Load Testing of Containerised Web Services

Christoffer Hamberg
Abstract

Load Testing of Containerised Web Services

Christoffer Hamberg

Load testing web services requires a great deal of environment configuration and setup. This is especially apparent in an environment where virtualisation by containerisation is used with many moving and volatile parts. However, containerisation tools like Docker offer several properties, such as; application image creation and distribution, network interconnectivity and application isolation that could be used to support the load testing process.

In this thesis, a tool named Bencher, which goal is to aid the process of load testing containerised (with Docker) HTTP services, is designed and implemented. To reach its goal Bencher automates some of the tedious steps of load testing, including connecting and scaling containers, collecting system metrics and load testing results to name a few. Bencher’s usability is verified by testing a number of hypotheses formed around different architecture characteristics of web servers in the programming language Ruby.

With a minimal environment setup cost and a rapid test iteration process, Bencher proved its usability by being successfully used to verify the hypotheses in this thesis. However, there is still need for future work and improvements, including for example functionality for measuring network bandwidth and latency, that could be added to enhance process even further. To conclude, Bencher fulfilled its goal and scope that were set for it in this thesis.
## Contents

1 Introduction ..................................................... 1
   1.1 Motivation .................................................... 2
   1.2 Delimitations .................................................. 2
   1.3 Outline ....................................................... 2

2 Web Server Architecture ........................................ 4
   2.1 Background .................................................... 4
      2.1.1 Common Gateway Interface .............................. 4
      2.1.2 FastCGI and Apache Modules ........................... 5
      2.1.3 Application Server ...................................... 5
   2.2 Hypertext Transfer Protocol ................................. 5
   2.3 Reverse Proxy ................................................. 6
      2.3.1 Life of a Request ....................................... 6
   2.4 Web Server Architectures .................................... 8
      2.4.1 I/O models ................................................. 9
      2.4.2 Multi-Process and Multi-Threaded ..................... 9
      2.4.3 Event Driven .............................................. 11

3 Web Applications ............................................... 13
   3.1 Characteristics .............................................. 13
      3.1.1 CPU- and I/O-bound Applications ........................ 13
      3.1.2 Slow Clients ............................................. 14
   3.2 Ruby Applications ........................................... 14
      3.2.1 Implementations of Ruby ............................... 14
      3.2.2 Concurrency and Parallelism in Ruby ................. 15
      3.2.3 Rack ..................................................... 16

4 Ruby Web Servers ............................................... 18
   4.1 Overview ...................................................... 18
   4.2 Selection process ............................................ 19
   4.3 Unicorn ....................................................... 20
      4.3.1 Architecture ............................................. 20
      4.3.2 Configuration ............................................ 20
   4.4 Puma .......................................................... 21
      4.4.1 Architecture ............................................. 21
      4.4.2 Configuration ............................................ 21
   4.5 Thin .......................................................... 22
      4.5.1 Architecture ............................................. 22
      4.5.2 Configuration ............................................ 22
   4.6 Comparison and Hypothesis ................................. 22
1 Introduction

With the rapid growth of connected devices, up 6.4 Billion in 2016 according to Gartner [4], the demands on the services supporting these devices are under heavy demand. One of the most common protocols used for IP network communication is the Hyper Text Transfer Protocol (HTTP).

HTTP is most likely known for being the protocol used for the World Wide Web (WWW) where web servers, also known as HTTP servers, serve resources to requesting clients. Resources can be of different types, for example, static files or dynamically generated web pages. HTTP is also commonly used as a communication protocol for Application Programming Interfaces (APIs) between applications. Needless to say, web servers have to be able to handle extreme loads of traffic.

Thankfully, web server architecture is, generally speaking, quite simple and are therefore a good target for both horizontal (adding more capacity) and vertical (more resources) scaling. However, neither of these are either cost- or energy-efficient. It is better, therefore, to utilise as much as possible of the given resources.

To measure the capacity of web server properly is a non-trivial task. There are numerous variables in the equation that may effect the results. The main goal of this thesis is to implement a tool that allows users to more easily setup and run benchmarks on HTTP services in an environment that resembles a modern way of running web services. Namely, containerisation.

In this thesis, a number of experiments are done in order to verify the tool created. The experiments are based on testing a set of hypotheses (see Section 4.6). These hypotheses are derived from an analysis made in this thesis related to the characteristics of architecture types of Ruby, a dynamic object-oriented interpreted language that in the last decade gained a lot of popularity for web application development, web servers and applications.
1.1 Motivation

Setting up, executing and retrieving results of an HTTP benchmark is a tedious task, especially in a containerised environment. This thesis’ main goal is to implement a tool to support this process.

To provide a full understanding of the space, web server architectures, concurrency models and application characteristics are explained and analysed. The review is first made in a language agnostic way, followed by how it applies to the Ruby ecosystem. Based on this knowledge, hypotheses are made regarding Ruby’s performance characteristics. Thus, the second goal is to analyse the differences between Ruby web servers and then test the differences with the tool created. Based on this, the problem statements of this thesis are:

1. How could a tool for containerised HTTP service testing be designed with the focus on simplicity?
2. What are empirically measurable differences of Ruby web servers, and can the differences be observed with the implemented tool?

1.2 Delimitations

There are numerous of different ways of running containerised services. Nevertheless, this thesis only looks at containers running with Docker (see Section 6.3), as it can be, at the moment, considered as the most widespread technology used to run container services.

The standard Ruby interpreter implementation, Ruby MRI, is the only Ruby implementation included in the experiment. Other implementations’ effects on concurrency model as well as other characteristics are discussed in Section 3.2.1. The sole purpose of the experiment is to test the hypotheses – the suitability of the different Ruby web servers is not addressed.

1.3 Outline

Section 2 starts by giving an introduction to web servers and the components related to them. To give a full understanding of how HTTP resources are served, a request’s lifecycle is examined. This acts as a base for the rest of the section where concurrency and architecture models for implementing concurrent web servers as well as their differences are discussed.
Section 3 introduces web applications and their characteristics. After the generic introduction of web applications, this section focuses on the Ruby language. Key concepts of the Ruby language in terms of its concurrency model and fit as a web application language are discussed.

Section 4 presents different Ruby web servers putting an impact on their characteristics and differences. Based on the knowledge from Section 2, a selection of web servers of different architecture types is made. These servers are characterised and compared in order to form the hypothesis that is used to verify the experiment in Section 7.

Section 5 provides methods and guidelines for setting up and running a benchmark of web services. In addition, the section also includes recommendations of for example metrics to collect and load testing tools to use.

Section 6 describes the design and implementation of the benchmarking tool, set as the primary goal for this thesis. Design decisions, based on Section 5, are motivated and, additionally, details regarding tools of use, for example, Docker, are explained.

Section 7 presents the setup of the experiment and the execution and results which are based on the hypotheses made in Section 4 and tested with the tool implemented in Section 6.

Section 8 concludes the thesis and discusses the findings and conclusions that were drawn. At last, ideas for future work is proposed.
2 Web Server Architecture

This section gives a broad overview of web servers. It first presents the historical background, development as well as the current state of web servers. Following this there are detailed explanations of some of the key components of web server architectures including; the HTTP protocol and internals of an HTTP request, proxy servers, the different architecture types that can be found in web servers and different I/O models.

2.1 Background

A web server’s role is to serve resources to requesting clients. These resources are most commonly served over HTTP (HyperText Transfer Protocol). HTTP is a protocol developed for the purpose of serving static and dynamically generated files over the web. In the early days of the Internet era, web pages consisted of static files that were served directly from the web server’s file system. In order to make dynamic and interactive web pages, something a static page at that time could not handle, a way of dynamically generating the pages was needed. Server-side scripting was the solution to that.

2.1.1 Common Gateway Interface

Server-side scripting was initially achieved with Common Gateway Interface (CGI). CGI works in the way that instead of serving the raw resource from filesystem, that resource, an executable program, is instead executed by the operating system and the program output is then served to the client. By reading parameters sent with the request the response can be fully customised.

CGI is based on a process model – meaning that each incoming CGI request spawns a new process of the program. This is both memory consuming and makes for instance sharing state between requests difficult. However, the simplicity of the CGI protocol, using OS built-in \texttt{STDIN} and \texttt{STDOUT} as communication channels and a process model, made it a good fit at time. Eventually the limitations caught up with the simplicity and the nature of CGI did not scale with the increased demands and complexity as the web gained popularity.
2.1.2 FastCGI and Apache Modules

FastCGI was introduced to solve this issue by not spawning a new process for every request. Instead the processes were kept persisted by the FastCGI server and could be reused between requests. This allowed FastCGI to handle more requests than traditional CGI. An alternative to FastCGI was introduced around the same time. It is based on including the program interpreter in the web server. Apache HTTP server is one of the most famous web servers which has this functionality. This integration between web server and program runtime results in a performance increase which is a positive outcome. On the other hand, a negative side-effect is the tight coupling of web server and program execution, compared to FastCGI that loosens the coupling.

2.1.3 Application Server

However, neither of these solutions reduced the boilerplate of developing web applications in terms of how to handle reusable resources, for example database connections, state management and output formatting. Tools such as Netscape’s LiveWire emerged to do the heavy lifting. This was the start for the web’s application servers. As more and more functionality was incorporated these tools turned into core services in the technology stack.

2.2 Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) is an application-level protocol developed by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) for the purpose of serving the web. It follows the client-server architecture and is stateless by its initial specification [2]. The communication between a client and a server is in a request/response manner where a client requests resources by Uniform Resource Identifiers (URIs) that a server responds to. Only a client can initiate the communication.

HTTP requests and responses each consist of three different parts. The request consists of a request line which specifies the resource, method and HTTP version followed by a set of headers describing the request. The third part is the message body, which is optional and may contain any binary data. The response, on the other hand, consists of a response line with a status code for the response followed by response headers. The response, as well, includes an optional message body.
Different HTTP methods have different semantics when used with a resource. Most commonly used is the GET method for requesting a resource. GET is considered as both idempotent (subsequent GET requests’ responses are identical) and safe (the state of the server does not change). Compared to the POST method, used for creating new resources, that is neither idempotent nor safe.

2.3 Reverse Proxy

A reverse proxy works as an intermediate between a client and a web server. Instead of clients connecting directly to the web server they connect to the reverse proxy that makes the request on their behalf. This enables a number of practical features such as load-balancing between multiple web servers, re-writing of URLs, caching of resources, serving static assets directly from the proxy and connection handling. For web servers of the process fork model – where a process is forked for every incoming request – a reverse proxy is a necessity. Without it the web server simply cannot handle more connections than the number of processes available.

Security is one other reason for using a reverse proxy. With public internet facing web servers, to which clients might send arbitrary data to, they are exposed to many possible attacks. Not handling I/O and connections properly can open up for various vulnerabilities or denial of service attacks only few possible problems to mention.

2.3.1 Life of a Request

In this section, a step-by-step overview of an HTTP request from a web server’s perspective is presented. The overview is a simplified version and some details are omitted. A GET request for the URI http://example.com/assets/cat.jpg is used as an example.

The request first needs to get routed to the web server available at where the hostname example.com points at.

---

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Hostname</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>example.com</td>
<td>/assets/cat.jpg</td>
</tr>
</tbody>
</table>

Figure 1: URI Breakdown.
Once the request has reached the target web server, it is put in a socket waiting to be accepted and read by the web server. How the server is notified and how it handles the connections differ depending on the architecture type (covered in Section 2.4). In order to be able to interpret and act accordingly to the request the server first needs to parse the request according to the HTTP specification.

```
GET /assets/cat.jpg HTTP/1.1
User-Agent: curl/7.37.1
Host: example.com
Accept: */*
```

Figure 2: GET Request Header.

Parsed from the request line is the requested resource and method. In this case; `cat.jpg` and `GET`. Headers and body are also extracted. Now with the request in a format that the web server can interpret it is able to decide how to process the request based on how the web server is configured. As an example the server can be configured in a way that resource paths starting with, for example, `/assets` are served directly from the web server to the client. Similar rules could apply for resources under `/cgi-bin` as they are executed as CGI scripts. In the case the resource points to an application server the request is usually simply passed to the upstream application server. Figure 2.3.1 shows a `GET` request of a static resource.

Once the requested resource has been located, in this example a static file on the web server, the web server then proceeds to assemble the response. The status code in the response line is set to 200 in this case since the resource existed. In addition to the response line a number of headers are also included, each with various degree of importance. `Content-Length` tells the size of the body, allowing the client to determine when the whole body has been transferred. `Content-Type` header tells which media type is used in the message’s content. Also known as MIME (Multipurpose Internet Mail Extensions) type, it tells the receiver how to properly handle the content. Examples of MIME types are `text/html`, `application/json`, `image/jpeg` (used in this example). Last part of the response is the body, in this case the requested image file.
Another header of interest is the Connection header. In Figure 2.3.1 the connection line specifies that it wants to use the HTTP 1.1 protocol. The response line from the server, Figure 2.3.1, acknowledges the use of HTTP 1.1 in its headers. By HTTP 1.1 specification, Connection is per default set to Keep-Alive. This means that the client should keep the connection open and re-use it for consecutive requests until either one explicitly closes it or times-out. Whereas in HTTP 1.0 the connection is closed after each request/response, which impacts scalability negatively. The Connection header was initially introduced as a non-standardised header already in HTTP 1.0 and became part of the specification in 1.1. In detail its advantages include: lowering the CPU and memory utilisation, limiting network congestion and reducing latency between consecutive requests.

2.4 Web Server Architectures

A web server’s work may seem rather trivial as how it is described in Section 2.3.1. In essence; wait for a request to arrive, process it, respond with the result and then wait for the next incoming request. A naive and simple implementation is to have a sequential loop – waiting for new connection requests to arrive and process them one by one. However, this does not scale with multiple incoming requests. The challenge for web servers is to do this concurrently for thousands of requests per second and as efficient as possible in terms of resource utilisation. In this section, three different architecture types, commonly used by web servers to handle concurrency are described.
2.4.1 I/O models

To fully understand the different architectures types, it is important to understand how the operating system handles I/O. I/O, seen from a computer architecture point-of-view, is often referred as the process of transferring data to or from the CPU or memory. For example, reading from a network socket or disk. There are multiple ways for an operating system to handle I/O, which also reflects on the way of the overall architecture. Let us first define two common concepts of I/O.

**blocking and non-blocking I/O** Describes how the operating system accesses I/O devices. In blocking mode, the operation does not return to the caller until the operation is complete. Non-blocking mode returns immediately to indicate the status of the operation. Multiple calls, polling, might be needed to determine if the operation has completed.

**synchronous and asynchronous I/O** Describes the control flow of I/O operations. Synchronous calls remain in control and wait (blocking) until completion until returning. Asynchronous returns immediately and the execution continues.

Blocking and synchronous, respectively non-blocking and asynchronous, are often used interchangeable in literature.

2.4.2 Multi-Process and Multi-Threaded

Multi-process/threaded web servers work in the way of associating each incoming request with a dedicated process/thread. This maps well to the synchronous blocking I/O model and encourages a straight-forward sequential programming model.

**Multi-Process** A Multi-Process model achieves its concurrency with, as its name implies, processes. Each request is handled by its own process. The first web server, CERN httpd [6], was based on this process model.

The process model both has its advantages and disadvantages. With each request running in a separate process context, there is no shared memory between different requests. This provides a good isolation and does not force the application to be thread-safe. Shared memory does however simplify, for example, caching across requests. Since processes are by nature costly OS
In terms of memory consumption and startup time, a technique called pre-forking is usually used to solve the latter. Pre-forking means that processes are pre-spawned in the web server on startup, instead of a more naive way of spawning them on-demand for each new incoming request. Each spawned process then has access to a shared connection socket descriptor from which they pick up and handle new incoming requests. The large memory footprint that processes induce is a limitation in terms of scaling the number of concurrent connections per server. This causes a trade-off in terms of concurrency per memory. Concurrency is therefore especially limited for long polling requests and slow clients that block processes.

**Multi-Threaded** The multi-threaded model, similar to the process model, uses a thread-per-request technique. With threads being more lightweight than processes it can achieve a much better concurrency per memory ratio than the process based one. Since threads have a shared memory space, it is possible to share state between requests, which is useful for caching to name an example. The drawback is that concurrent programming is more difficult to reason about. The developer has to write thread-safe application code to avoid deadlocks and race conditions that might cause data corruption. More on this is found in Section 3.2.2.

A common technique is to have a pool of threads, called a thread-pool, that is spawned on startup and which then waits to receive requests to process, much similar to the pre-forked process model. Usually, a dedicated thread is
used for connection handling. This dedicated thread serves the connection pool with requests that worker threads process. Thread-pools are good for achieving predictable performance. However, if the number of threads is large, performance might suffer. This is due to CPU time wasted on thread context-switching and also on increased memory consumption.

Both multi-process/threaded models are commonly implemented in a synchronous blocking way of handling I/O. The sequential programming flow model maps well to the I/O that mode and provides a good abstraction for the programmer, with requests flowing sequentially.

2.4.3 Event Driven

The event-driven architecture takes another approach when handling I/O compared to the process/thread variants. Instead of synchronous blocking I/O model it uses the asynchronous non-blocking model. As a result of this, the thread/process per request model is not applicable, and a common solution is to let a single thread handle multiple requests instead. New events (requests) are pushed to an event queue that is consumed by a so-called event-loop, which continuously processes the events in the queue (Figure 6). In order to process an event, a registered event handler or associated callback code for the event needs to be in place beforehand. This resolves the program flow to be a cascade of callbacks being executed in a non-sequential manner making the control flow a bit inverted and arguably more difficult to follow when the execution flow is not sequential.

Historically, event-driven architectures have been constrained by the limitation of OS level asynchronous/non-blocking I/O operation support and performant event notification triggers.
Performance and scalability wise, event-driven architectures differentiate from the process/thread-models. Only using one thread reduces the memory consumption with the need of only one thread stack per process. The thread context-switching overhead is therefore also reduced. In the case of the event-loop becoming overloaded, in other words, the loop is unable to process the growing queue of events, event-based systems usually still provide a throughput but with an increasing delay. Unlike process/thread-based models which commonly stops accepting new requests once overloaded because all resources are locked up.
3 Web Applications

Web applications are commonly defined as client/server applications and are often separated into three different tiers. Namely; presentation-, application- and data-tier. The tier of interest in this thesis is mainly the application tier where both web- and application servers are found. In this section an overview of different general characteristics of web applications is presented. Following this, a closer look at the Ruby programming language and its concurrency primitives and HTTP APIs are made.

3.1 Characteristics

Seen from a three-tier architecture view, the application tier and its characteristics are where the interest and scope of this thesis lie.

3.1.1 CPU- and I/O-bound Applications

In the application tier there are two interesting characteristics, CPU- and I/O-bound, that are now presented.

CPU-bound A CPU-bound application is characterised by that most of its execution time is spent on CPU cycles doing, for example, calculations. To improve the performance of a CPU-bound application, more CPU power can be added and the overall execution speed should improve.

I/O-bound An I/O-bound application spends most of its time on I/O operations. This can, for example, be reading from disk/network sockets or performing database calls. In a similar manner to CPU-bound performance a faster I/O device usually improves the overall execution speed.

Web applications are by nature heavy I/O-bound. Database calls, communicating with other network APIs and serving static assets, all fall under the I/O-bound category and work commonly seen in web applications.

Other common application characteristics are, to name a couple; cache- and memory-bound. These do not, however, correlate as strongly with the characteristics of web servers in the case that is of interest in this thesis and are therefore not further discussed.
3.1.2 Slow Clients

Slow clients are, as its name implies, web clients which send/receive requests/responses slowly. Simply put, any non-local network client can be considered as a "slow client". A web server without proper connection handling could have all its resources locked up while waiting for I/O to complete and not be able to handle new requests due to slow clients. This influences the number of concurrent connections it can handle simultaneously. Slow clients are especially common among mobile clients, where connection delays are often significant. Common ways of mitigating this problem are to have a load balancer or reverse proxy in front of the web- and application server to handle request queueing.

3.2 Ruby Applications

Ruby, a dynamic object-oriented and interpreted language created by the Japanese computer scientist Yukihiro Matsumoto in the early 90s. "I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That’s why I decided to design my own language." [3]. Ruby is strongly influenced by Perl, Smalltalk and Python and with the philosophy in mind to be understood by humans first and computers second. It was for many years quite unknown outside of the Japanese community and received its first English mailing list in the late 90s.

The big pivot for Ruby came in 2005 when a web application framework named Ruby on Rails was released and quickly received vast popularity.

3.2.1 Implementations of Ruby

Ruby has a number of implementations of it, where the most popular ones are; Ruby MRI, JRuby, and Rubinius. Ruby MRI is what is called the "reference implementation" of the Ruby language. Originally developed by Matsumoto, it is the most popular implementation of them all.

What differs between these different implementations are a number of properties. To take JRuby as an example. JRuby runs in the JVM (Java Virtual Machine) and thus allows for example; JIT (Just In Time) compilation, OS native threading and embedding Ruby code in Java applications and vice versa. One of the main downsides of JRuby is the increased boot-up time that the JVM brings. In this thesis, however, only Ruby MRI is used in the experiment.
3.2.2 Concurrency and Parallelism in Ruby

Ruby MRI has an often misinterpreted concurrency model. In Ruby MRI a locking mechanism called the Global VM Lock (GVM), also known as the Global Interpreter Lock (GIL), exists. Its function is to make sure that only one thread can execute at any given time in a multi-threaded environment. GIL exists due to the fact that many core parts of Ruby, for example, the Array implementation and native C extensions are not thread-safe. What GIL does is to enforce Ruby MRI programs to be thread-safe by nature. GIL both has its benefits and drawbacks. One benefit was already mentioned, ensuring thread-safety "out of the box". GIL also reduces the burden of concurrent programming which Sutter et. al describes as "... concurrency requires programmers to think in a way humans find difficult." [8]. JRuby however, does not have a GIL and thus allows access to native OS threads, in other words, putting the responsibility on the developer to write thread-safe code.

One obvious drawback of GIL is that it can never achieve true parallelism. With parallelism defined as running multiple tasks in parallel in for example multi-core environment. In comparison with concurrency where the execution of tasks is often multiplexed. To fully utilise all cores of a processor with Ruby MRI, multiple Ruby processes need to run simultaneously. An advantage of that is that scaling out with processes instead of threads also encourages the design of horizontally scalable architectures, which also fits well in a containerised environment that later will be discussed.

Important to note is that MRI’s GIL does not block for I/O-calls. It automatically switches to another thread when a blocking I/O call is performed. However, if a library uses C extensions (C code with a Ruby wrapper), it is required to explicitly tell the Ruby VM from the C code when it is allowed to do a context-switch. If the C implementation fails to do so, the I/O call prevents a context switch from occurring. This was earlier an issue in for example old versions of the official MySQL database driver causing decreased performance. I/O heavy applications, like web applications, are therefore not as severely exposed to the limitations of GIL as many people expect.
3.2.3 Rack

Rack is a middleware that sits between a Ruby web server and a Ruby web application. Prior to the release of Rack in 2007, in order for an application and web server to interoperate with each other, both had to implement each others APIs. In other words, each web server had to have application framework specific code and vice versa. Rack provides a middleware API that lets both web server and application framework to communicate with each other as long as they both implement the Rack API. This made it possible to remove the tight coupling between application and server and allows the web server to be seamlessly switched. A side-effect of this was also that new web servers emerged when only one API needed to be maintained. For application frameworks this also meant that they no longer needed to bundle server vendor specific code in the application framework.

A minimal Rack application (Figure 8) is an object with a call method that takes the incoming request as a hash and returns an array of three elements consisting of a status code, a hash of headers and an optional response body.

```ruby
class Application
  def call(env)
    [200, {"Content-Type" => "text/plain"}, ["Hi!"]]
  end
end
```

Figure 8: A simple Rack application.
Rack applications are built on layers of middleware that are composed and chained in a pipelined manner. Requests are passed through the chain where each middleware performs its objective and then passes it on to the next middleware in the chain. Each middleware is typically designed for one sole purpose, for example, logging of requests, connection rate limiting, error handling or other functionalities that could be performed in the layer that Rack operates in. A middleware worth mentioning, in the context of this thesis, is `Rack::Lock`. What it does, is to wrap each request in a mutex lock, ensuring that only one request can be executed at any given time. This can be used if the application code is not thread-safe.
4 Ruby Web Servers

In this section Ruby web servers are presented and analysed based on knowledge from previous sections. First a definition and brief history of Ruby web servers are given, followed by its current ecosystem. After this, a set of different web servers is chosen for further analysis and characterisation. Previous sections act as the foundation for the characterisation. In addition to this, new properties of web servers are also be looked at. Once all relevant properties are identified, a comparison between the servers is made. Finally, a hypothesis on how these selected servers perform is made based on this comparison and previously gained knowledge. This hypothesis is later tested in Section 7.

4.1 Overview

Running a Ruby web application is in its essence a simple and straightforward process. The standard way of setting up a web application is to include the web server in the application’s dependencies, meaning that everything needed to serve the application is included in the project. This may seem a bit inverted compared to for example the Java ecosystem, where the web application is usually loaded into a separate application server container. Historically speaking this approach has not always been the case for Ruby. Initially, CGI and FastCGI were used to serve Ruby web applications, followed by running Apache HTTP server with a Ruby module, mod-ruby, embedding a Ruby interpreter into the web server. At that time, Ruby was not as widespread for web applications as it is nowadays.

Shortly after the release of Ruby on Rails, an alternative way of serving web applications gained popularity called Mongrel. Mongrel is a single-threaded web server written in Ruby that presents a standard HTTP interface. This allows it to easily integrate with existing HTTP load balancers and reverse proxies. Running multiple instances of Mongrel behind an Apache HTTP load balancer became a popular way of running Ruby web applications.

Mongrel did however not solve the operational burden of setting up, deploying and running Ruby web applications. There is a lot of configuration to be done and not always a straightforward task to get it running without systems administration knowledge. This is where Passenger, initially called mod_rails, came to shine. With a simple setup as an Apache module it provided a web server, load balancer and Ruby application server combined into one package. It soon became the recommended way of deploying Ruby on Rails.
applications after its release. It was later extended with Rack support –
allowing any Rack supported application to run in Passenger.

With the release of Rack several new Ruby web servers emerged. Some
more adopted than others. In this thesis, the most popular ones of each
architecture model are selected to be used in the hypotheses.

4.2 Selection process

As mentioned in the previous section, Ruby web servers are commonly bun-
dled in as a package dependency to the web application. The standard
package manager in Ruby is called RubyGems where a "gem" is a package/li-
brary, hence the name. RubyGems has a public repository\(^1\) that provides
download statistics each for gem. With the help of The Ruby Toolbox, a
site that gathers gem statistics from different sources, a list\(^2\) of the most
popular web servers was used for the initial screening. From this list each
of the most popular process-, thread- and event-based of Rack supported
servers were selected. The total download count was verified with RubyGems.
To support the selection of these three web servers, a search on GitHub,
the worlds largest code host \([5]\) was conducted. Gem dependencies for an
application are commonly specified in a "Gemfile". GitHub offers a search
tool that allows searching through all publicly available code repositories.
Knowing the format of how a RubyGem dependency is defined makes it
possible to craft a search query to find all its occurrences. The query used
was "gem 'web_server_name'" and only to include Ruby files.

The result of the search is shown in Table 1 and is likely not surprising for
someone familiar with the Ruby ecosystem that Puma, Unicorn and Thin
ended up on the list. A few interesting observations can be made from the
results. Thin being the oldest and with the highest total download count. It
has however, in proportion, a low occurrence count in Gemfiles compared to
Unicorn. This could be related to a number of things; GitHub was launched
in 2008, and its popularity came at the same time as Unicorn was released.
Unicorn was also for a long time the recommended web server by many
big influencers in the Ruby community like; Ruby on Rails and the hosting
platform Heroku. At the timing of writing this, Puma is the recommended
server by them both.

\(^1\)http://www.rubygems.org
\(^2\)https://www.ruby-toolbox.com/categories/web_servers
### 1. Table 1: Statistics collected from Rubygems.org and GitHub 2015-07-14.

<table>
<thead>
<tr>
<th>Model</th>
<th>Release year</th>
<th># downloads (RubyGems.org)</th>
<th># in Gemfile (GitHub)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thin</td>
<td>Event</td>
<td>12,764,468</td>
<td>57,642</td>
</tr>
<tr>
<td>Unicorn</td>
<td>Process</td>
<td>12,245,596</td>
<td>399,369</td>
</tr>
<tr>
<td>Puma</td>
<td>Thread</td>
<td>3,545,214</td>
<td>15,868</td>
</tr>
</tbody>
</table>

### 4.3 Unicorn

Unicorn’s philosophy is stated as "Being a server that only runs on Unix-like platforms, unicorn is strongly tied to the Unix philosophy of doing one thing and (hopefully) doing it well." [1]. Its focus is to serve fast and low latency clients.

#### 4.3.1 Architecture

Unicorn is of the pre-forked process model with blocking I/O. As mentioned in Section 2.4.2, the process model gives the advantage of not requiring the application code to be thread-safe. Threads are not used at all in Unicorn internally. Designed for fast clients, Unicorn needs a reverse proxy in front to handle request queueing since it simply can not accept new requests if all its workers are busy.

On startup, a master Unicorn process is launched. This process forks itself and its children become process workers that handle requests. During runtime, the master process “reaps” and forks new workers in case they become unresponsive or exceed the maximum runtime timeout threshold. Load balancing requests between worker processes are done on an OS kernel level with the workers reading requests from a shared connection socket. This means that the need of a load balancer between the reverse proxy and the web server can be omitted if only one Unicorn server is being used.

#### 4.3.2 Configuration

To maximise Unicorn’s concurrency, one has to determine how many processes that can fit in the memory available. 

\[
\frac{\text{Memory available}}{\text{Application memory}} = \text{Max # processes.}
\]

After that, the number of available CPU cores is a limiting factor. The threshold, where the penalty of context switching exceeds the gain can be found by simply testing the different number of processes under load.
4.4 Puma

Puma started as a rewrite of Mongrel back in 2011. It had two sets of goals initially. Only to support Rack – Mongrel was created in a pre-Rack era and thus contained a lot of coupling to different frameworks. Second goal was to focus on Ruby implementations supporting true parallelism, such as JRuby and Rubinius.

4.4.1 Architecture

In the core of Puma, a thread-based connection handling architecture is found. Where each request is handled by a separate thread. In addition to this, Puma also supports a "clustered" mode where multiple processes are used – like in Unicorn. With the addition that in Puma, each process can also run multiple threads. Due to its threaded model, Puma requires the application code to be thread-safe.

Although the recommended Ruby implementations to run Puma with are the ones supporting true parallelism it does not prevent Ruby MRI from taking advantage of the multi-threading environment. MRI does allow non-blocking I/O if the underlying library extension supports it, as discussed in Section 3.2.2. With web applications usually being heavily I/O-bound – it is, on paper, a great improvement also for MRI applications.

Unlike Unicorn, Puma has a built-in "protection" against slow requests. Using something that can be seen as a Reactor pattern [7]. Threads detect slow client requests with a timeout threshold and automatically move that request to a thread dedicated to handle slow requests. Once request has been completely transferred, the request is then put back to the thread-pool for a new thread to pick it up and complete the request. Since this only happens for incoming requests, the response part, when the server sends data to the client, still remains unshielded.

4.4.2 Configuration

Tuning the correct concurrency settings in Puma is similar to Unicorn’s, except that it adds one parameter to the equation – thread count. Once the process count is set, optimising thread count can be done by increasing the thread count under application load. At the point when adding more threads does not give any performance improvements the limit has been found.
4.5 Thin

Thin, released in 2007, is a minimal web server that only provides a Rack interface. It is the oldest of the web servers selected in this comparison and also as of today, the one with the least active development. Unlike the other web servers in this comparison, it is of the event-driven I/O model. However, it has the similarity of the others that it can run in a multi-processes/clustered mode where each process consists of single-threaded event-loop.

4.5.1 Architecture

The event model allows Thin to handle long-polling and keep-alive connection without the need of a reverse proxy between itself and the client. In the core of Thin a network I/O library, EventMachine\(^3\), is used to provide the event-based architecture.

In order for Thin to fully glance, it is required for the application layer to implement evented I/O as well. For an application framework like Ruby on Rails, which has a large number of dependencies, it requires each library to provide evented asynchronous I/O support. Something that is not quite feasible. Without this support, Thin’s architecture should not yield any direct performance gains.

4.5.2 Configuration

Thin provides a rich configuration API for setting, for example, the maximum number of persistent connections, file descriptor limits and other low-level parameters. Its clustered mode is similar to Unicorn’s model of running multiple single-threaded processes. A key difference is that Unicorn shares a connection socket between all processes while Thin exposes a separate socket for each process. These processes are therefore referred to as separate Thin "servers". Hence, a reverse proxy is required to load balance between the sockets when running Thin in cluster mode.

4.6 Comparison and Hypothesis

In order to make a somewhat fair and realistic comparison between all the web servers, it is assumed for the hypothesis that a reverse proxy is used in

\(^3\)http://rubyeventmachine.com
front of all of them. This also holds true for experiment when testing the hypotheses.

Thin and Unicorn both share the property of being single-threaded and can run with multiple processes. If configured to run with the same number of processes, they are likely to be roughly the same performant with Unicorn being the one slightly more performant due to the overhead that Thin’s EventMachine architecture adds and Unicorn’s OS native load balancing of multiple processes. However, Thin should theoretically scale only with one process. Something that Unicorn does not do.

For I/O-bound workloads implemented sequentially, Puma is most likely the most performant among them all thanks to its threading architecture. This assumes that the underlying I/O library correctly handles thread signalling as discussed in Section 3.2.2. However, in the case of I/O-bound work implemented in an evented style, Thin should outperform the others with less resources utilised.

Due to MRI’s GIL, the process count is likely the configuration parameter that is of most importance. In other words, Puma likely needs to match its process count with Unicorn’s before any benefits of threading are seen except for I/O-bound workloads where the thread count in Puma matters more.

A number of hypotheses are concluded based on these facts.

- Puma is overall the most performant one for sequential thread-safe I/O-bound applications.
- Puma and Unicorn are equally performant for CPU-bound applications.
- Thin’s performance will closely match Unicorn’s – with a slight advantage for Unicorn.
- Thin is the most performant one for event-based applications.
5 Benchmarking Methodology

This section presents common benchmarking practises and recommendations ranging from for example, what metrics to collect, creating a test environment and different tools to use when performing load testing on HTTP services.

5.1 Overview

When performing a benchmark of a web server, there are numerous of aspects that should be taken into account for the benchmark to provide somewhat reliable results. In this section, a number of important factors, gathered from various sources are presented. The design and usage of the benchmarking tool implemented in this thesis have been developed with these factors in mind.

The benchmark type this thesis focuses on is of macro type, meaning that the overall performance of the web server seen from a client’s perspective is tested. The opposite of a macro benchmark is a micro benchmark, where detailed performance metrics of functional level, for example, request parsing are measured.

5.2 Performance metrics

There are a number of different performance metrics that can be used when measuring web server performance. Knowing which ones that are relevant and to compare may depend on the type of application being served. Different metrics could also give an indication of where potential performance bottlenecks are located in the system. For example, a low throughput (kilobits per second) could be due to limitations in network bandwidth. It is, therefore, important to determine the limiting factors that may affect the overall result.

One of the most commonly used metric when doing web server benchmarking is unarguably the number of requests per second (req/s) the server can handle. Requests per second together with latency (milliseconds) and throughput usually provide a good estimate of the general performance. As previously mention, the type and characteristics of an application can make the metrics irrelevant for the end client depending on the use case. For example, a limited number of requests with a high throughput can be acceptable for a web server serving large (in terms of file size) resources to a few number of clients. Table 2 shows an overview of these metrics of interest.
Other important metrics that are only visible from a web server’s perspective are, for example, resource utilisation in terms of computing power and memory consumption. Combining this data with the previously mentioned metrics gives an indication of the resource efficiency and scalability. For example the number of requests per megabyte of memory ratio that can be achieved with the server.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests per seconds</td>
<td>req/s</td>
</tr>
<tr>
<td>Simultaneous connections</td>
<td>conn/s</td>
</tr>
<tr>
<td>Latency</td>
<td>ms</td>
</tr>
<tr>
<td>Data throughput</td>
<td>KB/s</td>
</tr>
<tr>
<td>Memory utilisation</td>
<td>%</td>
</tr>
<tr>
<td>CPU utilisation</td>
<td>%</td>
</tr>
</tbody>
</table>

Table 2: Overview of performance metrics of interest.

5.3 Environment

A common mistake that often is observed in web server benchmarks made outside the academia is that the test environment is not properly setup for the results to be truly reliable. One of the most common mistakes is to run the web server on the same machine as the load generator which has a number of implications. Firstly, the load generator could "steal" compute resources from the web server. Secondly, tests running only over localhost omits the network factor. This could be of benefit for some tests, however, while doing macro benchmarks it should not be omitted. It is important to make sure that the network capacity is not a limiting factor in the test environment.

Another common mistake is to overload by under-dimensioning the load generator in comparison to the web server. A faster machine, or multiple, should be used for the load generator. Related to this is the question whether it is acceptable to use virtual machines or not. Using virtual machines when conducting benchmarks is twofold. An advantage is that it provides a controlled and repeatable system environment. On the other hand, it adds the extra layer of abstraction that consumes resources in an unpredictable way. This unpredictability may not be acceptable for all types of benchmarks. Nevertheless, the benchmarking tool implemented in this thesis is designed for virtualised HTTP servers.
5.4 Testing

An interesting and important behaviour to observe is the overload behaviour of a web server. Most servers loose some capacity when overloaded. This means that the load applied should be increased up to that point where it gets overloaded in order to find the maximum capacity and its behaviour while being overloaded. If graphed, it is represented as a peaking curve that then backs off. The back off shows how the server handles the overload. Therefore, it is important to test the whole range of load and not simply apply the maximum load from the beginning.

Test runs should be running long enough for buffers and caches in the different layers of stacks, for example, OS, application or network, to stabilise. Between each test run, there should also be enough time for connections to clear out.

A common mistake seen outside academia in HTTP server testing is the selection of what kind of requested resource that is used. Often a single "trivial" one is used, for example returning a static body of "Hello world". This does not realistically simulate a real use case scenario. For example, in the experiment to verify the hypotheses in this thesis a number of different resources were crafted to simulate the different application characteristics. This includes waiting for I/O, varying the response body size or doing heavy computational work to name a few.

5.5 Load generators

There is a large number of HTTP load generators available. Spanning in the whole spectrum of simple command line tools to full-blown commercial test runners with graphical user interfaces.

A recommendation is always to test with more than one tool to verify the results. This is to detect any abnormalities in the interaction between the client (HTTP load generator) and server (HTTP server). In addition, it is important to know that the HTTP loader is performant enough to overload the server. In the experiment in this thesis, it is obvious that a load generator is not going to be the limiting factor – the Ruby web servers will.

In academia httperf, developed by HP Labs\(^4\), was found to be one of the most popular tools used. Apache Benchmark\(^5\) is also a popular but less feature-rich tool available. The main requirements for the tool needed for

\(^4\)http://www.labs.hpe.com/research/linux/httperf/
\(^5\)https://httpd.apache.org/docs/2.2/programs/ab.html
this thesis are that it should be open source and have a well-defined control API.

The choice fell on autobench\(^6\) which is a Perl script that internally wraps httperf. Autobench has a feature of defining tests to automatically run over specified ranges of load and collect test results. No other tool was found to offer this feature. Other features includes running in a cluster mode allowing multiple remote loaders to be controlled by a single controller daemon. Being built around httperf also adds to its validity.

5.6 Load test characterisation

For a faster iteration speed when attempting to detect certain thresholds or hotspots in the application characteristics, it is advantageous if parameters that may effect that behaviour can be set at runtime. Preferably directly by the load generator, via for example resource path, query parameters or in headers.

To replicate the different application characteristics discussed in Section 3.1, resources could be created with the only purpose of for example blocking for I/O for \(N\) seconds, where \(N\) is set from the load generator’s request at runtime by a query parameter.

This method is used for the experiments in Section 7, where a number of resource paths in combination with query parameters are available to simulate different application characteristics. Table 3 shows an overview of them.

\(^{6}\)http://www.xenoclast.org/autobench/
6 Implementation - Bencher

This section describes in detail the requirements, design, architecture and implementation of the tool, from now on referred to as *Bencher*, that will be used to verify the hypothesis stated in Section 4.6.

6.1 Goal and Requirements

The goal is to create an HTTP benchmarking command line tool for containerised web services that combines the orchestration of both the load generator and web service to be benchmarked in a real-world like environment. This means less configuration and setting up the actual testing environment, which often is a tedious task.

Bencher is designed with the focus on simplicity and repeatability in mind. Simplicity, meaning that the tool should be easy to setup and use. Repeatability, meaning that the same test configuration can be easily replicated and performed by another user in another environment.

Presenting relevant and comprehensible test results is an important requirement that Bencher aims to improve compared to other benchmarking tools available. For example, Section 5.2 brings up metrics that are important to collect when performing a benchmark. These metrics consist of both the load testers results and system metrics from all running components. Since Bencher has control of all components – these metrics become automatically accessible for Bencher to collect. Bencher can therefore collect and present metrics, such as CPU-utilisation on both the HTTP service and load generator, in a uniform way.

6.2 Tools

What differentiates Bencher from many other benchmarking tools is its extensive use of containerised applications. Each component runs inside a container, from the CLI tool itself, to each individual component.

Containerisation, also know as Operating-system-level virtualisation, have gained a lot of momentum in the last years. The rise of containerisation is mainly due to the tools around it have evolved making it more accessible to the masses. For example, Platform-as-a-Service vendors (PaaS), such as Heroku, have been using Linux Containers (LXC) for many years already. Bencher utilises an application named Docker for containerisation.
6.3 Containerisation with Docker

As mentioned previously, containerisation is a form of virtualisation that, compared to classical virtual machines, is a form of operation-system level virtualisation. Meaning that multiple isolated user-space instances (containers) run in only one kernel. Sharing kernel space allows containers to run without each having their own operating system. Instead, they rely on the host’s kernel to provide core functionality.

Containerisation gained tremendous popularity in the recent years mainly due to an application named Docker that was released at the beginning of 2013. Docker provides an abstraction layer for operating-system-level virtualisation to allow deployment of applications inside containers.

Docker containers are launched from Docker images. A Docker image can be described as a template containing all binaries required for the container’s intended use, including the operating system. For example Ubuntu with the web server nginx installed can be described in an image. A key feature is that images are organised and built in layers. This means that when a Docker image is changed – only the affected layers are updated (or added). Due to this, the whole image does not have to be rebuilt and distributed, which commonly is the case for virtual machine images. Images can therefore effectively be based on other images, so-called base images, to share the operating system, applications and so forth. Figure 6.3 shows the base Docker image used in experiment in Section 7 by the different web servers.

6.4 Architecture

Benchconsists of a number of components that can be divided into four logical groups. This section covers the architecture and implementation details of each component. Figure 10 shows the different components an below follows a short description of each one.

**Bench** CLI to initiate and collect test runs.

**Loader** HTTP load generator.

**Proxy** Reverse proxy.

**Target** HTTP service to test.
FROM ruby:2.2.0 # Set base image

RUN apt-get update -qq # Update and install packages
RUN apt-get install -y build-essential libpq-dev

RUN mkdir /usr/app
WORKDIR /usr/app # Set Docker work directory

COPY tmp/Gemfile /usr/app/
COPY tmp/Gemfile.lock /usr/app/
RUN bundle install # Install Ruby libraries

COPY . /usr/app # Copy web application to image

ENV PORT 8080 # Set environment variable
EXPOSE 8080 # Expose container network port
ENTRYPOINT ["foreman", "start"] # Set entry point

Figure 9: Docker image used as base image for Target components in the experiment.

Figure 10: Architecture overview of Bencher and its components.
6.4.1 Bencher

In the core of Bencher itself is the CLI tool that initiates and ties together all other components. Each component runs in separate Docker containers, orchestrated by Bencher itself. Bencher is written in Ruby and has no other external dependencies except Docker.

A test run in Bencher is defined with a YAML-formatted configuration file. This configuration file contains two important parts. Firstly, connection details to the Docker hosts that are used for the test-run. Secondly, test parameters that define the characteristics of the load test to run. Figure 6.4.1 shows an example configuration. These test configuration options match a subset of Autobench’s configurations. Each configuration parameter can be overridden by passing in a command line argument too. This can be used to vary easily, for example, the HTTP endpoint to test, and have the configuration file acting as a base template.

Bencher uses the Docker API on the host machines to orchestrate and control all components. The Docker API allows full remote control of the Docker application running on the host. Bencher uses it for starting and stopping containers and retrieving system events and logs. Since the Loader component should be running on a separate machine as the Proxy/Target components (see Section 5.3, Bencher supports the option to control two Docker hosts. By default, the Docker daemon listens to a non-networked Unix socket. For Bencher to communicate with a Docker daemon on a remote host, Docker needs to be started in HTTP TLS mode. The authentication method is passwordless and requires certificates to be exchanged and setup between Bencher and the Docker hosts before a test can be run.

A test’s completion is determined from the state of the Loader component. While running, Bencher samples metrics from all components via the Docker API. These metrics include memory usage, CPU utilisation and other system metrics of interest as discussed in Section 5. Once the Loader’s container has terminated with a successful exit code, Bencher retrieves the Loader’s system log where its test result can be found.

6.4.2 Loader

The HTTP benchmarking component, referred to as Loader, is a Docker image built with autobench (see Section 5.5) installed and pre-configured. The image has the entrypoint, which specifies the executable to run when the containers starts, set to autobench. This means that the container can be run as if it were the autobench binary itself.
hosts:
  host1:
    hostname: loader.bencher.com
docker:
    url: tcp://loader.bencher.com:2376
  host2:
    hostname: target.bencher.com
docker:
    url: tcp://target.bencher.com:2376
loader:
  host: host1
test:
    hostname: target.bencher.com
    port: 80
    endpoint: /io?sleep=0.05
    low_rate: 100
    high_rate: 200
    rate_step: 10
    num_call: 10
    timeout: 5
    const_test_time: 300
target:
  host: host2
  image: bencher/target
  cmd: puma
  env:
    app_config: app.ru
    port: 8080
    web_concurrency: 1
    min_threads: 1
    max_threads: 16

Figure 11: Example YAML configuration file for a test run in Bencher.
Loader therefore only accepts standard autobench run parameters and returns the test result to standard output in a comma separated value (CSV) format after the test run is complete. The test parameters are passed from the test configuration via Bencher when launching the Loader container.

The completion of a test run is determined by the running state of the Loader container. Once terminated, Bencher can read the system log from the container and parse the test results into a Bencher’s internal test result representation format.

6.4.3 Proxy

To make benchmarks realistic and fair, a reverse proxy is needed in Bencher. Exposing web servers directly to end clients is a bad practice and in the case of containers, where containers should be treated as volatile components, the proxy is used to route the requests to the correct containers.

As the reverse proxy server nginx\(^7\) was chosen due to its widespread use and popularity in the industry. Alongside with the need of connection handling, the possibility to load balance between multiple upstream servers is also required.

The reverse proxy runs, like all other components, in its own separate container. A problem that needs to be solved is how to configure nginx to point to the correct Target containers. There are a number of ways to achieve this. For example, statically set the upstream servers IP addresses or hostnames in the nginx configuration and statically set the Target container IP address. This solution is neither pretty or good, static configuration with a pre-configured state in a container environment is an anti-pattern and too much responsibility is shared between the components. Ideally, the proxy configuration should dynamically change as new Target containers launches.

Docker-gen\(^8\) is a tool that renders file templates from Docker container metadata. This can be used to dynamically generate configuration files based on the running Docker environment. Using the Docker host API, it can get information and metadata about, for example, hostnames, IP addresses and forwarded ports from the running containers. This combined with a templating language gives a powerful way of dynamically generating configuration files with data from containers. In our case, docker-gen is used to generate nginx configuration files with the correct IPs to the upstream Target containers.

\(^7\)http://nginx.org/
\(^8\)https://github.com/jwilder/docker-gen
To automate this process of having a nginx automatically update its configuration when a new Target container is launched requires a couple of steps: a way of watching for specific Docker events (i.e. a new Target container is launched), render a new nginx configuration and finally reloading nginx. All this can easily be achieved thanks to the built-in functionality of docker-gen. docker-gen has a parameter to watch continually for container changes. If a change is detected a new file is rendered, and a notification command is triggered on success. The following command is used in Bencher to achieve this:

```
docker-gen -watch -notify "nginx -s reload" nginx.tmpl nginx.conf
```

Worth noting is that this requires that both nginx and docker-gen is running inside the same container.

In conclusion, the Proxy component is completely stand-alone and autonomous, i.e. once started it is self-managed. Proxy is, like all other components, terminated between test runs in Bencher in order to follow the guidelines defined in Section 5.

### 6.4.4 Target

Target is the HTTP service Bencher is to benchmark. The service itself has to be available as a Docker image and expose an HTTP service on a predefined port in order for Bencher to be able to perform a benchmark. This allows Bencher to benchmark almost any HTTP service that can run inside a Docker container.

Before a Target container is ready to service requests from the Loader, it first needs to register in the Proxy component. This is achieved by setting a specific environment variable in the Target container at launch. This environment variable sets the hostname in Proxy, under which the Target is set as an upstream server for. Proxy detects this with docker-gen and updates its configuration accordingly.

That same functionality also enables Bencher to launch multiple Target containers, running simultaneously in a test run. Proxy load balance requests between the Targets. The Target containers and Loader are unaware of the number of Target containers running – only Proxy and Bencher itself are aware of this. Hence, there are no dependencies between Targets.

The benefits of Docker and containers, in general, are obvious in the case of the Target component. As long as a Docker image can be created with the service – Bencher can test it.
6.5 Presenting the data

Bencher presents metrics from two main sources, Docker Stats API from each server and autobench’s test result from the Loader.

Each component container is probed from Bencher on a regular interval for system metrics via the Docker API (default is set to every other second). Examples of metrics are CPU and memory usage. With this data, it is possible to show that, for example, the Loader has a 100% CPU resource utilisation during the test run, which is something autobench (httpperf) recommends for a correct test.

All of the probed metrics are available in the final test result and are saved on disk in a JSON format, allowing it to be plotted as time series. In addition to this, autobench’s test result is read and parsed from the Loader’s container system log once the test run is completed.

Bencher includes functionality to generate simple graphs using the Google Charts API\(^9\) and other statistics of the test run, for example, average memory/CPU consumption for the different components.

After a completed test run, all data, including test configuration and test results, is saved to disk. This allows Bencher to easily re-run or re-graph previous test runs.

---
\(^9\)https://developers.google.com/chart/
7  Experiment and Results

This section describes how Bencher was used to test the hypothesis stated in Section 4.6, on how the web servers; Puma, Unicorn and Thin, all of different architecture types, perform depending on the application characteristics. First, the setup of the test application and environment is described, followed by the results.

7.1  Web application

In order to test the hypothesis, a test suite in the form of a web application had to be developed. It consists of a simple HTTP service, which exposes a number of resources, which each has the responsibility to simulate a specific application characteristic. These application characteristics were discussed in Section 3.1.

A test application was created using the Rack API. The choice of Rack was to simply avoid any extra dependencies that could interfere with the performance characteristics. An alternative would have been to use, for example, Sinatra, which is a web framework that provides a simple Domain Specific Language for mapping HTTP resources to blocks of code and handles parsing of query params and headers.

7.1.1  Application characteristics

To simulate different application characteristics, a number of different resources are created, each attached to its own path and some allowing query parameters to be passed to tune its characteristics. For example, GET /io?time=0.05 simulates a blocking I/O for 5 seconds. Internally the implementation is a Kernel.sleep method call. The actual implementation in Rack can be seen in Figure 7.1. All the different simulation paths are listed in Figure 7.1.1 and source code is available in Appendix A.

<table>
<thead>
<tr>
<th>Resource path</th>
<th>Description</th>
<th>Query parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/io</td>
<td>I/O-bound work with sleep</td>
<td>Sleep time</td>
</tr>
<tr>
<td>/cpu</td>
<td>CPU-bound work with calculations</td>
<td>CPU intensity</td>
</tr>
<tr>
<td>/evented</td>
<td>Perform evented I/O call</td>
<td>Sleep/Schedule time</td>
</tr>
<tr>
<td>/nop</td>
<td>Empty response</td>
<td>Not available</td>
</tr>
</tbody>
</table>

Table 3: Overview of test resources.
class Target
  def call(env)
    case env['PATH_INFO']
    when '/nop'
      nop
    when '/io'
      sleeping(env)
    when '/cpu'
      cpu(env)
    when '/evented'
      evented(env)
    else
      [404,
       {"Content-Type" => "text/html"},
       ["Invalid path"]
      ]
    end
  end
end

def sleeping(env)
  params = Rack::Utils.parse_nested_query(env['QUERY_STRING'])
  time = (params['time'] || 0).to_f
  sleep(time)
  [200,
   {},
   [
   ]
  ]
end

...
end

Figure 12: Rack application used for the experiment.
7.2 Target setup

For each web server in the experiment, a Docker image is created. Thanks to the Rack API and Docker’s layered images, it is possible to decouple the web application and web server in a convenient way with the use of Docker images.

First, a Docker image, target-master, was created which is based on Ubuntu 14.02 with Ruby 2.2.0 installed. In the target-master, the web application and its dependencies (Rack 1.6.4) are also added. This image could now be used serving the web application with WEBrick, a web server that comes built-in to Ruby.

Now, due to the target-master image, it is possible to create the web server specific images based on it. Each web server image simply needs to install the web server and corresponding configuration file and have it pointed to the Rack configuration in the web application in order for it to run.

7.3 Environment

The experiment was performed on cloud infrastructure hosted by Rackspace\textsuperscript{10}. As discussed in Section 5.3, it is important that the Loader’s capacity is dimensioned to be able to fully saturate the Target’s resources. Therefore, in order to fulfill this requirement, the two instances were equipped with different resources.

CPU and network capacity are the most crucial resource factors for the Loader instance and was therefore dimensioned with more capacity than the Target instance. Loader instance runs a 2.8 GHz, 10 core Intel E5-2680 v2 CPU with 32 GB RAM and 10 Gb/s network adapter. While the Target instance runs on a virtual instance with 6 virtual CPUs available, 30 GB RAM and 2.5 Gb/s network adapter. A network bandwidth and latency test were done with a tool called qperf. Between Loader and Target bandwidth was measured to $\sim$600 MB/s with a $\sim$160 $\mu$s latency, which gives enough room to avoid network saturation.

As the operating system, CoreOS, a lightweight Linux-based operating system, was used in both instances. The reasoning behind using CoreOS is based on its wide adoption in container hosting space and Docker comes built-in. CoreOS version 899.6.0 and Docker 1.9.1 was used for all tests. On both hosts, the maximum number of open file descriptors were set to infinite.

\textsuperscript{10}http://www.rackspace.com/
7.4 Results

In this section, Bencher is used to test the hypotheses stated in Section 4.6. All tests have a number of common test parameters that is used for all tests. These include: request timeout is set to 5 seconds, this means that any test with a request taking longer than 5 seconds will be classified as a failed test. Also, each test is run for 300 seconds to level out any spikes. Before the 300 seconds tests, a shorter test run of 30 seconds is done, to faster detect where in the request rate span the real tests should be targeted. The loader components CPU utilisation was confirmed to be close to 100% during all tests. This is per recommendation from Autobench in order to provide reliable test results. The CPU metrics were gathered from Bencher’s result output.

Not all possible test run combinations are tested in this section. Only the ones that are relevant to show the hypotheses are performed. For example, Thin is only tested with 1 process. The versions of the web servers tested are as follows; Unicorn 5.0.1, Puma 2.16.0 and Thin 1.6.4. In addition, the maximum number of processes used per server test is limited to 4 in order not to use up all 6 available cores in the Target host instance.

7.5 I/O-bound application

The first hypothesis to test was "Puma is overall the most performant one for sequential thread-safe I/O-bound applications."

I/O-bound was tested with /io?time=0.05 (Kernel sleep for 50 ms). A sleep of 50 ms was picked based on not being too small nor large, in the context of Ruby web applications. Any test with an average response time outside the 95th percentile of the sleep time of 50 ms is considered as a failed test. This applies to all future tests too.

Puma was first used to find a baseline for this test. Configured to run 1 processes and with 1 thread, Puma managed to serve 15.0 req/s while maintaining an acceptable average response time. Above 15 req/s, the average response time increased drastically until the hard 5-second limit was reached. After this, both Unicorn and Thin were configured with the same concurrency level. Both matched Puma’s test results (Table 7.5).

All servers were then configured to run with 4 processes. Theoretically, they should all be able to serve approximately 4 times the number of requests. Running Bencher with an updated configuration of 4 processes showed that
<table>
<thead>
<tr>
<th>Server</th>
<th>Concurrency</th>
<th>Rate</th>
<th>Latency</th>
<th>Throughput</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicorn</td>
<td>1 proc</td>
<td>15.1 req/s</td>
<td>51.4 ms</td>
<td>3.6 KB/s</td>
<td>4.1%</td>
<td>14.2 MB</td>
</tr>
<tr>
<td>Unicorn</td>
<td>4 proc</td>
<td>68.2 req/s</td>
<td>52.2 ms</td>
<td>16.5 KB/s</td>
<td>5.5%</td>
<td>61.3 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>1 proc, 1 thread</td>
<td>15.0 req/s</td>
<td>50.8 ms</td>
<td>3.5 KB/s</td>
<td>5.3%</td>
<td>15.1 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>4 proc, 1 thread</td>
<td>23.0 req/s</td>
<td>51.8 ms</td>
<td>5.4 KB/s</td>
<td>9.5%</td>
<td>62.1 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>1 proc, 16 threads</td>
<td>305.5 req/s</td>
<td>51.1 ms</td>
<td>71.3 KB/s</td>
<td>12.1%</td>
<td>89.6 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>4 proc, 16 threads</td>
<td>698.8 req/s</td>
<td>52.5 ms</td>
<td>163.8 KB/s</td>
<td>23.3%</td>
<td>122.1 MB</td>
</tr>
<tr>
<td>Thin</td>
<td>1 proc</td>
<td>15.0 req/s</td>
<td>51.0 ms</td>
<td>3.6 KB/s</td>
<td>4.9%</td>
<td>15.7 MB</td>
</tr>
</tbody>
</table>

Table 4: Highest request rate results among the servers with corresponding average metrics for /io?time=0.05.

Figure 13: Target CPU usage of I/O-bound application for Unicorn’s and Puma’s highest request rate.

Unicorn followed this theory almost linearly at 68.2 req/s while remaining an acceptable response time. Puma, however, only managed to serve 23.0 req/s with this same setup. This is highly likely due to Puma’s shared connection request dispatcher being the bottleneck.

So far, Puma has not been tested in the way it is intended to (running multiple threads per process). The same test setup was repeated, although this time with Puma configured to 4 processes, each with 16 threads available. Under these circumstances, Puma performed exceptionally well, managing to serve 698.8 req/s while remaining the average response time 52.5 ms. Looking at the memory graph of consumed memory Puma, it’s clear to see the lightweight of threads and Puma clearly being the most performant for I/O-bound applications among the tested servers.
vigure .?; Target memory usage of I/O-bound application for Unicorn’s and Puma’s highest request rate.

7.6 CPU-bound application

In this test, the objective is to verify the hypothesis that "Puma and Unicorn are equally performant for CPU-bound applications". In other words, Puma’s threading model should not yield any significant performance gains compared to Unicorn for CPU-intense applications.

The first step is to decide on the CPU intensity that is to be used throughout this test. Once decided, a baseline like in the previous test needs to be detected.

To easily compare with the previous test it was decided that the CPU intensity should be equivalent to the I/O-bound application’s sleep time. In other words, a workload size that yields an average response time of 50 ms for a single request had to be found. Since the workload size is easily set with a query parameter, Bench was configured to use Unicorn with 1 process with 1 concurrent request, and with binary-search, \( \text{/cpu}\text{size}=4200 \) was found to match the criteria. In addition, with this configuration, Unicorn is able to serve 18.0 req/s while maintaining an average response time of 50.1 ms.

Increasing the process count to 4, Unicorn follows the same, almost, linear pattern as it manages to serve 71.9 req/s before becoming saturated. Puma was then tested with the same configurations, first 1 process followed by 4 processes, each with 1 thread per process and the results came back almost identical as Unicorn at 20 req/s respectively 75 req/s with an average response time of 50 ms. Following this test, Bench was set to run Puma with 4 processes, each with 16 threads. As the results show, Puma did not gain any performance with this setup, and the hypothesis holds true. Request
<table>
<thead>
<tr>
<th>Server</th>
<th>Concurrency</th>
<th>Rate</th>
<th>Latency</th>
<th>Throughput</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unicorn</td>
<td>1 proc</td>
<td>18.0 req/s</td>
<td>50.1 ms</td>
<td>4.3 KB/s</td>
<td>91.1%</td>
<td>25.0 MB</td>
</tr>
<tr>
<td>Unicorn</td>
<td>4 proc</td>
<td>71.9 req/s</td>
<td>51.9 ms</td>
<td>5.0 KB/s</td>
<td>349.6%</td>
<td>94.9 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>1 proc, 1 thread</td>
<td>21.2 req/s</td>
<td>52.5 ms</td>
<td>5.0 KB/s</td>
<td>80%</td>
<td>30.1 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>4 proc, 1 thread</td>
<td>70.1 req/s</td>
<td>50.4 ms</td>
<td>16.5 KB/s</td>
<td>310.1%</td>
<td>110.0 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>1 proc, 16 threads</td>
<td>19.0 req/s</td>
<td>49.8 ms</td>
<td>4.4 KB/s</td>
<td>89.9%</td>
<td>45.4 MB</td>
</tr>
<tr>
<td>Puma</td>
<td>4 proc, 16 threads</td>
<td>67.8 req/s</td>
<td>52.5 ms</td>
<td>15.7 KB/s</td>
<td>326.2%</td>
<td>320.0 MB</td>
</tr>
<tr>
<td>Thin</td>
<td>1 proc</td>
<td>19.0 req/s</td>
<td>51.1 ms</td>
<td>4.6 KB/s</td>
<td>90.0%</td>
<td>30.5 MB</td>
</tr>
</tbody>
</table>

Table 5: Highest request rate results among the servers with corresponding average metrics for /cpu?size=4200.

![Graph](image)

Figure 15: Target CPU usage of CPU-bound application for Unicorn’s and Puma’s highest request rate.

The rate stayed at 67.8 req/s, a slight drop – most likely related to the increased context-switching of threading.

In addition, it is clear from the results that Thin’s performance characteristics, in this and the previous I/O-bound test, matches Unicorns’s. Which concludes the hypothesis that ”Thin’s performance will closely match Unicorns – with a slight advantage for Unicorn.”

### 7.7 Evented application

To test an evented application, two different test cases are used. Both uses the event I/O library EventMachine – the same library Thin is built with (see Section 4.5.1). The first test simulates a long polling request, meaning in this case: when a request comes in, it is scheduled to run $N$ ms later in a separate Reactor thread. The second test works similarly, however, the
Figure 16: Target memory usage of CPU-bound application for Unicorn’s and Puma’s highest request rate.

<table>
<thead>
<tr>
<th>Server</th>
<th>Concurrency</th>
<th>Test</th>
<th>Rate</th>
<th>Latency</th>
<th>Throughput</th>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thin</td>
<td>1 proc</td>
<td>Deferred</td>
<td>333 req/s</td>
<td>50.1 ms</td>
<td>73.8 KB/s</td>
<td>12.8%</td>
<td>36.3 MB</td>
</tr>
<tr>
<td>Thin</td>
<td>1 proc</td>
<td>Scheduled</td>
<td>1200 req/s</td>
<td>50.8 ms</td>
<td>264.4 KB/s</td>
<td>31.3%</td>
<td>35.7 MB</td>
</tr>
</tbody>
</table>

Table 6: Highest request rate results with corresponding average metrics for an evented application with Thin.

request is deferred to the Reactor thread immediately where an I/O operation is performed. This I/O operation is the same as in previous I/O tests (50 ms kernel sleep). Only Thin support these operations and is, therefore, the only server that can be tested in this test.

Like in all the previous tests, 50 ms was decided to be used as both the scheduled delay time and sleep time for the two tests. With Thin configured to run with 1 process, it managed in the first test case to maintain an average response time of 50 ms all the way up to 1200 req/s – which clearly shows its capabilities of connection handling. The second case, with a kernel sleep of 50 ms on the reactor-thread yielded a request rate of 333 req/s. Which correlates to Puma’s best result of /io?sleep=0.05 when running with 1 process. A possible reason behind this is that EventMachine internally uses a thread pool – like Puma does.
Figure 17: Target CPU usage of evented application for Thin’s highest request rate.

Figure 18: Target memory usage of evented application for Thin’s highest request rate.
8 Conclusions

In this thesis, Bencher, a tool to easily and realistically benchmark any HTTP service running in a Docker container was implemented. To test and verify Bencher’s functionality and correctness, a number of hypotheses were formed in Section 4.6 related to set of web servers characteristics in the domains of the Ruby programming language.

Bencher was successfully used to test and verify all the hypotheses and managed to show that it fulfilled its own design goals, namely: simplicity and repeatability. With a single point of configuration to change, a wide number of configuration options for a test showed invaluable when conducting the experiments.

Except verifying the hypotheses, it is possible to observe a number of interesting characteristics in the test results. Puma, for example, uses more than 3 times the memory compared to Unicorn in the CPU-bound test while serving the same amount of requests. Its memory is also slowly increasing throughout test runs (Figure 14 and 16) – possibly a sign of a memory leak.

In addition to the implementation of Bencher, this thesis provides a thorough explanation of web server architectures on a both generic level and in terms of the Ruby programming language.

8.1 Future work

Bencher satisfied the needs for the work done for this thesis. There are however a number of improvements that could be made to Bencher in order to make it more effective and useful.

Measurement of bandwidth and latency was done manually in the experiment. This is, however, something that easily could be added as a built-in component to Bencher.

In order to simulate a more realistic environment with multiple request sources Bencher needs to support running multiple Loader components simultaneously. A possible partial solution to this could be to use Autobench’s built-in cluster mode.

Collecting and presenting test results across multiple test runs are not in the focus of Bencher, that is up to the user her/himself. However, extending the current result output with better graphs and calculated results would undoubtedly be a useful feature.
References


A Web application - Rack Resources

A.1 CPU workload

def call(env)
    params = Rack::Utils.parse_nested_query(
        env["QUERY_STRING"]
    )
    size = (params['size'] || 1000).to_i
    pi = calc_pi(size)
    [200, [
        "Content-Type" => "text/plain",
        "Hello"
    ]]
end

private
    # Found at: http://goo.gl/fpjeni
    def calc_pi(digits = 10000)
        fudge = 10
        unity = 10**(digits+fudge)
        pi = 4*(4*arccot(5, unity) - arccot(239, unity))
        pi / (10**fudge)
    end

def arccot(x, unity)
    xpow = unity / x
    n = 1
    sign = 1
    sum = 0

    loop do
        term = xpow / n
        break if term == 0
        sum += sign * (xpow/n)
        xpow /= x*x
        n += 2
        sign = -sign
    end
    sum
end
A.2 I/O workload

def call(env)
    params = Rack::Utils.parse_nested_query(env['QUERY_STRING'])
    time = (params['time'] || 0).to_f
    sleep(time)
    [200, {}
        "Content-Type" => "text/plain",
        ["Hello"]
    ]
end

A.3 NOP workload

def call(env)
    [200, {}, []]
end

A.4 Evented workload – Deferred

def call(env)
    EM.defer do
        sleep 0.05
        env['async.callback'].call([200, {}, ["Hello"]])
    end
    # Signal async response
    Rack::Async::RESPONSE
end
A.5 Evented workload – Delayed

def call(env)
    EM.add_timer(0.05) do
        env['async.callback'].call([200, {}, "Hello"])
    end
    # Signal async response
    Rack::Async::RESPONSE
end