Web routers:  
An explorative performance review

A Dissertation

submitted in partial fulfilment

of the requirements for the

B.Science(Honours)

at

Lincoln University

By

Richard Andrew Cattermole

Lincoln University

2017

Abstract of a Dissertation submitted in partial fulfilment of the

requirements for the B.Science(Honours)

Abstract

Web routers: An explorative performance review

by

Richard Andrew Cattermole

In society, today almost everybody has used the internet and by extension the World Wide Web. Over the past 30 years a lot of work has gone into infrastructure creation for the World Wide Web. A common piece of infrastructure for the World Wide Web is the routing mechanism in web servers. When used appropriately it is nearly transparent to a profiler, when it isn’t it can slow down a website significantly. The routing mechanism can have many different designs, and this research is to compare some of these implementations regarding performance. This dissertation concludes with an overview of the worse possible design and what is a better alternative.

**Keywords:** www, web, router, http, server, client, data structures, tree, list, regex, optimize, performance

Acknowledgements

Enter your acknowledgements here. This area also uses the ‘Body Text’ style. The heading ‘Acknowledgements’ uses the ‘Heading MyL1’ style. Use the ‘Heading MyL1’ style on page headings like ‘Acknowledgements’, ‘Table of Contents’, ‘List of Tables’, ‘List of Figures’, ‘References’, etc., ie all page headings except for Chapter titles. To apply the ‘Heading MyL1’ style, place the cursor on the heading, and click on the ‘Heading MyL1’ button (it looks like ‘Heading...’) in the Styles Group of the Home ribbon.

Table of Contents

[Abstract 1](#_Toc485947578)

[Acknowledgements 2](#_Toc485947579)

[Table of Contents 3](#_Toc485947580)

[List of Tables 4](#_Toc485947581)

[List of Figures 5](#_Toc485947582)

[Chapter 1 Introduction 6](#_Toc485947583)

[Chapter 2 Literature Review 7](#_Toc485947584)

[2.1 The World Wide Web 8](#_Toc485947585)

[2.2 The Server 9](#_Toc485947586)

[2.2.1 Performance Issues 10](#_Toc485947587)

[2.3 Data Structures 10](#_Toc485947588)

[2.4 Current Routing Processes 11](#_Toc485947589)

[Chapter 3 Method 13](#_Toc485947590)

[3.1 Implementation 14](#_Toc485947591)

[3.1.1 Data sets 14](#_Toc485947592)

[3.1.2 The Routers 15](#_Toc485947593)

[Chapter 4 Experimental Results and Discussion 17](#_Toc485947594)

[4.1 Time-period of Results 17](#_Toc485947595)

[4.2 Case Studies 18](#_Toc485947596)

[4.2.1 Case Study: List 19](#_Toc485947597)

[4.2.2 Case Study: Tree 21](#_Toc485947598)

[4.2.3 Case Study: Regex 24](#_Toc485947599)

[4.3 Case Study Overview 27](#_Toc485947600)

[Chapter 5 Conclusion 30](#_Toc485947601)

[Chapter 6 Recommendations for Future Work 31](#_Toc485947602)

[Chapter 7 Appendix set\_10\_sites\_5.csuf 32](#_Toc485947603)

[Chapter 8 References 35](#_Toc485947604)

List of Tables

[Table 1 Data set creation parameters 14](#_Toc485947100)

[Table 2 Example routes 15](#_Toc485947101)

[Table 3 Time period R2 values 17](#_Toc485947102)

[Table 4 Case study list min-max 19](#_Toc485947103)

[Table 5 Case study tree min-max 21](#_Toc485947104)

[Table 6 Case study regex min max 24](#_Toc485947105)

[Table 7 Case study overview ratio above:below:at 28](#_Toc485947106)

List of Figures

[Figure 1 Web server operational theoretical process 7](#_Toc485947107)

[Figure 2 The web (HTTP) request + response cycle 8](file:///C:\projects\Lin-Hnr-Dissertation-Head\draft2.docx#_Toc485947108)

[Figure 3 General HTTP request + response processing activities 9](file:///C:\projects\Lin-Hnr-Dissertation-Head\draft2.docx#_Toc485947109)

[Figure 4 R2 for routers over input sets 18](#_Toc485947110)

[Figure 5 list all min-max graphed 20](#_Toc485947111)

[Figure 6 list iteration min-max graphed 20](#_Toc485947112)

[Figure 7 tree iteration min-max graphed 22](file:///C:\projects\Lin-Hnr-Dissertation-Head\draft2.docx#_Toc485947113)

[Figure 8 tree all min-max graphed 22](#_Toc485947114)

[Figure 9 optimized tree all min-max graphed 23](#_Toc485947115)

[Figure 10 optimized tree iteration min-max graphed 23](#_Toc485947116)

[Figure 11 regex all min-max graphed 25](#_Toc485947117)

[Figure 12 regex iteration min-max graphed 25](#_Toc485947118)

[Figure 13 optimized regex all min-max graphed 26](#_Toc485947119)

[Figure 14 optimized regex iteration min-max graphed 26](#_Toc485947120)

[Figure 15 average time per router all 27](file:///C:\projects\Lin-Hnr-Dissertation-Head\draft2.docx#_Toc485947121)

[Figure 16 average time per router iteration 28](#_Toc485947122)

# Introduction

In today's societies computers are taking more and more prominent role in our lives; mobile, Personal Computers (PC’s) and tablets based devices allow us to interact and communicate with a vast array of people and services at a push of a button. The most common communication mechanism used today is the World Wide Web using Hypertext Markup Language (HTML) as the basis of transmission of information to the user. The rendering of HTML is performed by a web browser, and is sent to the browser from a web server. Web servers can contain applications either separated by a process or embedded into it.

During optimization stages one of three different areas is considered. Client side, the Javascript being run, how it renders and what its doing in the background all determine its performance and how long it takes to operate. The server side where by optimizing interaction with system resources such as sockets and the file system to prevent sleeping. Lastly determinace of how resources get from the server to client, typically this is done by caching of resources. Each of the three optimization methods directly affects render times to users.

The focus of research in recent times has been on the client side of web development, as defined by the Hyper Text Transfer Protocol (HTTP) specification**Invalid source specified.**. The main body of research has been to produce better ways to present information, and with the interaction to the user.

The focus on the server side has been in making dynamic content (e.g. address book entries based upon a query) more easily created and manipulated. Anecdotally there does not appear to be much development in this area. This has left certain technologies in use by the web servers with limited work done on them; an example of this would be the web router.

Web routers are the core technology that maps HTTP requests to the web service (code) that produces the response to be sent back to the client. A web router does not interact with the user directly, instead it is configured by descriptions (routes) of websites. Software developers implement a web service to produce dynamic content for a website. Dynamic content and static content (e.g. a websites logo) utilize a web server router to locate the resource handling code to produce a response to the HTTP request.

While the web router implementation is unseen its performance is crucial to the overall of the response times for a request. This research is to compare the performance of various web router implementations to identify potential improvements.

# Literature Review

A web server has a lot of moving parts, web router, known routes/sites, scheduler and request gathering. The diagram Figure 1 Web server operational theoretical process below features a general flow of how a web server works internally. To understand this diagram and where web routers fit into the picture, this chapter is split into, World Wide Web (general background knowledge), the server, data structures and current approaches for implementing the router.



Figure Web server operational theoretical process

## The World Wide Web

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

TODO: references

During the early days, many different web browsers and servers were created, the 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):

TODO: references

1. 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.
2. 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.
3. 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.

Figure 2 The web (HTTP) request + response cycle

These components are displayed in Figure 1.

With greater usage of the internet coinciding and with greater speeds during 1990s as predicted by Nielsen’s Law, companies and developers alike experimented with dynamic web pages allowing for user interactions not possible with static web pages alone. The Common Gateway Interface[[3]](#footnote-3) was created to allow for external program to be executed as part of the web page processing by a server. From this point on existing programming languages gained new uses that were not seen before, which helped to introduce other new programming languages. Some examples of this would be PHP and JSP (Java Server Pages) which have the primary purpose of dynamic page creation on each request by the client.

TODO: references

## The Server

Web servers and web (server side) APIs alike are a field of research that continues to introduce new areas of study in both academic and in informal capacities. Combined they share a very similar technology set, with only slightly different purposes and entry points. The web router resides on a server and is a required component for the operation of both web servers and web APIs.

Figure 3 General HTTP request + response processing activities

A web routers, primary goal is to map an incoming request from a socket to a function (code) to process it. The execution and processing of a request once mapped can be done on the server or specific route handling code in a separate process. The handler can be written in any language. This is shown in Figure 2 General HTTP request + response processing ; it is based upon HTTP 1.x.

The primary focus of web developers is coding 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. There may be little consideration by those who use an implementation about, its performance. This can cause delays such as the time it takes to handle a request from getting it to sending a response back to the client. The algorithms used in web routers (e.g. linked list, B-Tree) and data structures were originally created for database engines. In the context of a database they have been optimised and analyzed for best performance. For a web router, these algorithms may give improved performance once they have been analyzed and optimised for this use case.

### Performance Issues

The standard implementation of a web server has the following processing:

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

In these steps there are a number of potential performance issues:

* Non-blocking asynchronous socket listening verses blocking synchronous socket listening
* Scheduling of the handling code on a thread/fiber
* Blocking operations (e.g. communicating with the database, file reading/writing)

These have been worked on by developers since the beginning of the web, but comparatively little has been done in the routing component. Hence the focus of this study is on improving the performance of the web router.

## Data Structures

In computer science there are two primary data structure families. Lists and Tree graphs. These data structures are built on top of heap memory using two techniques, struct/class based storage and arrays in the form of either static or dynamic. A dynamic array is a pointer to a location within the heap or stack with a length to indicate how many indices has been allocated for.

Lists are a more expensive method (in amount of memory allocated and lookup times) to perform storage of linear data. The benefit of using a List over an array is having faster insertion and removal times.

A Tree graph instead of storing data linearly, stores data separated with multi layers of parents and children. This separation is very good for decreasing lookup times for data that is similar but with vast differences later on, which is the case with routes. Typically there is a root node with a set of children associated with it. The root node itself doesn’t have a value but is the starting point for lookup. Each node has children and a parent.

TODO: references!

## Current Routing Processes

A web router maps incoming requests to a route handler (code). The approaches that are available to implement the routing can differ in functionality and performance. These different approaches each have a different set of costs (memory and CPU) and the performance will depend upon the profile of the web traffic.

There is a variety of different methods used in implementing a web router. The main ones used are: tree graphs such as a Red-Black as implemented in Nginx[[4]](#footnote-4) or a single Regular Expression (regex). A single regex can simplify the code required but will result in a limited functionality

TODO: references

Current implementations typically use the path from the HTTP header to perform lookups. Routes may contain constants as path segments, parameters or a “catch all” for all following path segments. Existing basic data structures can handle paths in this format. E.g. regular expressions are typically used to implement them. These can cover most cases by utilizing multiple instances of the router implementation. Each instance of the router handles a different HTTP method such as GET and POST.

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

In non-regex approaches, more information is stored in the data structure (HTTP method, port etc.). This allows the routing algorithm to use other conditions such as the HTTP request fields of User-Agent, Referer or Host. Support for this significantly increases the complexity of the implementation and limited research into this area was discovered in the creation of this proposal.

The data structure that the web router utilizes can take many forms including a list or a tree graph. These data structures are simple in design but have many optimisation opportunities (such as cache locality for children in a tree graph) which can improve performance by many magnitudes.   
TODO: references

# Method

A benchmark harness will be created to execute tests to record the timings for routing of requests. The harness will provide a common interface to allow the timings to be gathered the same way for each web router implementation to be tested and compared.

The input data for the benchmarking comprise two different sets of information. The first is the routes to be stored into the relevant router data structure, and the second set is of requests to be executed by the router implementation. The routes will be preloaded before testing and will be fully optimised by the router before the execution of the benchmarking.

The design of the routes used in the data sets include: static paths “/my/path/goes/here” with variable number of parts “/part”, a number of parameters (aka “variables”) “/my/path/:variable” and a catch all “/my/path/\*” for all values following the previous values. These will be combined into the forms: “/path/:vars/\*”, “/path/\*” and “/path/:vars” with path and vars being variable in number. The combination and complexity will be produced algorithmically for the purposes of testing as many corner cases as possible.

Different input sets will be generated by: how many path parts they contain, the number of parameters/variables and how many have a catch all.

Each implementation is expected to run solely within a single thread. All input sets will be stored in memory and initialisation will have been performed before a test starts. Each input set is executed in multiples of a ten giving: 10, 100, 1000, etc. The result for each multiple is then averaged for comparison with the same multiple for all other inputs sets. Each test will occur without any breaks (e.g. no thread sleeping) and will not include the time for pre-loading and optimisation.

The computer that will perform the benchmarking has the following specification:

* CPU: Intel Xeon v3 E5-2630, 8 cores at 2.4ghz base frequency and 20mb cache
* Memory: DDR4 64GB at 3200mhz
* OS: Windows 10

Storage of input and results will be done in memory. Once results are fully generated and testing has concluded for a test set, they will then be stored on the hard disk. This will prevent performance penalties associated with hard drive storage appearing in results.

Once the results have been gathered, graphs comparing multiples for a test set as well as comparing specific multiples between implementations for a test set. Particular attention will be paid to outliers within these graphs to determine problems of the implementation. Comparisons will also be made between test sets to determine how the type of web requests impacts performance.

## Implementation

For the implementation two sets of artifacts were created, code and the data sets. The data sets comprise of a variety of values generating a set of website route descriptors that are to be benchmarked. Secondly the routers themselves as code within the benchmarking framework that was as well created.

Three routers were created. List, tree and regex. Of the three they share a similar implementation of a site lookup using an array which contained all parameters to decide the set of routes to search over including, SSL status, HTTP error code and port. A site could include multiple hostnames, ports and ssl support.

### Data sets

The benchmark input data was generated using a generator which had its input provided by a script. The input generator produces a unique set of routes given a max number of entries, parts, parameters and tests per route. These can be combined together to produce a multi-site input into the router benchmarker.

The benchmark data generator runner script used a variety of values for max number of entries, parts, parameters, tests, specific number of sites and iterations. The number of catch all was derived from max parts and max parameters. The set of values that was used were:

Table Data set creation parameters

|  |  |
| --- | --- |
| Max entries | 10, 20, 50, 100, 200 |
| Max parts | 5, 10, 20, 30 |
| Max parameters | 4, 10, 20 |
| Max tests | 1, 2, 3, 5, 10, 20 |
| Site count | 1, 2, 3, 5, 10, 20, 30, 100 |
| Iterations count | 1, 10, 100, 1000 |

The above table is used using the pseudo code:

Foreach num-entries in set max-entries

Foreach num-parts in set max-parts

Foreach num-parameters in set max-parameters

Foreach num-tests in set max-tests

Foreach num-sites in set site-count

Foreach num-iterations in iterations-count

Perform operation

set-count = set-count + 1

The naming of a data set is contributed by set-count and num-sites. Into the form of “set\_<set-count>\_sites\_<num-sites>.csuf”. For output files it was “set\_<set-count>\_sites\_<num-sites>\_iterations\_<num-iterations>\_result.csv”. Each entry has a number of test URI’s associated with it. Each URI is made up by a number of parts, parameters and optionally a catch all. The number of iterations used increases the reliability of the results by making it repeat them X number of times.

The values chosen were picked to try and get a range that fitted most web servers and web service frameworks use cases, but because of how many web sites that exist today it is impossible to know if it fits correctly to the use case that is 2017. A word dictionary was used to produce unique words per path part, using a random number to pick which one to replace at each node for a tree graph. The tree graph got walked to produce tests and route definitions. Each router implementation was tested during development to determine the correctness of routing. What the router returned as part of the benchmarking process was assumed to be correct and no checking went into this. The data that was created came in the form (an except from *set\_10\_sites\_5.csuf*):

Table Example routes

|  |  |  |
| --- | --- | --- |
| Spec | Tests | Matches? |
| /Hollie/skyline/phlegmatic/  apical | /railwayman/coil\_s/ulster/noncorrosive/belletristic/  Lebanese/mechanic/mercilessness | No |
| /Hollie/skyline/phlegmatic/apical | Yes |
| /witticism/:aerodynamics/  :greenmail/chest/\* | /witticism/Joy/hookah/chest/proscription/link/Batu/  latecomer/bugler/dreariness | Yes |

A route is seperated into its path parts. The path parts being constant, parameter and catch all. To understand this, it is shown for the second example in Table 2 Example routes in the Table 3 Example route explained.

Table Example route explained

|  |  |
| --- | --- |
| **Path part** | **What it means** |
| /witticism | This is a constant, it only matches “/witticism”. |
| /:aerodynamics | This is a parameter, it can match anything except “/”. You can encode it using base64 encoding but this will not be matched. As an example it can be matched by “/foo” or “/cats”. |
| /:greenmail | Like “/:aerodynamics” this is a parameter. |
| /chest | Another constant just like “/witticism”. |
| /\* | This is a catch all, it can accept anything. However if there is a route that is a constant or parameter that would go in this place then those would be preferred instead. An example of this can be “/foobar” or “/abc/def”. |

From this the data set were created with each entry having a specification, tests (that may or may not match it). In total 287 benchmark input sets were created as part of the generation process. Only 150 of these were run and took over a week to complete. The last of these took over 24 hours to run and the following sets would only become longer in time as the complexity could only rise as per the table.

### The Routers

Three routers were implemented. A tree graph, list (array) and regular expression (regex). The tree graph was based on a rooted child array approach with multitude of specific node pointers per node to support parameters, catch all and other children. The list was an array sorted by the route value it contained. Lastly the regex router was implemented using D’s standard library implementation (std.regex). This implementation does not perform JIT’ing or any fancy/complex tricks to make it significantly faster making it a good base for how good/bad it can be.

The list router utilised an array that was formatted based upon hostname, if there was a catch all and the path itself. This was done so that once it started matching and didn’t find any more matchable entries it would stop.

The tree graph used a hierarchical set of nodes with each node have its own children (the path segment) and if it was a variable. Preference was given to non-variables but if one could not be found variable and finally catch all was used for routing. Finally the last router regex was implemented by using D’s std.regex (available in the standard library).  
The optimized version of the tree graph allocates once for all nodes and put all the children together before going down. The optimized regex router combines all the separate string into one large one.

All routers and benchmarking suite was compiled using dmd 2.073.0 in release mode with optimizations turned on. LDC (LLVM) during implementation was unable to compile the suite as it required a feature from std.regex that was only available in the latest version.

# Experimental Results and Discussion

The results were gathered in a continuous time period of over a week. In initial analysis results gathered earlier in the benchmarking process appeared to be most representative of real world usage and have less irregularities visible. This was based upon the time taken for the amount of work done per input set and resulting data. Because of this the assumption that 100-150 results had irregularities and should be considered less viable was taken.

It was observed during the benchmarking process that every 50 input sets contributed to a significantly longer time to complete, the increase was done slowly in a curve. This increase in time with the added potential of irregularities can be studied by breaking down a routers benchmark results into three groups (chosen for the even group numbers) of 50. Breakdown of the results occurs in 4.1 Time-period of Results with discussion.

Each router implementation results are studied in separate case studies in an attempt to understand each of there's characteristics.

## Time-period of Results

The generation of results required long running processes which produced irregularities within the output. The question to be taken is how much of an error compared to the earlier ones executed was it? This answers a new question, was the input data set generated too big given the problem domain.

The following table of R2 was calculated from the means of the times taken to benchmark a given input set to a router, not per iteration. A trend line was added to the given data set chosen using a polynomial form with 8 degrees of points. The usage of three groups of 50 was chosen arbitrarily.

Table Time period R2 values

|  |  |  |  |
| --- | --- | --- | --- |
|  | Set 0-50 | Set 50-100 | Set 100-150 |
| List | 0.05523 | 0.04737 | 0.11400 |
| Tree | 0.04176 | 0.03878 | 0.09072 |
| Regex | 0.03768 | 0.04476 | 0.07980 |
| Optimized Tree | 0.04018 | 0.03989 | 0.09001 |
| Optimized Regex | 0.03910 | 0.05197 | 0.05632 |

The table was graphed to determine if any trends were occurring in data matching.



Figure R2 for routers over input sets

The graph shows three very noticeable traits, firstly 0-50 and 50-100 are very close to being similar in value. Second the 100-150 sets of data tend to be significantly higher than the other two sets. Lastly four out of the five router instances have the same curve shape. The router instance with a different one is the optimized regex which shows a much more predictable result. For 50-100 input sets mark, regex implementation (both optimized and unoptimized) was the only one to increase in R squared which would align with it being very consistent but having set min/max consistency.

## Case Studies

Each router type (List, Tree, Regex) is separated into its own case study, if included an optimized router implementation is considered as part of the study. The notation used in the graphs below is: <name>A, <name>I, <name>OA, <name>OI. The A stands for all, I stands for iteration, O for optimized and if O is not present means unoptimized (except for list).

### Case Study: List

From the time series analysis Table 5 Case study list min-max it is apparent that the list router for 0-100 input sets scales to match the calculated formula highly. The differing R2 by around 0.008 is a reasonable number and is supported by looking at the results by min-min, min-max, max-min and max-max of the entire responses.

Table Case study list min-max

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Min (hnsecs) | Mean (hnsecs) | Max (hnsecs) |
| ListA – min | 400 | 32854723 | 2819994600 |
| ListA - max | 400 | 36521920 | 3030136700 |
| ListI - min | 200 | 32341 | 696800 |
| ListI - max | 400 | 39639 | 790700 |

The means are all fairly large which implies although the minimums are small, there is a high cost in larger number of sites + routes. This in turn is backed up by it being an array, array looks ups cannot skip elements but it can quite early assuming it is sorted. Larger the sites + routes, the longer time it takes to search it for a given route.

To observe the min/max of the results more carefully using the graphs Figure 5 list all min-max graphed and Figure 6 list iteration min-max graphed, observably all the results minimums were under 50% of the max and had a high concentration around 30-40%. This means that this router implementation will typically be able to complete in at a minimum of 30-40% of the maximum time. The dependency upon which route takes the absolute minimum time cannot be determined without the implementation making the list of routes available at any given point in time to be analysed. The sorting of the array which gives order to the searching is important to the functioning of the router so that it cannot support reordering with priorities for commonly accessed routes.

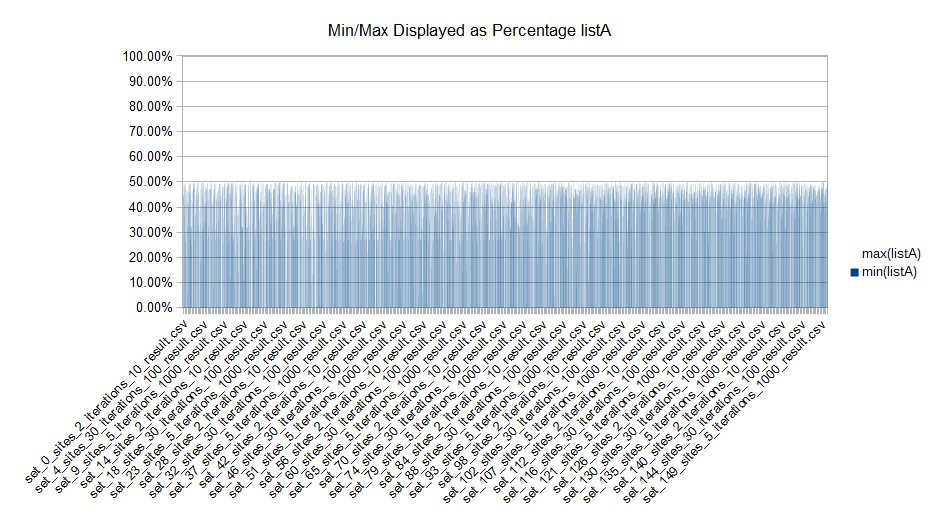


Figure list all min-max graphed

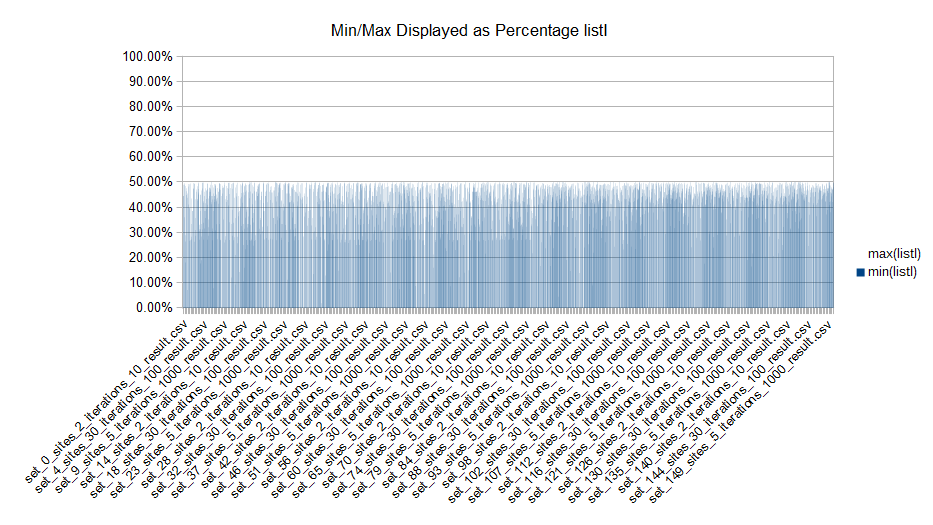


Figure list iteration min-max graphed

### Case Study: Tree

From the time series analysis Table 6 Case study tree min-max it is apparent that the tree router for 0-100 input sets scales to match the calculated formula highly. The differing R2 by around 0.003 is a highly desirable number but it is not validated in the data Table 6 Case study tree min-max. From the data it can be seen that there is quite a difference between minimum and maximum. However when compared to the list router, the mean is closer to min in the tree compared to the list.

Table Case study tree min-max

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Min (hnsecs) | Mean (hnsecs) | Max (hnsecs) |
| TreeA – min | 400 | 174044 | 10859900 |
| TreeA - max | 1200 | 208644 | 11556600 |
| TreeI - min | 100 | 276 | 2800 |
| TreeI – max | 100 | 530 | 5200 |
| TreeOA – min | 400 | 178081 | 10089300 |
| TreeOA - max | 400 | 208999 | 15245600 |
| TreeOI - min | 100 | 273 | 2400 |
| TreeOI – max | 100 | 449 | 11500 |

Comparing the optimized version to the unoptimized version of the routers it can be seen that there is almost no difference except that the best case scenario seems to be lower but worse case is a lot higher. The only difference between the two is a single memory block versus lots of smaller ones suggest that memory locality optimization could improve performance more than the currently tested implementation.

To observe the min/max of the results better, using the graphs Figure 8 tree all min-max graphed, Figure 7 tree iteration min-max graphed, Figure 9 optimized tree all min-max graphed and Figure 10 optimized tree iteration min-max graphed, observably all the results minimums were under 50% of the max and had a high concentration around 20-30%. This means that these router implementation will typically be able to complete in at a minimum of 20-30% of the maximum time. However this is not affected by memory optimization too much suggesting that perhaps such optimization may not be missed for the general case but is necessary for those that tweak web servers.

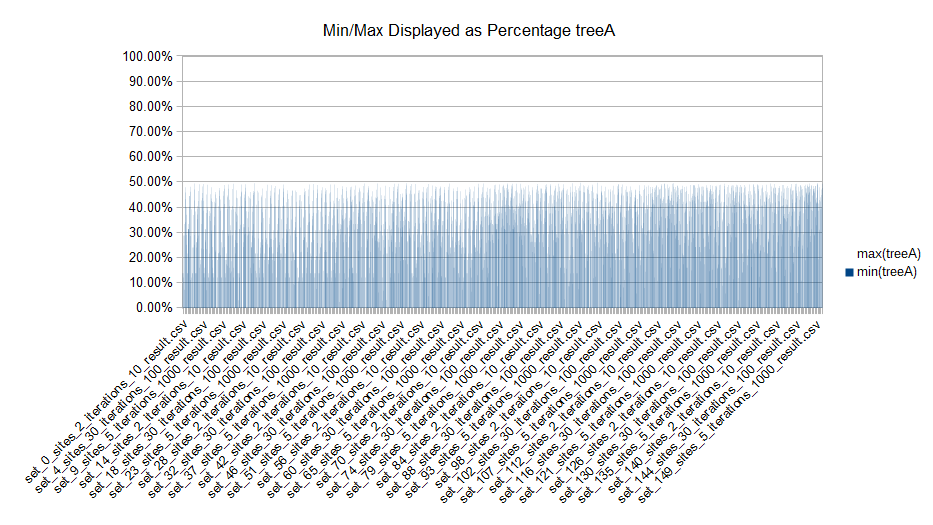
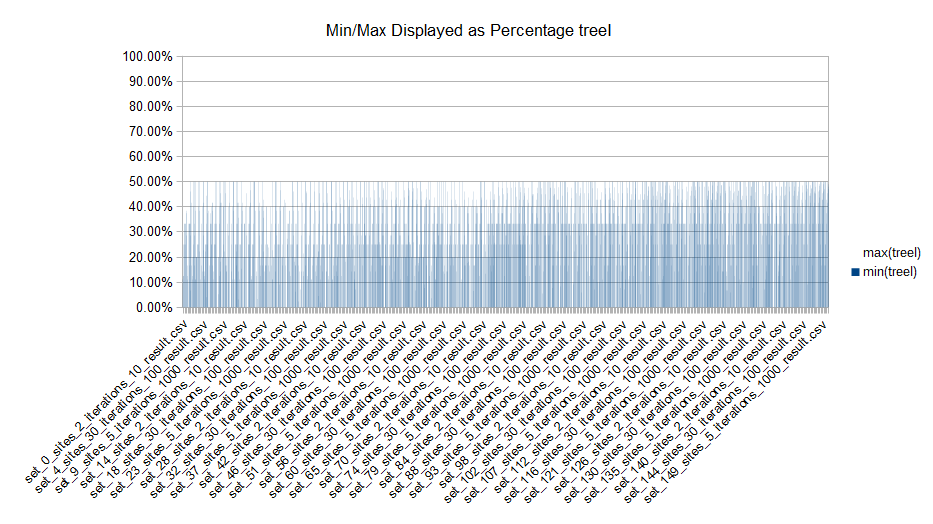


Figure tree iteration min-max graphed

Figure tree all min-max graphed



Figure optimized tree all min-max graphed



Figure optimized tree iteration min-max graphed

### Case Study: Regex

From the time series analysis Table 7 Case study regex min max it is apparent that the unoptimized regex router has the worse relationship to the data compared to the other implementations. The optimized regex router has a different relationship with 100 set being vastly higher than any others. In the unoptimized implementation the input sets 0-100 there is a differing R2 value of 0.07 and for 100-150 of 0.021. In the optimized implementation the input sets 0-100 there is a differing R2 value of 0.011 and for 100-150 of 0.004. From this we can see that given the input set that there can be very high minimum times which is backed up from the min-max’s of the time it took to perform. The max-max’s tend to be very close to to the min-max’s which implies a high overhead as the regex string grows.

Table Case study regex min max

|  |  |  |  |
| --- | --- | --- | --- |
| Metric | Min (hnsecs) | Mean (hnsecs) | Max (hnsecs) |
| RegexA – min | 800 | 133211401 | 8161259700 |
| RegexA - max | 8500 | 145157883 | 8567889600 |
| RegexI - min | 800 | 155253 | 2016600 |
| RegexI – max | 4500 | 187002 | 2235800 |
| RegexOA – min | 800 | 498155614 | 24742495900 |
| RegexOA - max | 2100 | 536130030 | 28392916800 |
| RegexOI - min | 800 | 554891 | 10276600 |
| RegexOI – max | 1700 | 635525 | 11899700 |

The means are all large which implies although the minimums are reasonably small, there is a high cost in larger number of sites + routes. The optimized versus unoptimized had one major difference, one regex string (internally) versus one per route. The optimized tended to do better for smaller sets of routes + sites, but for larger routes would do significantly worse for both min/max-max’s and min/max-mean’s.

To observe the min/max of the results better, using the graphs Figure 11 regex all min-max graphed, Figure 12 regex iteration min-max graphed, Figure 13 optimized regex all min-max graphed and Figure 14 optimized regex iteration min-max graphed, observably all of the results minimums were under 50% of the max and had a high concentration around 20-40%. This means that these router implementation will typically be able to complete in at a minimum of 20-40% of the maximum time. However this is not affected by single string optimization too much suggesting that perhaps such optimization may not be of benefit especially with larger times in max time. But this may entirely depend upon the regex engine.

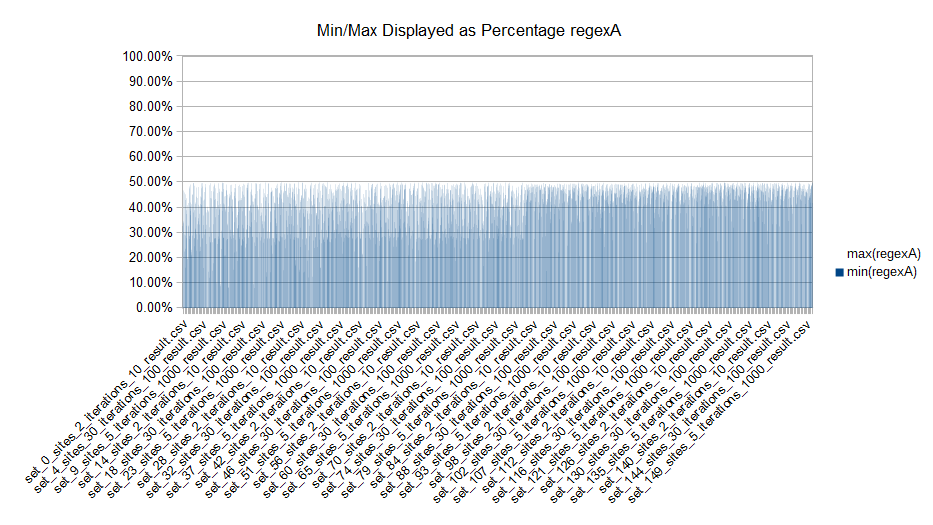


Figure regex all min-max graphed

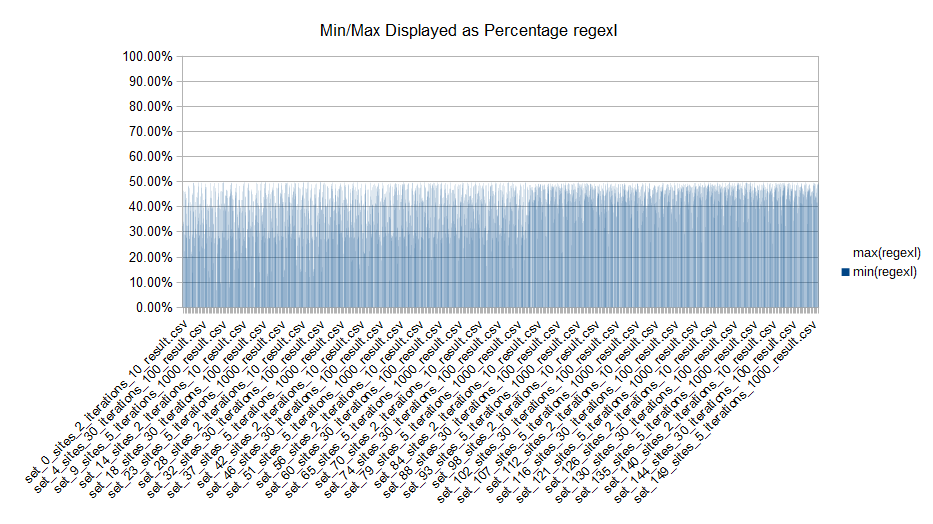


Figure regex iteration min-max graphed

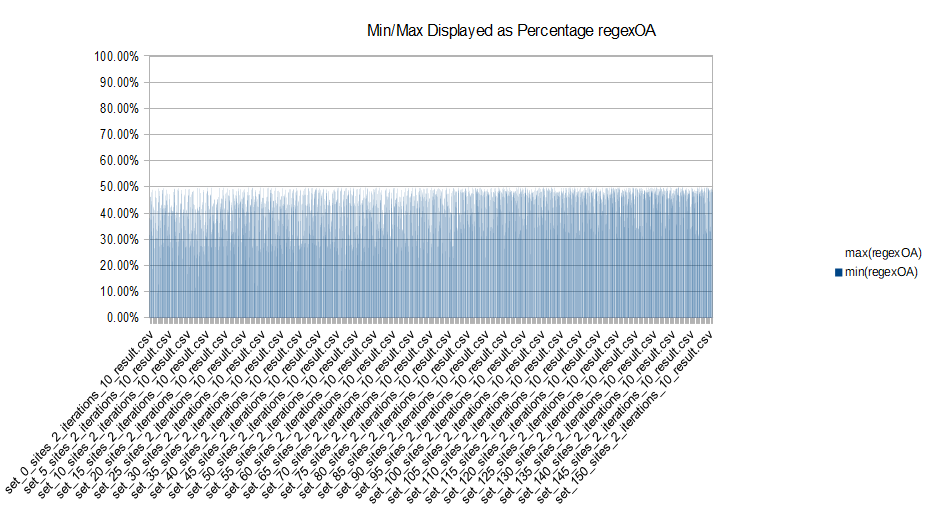


Figure optimized regex all min-max graphed

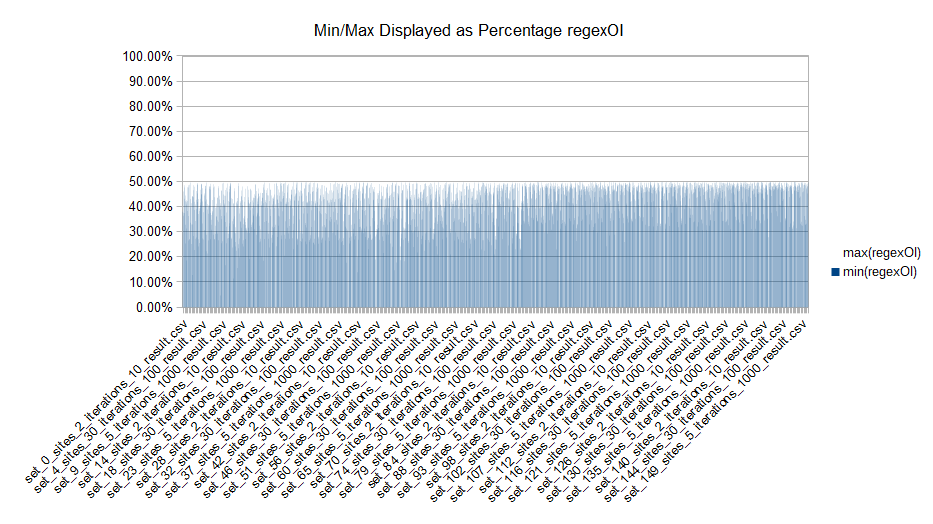


Figure optimized regex iteration min-max graphed

## Case Study Overview

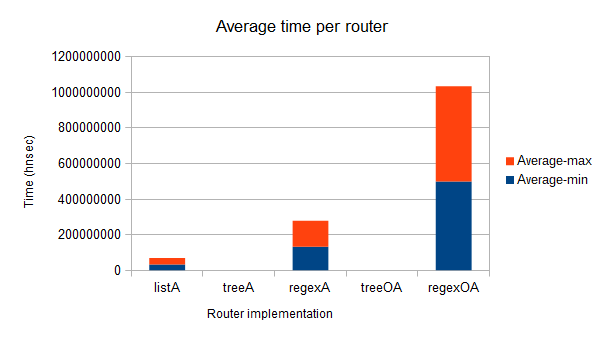
Each router implementation as a case study explore how a router performs through different inputs. In each of the case studies observations as seen in Figure 15 average time per router all and Figure 16 average time per router iteration, were made per the router implementation performance in different metrics. Over all the tree graph performs the best, list then regex. Optimized versus unoptimized this statement holds true. Comparing the average time per iteration and for all iterations the only change is the scale of the numbers, over all the pattern is the same.

Figure average time per router all

The treeA and treeOA do not show up on the graph yet are part of the data set. They are too small given the scale that the others are.

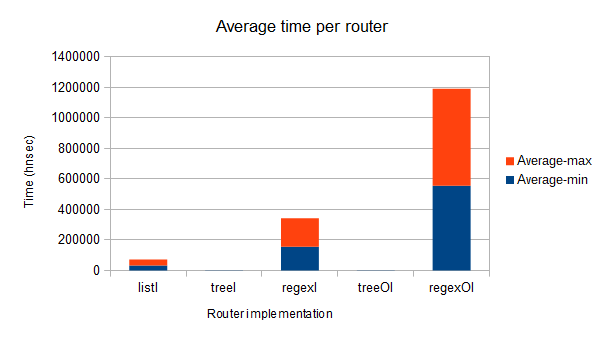


Figure average time per router iteration

For each router a comparison Table 8 Case study overview ratio above:below:at of the number of route benchmarks above versus below the mean was done.

Table Case study overview ratio above:below:at

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ratio above | Ratio below | Ratio at |
| ListA | 56.6% | 45.5% | 0.1% |
| ListI | 55.7% | 45.9% | 0.6% |
| TreeA | 53.0% | 48.2% | 0.9% |
| TreeI | 30.2% | 43.4% | 28.5% |
| RegexA | 56.8% | 45.3% | 0% |
| RegexI | 54.8% | 47.2% | 0.1% |
| TreeOA | 55.0% | 45.7% | 1.4% |
| TreeOI | 32.9% | 34.4% | 34.8% |
| RegexOA | 56.6% | 45.5% | 0.1% |
| RegexOI | 54.3% | 47.7% | 0.1% |

Majority of the results is around the 0.05 mark, this is a baseline for both above and below. Of note is that optimization on the tree graph did have significant impact on total number of routes which took longer or shorter and in having lower numbers. However for regex implementation optimization did not aid it.

# Conclusion

This research explores the question as to “what are the performance characteristics of common web router algorithms under a typical web request scenario”. Three research questions were established:

1. What are the current performance metrics associated with the request/response cycle?
2. What are common algorithms that web routers use and how do they relate to those used in other fields?
3. What are the performance characteristics of commonly used algorithms to implement a web router given a range of routing scenarios as input?

The scope of the work could have included socket handling, threading and other decoding processes used as part of the first sub question. This work would have included metrics of the entire operations of a web server and so for this work it was not included as it was outside of the scope of the web router. The first question cannot be answered in this dissertation but may in future work built upon this.

The second sub question was answered through the literature review, this highlighted that tree graphs and regular expressions are currently used in web router implementations. A list router was implemented for a comparison to what should be the slowest data structure lookup method.

Each of the routers results answer the last question, what are the performance metrics for the techniques involved? The tree graph was fastest over all, list was two magnitudes slower than the tree graph and regex engine was three magnitudes slower than tree graph. This is based upon the minimums of the means (for all data sets used). The optimization versus unoptimized status did not affect the results enough to show in magnitudes but may play a role in performance tuning of a web server.

In conclusion, this work has found that the Tree graph data structure is best suited towards web router implementation. It represents routes the most accurately. A common method for routing using regular expressions to represent routes was found to be slowest and should not be used unless pattern matching was extremely necessary in its usage.

# Recommendations for Future Work

This work did not cover exploring the request/response cycle and its associated metrics, this work would be built on top of this one. A limited number of techniques were used in this research. To further this several more web routers can be created:

* More Tree graph variants, Red-Black, Splay and AVL.
* Merging of a tree graph with regex on request, to provide more modeling power when required but reverting to a simpler algorithm lookup for performance.
* Using another data structure to represent sites to route storage. The current one used is a basic array with a child of the element to the root node.
* Validation of results during bench marking e.g. HTTP status codes.
* Using another regex implementation, PCRE2 and perhaps a JIT’d version.

Creating a different more randomized input set benchmarking. Instead of doing them linearly, randomize order and do some of them multiple times.

# Appendix set\_10\_sites\_5.csuf

.new

/convoke/rook/outermost/mtg/fiendish/loyaler/vastness/inn /Meyer/murky/trousseau/annuity /Meyer/murky/trousseau/annuity /convoke/rook/outermost/mtg/fiendish/loyaler/vastness/inn /convoke/rook/outermost/mtg/fiendish/loyaler/vastness/inn

..status\_code 200

..requiresSSL false

/Meyer/murky/trousseau/annuity /Meyer/murky/trousseau/annuity /Meyer/murky/trousseau/annuity /Meyer/murky/trousseau/annuity /convoke/rook/outermost/mtg/fiendish/loyaler/vastness/inn /convoke/rook/outermost/mtg/fiendish/loyaler/vastness/inn /Meyer/murky/trousseau/annuity /Meyer/murky/trousseau/annuity /Meyer/murky/trousseau/annuity

..status\_code 200

..requiresSSL false

.num\_requests 12

.len\_paths 87

.len\_requests 468

.new

/chortle/likableness/junior/centrist /chortle/likableness/junior/centrist /chortle/likableness/junior/centrist /chortle/likableness/junior/centrist /chortle/likableness/junior/centrist /chortle/likableness/junior/centrist /chortle/likableness/junior/centrist

..status\_code 200

..requiresSSL false

.num\_requests 6

.len\_paths 36

.len\_requests 216

.new

/Bermudan/Commonwealth/manipulation/steersmen/Novocaine/NT/sexless/milling/schoolmaster

..status\_code 200

..requiresSSL false

/ilea/insertions/gorse/attribution /ilea/insertions/gorse/attribution /ilea/insertions/gorse/attribution

..status\_code 200

..requiresSSL false

.num\_requests 2

.len\_paths 121

.len\_requests 68

.new

/rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship /rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship /rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship /rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship

..status\_code 200

..requiresSSL false

/rankle/inning/delineate/Mekong /rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship /rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong

..status\_code 200

..requiresSSL false

/rankle/recon/risky/wraiths /rankle/inning/spatial/laundromat/gesticulate/sparkler/gastronomical/girlfriend/discipleship /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/inning/delineate/Mekong /rankle/recon/risky/wraiths

..status\_code 200

..requiresSSL false

.num\_requests 17

.len\_paths 150

.len\_requests 889

.new

/cumber/:Polaris/Zosma/:groat/\* /cumber/batman/Zosma/Scorpius/antrum/stadium/annihilator/testifier/Marseilles/Fuchs /cumber/dailiness/Zosma/Flatt/lucubration/jurisdictional/teapot/casino/Merck/disdain /cumber/pluralization/Zosma/Scottish/nonoccurence/Herminia/Lyle/phoneme/hamburg/Erin /cumber/Gerber/Zosma/krona/homemaking/Bataan/architecture/Farley/rigid/succession /cumber/emphatically/Zosma/kvetch/inalienability/tilapia/phylum/calciferous/signora/parentage /cumber/kabbalah/Zosma/gendarme/aghast/PS/noncooperation/exterior/polyvinyl/via /cumber/went/Zosma/deprecatory/denominational/harmless/tarmac/righteousness/could/Portuguese /cumber/formulator/Zosma/overthrown/ophthalmic/BC/stipend/acquisitive/LBJ/quack

..status\_code 200

..requiresSSL false

/cumber/:Polaris/Zosma/:groat/desiccation /cumber/dailiness/Zosma/Flatt/lucubration/jurisdictional/teapot/casino/Merck/disdain /cumber/pluralization/Zosma/Scottish/nonoccurence/Herminia/Lyle/phoneme/hamburg/Erin /cumber/Gerber/Zosma/krona/homemaking/Bataan/architecture/Farley/rigid/succession /cumber/emphatically/Zosma/kvetch/inalienability/tilapia/phylum/calciferous/signora/parentage /cumber/kabbalah/Zosma/gendarme/aghast/PS/noncooperation/exterior/polyvinyl/via /cumber/went/Zosma/deprecatory/denominational/harmless/tarmac/righteousness/could/Portuguese /cumber/formulator/Zosma/overthrown/ophthalmic/BC/stipend/acquisitive/LBJ/quack /cumber/formulator/Zosma/overthrown/desiccation

..status\_code 200

..requiresSSL false

.num\_requests 16

.len\_paths 72

.len\_requests 1314

# References

**There are no sources in the current document.**

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