Development of a Concurrent Ruby Webserver using Behaviour-Driven Development

CMT4141

Jesper Kjeldgaard
M00318018
Suporvisor: Kai Xu
School of Engineering and Information Sciences
Middlesex University, London, United Kingdom

October 2011

Abstract

In this project, a concurrent webserver is implemented in Ruby using the Behaviour-Driven Development (BDD) method.

Topics relevant to the thesis, namely Ruby webservers, concurrent programming, the Ruby programming language and BDD are covered. The implementation of each requirement is described, along with selected technical problems and their solutions.

Yarn, the name of the implemented webserver, is benchmarked to other Ruby webservers and the results revealed that for static requests it performed sub par, but for CPU intensive applications it performed well.

Using Behaviour-Driven Development (BDD) enabled getting started on the implementation right away, without having to produce a big design up-front. This proved to be both an advantage and a disadvantage, as some design decisions which turned out not to work as expected, might have been avoided. As BDD employs a test-first approach, a large test-suite was produced and this enabled design changes late in the development, without having to worry about the correctness of the implementation.

Contents

Chapter 1

Introduction

With the Internet growing each day in terms of content and users, the role of webservers and especially their performance is increasing. Furthermore, as websites must be able to service multiple visitors at a time, the webserver serving it must employ some form of concurrency. Various concurrency models exist, and each comes with a set of advantages and disadvantages all depending on the context of the webserver and the content being served. Concurrency both enables serving multiple requests at a time, but can also increase the performance of the webserver. Most computers has multi-core processors, but it is up to the webserver itself to properly utilize this to handle requests in parallel. This project aims at developing a concurrent webserver which utilizes multi-core processors and can handle parallel requests.

Besides developing a webserver, focus is put on using a development method throughout the project. Two agile test-first development methods are examined, and the best candidate will be used.

1.1 Readers Guide

Background information required for the remainder of the report is covered in Chapter 2. Chapter 3 covers the requirements for the software developed. Chapter 4 regards implementation specific aspects of the software and Chapter 5 describes the testing done during the development. In Chapter 6, the end product is demonstrated, evaluated through a series of benchmarks and compared to other Ruby webservers. In Chapter 7 the project process is reflected upon, and Chapter 8 concludes the project and suggests future improvements to the software. Summaries are included on larger chapters to recap the covered material.

For source code listings, the caption will show which file and on what line the code is taken from. The source code is included on a CD, and also available at `https://github.com/thejspr/yarn`.

1.2 Acknowledgements

I would like to thank Kai Xu for invaluable supervision and feedback throughout this project.

Chapter 2

Background

This chapter covers technology and methods used throughout this project and looks at the current Ruby webservers. Initially, the development method is described, then the topics; webservers, concurrent programming and Ruby are briefly introduced.

2.1 Development Method

Software development methods are often used to manage the scope and complexity of software projects. Some methods follow a linear approach and dictates first the system is designed, then implemented and finally tested. Whilst this approach works in many situations, it is critized for it's inability to handle difficulties arising in the process. Another type of methods grouped as agile, provides an interative process where the software is designed, developed and tested through a series of iterations [6]. Two agile methods were considered for this project; Behaviour-Driven Development (BDD) and Test-Driven Development (TDD).

BDD is an evolution of TDD which focus more on the behaviour of the software instead of the software structure. A problem with TDD is that it focus on the internals of the implementation structure, making tests fail even when the behaviour of the code hasn't changed. In short TDD tests what an object *is* and BDD tests what an object *does* [1]. Hence, BDD ensures business value is created and the software works as intended, TDD ensures that the code is correct but is blind to whether it provides the intended business value.

BDD was chosen as the development method for this project due to it being seen as an improvement on TDD and allowing for rapid development without needing to design the system up front. This agility makes it easier to handle difficulties and caveats during the process, leaving more room to adapt. With BDD (and TDD) the design is constantly evolving as more code is added and former code is constantly reviewed and refactored. This form of design is referred to as an emergent design. The following describes the BDD development cycle and the tools used [6].

2.1.1 Behaviour-Driven Development

BDD revolves around writing specifications for what a program is intended to do. This process is made up of an outer and an inner loop. The outer loop concerns passing an acceptance test covering one feature i.e. logging functionality. The inner loop concerns passing RSpec behaviour specifications describing the behaviour of methods and objects. Figure 2.1 depicts the BDD cycle.

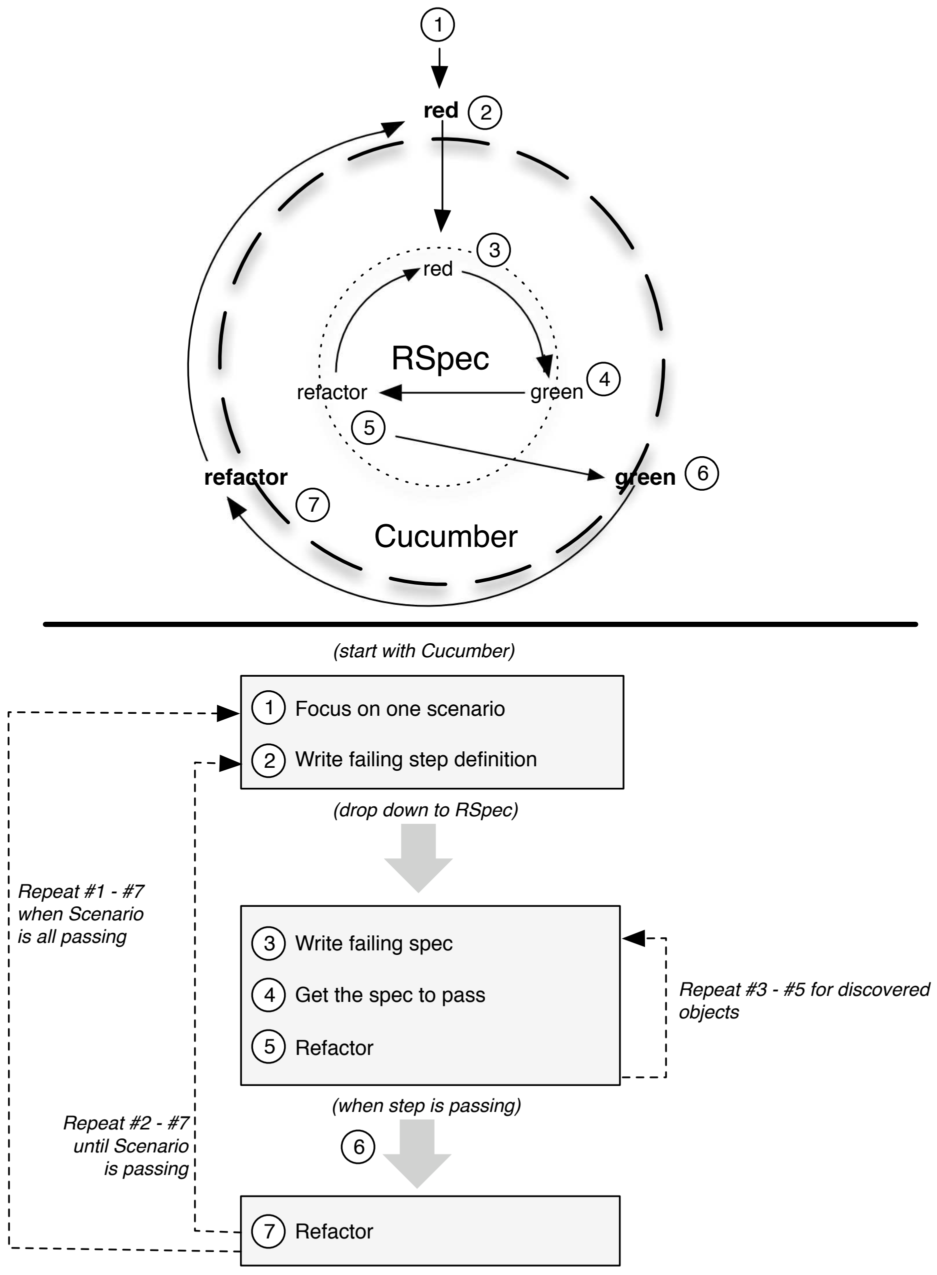


Figure 2.1: The BDD development cycle [1].

At the beginning of the iteration an acceptance test is written and executing it should reveal that the functionality is not yet implemented according to it's requirements. Cucumber is a framework for writing acceptance tests in a natural language as a set of steps in one or more scenarios. An example Cucumber acceptance test is included in Listing 2.1.

```
 1 Feature: Logging
 2   As a webdeveloper
 3   I want to have logging of HTTP requests
 4   In order to debug errors or attacks on the webserver
 5
 6   Scenario: Logging of HTTP requests
 7     Given the webserver is running
 8     When a user visits http://localhost/posts
 9     Then the log should contain "GET /posts"
10     And the log should contain "SENT /posts SUCCESS"
```

Listing 2.1: Cucumber acceptance test example

The acceptance test in Listing 2.1 covers the logging functionality of a webserver and tests it by starting the webserver, making a request and then checking whether the log contains the correct entry. The first line names the feature covered by the test. Lines 2 through 4 states the user of the feature, what it should do and lastly why and what business value it creates. Line 6 states a specific scenario, and often a feature has multiple scenarios and needs to test them separately. Scenarios contains separate steps which defines either a prerequisite (Given), an action (When) or a result (Then)[1]. The semantics of the steps are defined in a set of step definitions written in Ruby. Listing 2.2 is a step definition of the step used on line 9 in Listing 2.1.

```
1   Then /^the log should contain "([^"]*)"$/ do |entry|
2     Server::log.should have(entry)
3   end
```

Listing 2.2: Cucumber step definition

The step definition checks whether the server log contains given entry using RSpec. Steps are matched with regular expressions and `"([^"]*)"` matches the string "GET /posts" on line 9 in Listing 2.1. Step definitions returns true if it passes and false if it didn't, and in the case of a failure displays a detailed debug message.

RSpec is a BDD framework for Ruby used to write behaviour specifications (specs). Specs are to BDD what unit tests are to TDD. Once the acceptance test is written and run, steps will fail as they are not implemented (step 2 on Figure 2.1). Once implemented, steps might reveal application code which is not yet available and therefore fail. In the example above, this error could be caused by `Server` not having the method `log`. At this point (point 3 on Figure 2.1, an RSpec spec is

[1]"And" continues the previous step type

needed to define the behaviour of the `log` method. Listing 2.3 shows how such a spec might look.

```
1 describe Server do
2   describe "#log" do
3     it "should return an array" do
4       Server.log << "Test string"
5       Server.log.should.return(["Test string"])
6     end
7   end
8 end
```

Listing 2.3: RSpec (spec) example

Lines 1 and 2 in Listing 2.3 describes that we are specifying the `log` method on the `Server` class. Line 3 describes the intended behaviour of the `log` method; namely that it should return an array. Then a test string is written to the log enabling the assertion that calling the `log` method should contain that exact string. The next step in the BDD cycle is now to get the spec to pass (go from red to green) with the simplest solution possible. In case of the example, this would require implementing the `log` method to return an array of log entries. Once the spec passes, the code can be refactored if possible, and then the developer moves on to the next failing step definition. This process is continued until the acceptance test passes, meaning the feature is done and behaves as intended.

2.2 Webservers

The goal of a webserver is to listen on a port for incoming requests, handle them by either returning the contents of a file, run a program and return it's output, or return an error message if an error occurred. Since the Internet was made public over 20 years ago, there have been countless webservers created, most of which focus on handling websites written in a specific programming language or framework. Multi-core processors have long been available in server environments and hence, many webservers employ techniques to raise their performance by handling HTTP requests in parallel.

This section looks at some of the available Ruby webservers and the concurrency models they employ.

2.2.1 Rack

Rack is not an actual webserver, but an interface between webservers supporting Ruby and Ruby web-frameworks. It is covered here as the Ruby webservers covered next implement it. Rack decouples the link between the webserver and web-application (written in one of the many Ruby web-frameworks) and hereby makes it possible to change webserver without having to make any changes to the web application. This is achieved by wrapping requests and responses in a Ruby object. By implementing the Rack interface, a webserver would immediately become compatible with more than 15 Ruby frameworks [2].

2.2.2 Thin

Thin is a webserver written in Ruby which include a HTTP parser written in C, implements the Rack interface and uses EventMachine to handle requests in a concurrent non-blocking manner [4]. EventMachine is a library which enables event-driven (non-blocking) I/O by utilizing the reactor design pattern. The reactor pattern works by having a dispatcher (reactor) passing on work to various handlers, which then in turn fire an event when finished. This enables Thin to focus on other requests instead of waiting on blocking actions such as waiting for a file to be written to disk or fetching data from a remote API [11].

Thin works by running the EventMachine event loop in a single thread, and does therefore not employ any parallel processing, but does allow for concurrency due to the asynchronous nature of the reactor pattern. Thin is usually setup as a cluster of processes behind a proxy server or load balancer.

2.2.3 WEBrick

WEBrick is a webserver which is included in the Ruby standard library. It is written entirely in Ruby and employs concurrency by creating a separate thread for each incoming request. WEBrick is mostly used as a development server as it does not perform well under heavy loads due mainly to poor performance of it's HTTP parser [3].

2.2.4 Unicorn

Unicorn is a Ruby webserver which employs parallel processing by means of forking separate worker processes [15]. It uses the same C extension HTTP parser as Thin. By using processes, Unicorn allows for parallel processing without the need to used a proxy server or load balancer as load balancing is performed by the operating system.

2.3 Concurrent Programming

Concurrent programming has become an increasingly important issue over the last decade due to the introduction of multi-core processors. As CPU malefactors reached a plateau in the clock speed due to heat issues, they started adding more cores to CPU's to make them faster. This poses a problem for software developers as they now have to explicitly program to multi-processor architectures in order to reap their benefits [13].

Concurrency refers to code which is executed simultaneously on several processors, or cores in a single processor. Concurrent programming refers to programming language constructs which enables code to be executed concurrently. Concurrent code executed on a single-core processor will look like it's executed at the same time, but in reality the processor executes increments of each piece of code by switching back and forth between them. The way a processor switches between concurrent tasks is referred to as context-switching, and can easily bog down a system as the context (program state and variables) needs to be loaded for every context-switch. Concurrent code executed on a multi-core processor has the potential to run in parallel, e.g. at the same instance in time. True concurrency can occur on a computer with either multiple processors or one processor with multiple cores.

In most programming languages code is executed one statement at a time in a sequential manner. Most programming languages does however include constructs to allow for concurrent programming. One of the most common constructs is *threads*. A thread is an abstraction for a piece of code which is executed as a subroutine of the running program. This allows for executing several threads concurrently or in parallel. In Section 2.4, the constructs available in Ruby are covered in more detail.

Concurrency introduces several complications which must be taken into account, mainly; race conditions and deadlocks. A race condition occurs when two or more pieces of code tries to manipulate the same variable, and the outcome of the program changes depending on the order of the manipulations. An example would be if two threads, A and B, stored a counter in a global variable. Imagine thread A reads the value of the counter and then gets preempted by the scheduler and thread B changes the counter and saves it. Then when thread A gets resumed, it would have an incorrect counter value, and changing the counter based on this would render thread B's previous change. Race conditions can be fixed by synchronizing access to shared resources such as variables. A common way too synchronize access to shared resources is to add a mutual exclusion lock (mutex) around the code which accesses the shared resource. A mutex could ensure that only one thread uses the shared counter variable at a time. Deadlocks can occur when shared resources gets locked by separate threads which then each waits for the other thread to release the resource.

Race conditions occurring in a webserver could lead to erroneous data and deadlocks could result in the server becoming unresponsive. Hence, they should be avoided at all costs.

2.4 Ruby

Ruby is an dynamic, reflective object-oriented general-purpose programming language. It is dynamic in the sense that it is interpreted at runtime, and reflective in the sense that it can modify and inspect program behavior during runtime. Ruby has a dynamic type system where the type of the object is determined by what the object can do in terms of which methods are available for the object. This type system is referred to as duck typing, i.e. if it walks like a duck and talks like a duck, then the interpreter will treat it like a duck.

The latest version of Ruby, version 1.9, holds several improvements over the older but still maintained version 1.8. Besides small syntactic changes and including a new VM, Ruby 1.9 introduced the concept of fibers. Ruby 1.9 also includes RubyGems which enables easy packaging, installation and distribution of Ruby software. RubyGems, mostly referred to as *gems*, can be used to modify or extend functionality within a Ruby application or to split out reusable code for others to benefit from [9]. There are currently over 27.000 gems available and this greatly reduces the need to "reinvent the wheel", hence improving productivity.

Ruby was chosen for this project because it is dynamic, expressive, productive and terse, all adding to the speed and joyfulness of developing software. Furthermore, Ruby has several constructs available for concurrent programming which are covered below.

2.4.1 Concurrency in Ruby

Ruby comes with several constructs for concurrent programming; processes, threads and fibers. The following describes the advantages and possible disadvantages of each of these concepts.

Processes

A process is a running instance of a program, e.g. a running webserver or an open text editor. Processes are handled by the operating system which will utilize the available processors to run processes in parallel. Hence, parallelism can be achieved by running multiple instances (processes) of the same program.

Ruby includes a library for creating, killing and inspecting processes. Creating a new process can be done by *forking* a new process from the current one. Forking starts a new Ruby VM with the executing program as a new process. The forked and the original process does not have access to each other memory, and thereby eliminates many of the problems regarding race-conditions and deadlocks. The disadvantage of using multiple processes is that, as they cannot easily share state, and they have to communicate through an explicit protocol like inter-process communication [14].

Threads

A thread is an encapsulation of a set of instructions which execution can be managed. A thread can be seen as a lightweight process which has access to the current programs memory. Threads are managed by a thread-scheduler which decides when threads are run or preempted to enable other threads to run. Using threads introduces complex issues as race-conditions can easily occur due to all threads having access to the same memory.

An advantage of using threads is that they can easily share data, but that is also one of it's disadvantages as it can easily become quite complex and bugs concerning multi-threaded code can be very hard to discover [14].

Fibers

A fiber in Ruby is a coroutine mechanism which enables pausing and resuming code blocks to achieve cooperative concurrency. A code block is a piece of ruby code encapsulated in an object, and can be passed into methods, saved in variables, and executed on demand. Fibers resembles threads in that they can be controlled from outside, but whereas threads are managed by a scheduler, fibers must be scheduled manually.

The advantages of fibers over threads are that they have a significantly lower memory footprint and doesn't introduce the complexity of a scheduler [10].

2.4.2 Ruby Implementations

Ruby code is run in a virtual machine (VM) of which there exists several implementations. The official VM, for Ruby 1.9, is called YARV (Yet Another Ruby VM). As there is no specification for Ruby, YARV serves as the reference implementation for the Ruby language. The following describes two widely used Ruby implementations, namely YARV and JRuby, and how they differ in regards to concurrent programming.

Yet Another Ruby VM

YARV is developed and maintained as the official Ruby implementation and replaced the old VM MRI (Matz[2] Ruby Interpreter) as of Ruby version 1.9. Given YARV is the one driving the Ruby language implementation, it always has the latest language features and updates, whereas other Ruby implementations tend to be a bit behind. The latest stable version of YARV is 1.9.2, with 1.9.3 being just around the corner. YARV supports threads, fibers and processes, but includes a Global Interpreter Lock (GIL) which limits the VM to only running one thread at a time. The GIL is included to ensure non-threadsafe C-extensions does not cause problems, and effectively bars YARV from utilizing true parallelism from fibers and threads [14].

JRuby

JRuby is a Ruby implementation that runs on any Java Virtual Machine (JVM) and therefore enables using Java libraries from within Ruby. Java is a very mature and large ecosystem with many useful libraries, language constructs and concurrency models [7].

[2]Yukihiro Matsumoto, a.k.a. Matz is the creator of the Ruby programming language

JRuby does not include a GIL as YARV and therefore allows true parallelism for threads, as opposed to only having one thread/fiber run at a time. JRuby does however not support process forking as it is not supported by the JVM [12].

2.5 Summary

This chapter covered the BDD method and how it uses a test-first approach and focus the behaviour of the software. Several Ruby webservers were also covered along with theory on concurrent programming and the Ruby programming language.

In the next chapter, the requirements of the webserver to be implemented are defined.

Chapter 3

Analysis

In order to have a clear goal of what the software produced in this project should do, a requirements specification is essential. The following reasons for the seven requirements for webserver produced in this project.

The webserver is named Yarn, and the latter of the report will refer to it by this name.

As covered in Section 2.2, the main purpose of a webserver is to listen for HTTP requests and then return a response according to the request. Therefore, Yarn should be able to parse HTTP requests in order to know how to respond. The parser should parse HTTP requests according to the RFC 2616 specification.

In order to analyse and debug Yarn, it should be able to log vital messages to the console. This feature would greatly improve troubleshooting and serve as a backlog of events in the case of an error or attack.

To increase the performance and throughput of Yarn it should be able to handle multiple requests at a time in a non-blocking manner. This would keep a slow request from blocking other requests from being processed.

Yarn should be able to serve static files like HTML, cascading stylesheets, JavaScript and images in order to make webpages work as expected. If a folder is requested, then Yarn should return a HTML formatted directory listing with links to the files in the given folder.

If a resources doesn't exist, or an error occurs during processing, the webserver should return an error page with a short message on what went wrong.

Yarn should be able to serve dynamic content by executing Ruby scripts and returning the output. This enables client interaction instead of simply being served static content.

In order to use Yarn with one of the many Ruby web-frameworks, it should implement the Rack interface. This would instantly enable developers to use Yarn with their Ruby web-applications.

3.1 Requirements Specification

The list below summarises the requirements for Yarn.

- Parse HTTP/1.1 requests in accordance with RFC 2616.
- Logging of requests and responses.
- Handle concurrent requests in a non-blocking manner.
- Serve static files and directory listings.
- Return an error page if an error occurs.
- Serve dynamic content by executing Ruby scripts and returning the output.
- Implement the Rack interface.

As this project was developed using BDD, the requirements specification was translated into a set of acceptance tests to get concrete feedback on completed requirements. These acceptance tests are included on the CD in `Yarn/features`.

3.2 Project Planning

Prior to starting development, the requirements were estimated and the development split into several iterations, each driven by a requirement. Figure 3.1 shows the initial planning of the development.

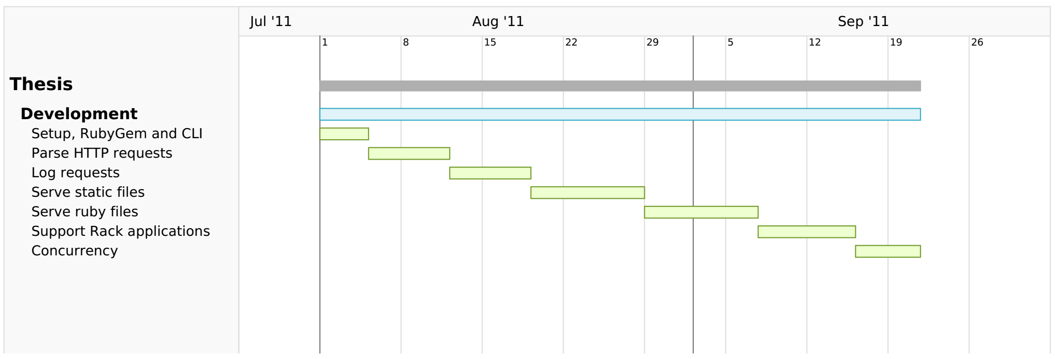


Figure 3.1: Development Gantt Planning

The development was planned to take place between August 1 and September 20. The first three days were to be spent setting up the project, RubyGems packaging and writing the command-line interface (CLI). Each requirement is then implemented in turn, with varying time set aside, depending on the complexity of the task.

As Behaviour-Driven Development was being used, no design was needed up-front, and development could start head on. Design and architectural decisions were to be made once the need arose, and tests were to be written prior to any functionality.

Chapter 4

Implementation

The development of each requirement was driven by aiming to satisfy an acceptance test which stated the intended behavior of the feature. The acceptance tests were written in a natural language using Cucumber, and once the acceptance test passed the feature was considered done. All requirements were satisfied, and this chapter covers some of the problems, and their solutions, that occured during the eight weeks of development.

Initially, an overview of the webserver, it's parts and how it works is described. Subsequently, selected interesting parts of the system is covered. The order of this chapter mirrors the order in which the features were developed.

4.1 System overview

The user interface with Yarn through a Ruby script which accepts a hostname and port number for to listen for incoming requests on, the number of processes to fork and a Rack file. All parameters are optional, and default values are used if none are given. By default the webserver will listen on `127.0.0.1:3000`, with four worker processes serving static and dynamic content. If a Rack file is given, then Yarn will serve requests by invoking the Rack application. For debugging purposes, the user can supply a parameter to include debug messages in the log. The following describes how Yarn handles requests when serving static and dynamic files, and then how it serves Rack applications.

When Yarn is started it will create a TCP server that listens for connections on either the given or default port and hostname. If a Rack file is given, then the application will be loaded. Then the process will fork into the given number of workers and Yarn is ready to handle incoming requests. Each worker contains a loop which, when registering an incoming connection, will call either the `RequestHandler` or `RackHandler` depending on whether a Rack application is loaded or not. Listing 4.1 displays the loop run in each worker process.

```
1 loop do
2   @session = @socket.accept
3   configure_socket
4   handler.run @session
5 end
```

Listing 4.1: Worker loop (Yarn/lib/yarn/server.rb:90)

For each iteration the loop will wait at line 2 for an incoming connection, and not busy-wait and run again. Once a connection is registered, the socket is configured with some TCP optimizations and then the supplied to the instantiated handler. The handler classes takes care of all the rest and closes the connection when finished. If the user sends the interrupt signal (usually by pressing Ctrl-c), Yarn will kill the worker processes, close the TCP server and exit.

4.2 Parser

The HTTP request is received as a string of text, and in order to know the how to respond, it needs to be checked for it's validity. Furthermore, it should be stored in a more convenient data format, to enable easier looking up request information.

4.2.1 HTTP Requests

The current version of the HTTP protocol, version 1.1, is well defined in a specification called RFC 2616 [5]. The specification clearly defines how a HTTP request is formed and what it consists of. Listing 4.2 is an excerpt from the specification and shows the overall structure of a HTTP request.

```
Request = Request-Line
          *(( general-header
           | request-header
           | entity-header ) CRLF)
          CRLF
          [ message-body ]
```

Listing 4.2: HTTP request structure

Listing 4.2 translates to; a `Request-Line`, followed by zero or more headers, and after a `CRLF` an optional `message-body`. Headers include information such as cookies present in the browser, browser type, size of the `message-body`. `CRLF` is a newline (line break), and is used to define the structure of the request, e.g. each header is a separate line and an empty line separate the last header and the `message-body`. The `Request-Line`, as shown in Listing 4.3, consists of a method, URI and HTTP version.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Listing 4.3: HTTP Request-Line

Listing 4.4 shows an example HTTP request.

```
GET /path/file.html HTTP/1.1
User-Agent: Internet Explorer
Content-Length: 21

field=value&show=true
```

Listing 4.4: Example HTTP request

4.2.2 Parsing Strategy

The parser was written using a Ruby library named Parslet. Parslet works by defining a set of rules and then applying them to check whether the input satisfies the rules, and then extracts defined values. The rules consists of a mix of regular expressions and boolean logic. Listing 4.5 shows the rule for the `Request-Line` as shown in Listing 4.3.

```
rule(:request_line) do
  method.as(:method) >>
  space >>
  request_uri.as(:uri) >>
  space >>
  http_version.as(:version) >>
  crlf.maybe
end
```

Listing 4.5: Request-Line parser rule. (Yarn/lib/yarn/parser.rb:61)

When a request is successfully parsed, a `Hash` containing the request data is returned. E.g. running the parser on the example HTTP request from Listing 4.4 will produce the following `Hash`:

```
{ :method=>"GET"@0,
  :uri=>{:path=>"/path/file.html"@4},
  :version=>"HTTP/1.1"@20,
  :headers=>
    { "User-Agent"=>"Internet Explorer",
      "Content-Length"=>"21" },
  :body=>"field=value&show=true"@105 }
```

Listing 4.6: Example `Parser` output

With the request formatted in a `Hash` it becomes very easy to query information about it. The numbers in Listing 4.6 preluded by a @ denote the elements' start position in the request.

4.3 Static and Dynamic Requests

Serving static and dynamic content was developed as two separate features and handler classes `StaticHandler` and `DynamicHandler`, but later merged into what is now the `RequestHandler` class. From the beginning it was clear that the handler classes would have much functionality such as reading the request, invoking the parser and returning the response in common, but also some distinct functionality. The process of handling requests can be split up into the following steps:

1. Parse the request.
2. Prepare the response.
3. Return the response to the client.
4. Close the connection.

Given this process, the part that differs between handling static and dynamic files is preparing the response. A way to vary an algorithm whilst keeping much of the functionality the same is to use the Template Method design pattern. The Template Method provides a convenient way to vary the implementation of how the response is prepared [8]. Figure 4.1 shows how the relationship between the handler classes as implemted using said pattern.

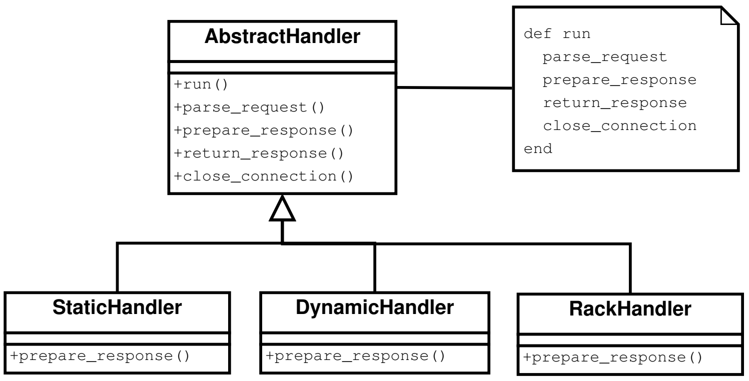


Figure 4.1: Initial handler implementation using the Template Method.

Invoking `run` on each handler class would then follow the same process, but each have their own implementation of `prepare_response` method. Once both handler classes were implemented it became clear that it would make more sense to merge the two together to mirror how Yarn can be run to serve either static and dynamic files or a Rack application. Figure 4.2 shows how the final handler classes were implemented (`RackHandler` is covered in Section 4.4).

4.3.1 RequestHandler

The `RequestHandler` works by checking whether the requested path is a file or a directory. Listing 4.7 show the logic determining what is being served.

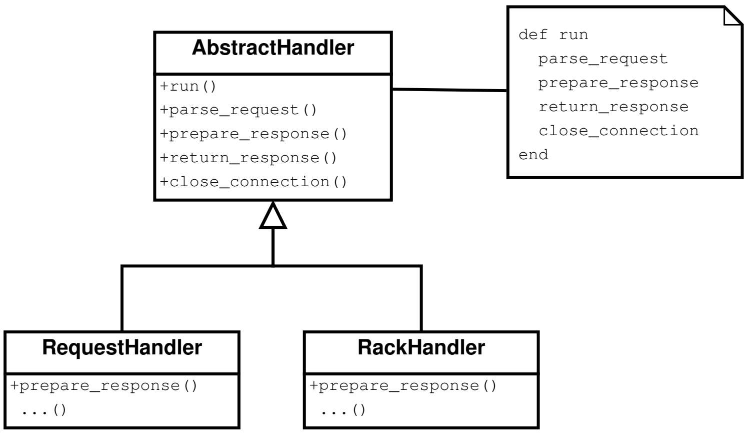


Figure 4.2: Final handler implementation using the Template Method.

```ruby
begin
  if File.directory?(path)
    serve_directory(path)
  elsif File.exists?(path)
    path =~ /.*\.rb$/ ? serve_ruby_file(path) : serve_file(path)
  else
    serve_404_page
  end
rescue ProcessingError
  log "An error occured processing #{path}"
  serve_500_page
end
```

Listing 4.7: `RequestHandler` logic. (Yarn/lib/yarn/request_handler.rb:12)

If the requested path is a directory, then a directory listing is served. In case the requested file ends with `.rb` (Ruby extension), then it is executed and the result is returned. In case the file does not exist or an error occurs executing a Ruby file, then an error page is returned. Ruby files are served by evaluating the contents of the file using a new Ruby VM and the output is then stored as the response body.

4.4 Rack Applications

Implementing support for Rack applications consisted of the following two parts:

1. Loading the Rack application from a file.
2. Communicating requests and responses between the server and the application.

Rack applications are specified in a Ruby file which needs to be evaluated to load the application. An example application which returns a response with HTTP status 200, a Content-Type header, and a body with the test ”Rack works”, is included in Listing 4.8.

```
run Proc.new { |env|
  [200, {'Content-Type' => "text/plain"}, ["Rack works"]]
}
```

Listing 4.8: Sample Rack application. (test_objects/config.ru:4)

When a Rack application file is supplied when invoking Yarn, the contents of the file is read, passed into the `Rack::Builder` constructor which is then evaluated. Listing 4.9 contains the lines from the `Server` class where the Rack application is loaded.

```
config_file = File.read(app_path)
rack_application = eval("Rack::Builder.new { #{config_file} }.to_app",
    TOPLEVEL_BINDING, app_path)
```

Listing 4.9: Loading a Rack application. (Yarn/lib/yarn/server.rb:46)

Once loaded, a Rack application is invoked with a `call` method which returns an array containing the HTTP status code, HTTP headers and message body. The `call` method takes one parameter which is a `Hash` of values the Rack application needs such as the path, hostname, port, request body etc. [2].

4.5 Concurrency

Concurrency was developed lastly as to be able to properly test it with both Rack applications and static and dynamic files. This section covers the stages of development the concurrency feature went though, from using threads to switching to processes.

4.5.1 Initial Thread-per-request Implementation

The initial implementation handled each incoming request in a new thread. This however, quickly proved to introduce too much overhead in creating a new threads. Furthermore, as the amount of threads hobbed up, memory use would rise and eventually bog down the system. It became clear that another solution was needed, and a system devised of a thread-pool and a job queue was implemented.

4.5.2 Thread-pool and Job Queue Implementation

The thread-pool was an array of threads, each running a handler which listened for requests to be added to the job queue. Each incoming request would be added to the job queue and then processed by one of the workers (handler in the thread-pool). The pros of this solution was that, with a fixed number of threads, memory use was limited. Furthermore, the job queue made sure that no requests were dropped if all threads were busy serving other requests.

Once implemented, several issues occurred which led to a rethink of the chosen concurrency model. One problem was that requests would often hang indefinitely and never return any content to the client. Exactly what caused this bug was never discovered though many hours were spent trying the track it down. It was suspected that the threads somehow created a deadlock which in turn would make the client wait until the browser timed out. Never discovering the cause of the bug illustrates one of the key problems of threaded programming; bugs are often very hard to replicate due to the complexity of the thread model and the thread scheduler.

Another problem with the thread-pool implementation was that it performed sub par to the initial implementation of creating a new thread for each request. This seemed to be caused by the way threads were scheduled, but again it proved too difficult to pinpoint the exact cause of the poor performance.

Given the difficulty of a thread-based approach, it was decided to implement the concurrency feature using processes instead.

4.5.3 Process-based Implementation

Implementing a process-based solution turned out to have many benefits, as it both reduced the complexity of Yarn, but also greatly improved upon it's performance.

The process-based solution was implemented by using `Process::fork` from the Ruby standard library to fork new processes running the server. Each forked process listens for incoming requests on the socket created in the master process (the process which forked). Listing 4.10 shows the logic for creating new processes.

```
 1 def init_workers
 2   @num_workers.times { @workers << fork_worker }
 3 end
 4
 5 def fork_worker
 6   fork { worker }
 7 end
 8
 9 def worker
10   trap("INT") { exit }
11   handler = get_handler
12   loop do
13     @session = @socket.accept
14     configure_socket
15     handler.run @session
16   end
17 end
```

Listing 4.10: Process-based implementation (Yarn/lib/yarn/server.rb:75)

Once the `fork` method is called on line 6, a new process is started with a separate Ruby VM and which continues on from that point. Meaning, each process would invoke the `worker` method running the loop. On line 10, the interrupt signal (INT) is trapped, such that when the user presses `Ctrl-c`, all processes will exit and Yarn will stop.

The process-based implementation improved the performance from ∼250 requests/second to ∼700 requests/second. The performance is covered in more detail in Section 6.2. Another improvement of the process-based concurrency model is that Yarn can now run on in parallel on YARV, wheres the thread-based implementation would have to be run on JRuby to allow for parallelism.

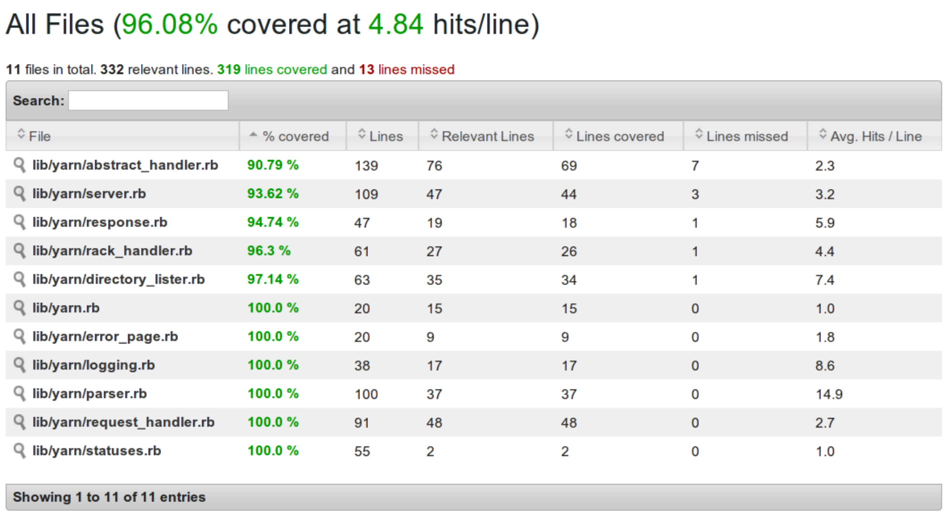
4.6 Summary

Throughout this chapter, parts of the implementation of Yarn has been covered. It described how the parser, concurrency model and request handlers were implemented and some of the troubles that arose during development. In the next chapter, testing and how it is closely related to the implementation, is covered.

Chapter 5

Testing

Throughout the development of Yarn, acceptance tests and specifications were written prior to developing a feature. This provided a solid test coverage which in turn allowed for some radical refactorings to improve the quality, readability and performance of the system. By the end of development, all 13 acceptance tests (Cucumber features) and 80 specifications (RSpec specs) passed. The final test coverage was 96.08%, as depicted on Figure 5.1.



All Files (96.08% covered at 4.84 hits/line)

11 files in total. 332 relevant lines. 319 lines covered and 13 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
lib/yarn/abstract_handler.rb	90.79 %	139	76	69	7	2.3
lib/yarn/server.rb	93.62 %	109	47	44	3	3.2
lib/yarn/response.rb	94.74 %	47	19	18	1	5.9
lib/yarn/rack_handler.rb	96.3 %	61	27	26	1	4.4
lib/yarn/directory_lister.rb	97.14 %	63	35	34	1	7.4
lib/yarn.rb	100.0 %	20	15	15	0	1.0
lib/yarn/error_page.rb	100.0 %	20	9	9	0	1.8
lib/yarn/logging.rb	100.0 %	38	17	17	0	8.6
lib/yarn/parser.rb	100.0 %	100	37	37	0	14.9
lib/yarn/request_handler.rb	100.0 %	91	48	48	0	2.7
lib/yarn/statuses.rb	100.0 %	55	2	2	0	1.0

Showing 1 to 11 of 11 entries

Figure 5.1: Yarn test coverage screenshot.

During the development, the test coverage was continously monitored to ensure all parts of the system was exercised in the test suite. The test coverage analysis is available on the CD in the `coverage` folder. The following looks at how testing for concurrency was performed, and how the HTTP parser was developed with BDD.

5.1 Testing for Concurrency

Some features were harder to test than others, and testing for concurrency was especially tricky. The solution was to invoke two requests to a Ruby file, which invoked the `sleep` method for a given amount of time before returning the response. Firing a request which slept for a second would block other requests from being served during that second. But if the server could handle concurrent requests, a fast request fired after the slow request, should be able to get it's response prior to the slow request completing. Listing 5.1 shows the acceptance test for concurrent requests.

```
Scenario: Perform two requests in parallel
  Given the server is running
  Given a client "A"
  And a client "B"
  When client "A" makes a "1" second request
  And client "B" makes a "0.1" second request
  Then client "B" receives a response before client "A"
```

Listing 5.1: Concurrency acceptance test (Yarn/features/concurrency.feature:7)

Having this acceptance test proved especially valuable during the reimplementation of the concurrency feature from using threads to using processes.

5.2 Testing the HTTP Parser

The development of the HTTP parser was initiated by writing an acceptance test as shown in Listing 5.2.

```
Scenario: Parse HTTP request
  Given a HTTP request "GET /index.html HTTP/1.1\r\nUser-Agent:
      cucumber\r\n"
  And a parser
  When I feed the request to the parser
  Then the result "method" should be "GET"
  And the result "uri" should include "path" with "/index.html"
  And the result "version" should be "HTTP/1.1"
  And the result "headers" should have "User-Agent" with "cucumber"
```

Listing 5.2: HTTP parser acceptance test (Yarn/features/parser.feature:7).

The acceptance test exercises the parser by supplying it with a HTTP request string, and then expects the result to contain the values of the request. Running the acceptance test revealed that the steps were not defined, and implementing them was the next step. Listing 5.3 shows two of the step definitions.

```ruby
Given /^a parser$/ do
  @parser = Yarn::Parser.new
end

When /^I feed the request to the parser$/ do
  @result = @parser.run(@request)
end
```

Listing 5.3: Parser step definitions excerpt (Yarn/features/step_definitions/parser_steps.rb:1).

With all the step definitions implemented, running the acceptance test again revealed that the class `Yarn::Parser` did not exist. Creating the class and running the test again revealed that the class did not have a method named `run`. The development continued in this loop of first asserting certain behavior from the code, discovering it did not behave as asserted, and then implementing the expected behaviour.

Each element which should be parsed was added incrementally by first writing a RSpec specification, and then adding the behaviour to the `Parser` class. For instance, the expected behaviour of being able to parse query parameters was written in a specification (Listing 5.4).

```ruby
it "parses query parameters in the URL" do
  result = Parser.new.run("GET /page?param1=1&param2=2 HTTP/1.1\n")
  result[:uri][:query].should == "param1=1&param2=2"
end
```

Listing 5.4: Parser query parameter support specification (Yarn/spec/yarn/parser_spec.rb:54).

To begin with, the parser is executed with a HTTP request. Then the result `Hash` is inspected to see whether the values match. Running the above specification would output an error message as the parser could not yet handle the query parameter. The behaviour, Listing 5.5 was then added to the `Parser` class.

```ruby
1 rule(:path) do
2   match['^\?'].repeat(1).as(:path) >>
3   str("?") >>
4   query.as(:query) | match['^\s'].repeat(1).as(:path)
5 end
6
7 rule(:query) do
8   match['\S+'].repeat(1)
9 end
```

Listing 5.5: URL query parameter support (Yarn/lib/yarn/parser.rb:31).

The behaviour added was lines 3,4 and lines 7 through 9. The path is checked for the occurrence of a question mark, and if it is found the sequence of characters until a whitespace character is parsed as the query string (line 8).

The implementation of all rules in the `Parser` class was developed this way, totalling in 17 specs and one acceptance test for the `Parser` class.

5.3 Summary

This chapter revealed how developing functionality through first describing the expected behaviour in a test, worked out well and always made it clear what the next step to completing a feature would be. It was also covered how using a tool to monitor test coverage, ensured that all execution paths were tested and thereby improved the value of the test suite. In the next chapter the final version of Yarn is demonstrated and benchmarked.

Chapter 6

Demonstration and Benchmark

This chapter demonstrates the final version of Yarn, and benchmarks it's performance compared to other Ruby webservers.

6.1 Demonstration

For ease of use, Yarn exposes a help command to show the options available when launching the server. Figure 6.1 shows the output of the help command.

```
[~]$ yarn --help
Yarn v0.1.1 is a multiprocess webserver written in Ruby 1.9

Usage: yarn [options]
where [options] are:
     --host, -h <s>:   Hostname or IP address of the server (default: 127.0.0.1)
     --port, -p <i>:   Port number to listen on for incomming requests (default: 3000)
  --workers, -w <i>:   Number of worker threads (default: 4)
     --rack, -r <s>:   Rackup file <config.ru> (default: off)
          --log, -l:   Enable logging
        --debug, -d:   Output debug messages
      --version, -v:   Print version and exit
         --help, -e:   Show this message
```

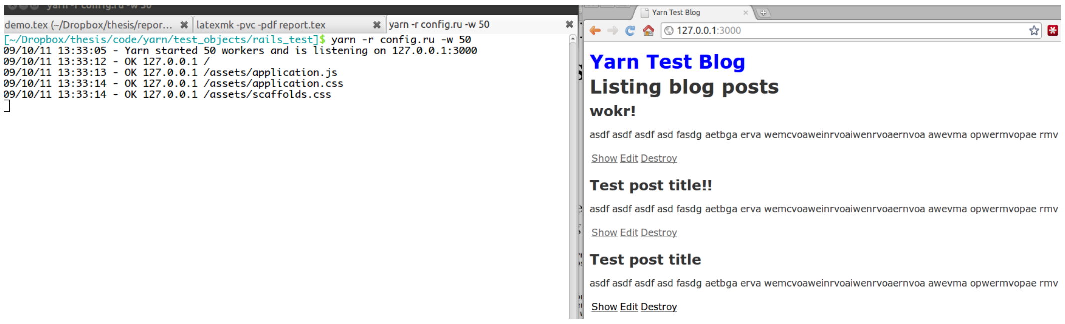
Figure 6.1: Yarn help output.

Figure 6.2 shows the output when Yarn is serving static and Ruby files with 32 worker processes and logging enabled.

```
[~]$ yarn -l -w 32
11/10/11 08:38:59 - Yarn started 32 workers and is listening on 127.0.0.1:3000
11/10/11 08:39:28 - OK 127.0.0.1 /
11/10/11 08:39:49 - OK 127.0.0.1 /test_objects
11/10/11 08:40:06 - OK 127.0.0.1 /test_objects/app.rb
11/10/11 08:40:46 - OK 127.0.0.1 /README.md
^C 11/10/11 08:40:52 - Server stopped. Have a nice day!
```

Figure 6.2: Yarn static/dynamic output.

Figure 6.3 shows Yarn serving a sample Ruby on Rails application.


demo.tex (~/Dropbox/thesis/repor...
latexmk -pvc -pdf report.tex
yarn -r config.ru -w 50
Yarn Test Blog
127.0.0.1:3000

```
[~/Dropbox/thesis/code/yarn/test_objects/rails_test]$ yarn -r config.ru -w 50
09/10/11 13:33:05 - Yarn started 50 workers and is listening on 127.0.0.1:3000
09/10/11 13:33:12 - OK 127.0.0.1 /
09/10/11 13:33:13 - OK 127.0.0.1 /assets/application.js
09/10/11 13:33:14 - OK 127.0.0.1 /assets/application.css
09/10/11 13:33:14 - OK 127.0.0.1 /assets/scaffolds.css
```

Yarn Test Blog

Listing blog posts

wokr!

asdf asdf asdf asd fasdg aetbga erva wemcvoaweinrvoaiwenrvoaernvoa awevma opwermvopae rmv

Show Edit Destroy

Test post title!!

asdf asdf asdf asd fasdg aetbga erva wemcvoaweinrvoaiwenrvoaernvoa awevma opwermvopae rmv

Show Edit Destroy

Test post title

asdf asdf asdf asd fasdg aetbga erva wemcvoaweinrvoaiwenrvoaernvoa awevma opwermvopae rmv

Show Edit Destroy

Figure 6.3: Yarn Rack application output.

Yarn can easily be installed using RubyGems which is included in Ruby as of version 1.9. To install Yarn run the following command `gem install yarn`, and now Yarn is available from the command-line as shown above.

6.2 Benchmark

The benchmarks analyses the performance and scalability (handling concurrent requests) aspects of Yarn, WEBrick, Thin and Unicorn. It does not include topics like security and reliability, which are important for webservers, but out of scope for this project.

To evaluate Yarn, it's performance will be analyzed first by itself, then it is compared to the other Ruby webservers covered in Section 2.2. The measure of performance is how many requests the webserver can handle per second. The tests were run on a 64-bit Linux machine with four cores and 8GB RAM using Ruby (YARV) 1.9.3-rc1. The benchmarks were performed using Apache Bench, which is a tool made to measure the number of requests per second a webserver can perform.

6.2.1 Yarn Performance

To get an idea of the performance of Yarn, it was measured how many requests per second (req/sec) it could handle at different counts of worker processes. Figure 6.4 plots Yarn performance.

Test runs above 256 processes bogged the computer down and were not included. The cause of this was that context-switching and memory usage exceeded the capabilities of the system. The best result was achieved with 32 worker processes which maxed at 808 req/sec, and on average Yarn performed 706 req/sec. The test consisted of making 2000 requests, 200 at a time, to Yarn serving a simple Rack application (source/test_objects/config.ru).

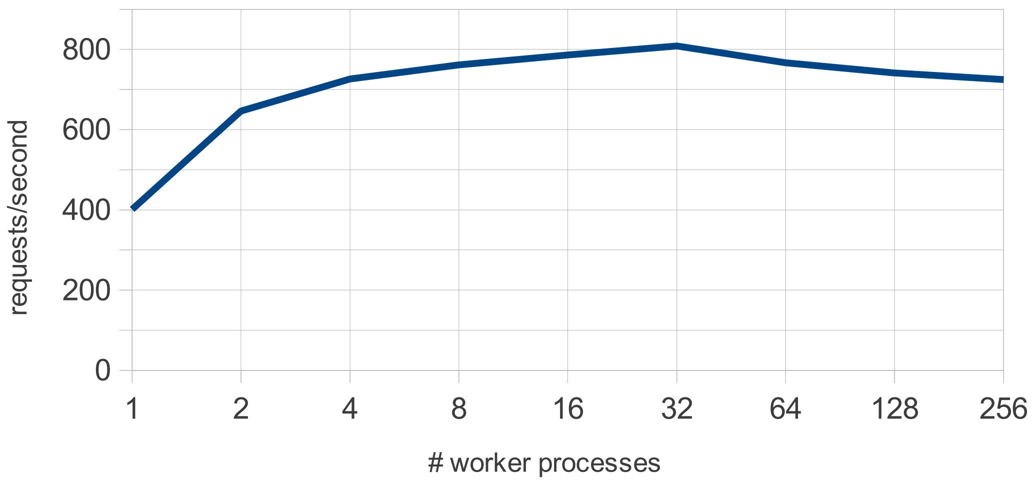


Figure 6.4: Yarn performance.

6.2.2 Yarn Vs. Other Ruby Webservers

The following benchmarks compares Yarn's performance compared to other Ruby webservers. Listing 6.5 shows the results of measuring the webservers performance serving a simple static Rack application. The test was performed doing 2000 requests, 20 at a time. The average of four test runs was used to minimize outside factors from polluting the results.

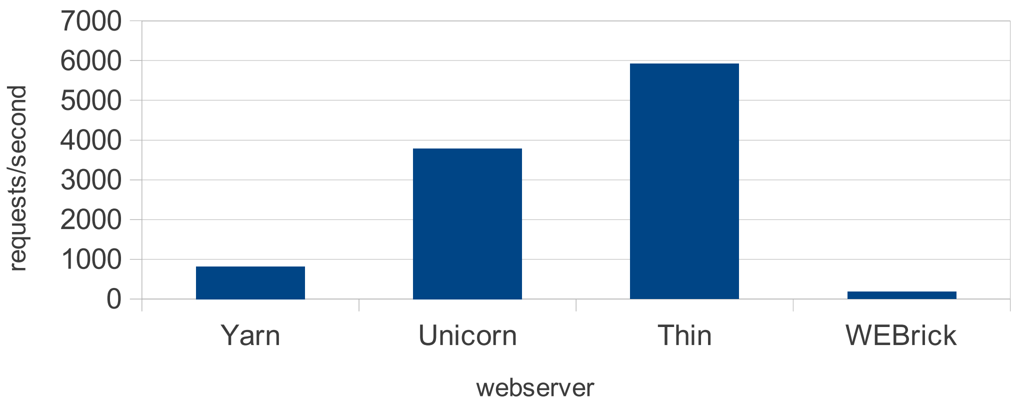


Figure 6.5: Serving a static Rack application.

Benchmarking with a static Rack application revealed that this was something that Thin managed really well with 5915 req/sec on average. Unicorn came second with 3792 req/sec, Yarn third with around 820 req/sec and WEBrick last with 180 req/sec.

Thin's excellent performance is interpreted to be the result of not having to do any context-switching as all requests are handled in one thread. Furthermore, the impact of Thin and Unicorn's fast C HTTP parser has a higher impact when the time spent parsing the request relatively high compared to the amount of time spent creating the response. The bottleneck in Yarn's implementation is currently the parser which performs a lot of string computations. Had more time been available it would have been interesting to see how it fared using the C parser from Unicorn.

The results were quite different when a CPU intensive Rack application was used. Listing 6.6 shows the results of benchmarking with a Rack application that calculates the Fibonacci sequence up to 10,000.

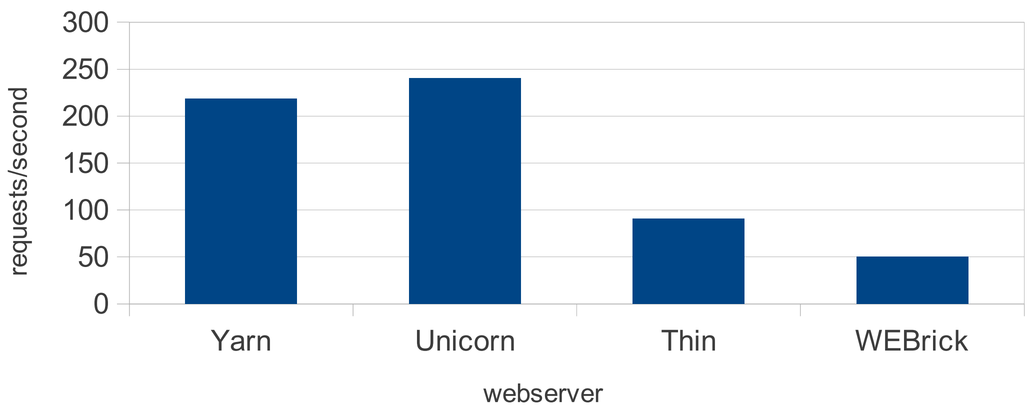


Figure 6.6: Serving a CPU intensive Rack application

With each request now having to run a CPU intensive computation, Unicorn came in first with 240 req/sec, Yarn second with 218 req/sec, then Thin with 90 req/sec and lastly WEBrick with 50 req/sec. With a CPU intensive application, the benefits of parallel processing is apparent, as is the case with Unicorn and Yarn. Thin and WEBrick would probably perform similarly if multiple instances and a load balancer was used.

6.3 Summary

In this chapter, the usage, installation and distribution of Yarn was covered. Yarn was also benchmarked in terms of how many requests per second it could serve. The results were compared to other Ruby webservers, and whilst yarn performed well for CPU intensive requests, it left room for improvement when serving static content. The performance bottleneck in Yarn is the HTTP parser. In the next chapter Yarn and the process of using BDD is evaluated.

Chapter 7

Critical Evaluation

This chapters looks critically at the final product and reflects upon the project process using BDD.

7.1 Product Evaluation

The final version of Yarn (0.1.1) includes all features from the requirement specification, and all acceptance tests and specifications passes.

Yarn becomes unstable at very high request volumes. However, when the requests to worker process ratio becomes too high, some TCP connections are dropped and no content is delivered to the client. It was not possible to fix this bug as no more time was left for development.

The performance of Yarn is quite satisfying for CPU intensive applications, but for static content it performs sub par to other Ruby webservers. The performance bottleneck is the HTTP parser, and a switch to using a C extension parser would be feasible.

7.2 Process Reflection

Developing software tests first proved to be difficult at times when it was hard to figure out how a piece of code would even work. Having done mostly non-test-first development prior to this project, it required a continuous concious effort to stick to writing tests before implementing features. Late in the development phase, it became a more natural effort to drive the implementation through testing, and the advantages of the work-flow were proved a great asset. Having the tests provided the courage which was sometimes needed prior to a large rewrite or refactoring of a feature, as it was easy to see whether the changes had broken any functionality. Furthermore, by writing the tests first, it was always clear what the next step was as running the test would reveal exactly what was missing in the implementation.

Given there was no design phase prior to starting the development, the design decisions were made continuously throughout the development. This resulted in some decisions made early on which later proved to be less fortunate. But due to the agile nature of BDD, and the confidence achieved by having a big test suite, drastic design changes late in the process were possible. Had the system been thoroughly designed upfront, such drastic changes might have been fatal to the success of the project.

Chapter 8

Conclusion

During this project a working webserver was created and all requirements were met. Yarn features logging, serving static and dynamic content, Rack applications and parallel processing of requests. Yard performed well when serving CPU intensive applications, but left room for improvements when serving static content. Using processes for concurrency gave the best results, as the threaded implementation was error-prone and did not perform well.

BDD works well, but using a test-first approach can be hard with new technology. Design flaws were detected late in the development phase, which might have been prevented with a more thorough design upfront. Having a thorough test suite did however make late design changes possible, and did also ensure that changes in general did not cause regressions.

8.1 Future Work

Given more time for development, there are a number of imminent features and a bug which should be added or fixed before Yarn will be stable enough to be used in a production environment.

The bug regarding dropped TCP connections should be fixed, possibly by queuing incoming requests to make sure they are being properly served even at high volumes. Implementing a request queue with the current process concurrency model will however require some form of inter-process communication protocol.

Yarn currently does not support persistent TCP connections, and for each request, a new TCP connection is used. This creates an overhead which could be spared by allowing clients to reuse existing connections.

The source-code for Yarn is open-source and publicly available on Github which is a web-based collaborative Git hosting site. This enables anyone to suggest or add improvements and features to the project for the good of all Yarn users. Given time to mature and possible interest from the Ruby community, Yarn might evolve to be a prominent Ruby webserver.

Bibliography

[1] D Chelimsky, D Astels, Z Dennis, A Hellesoy, and B Helmkap. The RSpec Book. page 448, 2009.

[2] Christian Neukirchen. The Rack Interface Specification. `http://rack.rubyforge.org/doc/SPEC.html`, 2010.

[3] Ruby community. WEBrick documentation. `http://ruby-doc.org/stdlib-1.9.2/libdoc/webrick/rdoc/index.html`, 2003.

[4] Marc-Andr Cournoyer. Thin - A fast and very simple web-server. `http://code.macournoyer.com/thin/doc/files/README.html`, 2010.

[5] Network Working Group. Hypertext Transfer Protocol – HTTP/1.1. `http://tools.ietf.org/html/rfc2616`, 1999.

[6] Dan North. Introducing BDD. `http://dannorth.net/introducing-bdd/`, 2006.

[7] Charles O Nutter, Nick Sieger, Thomas Enebo, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java.* 2010.

[8] R Olsen. *Design patterns in Ruby.* Addison-Wesley professional Ruby series. Addison-Wesley, 2008.

[9] Nick Quaranto. RubyGems.org. `https://rubygems.org/`, 2011.

[10] Ruby community Ruby documentation. Class: Fiber. `http://ruby-doc.org/core-1.9/classes/Fiber.html`, 2011.

[11] Douglas C Schmidt. Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events. pages 1–11, 1995.

[12] Venkat Subramaniam. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors.* Pragmatic Bookshelf, 2011.

[13] Herb Sutter. A Fundamental Turn Toward Concurrency in Software. *Dr Dobbs Journal*, 30(3):16–20, 2005.

[14] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby 1.9.* 2009.

[15] Eric Wong. Unicorn: Rack HTTP server for fast clients and Unix. `http://unicorn.bogomips.org/`, 2010.