Web routers: An explorative definition

Sdf

Richard (Rikki) Andrew Cattermole

2016

Abcd

# Introduction

Web servers and web applications alike are an emerging field of research in the world of Information and Technology (IT). They share very similar technology only with slightly different purposes and entry points. Of which the router from which all fundamental mental models originates from is convceived.

A web routers primary goal is to map an incoming request to a function to 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 wrapper to communicate to another process to execute it.

TODO fast cgi reference

The execution of a request handling is quite an extensive place of research with many frameworks (ASP.net, Play Framework, JSP and Zend) being used in a vastly differing environments to each other but fundamentally doing the exact same things.

TODO: foot-notes for ASP.NET, Play Framework, JSP and Zend

Most users of such frameworks utilize higher forms of registration systems and do not interact with the routing engine directly instead preferring to allow for using reflection and utilizing attributes upon symbols to do most of the work for them. Without going into depth and understanding what the router itself is doing or how to improve its execution speeds.

This can be quite useful unless you need to start considering thread locality of the router representation along with can it be modified during the running of the application. Of course there is one other use case that is highly problematic in the case that is PHP, no global state to store the router except for what the interpreter itself does automatically.

TODO: reference about PHP and global state

TODO: “Thread locality of memory is problematic for data structures because … memory location yada yada”

TODO: non cached memory penalty on x86 cpus?

Because of the penalties taken when data is not within CPU cache different kinds of data structures are utilized to minimize this and along with it minimize search times within the memory for specific nodes of information. These types are fundamentally either list, map or tree graph of some form. Because of this routers quite commonly utilize existing data structures and algorithms represented in a list or a tree graph.

TODO: references regarding different data structures

Unlike the primal data structures web routers utilize complex conditional evaluation per node to determine if the request matches it. This could be for parameters passed by the query string or the path used. Which makes them far slower then anything the original data structures were designed to handle. Beyond this they quite often implement a feature known as rewriting which requires multiple executions of the router to discover the end route to execute. An example of this would be the Apache2 web server along with its rewrite module (extension).

TODO: apache2 web server routing for rewriting foot note

As shown in the research into data structures and there appropriate algorithms, the performance of each can vary widely to the point of differing structure can affect which one you should use for both improve performance and limit memory usage. This is the reason the difference in what the routing engine utilizes below the conditional nature of it could drastically affect it and without formal research into the matter at hand it is effectively an unknown quantity in the age of the world wide web.

TODO: reference into differing algorithm + data structure performance

For the above reasons the purpose of this dissertation proposal is to present an argument for a dissertation upon which a comparison and contrast of the given different types of routers that currently exist with consideration of performance for specific use cases.

# Literature review

At the core of a web router is the process of turning a request from Hyper Text Transfer Protocol (HTTP) 1.0/1.1/2.0 into a call into a procedure by which to handle and return result along with some meta information such as cache information. This is done by separate parts:

1. Socket listening
2. HTTP request processing
3. Routing to function call
4. HTTP response creation

The above list is a general overview and problems that an implementation may face. Commonly it is instead separated out into:

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

However this does not take into account the need for blocking actions such as file reading/writing and other socket related processing such as database access. So for this use case to get the best performance the usage of Fibers is commonly used. Fibers allow for using up as much of the time slice that the kernel is willing to give a thread while also switching out what code (and with it its stack) is being executed at any given moment. Improving the number of requests a single thread is capable of execution before its time slice is ended or the blocking operations takes precedent.

Asynchronous execution along with threads and fiber handling is a complex topic that will not be further discussed in this literature review. These topics alone are worth research

/\ TODO: see (a) (b) (c) for further information on these matters.

When a connection has been established and the handling code is available to being execution, 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 such as what is done with Apache Web Server version 2’s mod\_rewrite. The given procedure that goes on to execute the code for that route takes in the request representation along with a response representation ready to send the HTTP response representation to the client. The exact nature of the execution handler does not matter. It could be PHP or a web service framework written in the same language calling into its own router.

## The Hyper Text Transfer protocol

TODO: Basic what is it?

TODO: overview of basics of each version \/

Versions:

1. 1.0
2. 1.1
3. 2.0

TODO: why was 2.0 needed? Problem + its solutions

TODO: what does a request provide?

TODO: what does a response contain?

TODO: overview of how requests work in terms of sockets (need a nice pretty image)

## Optimization techniques

TODO: cache locality

TODO: memory layouts

TODO: JIT’ing

TODO: adding threads

TODO: more efficient instructions

TODO: less work

TODO: rearranging (e.g. one whole regex vs a whole lot of parts)

## The routing problem

Routers can be implemented using a variety of different ways. However the approaches differ significantly enough that feature sets and performance may not compare between them for any given set of data (routes + requests). This may be further exemplified by looking at the approach taken at searching for elements in lists and maps. However these primal data structures are not what is commonly used, instead the usage of tree graphs is preferred. Not because it is inherently better but instead it allows for skipping large sections or sub parts of a set of requests. Narrowing down the search for the specific request significantly more faster.

Most implementations only work with the path from the HTTP header. This allows them to be implemented using only regex or other more limited approaches such as using a map. This does cover most cases; by utilizing multiple instances it can be used for different HTTP methods such as GET and POST. However other rules such as rewriting are not possible and this is where tree graphs become quite useful.

Unlike the approach that regex typically takes, an extra data structure provided with the route function can be provided to allow for more complex operations such as rewriting or basing it off of other fields such as User-Agent or Referer. This can be used with lists, maps or tree graphs. Because of the extra information a router has access to and can query in its data store it has potentially many conditional arguments per search, without the possibility of rewriting of a request. This is a unique situation in computer science, the existing literature on the subject is thin as more commonly you are searching for a specific value instead of many potential points that need interpreting while searching.

TODO: something, ANYTHING on unconditional condition nature of data structures in CS, include about threading and how locking can affect this along with memory usage also cache locality of memory

TODO: ok this is a huge amount of text that is needed here

## Existing implementations

TODO: Apache 2 web server

TODO: Nginx

TODO: some comparison and referencing?

## Existing data structures

TODO: lists

TODO: maps

TODO: tree graphs and more?

# Time line

TODO: tasks and explanation

TODO: do it based upon the tasks that need to be completed and estimate how long each one will take

# 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 incounted.

# Outcomes

There is one outcome expected from this proposal and that is the dissertation. Supporting this will be code created to exemplify the different kinds of web routers that currently exist today.

TODO: the above is absolutely horrible but there needs to be something along those lines

TODO: there also needs to be something else here, what could that be?

## Dissertation

TODO: yes there is a dissertation with a methodology but what else can I do/need?

### Methodology

TODO: ok we got a methodology, but a) what is it b) why is it that c) what does that give us?

## Code

TODO: so an outcome is code, what does the code show and how are we doing it?

### Router types (known)

TODO: some router types are already known, so lets define them as part of the scope of the project

# References

TODO: do we need any more?

TODO: http 2.0 spec reference

TODO: asynchronous socket libraries

TODO: apache2 mod\_rewrite

TODO: fastcgi

TODO: PHP (webserver→PHP routing)

TODO: cache locality optimization strategies for data structures and general memory such as arrays