems. This means it's *not* supported on Windows or JRuby.

Another concern is the issue of using processes versus using threads. I'll save this discussion for the next chapter when we actually get to see threads.

Examples

- shotgun
- inetd