Web routers: An explorative performance review

Richard (Rikki) Andrew Cattermole

2016

# Introduction

Web routers are the core technology that allow for execution of code in response to the mapped requests from an HTTP client. This process is more often known as the request response cycle of web applications. Web applications or web services as they are more commonly are known by are the fundamental way that the world wide web operates for when requiring some form of server side operations. The process of HTTP request-response cycle uses the client-server topology to transmit and manage the control of information by a third party while also displaying content interactively to the user with the usage of HTML, Javascript and Cascading Style Sheets.

When the technology is being utilized in its designed manner, for the purpose of serving hundreds of thousands of requests per second however when used incorrectly of the technology stack design will result in performance degradation. Resulting in more resources required to serve the same amount of requests to be served.

The technology stack typically includes three different parts: the client, the Server, and server side code. The client is most commonly a web browser with an execution engine for Javascript and rendering engine for HTML. The content that is served comes from the server. The server operates as a means to map a single IP address to handle many different possible websites with the possibility of a multitude of different IP addresses. Server side code is external to the server and is code which operates within the confines of a singular web site context. This code allows for dynamic web pages and may include a secondary web router, commonly known as a web service or web application.

The focus of this research is the web router. The web router is core to the web stack however most web application developers do not place much importance into the implementation; instead their focus is the features and interfaces that it provides.

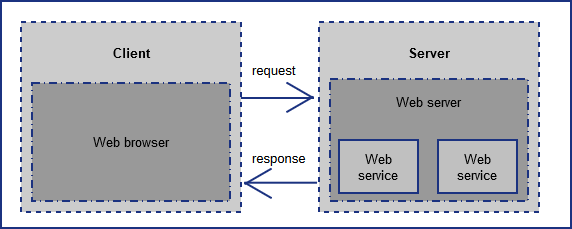
# Literature Review

The World Wide Web was conceptualized in 1989 (World Wide Web Consortium, n.d.) since then there has been a large uptake in its usage by everyone all across the globe to an estimate of 3.4 billion users as of October 8th of 2016 (Internet Live Stats, 2016). With every one of those users working with the standards of Uniform Remote Locator[[1]](#footnote-2), Hypertext Transfer protocol[[2]](#footnote-3) and Hypertext Markup Language[[3]](#footnote-4) in some form or another.

During the early days many different web browsers and servers were created. Majority of these have since long died off but the legacy that is the definition of each has not. As defined by the World Wide Web Consortium (W3C) (World Wide Web Consortium, 2014):

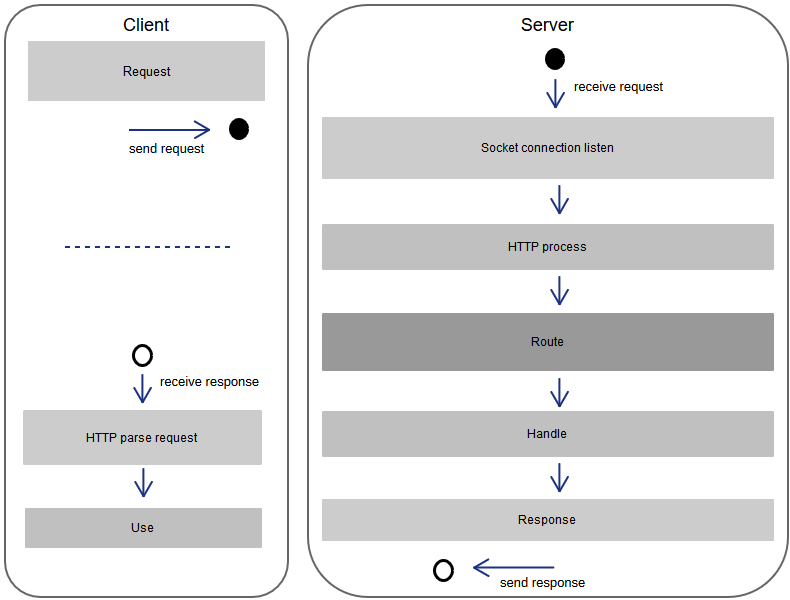
* Web browser  
  A program which allows the display and execution of a web page for a user. Interacts with a web server to provide any data required. This is the most common form of client.
* Web server  
  Retrieves files or resources from the file system or some form of backend such as a web application and sends them to the client as requested.
* Web Server API or service  
  A standalone piece of software that will dynamically create content to send to a client. It communicates in some form to the web server to serve up content to the client.

With an upsurge in internet speeds during 1990s as demonstrated by Nielsen’s Law (Nielsen, 1998), companies and developers alike experimented with dynamic web pages allowing for user interactions not possible with static web pages alone. The Common Gateway Interface[[4]](#footnote-5) was created to allow for external program to be executed as part as the web page processing by a web server. From this point on existing programming languages gained uses that was not seen before which helped to introduce other new programming languages. An example of a new programming language spawned by this would be PHP (The PHP Group) which has the primary purpose of dynamic page creation on each request by the client.

  
Diagram 1: The web (HTTP) request + response cycle

Web servers and web (server side) APIs alike are a field of research that continues to introduce new areas of study for research in both a formal and an informal capacity. Combined they share a very similar technology set only with slightly different purposes and entry points. Of which the web router that resides on a server from which all fundamental mental models originate is conceived.

A web router, primary goal is to map any incoming request from a socket to a function process it. The execution and processing of a request once mapped can be done in any number of languages and quite commonly utilities other protocols such as a Fast-CGI to communicate to another process to execute the request. This is shown in Illustration Illustration, the diagram is based upon HTTP 1.x.

  
Diagram 2: General HTTP request/response process

The tendency of web developers is to focus upon coding within the requests and manipulating of the response for the client side. For the server side the focus is upon handling the routes for a given purpose. When it comes to implementation of the libraries, frameworks and end user code there is little consideration by those who use a specific implementation and along with it, its performance. This can cause problems such as the time it takes to handle a request from getting it to responding to it back the client. These existing algorithms and data structures were created for the usage within a database engine. In the context of a database they have been optimized and analyzed for best performance. For a web server these algorithms and data structures may have improved performance once they have been analyzed with optimizations for this use case.

## The request-response cycle

At the core of a web router is the process of turning an HTTP request into a call to a procedure by which handles the request and returns the result along with some meta information (e.g. cache information). This process has several stages:

1. Socket listening & connection
2. HTTP request received
3. Routing to function call & execution of function
4. HTTP response creation
5. Response sending

The above list is a general overview of the different sequential parts that a request goes through on the server. Commonly it is implemented as:

1. Asynchronous socket listener
2. Thread/Fiber router (choose the thread to execute the request handling in)
3. HTTP request processing
4. Routing to function call
5. HTTP response creation and return

This overview does not take into account some of the problems that implementation all face. Not all information that is required to complete a request is held within memory while the program is executing. Instead it relies upon other software e.g. database to hold it. The process to communicate and retrieve this extra information is expensive compared to the time it would have been if it was within memory at the time of request. The usage of external resources is commonly implemented using blocking operations. A good example of this is file reading and writing which is critical to web servers if they serve content held within the file system. Blocking operations prevent other requests from being concurrently served.

Asynchronous execution of requests is a complex topic that can affect performance between web servers quite significantly. The difference is exemplified by Nginx and Apache2 httpd. Nginx uses asynchronous event based handling, while Apache2 utilizes a thread based approach (DigitalOcean, 2015).

When a connection has been established and the handling code is executed, it will translate the given binary stream into some form of programmatic representation such as classes or structs. Allowing for ease of use and modification by the routing engine. The handling code may be a wrapper to another protocol such as Fast-CGI to allow out of process execution and processing of the request.

The routing engine is responsible for manipulating this request representation into recognizing a specific route. This is primarily done by utilizing some kind of run-time look up and registration into the routing table. This allows for using language features such as attributes to map procedures to routes more organically. For example, the web server Nginx, utilizes Red-Black trees for files caches[[5]](#footnote-6) and Fast-CGI[[6]](#footnote-7) processing. From this the handling mechanism for the specific route is called with it.

## The routing problem

By using the definition of a router as the process to which the decision of which route handler is chosen per request and along with it the definition of what the routing problem is, it can be seen that this can be implemented in many different and varying ways. Because of this, the approaches can differ quite significantly. These different approaches each have a different set of costs and cannot be interchanged in a given context with the expectation of performance related problems appearing.

For example a regex based approach while producing simplified code will require a limited capability, subject only to the host name and URI path but will be subject to the overhead of the regex engine. Where as with custom engine, the utilization of a tree graph will allow for complex conditions but may result in slower and require more memory to be able to operate.

At the core of what a web router does is to take a set of known variables and return a function to execute with the potential to modify the known variables. These set of variables that must be utilized in each searching of the underlying structure are unique when compared with existing research into data structures which focuses primarily upon a single variable. The extension to multiple variables to check and a more complex search algorithm that may need to repeat itself mean existing data structures and algorithms may be used but modified to take into account that simple comparisons do not correctly relate entries to the search parameters.

Current implementations typically use with the path from the HTTP header to perform lookups. These require the least extension to existing data structures and algorithms. Regular expressions (regex) are typically used to implement them. These cover most cases; by utilizing multiple instances of the router implementation it can be used for different HTTP methods such as GET and POST without direct support within the elements of the data structure.

Some web routers support a feature known as rewriting. Rewriting is the process by which requests are modified into being another; however only internally. After a ‘rewrite’ of a request takes place it must be evaluated out as if it was a new request. Most web routers do not implement this feature because of its complex nature. The rules by which it can modify the request by can include the path, domain, time stamp, client IP address and any other HTTP request field.

In non-regex approaches, more information is stored using data structures. Such as a key in a map or to wrap the reference to the handler function. This allows the routing algorithm to use other conditions such as the HTTP request fields of User-Agent, Referer or Host. Support of this is a significant complexity increase and limited research into this area was discovered in the creation of this proposal.

The implementation of the storage mechanism that the web router utilizes can take many forms including a list or a tree graph. These data structures are fairly simple in design but have many optimization opportunities such as cache locality for children in a tree graph which can improve performance by many magnitudes.

TODO: reference Making B+- trees cache conscious in main memory, Jun Rao and Kenneth A. Ross, <http://dl.acm.org/citation.cfm?id=335449>

# Research context

The research proposal being presented here has the end goal of trying to make the world wide web faster, in general better for developers and businesses in terms. The classification for better in this case is the monetary cost of serving requests to the clients. In previous work a considerable amount of research has gone into making the web faster by focusing upon the total performance of the web request processing cycle with little investigation into optimization of server side performance in terms of the web router.

The overarching goal is to determine the performance of the web router algorithms for a given set of circumstances; what is the performance of a set of web router algorithm implementations?

From this a set of sub-questions is formulated to help answer it.

* What are the current performance metrics associated with the request → response cycle?
* What are common algorithms that web routers use and how do they relate to those used in other fields?
* What are the performance charactersitics for the common algorithms as determined by the performance metrics for a set of typical web route patterns

These sub-questions will allow the performance of each algorithm to be characterized allowing analysis to determine the focus for improvement.

# Method

This proposal presents a method for determining the performance of a set of web routers and determining there individual performance capabilities compared to one-another. To do this it is split up into the router implementations and the harness that executes upon each of these.

Each implementation of router provides a generic interface that hides the implementation details but allows for an optimization phase following the addition of all the routes. On top of this is the harness which operates upon the generic interface. The harness allows for testing routes against search patterns.

Once the routes and test patterns are created, the routes are emplaced into the resulting implementations. Each pattern is tested a number of times to determine the overall speed and overhead depending upon the input routes.

The analysis of each web router is implemented by comparing each implementation against each other, by using the recorded values and calculating the difference between the test sets.

# Time line

|  |  |  |
| --- | --- | --- |
| **Task** | **Week of** | **Expected length of time** |
| Router implementation(1)  Flat array | December 5th | 1 week |
| Test harness  Initial creation. This is expected to be a very rough implementation that does not get anywhere close to what is needed. | December 12th | 2 weeks |
| Router implementation(2)  Tree graph, no optimizations | January 2nd | 1 week |
| Router implementation(3)  Tree graph, optimizations | January 9th | 2 weeks |
| Router implementation(4)  Regex | January 16th | 1 week |
| Test harness update  Updates based upon what has been implemented with the routers. As well as include new features to make it more complete based upon what the routers are doing. | January 23rd | 2 weeks |
| Analysis upon router implementations performance | January 30th | Unknown |

# Budget

It is expected that a printing budget of $100NZD will be needed for final copies available for submission.

No other expenses has been expected to be incurred.

# References

DigitalOcean. (2015, 28 1). *Apache vs Nginx: Practical Considerations*. Retrieved from DigitalOcean: Cloud computing designed for developers: https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations

Internet Live Stats. (2016). *Number of Internet Users (2016) - Internet Live Stats*. (Internet Live Stats) Retrieved 10 8, 2016, from Internet Live Stats: http://www.internetlivestats.com/internet-users/

Nielsen, J. (1998, April 5). *Nielsen's Law of Internet Bandwidth.* Retrieved from Nielsen Norman Group: UX Training, Consulting, & Research: https://www.nngroup.com/articles/law-of-bandwidth/

The PHP Group. (n.d.). *History of PHP.* Retrieved 10 8, 2016, from PHP: Hypertext Preprocessor: http://php.net/manual/en/history.php.php

World Wide Web Consortium. (2014, March 14). *How does the Internet work*, 72360. Retrieved 10 8, 2016, from World Wide Web Consortium (W3C): https://www.w3.org/wiki/index.php?title=How\_does\_the\_Internet\_work&oldid=72360

World Wide Web Consortium. (n.d.). *Help and FAQ - W3C*. (W3C) Retrieved 10 8, 2016, from World Wide Web Consortium (W3C): <https://www.w3.org/Help>

1. https://tools.ietf.org/html/rfc3986 [↑](#footnote-ref-2)
2. https://tools.ietf.org/html/rfc2616 [↑](#footnote-ref-3)
3. https://www.w3.org/MarkUp/draft-ietf-iiir-html-01 [↑](#footnote-ref-4)
4. https://tools.ietf.org/html/rfc3875 [↑](#footnote-ref-5)
5. <https://trac.nginx.org/nginx/browser/nginx/src/http/ngx_http_file_cache.c?rev=953512ca02c6f63b4fcbbc3e10d0d9835896bf99> [↑](#footnote-ref-6)
6. <https://trac.nginx.org/nginx/browser/nginx/src/http/modules/ngx_http_fastcgi_module.c?rev=953512ca02c6f63b4fcbbc3e10d0d9835896bf99> [↑](#footnote-ref-7)