Decreasing Data Transfer on the Web with Templates

by

Daniel Robertson

BIT (Information Technology), Deakin University, 2013

Thesis submitted in partial fulfilment of
the requirements for the degree of
Bachelor of Information Technology (Honours)

in the
School of Information Technology
Faculty of Science, Engineering and Built Environment

Deakin University
November, 2013

FORM C

SCHOOL OF INFORMATION TECHNOLOGY

CANDIDATE DECLARATION: HONOURS

DECLARATION

I certify that the thesis entitled

Decreasing Data Transfer on the Web with Templates_____
(please print)

submitted for the degree of Bachelor of Information Technology (Honours) is the result of my own work and that where reference is made to the work of others, due acknowledgement is given. I also certify that any material in the thesis which has been accepted for a degree or diploma by any university of institution is identified in the text.

Daniel Robertson_____

Name of Candidate

_____ 14 / 11 / 2013

Signature of Candidate **Date**

Supervisor Approval

I certify that the thesis prepared by _____
entitled

_____

is prepared according to my/our expectations and that the honours coordinator can proceed to accept this submission for examination.

_____

Name(s) of ALL Supervisor(s)

_____ ___/___/___

Signature(s) of ALL Supervisor(s) **Date**

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Michael Hobbs, who has supported me throughout the work on my thesis. It was he who suggested that I undertake the Honours degree, so without his assistance and advice, this would never have been written.

I would like to thank the Honours coordinators, Jo Coldwell-Neilson, Sophie McKenzie, and Lei Pan, for organising and running the research training sessions and for your support during the year.

Finally, I would like to thank Nicholas DeBeen for his suggestions and an often-needed second opinion.

Abstract

Web content makes up a significant portion of network traffic on the Internet, with some sources pointing toward the percentage being greater than 50% (Gibson, Punera & Tomkins 2005) and increasing due to the rise of social media networks such as Facebook. Efforts to reduce this have spawned the creation of caching mechanisms such as local browser caches and Content Delivery Networks (CDNs) for some web objects, but these approaches can do little in addressing the problem of redundant HTML sent over the network for structuring content. It is proposed that by separating the content and structural elements of a web page, template structures can be defined which only requires a single instance to be sent to a client.

In this research, a new approach to calculating a web page's processing and rendering latency based on the amount of time it takes to process and render content on the screen is presented. In addition, a confirmation of previous results in studies (Benson et al. 2010; Garcia, FJ & Izquierdo 2012) of the effects template use has on web page size is conducted. Web browser processing and rendering latency along with the achievable web server throughput were also investigated. The results of this study show that the implementation of a template engine can reduce the size of a web page by approximately 85%, but only when the amount of content outweighs the inclusion of the necessary components for templates to be implemented. As a consequence of the reduced page size, web server throughput increased 615%. However, both tested template engines were unable to generate web pages faster than the existing traditional method.

Table of Contents

1 Introduction

Due to the nature of how web pages are constructed with HTML, it is very common for web pages to contain repeating element structures. Such pages require a web server to transmit the markup of each item within the overall HTML response across a network, even though each is likely to be very similar in its structure. The web page for a search engine result is a very good example of this. Using Google's search engine, for instance, to query a given search term returns a list of results, all of which have a very similar HTML structure within the overall returned HTML. The only difference between each result being the content each structure contains, such as the page title, URL, and summary information. The rest of the HTML used to structure a single result, which is approximately 61% of a single result's HTML, does not change. The structures for each result can therefore be seen as reusable templates to populate content with. However, as this process of combining templates with content is occurring on the server-side, each populated template must be included within the response to the client, unnecessarily increasing the size of the HTTP message, which previous research has shown to contribute to 40-50% of all traffic on the web (Gibson, Punera & Tomkins 2005).

1.1 Background

Previous studies by Tatsubori and Suzumura (2009), Benson et al. (2010), and Garcia, FJ and Izquierdo (2012) have attempted to address the problem of HTML redundancy with the creation of template engines – libraries of code which enable templates to be combined with content and injected into a web page. Each study concluded that the use of template implementations (including their own developed solutions) were able to substantially decrease the amount of data transferred when compared to the existing construction method. Two of these studies also conducted experiments highlighting client-side processing and rendering latency, and achievable web server throughput. However, the approach used by both of these studies to measure client processing is contentious, as they both use a methodology that underestimates when processing begins and overestimates where it ends. Furthermore, the Navigation Timing API which provides access to more precise measurements was not previously available for other studies to make use of in measuring browser performance.

1.2 Aims and Research Question

The contributions of this study are twofold. Firstly, to confirm the previous studies' results by showing that the use of a client-side template engine can decrease web page size and increase web server throughput. Secondly, to propose a new approach that more accurately measures client-side processing and rendering latency than methodologies used in previous studies (Benson et al. 2010; Garcia, FJ & Izquierdo 2012), and through its use, calculate the latency. The research question of this study is whether the use of templates can reduce the size of a web page.

1.3 Approach

In assessing the criteria outlined in Section 1.2 above, a mock Twitter timeline application was devised to implement the traditional and template construction methods. To vary the size of pages, a varying number of Tweets to be displayed were used. In determining whether the use of templates can reduce the size of a web page, a set of web pages implementing them and the traditional method were created, and their sizes, in bytes, were compared. To assess web server throughput, another set of web pages were created using the same implementation methods, with the omission of the template engine(s) to simulate client-side caching. The throughput of the web server was then tested by requesting each of the aforementioned web pages and the results compared. Lastly, in calculating client-side processing and rendering latency, another set of web pages were created using the same implementation methods. These web pages calculated and reported the amount of time spent by the browser processing and rendering the page, which were then compared with each other.

1.4 Structure of Thesis

The remaining sections of this paper are organised as follows: In Section 2, a number of approaches to reducing web content traffic are explored; Section 3 introduces Penguin, the template engine developed by the author; Section 4 discusses the methodology used; Section 5 details the experiments and results; and Section 6 concludes the study along with potential future research directions.

2 Literature Review

2.1 Introduction

This review covers four areas of web page construction and transfer, focusing on the production of redundant data and efforts in mitigating it. Firstly, the traditional construction method whereby web servers transmit a stream of HTML to a client and let the browser process and render the page (Figure 2.1). Secondly, technologies and methods used to cache web content, including local browser caches and Content Delivery Networks (CDNs) (Figure 2.2). The use of features introduced as part of the HTML5 specification, including localStorage, IndexedDB, and Web SQL Databases, and the role of JavaScript in Document Object Model (DOM) manipulation (Figure 2.3). Lastly, existing implementations for separating HTML structure from content in the form of template engines and their use (Figure 2.4). The scope of this review is limited to examining web pages and web content that is transmitted from web servers to clients in an uncompressed and non-encoded form.

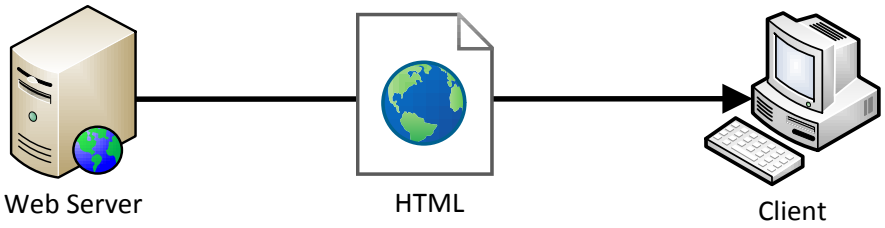

Figure 2.1 Traditional web page construction and transfer

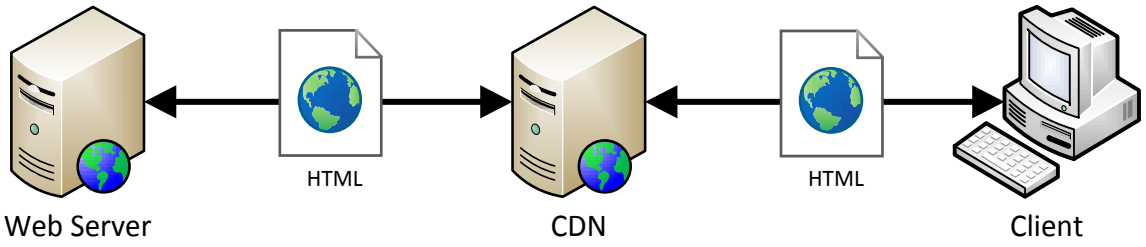

Figure 2.2 Caching technologies

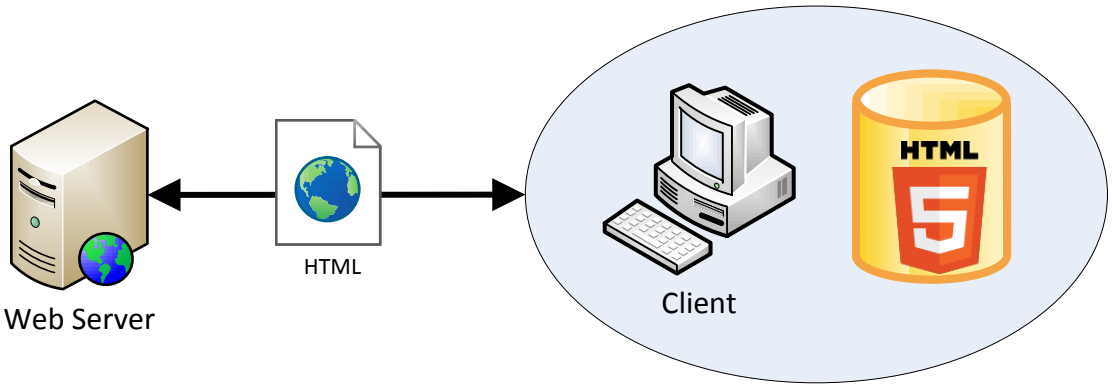

Figure 2.3 JavaScript and HTML5

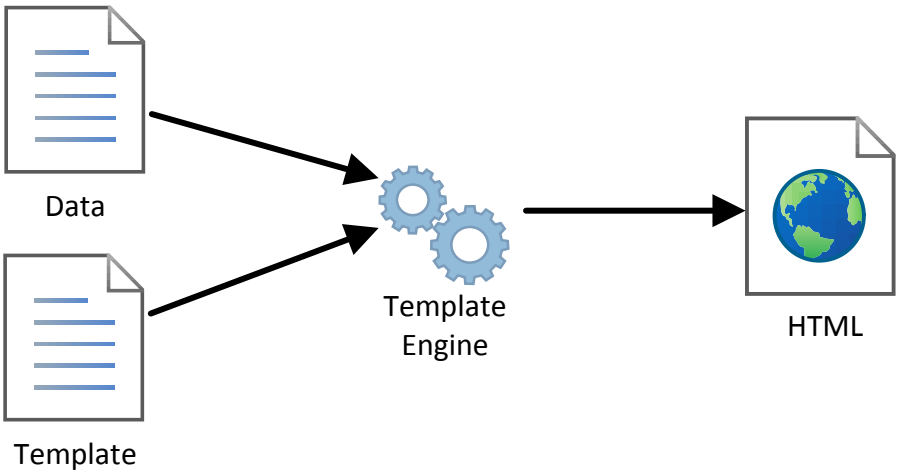

Figure 2.4 Existing template engines

2.2 Traditional Web Page Construction and Transfer

In the most simplistic approach to the generation of a web page, two tasks are involved. Firstly, a web server creates a string of data, places it within a HTTP response message, and sends it to the client where, secondly, it is handled by a web browser and rendered on the screen. Expanding on the first task mentioned, the creation of data more specifically refers to the combination of structural information (in this case, HTML) and content. For example, if a web server is tasked with handling search results, it might insert some text about each result into separate HTML paragraphs before sending them all to the requesting client. This has been referred to as the traditional approach when compared to others (such as the use of templates) in previous research (Al-Darwish 2003; Benson et al. 2010; Garcia, FJ & Izquierdo 2012; Tatsubori & Suzumura 2009) and is the focus of this section.

Looking at the specifics of how content can be combined with HTML, markup can quickly become very complex and repetitive depending on how it is used. Continuing with the example of a search engine, consider one that is implemented on a website with items for

sale and which returns a list of results for various matched items when queried, such as the search page on amazon.com. Each result might contain information such as an item's name, description, a URL to an image of what it looks like, its cost, and the quantity of the item in stock (Figure 2.5). To display each result in an elegant fashion, multiple structural elements such as divisions and lists might need to be used in addition to the elements which directly display the item's data, such as headings, paragraphs, and image elements. Extra elements for interactivity such as buttons might also be included as well. When the web server is generating these elements as HTML, it is likely traversing a collection of items and, for each of them, combining the data each item holds with a predefined set – or template – of HTML. At the end of traversing the collection, the server would have generated a string of HTML containing, among other markup, those now filled templates. While this is arguably the simplest method of generating such a page, a web server making this kind of response is including unnecessary data in the form of redundant HTML, consequently increasing the amount which must be transmitted over the network. This is because if the same process the web server uses to combine content with a template can be moved to the client-side, only a data source, such as an array, and a single, empty template would need to be sent to the client, as the latter can be reused and reapplied to the data source in the same way the server did (Benson et al. 2010).

```
<ul>
 …
 <li>
  <div class="productItem">
   <h2 class="productHeading">Product Name</h2>
   <img src="http://www.example.com/products/product3.jpg"
title="ProductName" alt="ProductTitle">
   <p class="productSummary">Product Summary</p>
   <p class="productSpecs">Product Specifications</p>
   <p class="productWarranty">Warranty Information</p>
   <div>
    Quantity: <input type="text" value="1">
    <br>
    <input type="button" onclick="addToCart('ProductId');"
value="Add to Cart">
   </div>
 </li>
 …
</ul>
```

Figure 2.5 Example HTML used to structure product information

Focusing more on the impact this has on the network, since the web makes use of the Internet to transfer web documents, it inherits the Internet's problems, especially in terms of performance of utilising a network to transmit data. For example, if the underlying transmission rate of the network is slow, the network itself becomes a bottleneck for a device (Zhang et al. 2010). Clearly this is a problem for web servers with constrained network capability if they wish to satisfy all incoming requests as efficiently as possible, but also for clients with limited capabilities if they want to quickly access web pages. A real-world example of this problem can be seen in the use of some applications, such as the popular blogging tool WordPress, which allows users to easily deploy a blog on a website (WordPress 2013a). These frameworks often contain a great deal of boilerplate HTML – markup which is part of a template used by the application to achieve a consistent structure among each web page. Websites using WordPress use these templates extensively, some of which are included on all web pages within the blog (WordPress 2013b). Unfortunately, websites which use such frameworks that contain templates greatly contribute to the amount of redundancy on the web, with approximately half all web content being attributed to the use of templates and increasing up to 8% per year (Gibson, Punera & Tomkins 2005). While other studies have shown the amount of redundancy in

HTTP traffic is as high as 32% (Anand et al. 2009). Compounding this problem, as the average web page size increases (Ihm & Pai 2011; Schulze & Mochalski 2009; Zhang et al. 2010), the impact this will have on network infrastructure will take its toll and, in fact, already is. According to Richardson (2010), several communications groups have expressed concerns about fibre-optic Internet backbones reaching their capacities in the near future due to increasing network traffic. This is especially true where HTTP is used as its use continues to rise each year (Erman et al. 2009; Schulze & Mochalski 2009).

2.3 Caching Technologies

Several studies (Datta et al. 2001; Erman et al. 2009; Gibson, Punera & Tomkins 2005; Schulze & Mochalski 2009; Spring & Wetherall 2000; Zhang et al. 2010) point toward the need for and successes of caching mechanisms for web content. In this section, the focus of the review turns to analysing caching techniques and their strengths and weaknesses.

Imagine that the developers of a website want to create a consistent theme throughout each of the pages they create which they do using Cascading Style Sheets (CSS). For each different web page they create, they replicate the same CSS into the page. However, by doing this they have inadvertently bloated the size of each of their pages. In addition to the maintainability issues, this has implications on the web server hosting them as well, since it is not only the page containing content being sent but the included CSS as well. This is an issue because it is unlikely that the CSS has been changed from when it was included in a previously requested page. Hence the reason for the existence of a local browser cache, which exists as a place to store frequently accessed web content such as images, scripts, and style sheets (Mozilla 2013c) to reduce the amount of network traffic (Nottingham 2012) . However, even though objects such as those mentioned can be cached, as well as whole web pages (Bower 2011), fragments of page's HTML cannot be stored in the browser cache (Mohan 2001; Padmanabhan & Qiu 2000) as there simply is no mechanism to do that. This is an issue as these fragments are analogous to the templates identified previously. If they cannot be cached locally, they cannot be reused by the browser.

There are also additional problems with browser caching, namely in the use and implementation of the HTTP caching functions themselves. For example, in Bent et al. (2004) it was shown that even when a website attempted to utilise caching features, the indiscriminate use of cookies invalidated any caching potential because of the uniqueness of a response a cookie value provides. Additionally, for a browser to avoid storing cached

content that is too old (referred to as the content that is stale) (Apache Software Foundation 2013b) it makes use of caching validation and validators (Nottingham 2012), but these still generate traffic (albeit not as much as an implementation with no caching mechanisms).

Another approach is to utilise Content Delivery Networks (CDNs) and proxy caches to store copies of web content as close as possible to clients in an effort to reduce the load on the origin server and latency to the client (Labovitz et al. 2010). While this is successful in reducing the amount of data the origin web server has to transfer, it still requires a client to download exactly the same amount of data as if it were stored in its original location. Merely moving the location of the server does not change the size of the content it is hosting or what the client must download.

2.4 JavaScript and HTML5

In this section, the focus of the review looks at the various storage methods that have become available through the HTML5 and WebStorage specifications as alternatives to browser caching discussed in Section 2.3, and how JavaScript can access them.

With the exception of proprietary browser extensions such as the now-deprecated Gears (Boodman 2011), previously, the only way to allow data in web applications to persists across browsing sessions and pages was with the use of cookies. As HTTP is a stateless protocol it does not retain information about the interactions between a client and server. Cookies are a method of providing this information. However, cookies are particularly limited in both the number of them that can be defined (Microsoft 2012) as well as how they inherently operate. That is, because cookies operate at the HTTP level, each and every applicable cookie will be included in every HTTP request for a given domain, even if they are not being used (Pilgrim 2011). The localStorage API, introduced within the Web Storage specification (Microsoft 2013), solves this by permitting arbitrary key-value pairs to be stored locally within the browser which persist indefinitely. What is particularly useful about the localStorage API is that it can be utilised from normal JavaScript as part of a web page. This is in stark contrast to other methods of storing data such as the browser caching mechanisms and cookies explained in Section 2.3. The HTML5 specification also introduces the Web SQL Database API and IndexedDB API as additional browser storage solutions, all of which can also be accessed via normal JavaScript, but allow for much larger amounts of data to be stored when compared to localStorage (Simms 2011).

What these APIs present to web developers are possibilities for storing data that was either not possible before or, in the case of cookies, was extremely inefficient. With JavaScript able to access them and perform arbitrary manipulations to a web page's Document Object Model (DOM) (Mozilla 2013d), caching and using HTML fragments is now a viable option.

2.5 Existing Template Engines

In Section 2.2, the concept of combing data with HTML was introduced with reference to the already established traditional method of generating web pages on the server-side, as was the premise behind wanting to move this process to the client-side. As noted in the previous section, JavaScript is able to modify a web page by changing the elements within the DOM. This functionality has allowed for the creation of JavaScript libraries which support the processing of templates within a browser – client-side template engines.

These engines are in abundance on the Internet, some of which include EJS, handlebars, mustache, underscore.js, and JavaScriptMVC (Basavaraj 2012). Most appear to advertise themselves as development tools for creating consistent web pages rather than an alternative processing architecture for pushing processing to clients. This is unfortunate as they do not provide any information on how well they perform when compared to the traditional method. However, in studies by Benson et al. (2010) and Garcia, FJ and Izquierdo (2012), two template engines were created, *Sync Kit* and *Yeast Templates*, and the effects of their respective engines explored. In the former, the implementation of Sync Kit resulted in a 94% reduction in the data transferred when compared to the traditional method, and the study involving Yeast Templates had a similar result of a 92% reduction.

Interestingly, some template engines may not be explicitly defined as an engine but still perform the same, or at least allow for, the same manipulations to occur. Facebook, for example, will retrieve comments for a post via a web service that returns JavaScript Object Notation (JSON) data. However, this does not contain any HTML, only information about posts such as ID values, timestamps, and posts' text. It can therefore be inferred that this data is used by other client-side JavaScript functionality to create and append the HTML elements which contain the aforementioned information on the page in the same way a template engine does. It is also worth noting that the implementation of an [implicit] template engine may help Facebook in scaling to meet the billions of requests it receives per day (Johnson 2010). This is because if the traditional approach creates larger sized web

pages than the use of templates, a web server is going to spend more time pushing the larger web page onto the network.

2.6 Summary

Previous research has shown that the traditional method of generating web pages with templates on the server-side leads to a large amount of redundant traffic on the web. While the desire to cache an instance of these templates as a fragment of a page has only become possible through the use of client-side data storage APIs accessible with JavaScript rather than with the browser's cache. That is to say, traditional caching mechanisms have not been able to facilitate caching HTML fragments. Template engines in the form of client-side JavaScript libraries have been created which are able to take fragments of HTML and combine them with a data source, resulting in element structures being created and injected into web pages and consequently an alternative method of constructing web pages established. Research by Benson et al. (2010) and Garcia, FJ and Izquierdo (2012) has shown the template construction method is extremely effective in increasing the throughput a web server can achieve and reducing the size of web pages when compared to the traditional method. However, as Benson et al. noted in their study, occasionally the size of the response from a web server is larger than the traditional approach due to the inclusion of the template engine.

3 Penguin

Penguin is this author's own implementation of a client-side template engine which was built with a number of key aims in mind:

- To [loosely] conform to the Model-View Controller (MVC) pattern;
- To help in development by providing explicit separation of each entity in the MVC pattern, specifically in how each relates to the HTML structure, logic (JavaScript), and content;
- To provide mechanisms for mitigating Cross-Site Scripting (XSS) attacks;
- To provide cross-browser compatibility;
- To follow the Object Oriented paradigm;
- To conform to HTML standards as closely as possible; and
- To ease in the development process of web applications.

Templates in Penguin are designed as a composition (Figure 3.1) of:

- A model, which defines a structure;
- A logical component, which is an optional description of how the model should be populated or otherwise modified; and
- Content, which is arbitrary input used by the logical component to populate the model with.

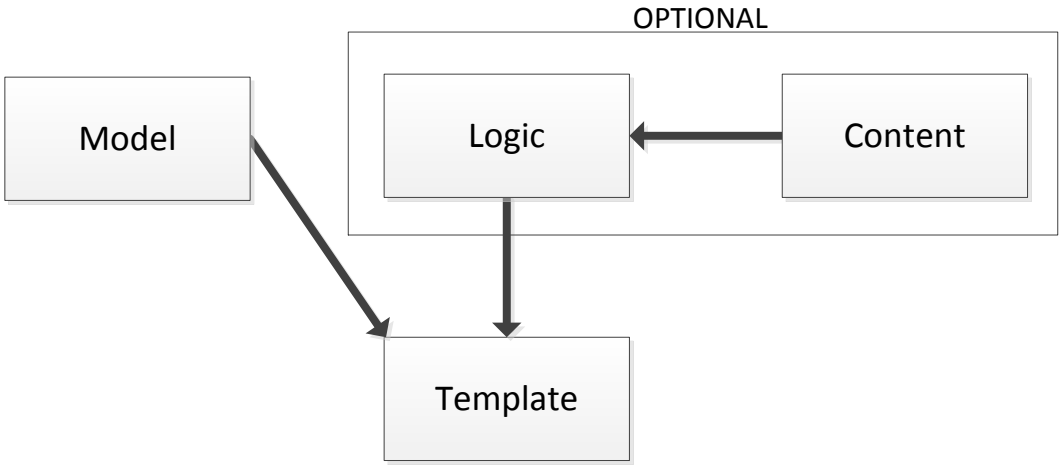

Figure 3.1 Penguin template design

As the logical component is an optional entity, a template can be created using only the structure contained within the model.

Physically, templates in Penguin are implemented using two elements: the model, which is a string of HTML; and an optional string containing the source code for a JavaScript function representing its logical component. This is in contrast to the template engines previously mentioned in Section 2.5 which mix template models with logical components (such as processing directives); Penguin explicitly keeps these separate.

Template models define the physical HTML structure of the template – they are a shell to be filled with content – and can define the structure for an individual component such as a search result or even an entire page. When the Penguin engine parses the string containing the model, it converts it into either a Document Fragment for individual elements or a Document object, depending on what the developer's aims are. Since these are both DOM objects, standard DOM manipulations can occur, such as appending a fragment directly into the current document or replacing it entirely. To support an interface into this structure, hooks into the HTML can be placed on elements as data-attributes under a customisable name. During the parsing process of templates, these attributes are optionally removed from the underlying DOM object, but references to the elements they are placed on are kept and become accessible in JavaScript under value of the attribute. Figure 3.2 below provides an example of template model.

```
<div>
 <img data-item-id="avatar">
 <span data-item-id="username"></span>
 <h2>About Me</h2>
 <p data-item-id="aboutMe"></p>
</div>
```

Figure 3.2 Example Penguin template model

Template logic defines how the model is to be populated with arbitrary data. When the logic portion of the template is parsed, the string containing the function code is converted into a native JavaScript function which is then bound to an instance of a Penguin template. This function then becomes callable from code making use of the template instance. Figure 3.3 below continues the example of the model given above in Figure 3.2 and demonstrates how data contained within the user object is bound to the "marked" elements of the model, which are accessible via the Refs object.

```
function(user){

 var f = this.GetTemplateFragment();

 f.Refs.avatar.setAttribute("src", user.AvatarURL);
 f.Refs.username.ReplaceWith(user.Username);
 f.Refs.aboutMe.textContent = user.AboutMe;

 return f.Fragment;

}
```

Figure 3.3 Example Penguin template logic showing data binding to elements within the template's Document Fragment and returning the fragment itself

Once a Penguin template has been instantiated using JavaScript, it can be used by either directly accessing the underlying DOM node to append it to the current document or by firstly binding data to it then appending it. Given the model and logic shown above in Figure 3.2 and Figure 3.3 respectively, Figure 3.4 below shows the result of processing this template by passing in an object containing three values: “http://www.example.com/image.jpg”, “johnsmith”, and “Hello, my name is John Smith”. Note how the ReplaceWith call removed the span element entirely and replaced it with the given text.

```
<div>
 <img src=”http://www.example.com/image.jpg”>
 johnsmith
 <h2>About Me</h2>
 <p>Hello, my name is John Smith.</p>
</div>
```

Figure 3.4 Example result of processing the model and logic shown in Figure 3.1 and Figure 3.2

Penguin templates also support the JSON format and include functionality to serialise a template object into a JSON-formatted string, or to take a JSON-formatted string containing template data and convert it back into an object. With this in mind, it becomes trivial to both cache the JSON string in the browser (via localStorage, for example) which can then be used later, and to output an entire template from a web service simply by serialising a server-side object into JSON format.

4 Methodology

4.1 Objectives

The objectives of this research are to observe the effects of altering the method of constructing web pages from the traditional method to the use of a template engine. Specifically, in addition to providing an answer to the research question – whether the implementation of templates can reduce the size of web pages – this research is concerned with measuring the effects each implementation has on the performance of a web browser and web server. The rationale being that even if page size is reduced, there are other factors which may be detrimental or advantageous to their use. For example, if a browser takes ten times longer to display a web page using templates than the traditional approach, justifying their use may be difficult because of the impact this has on the perceived loading time. For a web server, the direct effect of a change in the size of a web page is of interest, as the amount of data it needs to transmit will affect how quickly it is able to push it onto the network. Therefore, three criteria are defined:

- The amount of data needed to traverse the network to reach the client to be able to construct a web page;
- The amount of time spent by a browser processing and rendering data and elements up to a finished or loaded state; and
- The throughput that can be achieved by a web server in conjunction with client-side caching.

4.2 Requirements

4.2.1 Web Page Size

The total size of a web page is defined here as the sum, in bytes, of all HTML (including fragments) and all data used to populate the page. Importantly, this definition specifically excludes any HTTP related characteristics such as headers as well as any compression algorithms from the equation. That is, the measurement is the sum of the size of the uncompressed components of a generated web page and not of a HTTP message or any of the metadata associated with it.

4.2.2 Load Time

The loading time for a web page can only be calculated by a web browser, a device with one installed is needed as well as a location storing the necessary resources for the page. The browser needs to be able to report or calculate the amount of time it takes to parse the

required components of the page and render every element of constructed DOM on the screen.

4.2.3 Web Server Throughput

The throughput of the web server is determined by measuring the number of requests by a client that can be successfully satisfied by the web server per second. A web server is therefore needed to serve the resources for this assessment as well as a web client to initiate the requests for the test. The web client also needs to be able to make requests rapidly and calculate the rate of successful responses, so a benchmarking tool capable of this is also necessary. Needless to say, both the client and server also need to be connected via a network such as a Local Area Network (LAN).

4.3 Approach

4.3.1 Chosen Methodology

As the objective of this research is to observe the effects that have occurred after altering the construction method, a requirement should be to keep the resources used in the construction of each web page the same (or at least as similar as possible). This should be to ensure that any differences which do arise are in fact the result(s) of changing the construction method and not because of a difference in the quantity of data used on the page or due to unpredictable network functionality, for example (Key 1997). Thus, the tests performed will need to provide ways of controlling as many of the variables unrelated to each test as possible, such as those mentioned. For this reason, the experimental methodology is followed (Shuttleworth 2010); the construction method becomes the independent variable being altered, and the metrics identified in Section 4.2 above are those being measured as dependent variables. Furthermore, as Shuttleworth (2010) notes, the experimental methodology allows for control groups to be identified as baseline measurements for comparison. This is essential so as to be able to draw any conclusions about the differences between construction methods, such as whether implementing a template engine would produce a smaller web page size when compared with implementing the traditional method.

4.3.2 Web Page Size

One such possibility for determining a web page's size is to load the web page into a web browser, such as Firefox or Google Chrome, and use their inbuilt developer or inspection tools to examine details about the page. However, this approach is fundamentally flawed in

that the browser must be able to successfully load the web page in its entirety before being able to report the size. Furthermore, there is no guarantee that a browser will be able to load a web page containing a large amount of data before crashing, as some preliminary testing shows that Firefox fails to report the size of a web page via the page info dialog when the size increases to what appears to be some predefined threshold. An alternative approach is to examine the Content-Length header of the HTTP response message carrying the web page. This is a solution, however there is no guarantee it is the actual length of the HTTP message's body due to potential pre-processing by the web server prior to being pushed onto the network, interference from other server-side functionality, intermediary devices such as web proxies intercepting the request, or even the client itself modifying the response before being passed to the browser. This is because if the body of the message has been modified from its original size by one or more of the entities previously listed, the reported Content-Length may now be incorrect. It is also possible that a web server never outputs a Content-Length header (Rielding et al. 1999).

Software tools such as Fiddler (Telerik 2013) and Wireshark (Lamping, Sharpe & Warnicke 2013) which monitor traffic as they pass over a network can reveal information at relatively low levels. Wireshark and Fiddler, for example, can calculate the length of a HTTP response message, which is particularly useful as this does not rely on the Content-Length header. This is also an important distinction from the length of the entire HTTP message as this will include the headers as well, which may be considered both dynamic due to a web server's role in providing metadata in headers and largely irrelevant to the page generation process. However, one particular limitation of these tools is that the web page must be transferred over a network in some manner to be detected and allow inspection. Additionally, these tools may present sizes that may be ambiguous, as it may be unknown whether the sizes and lengths they report are those of the entire HTTP message or just that of the message body.

Lastly, if serving the web page is problematic in some way (because it may be too large to be handled by a web server, for example), none of the methods described above will work, in that they all require the client to request and then actually receive the page to either calculate or examine the size of the page. To overcome these problems, calculating the size of a web page on the server-side, irrespective of whether or not it can or will be served by the web server, is most desirable. Therefore, simply reading the file size of a web page is

the optimal solution in this case. This method reports the number of bytes the file containing the HTML of the web page has, is much faster and less cumbersome than loading web pages in browsers, does not need to involve third-party monitoring or sniffing tools, and eliminates any ambiguity caused by having it encapsulating in a HTTP response message.

4.3.3 Load Time

In a previous study of client-side template engines, the authors outlined their method of determining what they considered to be the "overall latency for a browser", which comprised of three separate tests using both client and server-side latency measurements (Garcia, F & Castanedo 2012). One of these tests involves measuring the load time of the document; by loading the web page from the local file system into a web browser and computing the time taken until the OnLoad event is called. From the perspective of eliminating any latency caused by the transfer of the page over the network, this is the correct approach as there simply is no latency to factor-in, and the focus of the test is solely on the browser's capabilities. There are, however, a number of issues with this approach.

Firstly, it appears as though the specific approach the authors have taken is to determine the start time with a JavaScript variable which is initialised at a point when the page data [from the file or network] is still being loaded into the browser and in the process of being parsed. This is problematic as it does not allow the browser the necessary time to read all file data, nor does it prevent the JavaScript variable containing the start time from holding a timestamp value that misrepresents when the browser began processing; it may hold a value which is later than when parsing actually began. The reason for this being that parsing must have already begun for the JavaScript variable initialisation on the page to have occurred. For both traditional and template construction methods, this previous point in time when the browser begins parsing is the actual start of processing. What may not be obvious though is why this is the case for template engines if they begin executing later on (since they are scripts). The answer is that the template engine code must still be parsed by the browser and injected into the document as a script so as to become available to run. Regardless, even with a known point in time now defined as the beginning, there is a still an issue with how to access it, as user code cannot run before it has been parsed. Fortunately, the most recent major browser versions provide this value through the PerformanceTiming interface (Mozilla 2013e) as part of the Navigation Timing API. This

exposes a number of different timestamps for various browser events that occur outside of what normal user code is able to access. For the purposes of determining the time taken to process a document, the domLoading property is ideal as a start time as it specifically defines the point in time from when a document was created and parsing began (Mozilla 2013f). Even in instances where data is still being read into the browser from a network or file, domLoading still represents the point in time when parsing began (W3C 2012).

Secondly, the authors have defined the end time for processing as the point where the OnLoad event is called by the browser, but this approach is also problematic as it too misrepresents when the browser has finished processing and rendering. A browser will invoke a bound OnLoad event handler when and only when all of the document's dependencies have completely finished loading (Mozilla 2013b). Unfortunately this also includes potentially external content, such as images, scripts, and style sheets. Therefore using the OnLoad event as the cut-off point for processing may produce an erroneous result due to the inclusion of wait times for loading extra content. This is also a specific problem in relation to the favicon.ico file, which is the small image provided by websites and used by browsers to identify a tab, window, or shortcut to the site. The favicon.ico file is an issue because may be included as a document dependency without specifying it in a web page's HTML, and because web browsers automatically request it (Heng 2008). However, this problem is easily solved by defining arbitrary points in time where processing ends based on the construction method used.

For the traditional construction method, the end of processing occurs when the document has been completely parsed, which can be determined using the DOMContentLoaded event (Mozilla 2013a). This is essentially the same as OnLoad, but does not wait for extra content to be loaded, which is the point of contention with using OnLoad. On the other hand, as template engines will inject elements into the document, the end of processing is simply defined as the point in time immediately after these actions have been completed, which will depend on how the engine operates. That is to say, a template engine may complete its processing routines at any point in a web page's lifetime after DOMContentLoaded has been called, as this is the earliest point in time one can begin modifying the DOM.

A comparison of the existing and proposed approaches discussed above is given below in Figure 4.1 and Figure 4.2, showing both approaches for the traditional construction method

and template method, respectively. Both figures represent the processing events after a browser has obtained a response, indicated as dashed vertical lines. The first event indicates when the browser creates the document and begins to process the response by creating the initial elements of the DOM, such as the head and body. The second event represents when the browser has completed parsing the response and invokes the DOMContentLoaded function. Lastly, the third event shows when any external dependencies have completely finished loading as previously mentioned above, and the OnLoad function is called. Dashed horizontal lines indicate variable points in time that cannot precisely be represented.

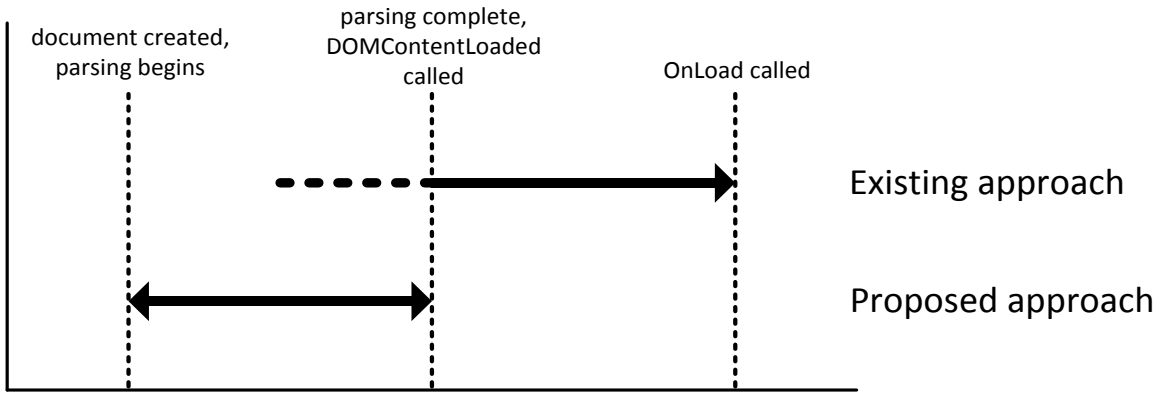

Figure 4.1 Approaches to calculating processing time for the traditional construction method

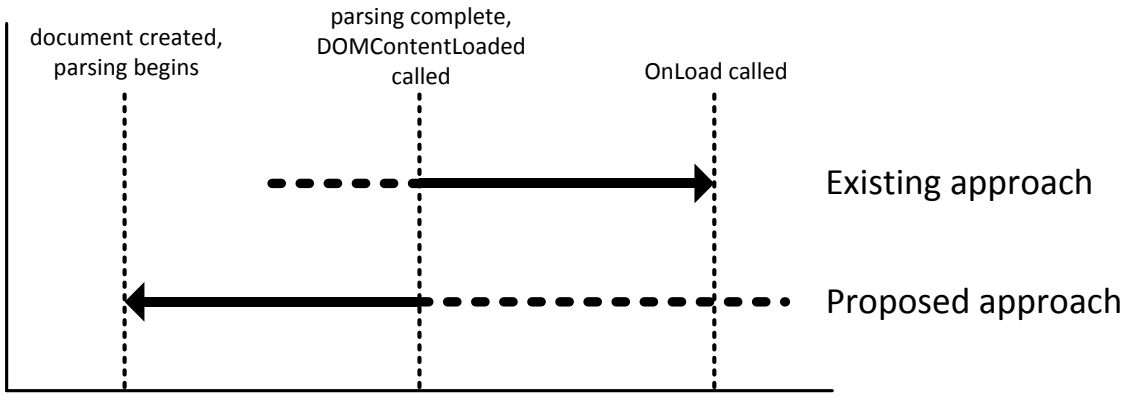

Figure 4.2 Approaches to calculating processing time for the template construction method

There is one more caveat to determining the load time, which is the time spent rendering the DOM on the screen and needs to be added to the time already spent processing. However, creating and appending elements to the DOM as part of the processing routine does not immediately render them on the page. To work around this problem, a timeout function can be used to wrap the end time calculation which allows any queued DOM manipulations to complete before proceeding (Koch 2013).

4.3.4 Web Server Throughput

The approach outlined in a previous study involving web server throughput is followed here, specifically, the use of JMeter as a benchmarking tool on a client computer to make requests (Garcia, F & Castanedo 2012). However, one limitation of JMeter is that it is incapable of parsing and rendering any responses as it is not a web browser (Apache Software Foundation 2013a), so an assessment of how efficient the application of caching template engines in a browser cache is not possible. Although a compromise can be made by assuming JMeter is a browser and using a script element referencing an external JavaScript containing the engine even though it does not exist. The inclusion of the script element is merely to include the information that would be present if this experiment were conducted using web browsers as clients and they were caching the engine. Thus, we can assume the local cache hit-rate for the engine is 100% for each request.

5 Experimental Results

In this section a comparison of the traditional and template construction methods is performed.

The following experiments take place within the context of a mock Twitter application displaying a timeline of Tweets. Each Tweet consists of multiple items of data, including: Tweet metadata, user information, displayable Tweet text, and timestamp information. This data set of 100,000 Tweets is randomly generated once and is shared amongst all experiments. That is, the first 10 Tweets, for example, are the first 10 in all tests; the data does not vary between them.

Within each experiment three implementations of each web page construction method are used:

- The traditional method of constructing a web page on the server-side, referred to as the PHP implementation (as the server-side environment uses PHP to accomplish this);
- Yeast, the implementation of the Yeast template engine; and
- Penguin, the implementation of the Penguin template engine.

To facilitate the generation of each timeline, a number of PHP scripts are used to create separate web pages where each contains the components necessary for the construction method to create equivalent pages. For the PHP implementation, the pages contain HTML already populated with Tweet data by the generation scripts themselves, and for both template implementations, each page contains the template engine itself embedded within a script element (except for testing web server throughput, as the engine is assumed to be cached in the browser), an array of JavaScript objects containing Tweet data, and the necessary template(s) to be applied to each element of the array. The respective sizes of the inserted template engines for Yeast and Penguin are 47,025 and 4,935 bytes. The generation process creates these web pages with varying numbers of Tweets in each of them and does so for each implementation. These experiments consider web pages with 1, 10, 40, 80, 100, 1000, 10,000, and 100,000 Tweets.

5.1 Experimental Environment

The testing environment consists of a LAN with multiple wired and wireless devices (Figure 5.1) with a selection of compatible major browsers installed (Table 1) to examine the influence of various devices and browsers:

- One PC with an instance of Apache 2.2 web server installed;
- Two PCs;
- Two Xbox game consoles;
- One laptop; and
- One mobile device (GS4).

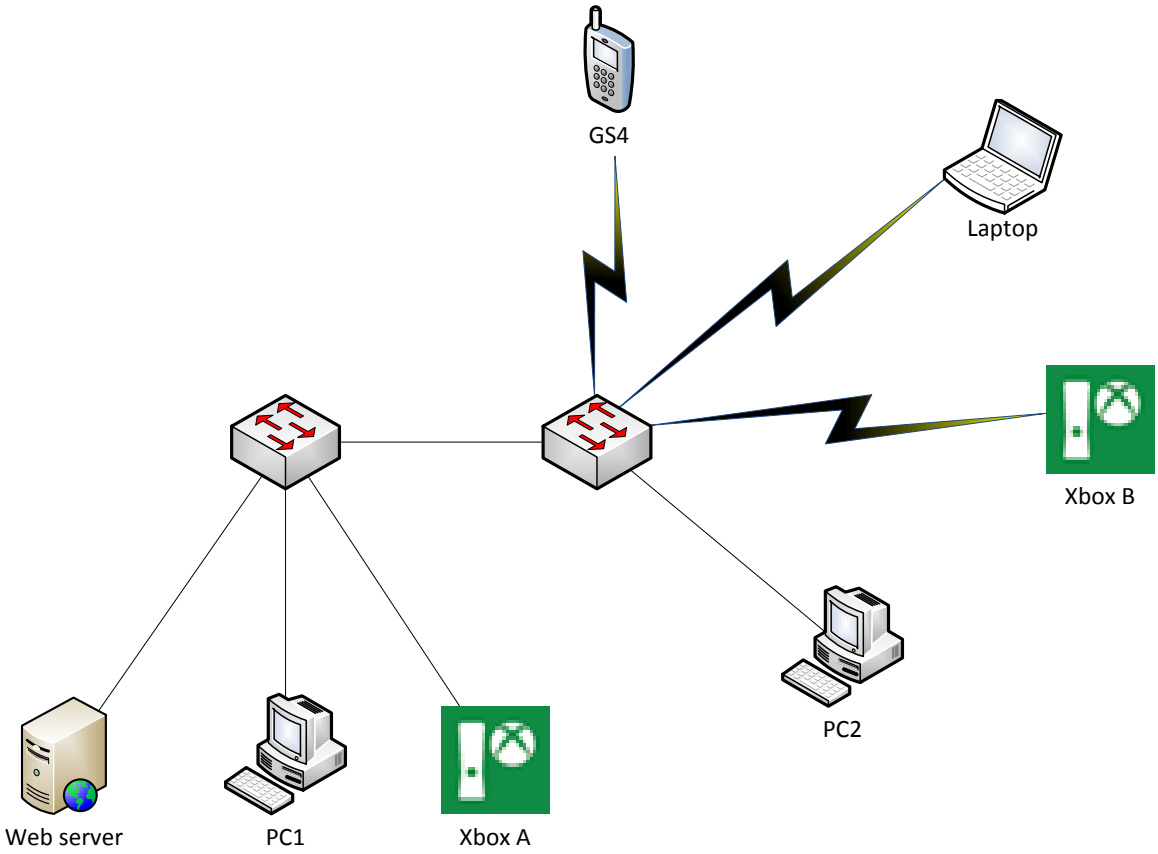

Figure 5.1 Devices used in the experiments

Table 1 Installed browser versions

Device	Internet Explorer	Firefox	Opera	Google Chrome
Xbox A	6.1.7601.17735			
PC1		23.0		28.0.1500.95
Web server	10.0.9200.16660	23.0	15.0.1147.153	28.0.1500.95 m
PC2	10.0.9200.16660	23.0	15.0.1147.153	28.0.1500.95 m
GS4		23.0	15.0.1162.61541	28.0.1500.94
Laptop	10.0.9200.16660	23.0	15.0.1147.153	28.0.1500.95
Xbox B	6.1.7601.17735			

5.2 Experimental Setup

Before beginning the experiments, the generation scripts were executed to create the shared Tweet data source file and each web page. Each web browser had any extensions and add-ons disabled and all non-essential programs and extraneous network functionality were terminated and disabled. All tests were performed independently to avoid interference with each other.

5.3 Experiment 1: Web Page Size

The web page size results were obtained by recording the Size property of each generated web page file via Windows' Properties window which was examined on the Web server alone. Table 2 below shows the raw file size of each generated page according to the number of Tweets the page contained and its implementation type. Each cell represents one web page.

Table 2 Generated web page sizes

		Raw File Size (MB)		
	Number of Tweets	**PHP**	**Yeast**	**Penguin**
Number of Tweets	**1**	0.17 MB	0.21 MB	0.17 MB
	10	0.19 MB	0.21 MB	0.18 MB
	40	0.25 MB	0.22 MB	0.19 MB
	80	0.34 MB	0.24 MB	0.20 MB
	100	0.39 MB	0.24 MB	0.21 MB
	1,000	2.39 MB	0.25 MB	0.50 MB
	10,000	22.44 MB	3.48 MB	3.44 MB
	100,000	222.95 MB	32.92 MB	32.88 MB

What is particularly interesting about these results is how much the size of the template engines affect the overall size when fewer Tweets are included. For example, the page with one Tweet with the PHP implementation had a size of only 0.17MB, but when Yeast is used, the size is around 40 KB greater, although the same cannot be said for Penguin as it is around 9.5 times smaller than Yeast. Also worth noting was how this difference between both engines' size persisted throughout each page; since they both used the same data source, the only difference was, again, the size of the engines themselves. Finally, when 100,000 Tweets was reached, both Yeast and Penguin were only using approximately 15% as much data as the PHP implementation.

5.4 Experiment 2: Load Time

Each device was used to manually load each generated web page for each implementation via the web server in each compatible browser until either a time was reported by the page via the alert box, an error occurred, or a maximum of one hour elapses from the beginning of the test. Times reported by the page were in milliseconds. In representing the end of processing, the PHP implementation defined a DOMContentLoaded callback wherein the end time is calculated, the Penguin implementation's time was calculated after processing occurred within a DOMContentLoaded callback, and the Yeast implementation used the onYSTProcessed function defined by the Yeast engine, which is called by the engine itself when it has finished. Figure 5.2 through Figure 5.8 below show the results of each implementation on each device, and, with the exception of both game consoles (as they only support Internet Explorer) and the mobile device, the results obtained using Firefox

are used to represent each figure as it was the only browser capable of loading a page with 100,000 Tweets without crashing.

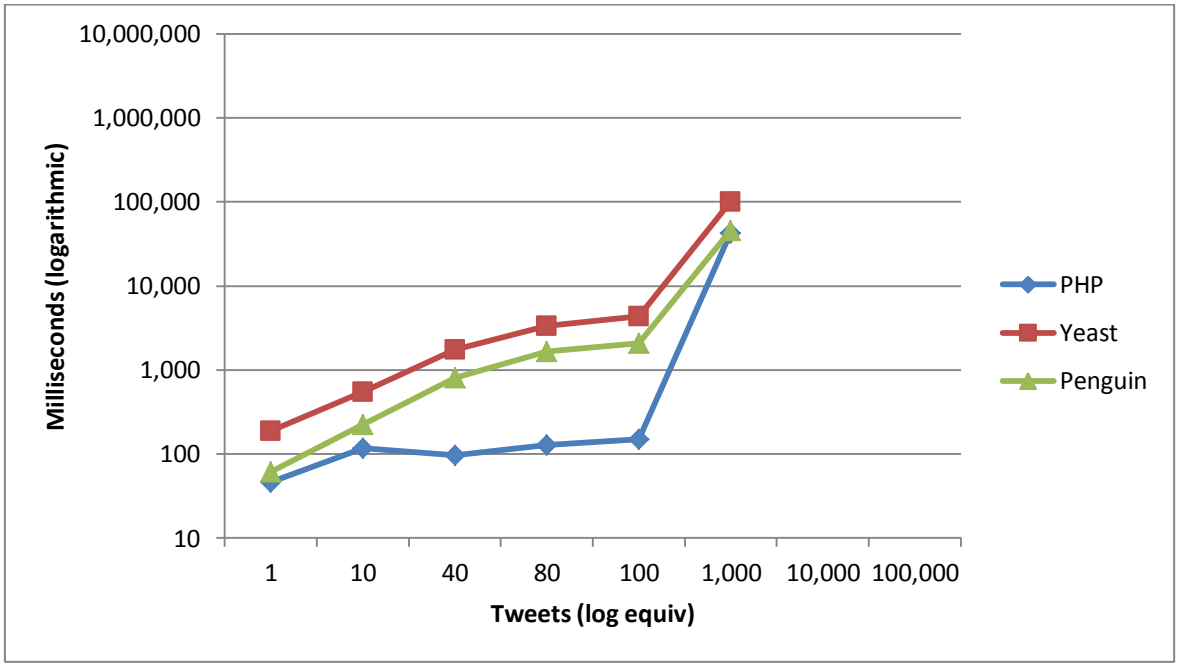

Figure 5.2: Timing results for Xbox A with Internet Explorer (wired)

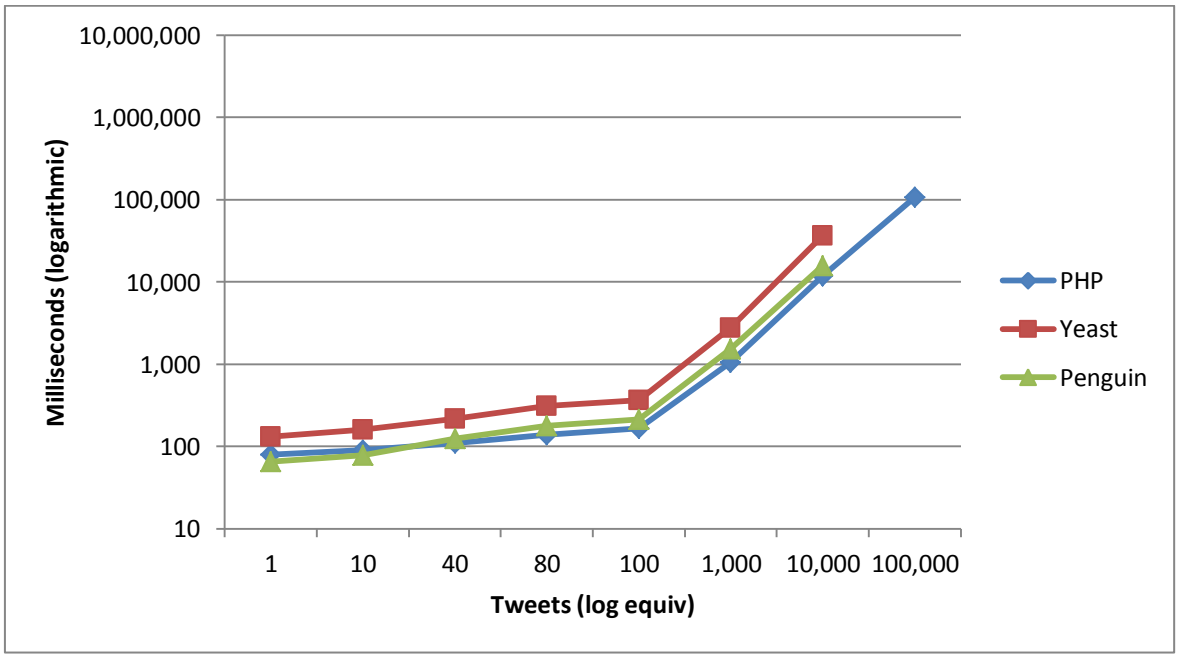

Figure 5.3: Timing results for PC1 with Firefox (wired)

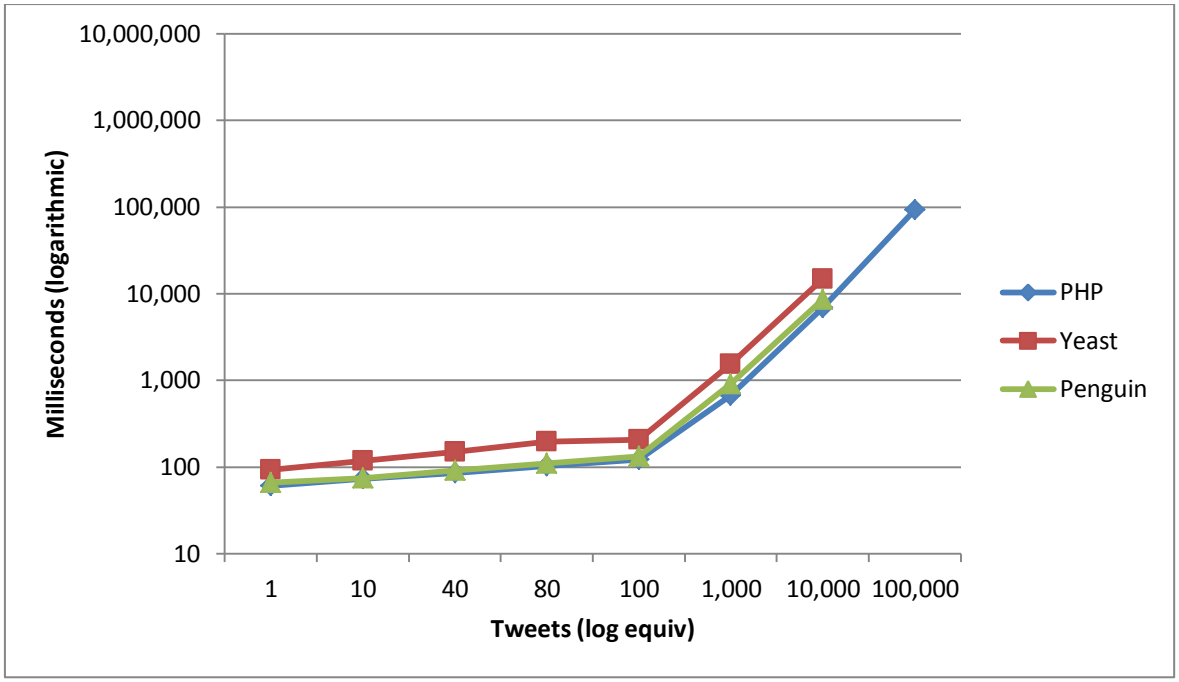

Figure 5.4: Timing results for the Web server with Firefox (wired)

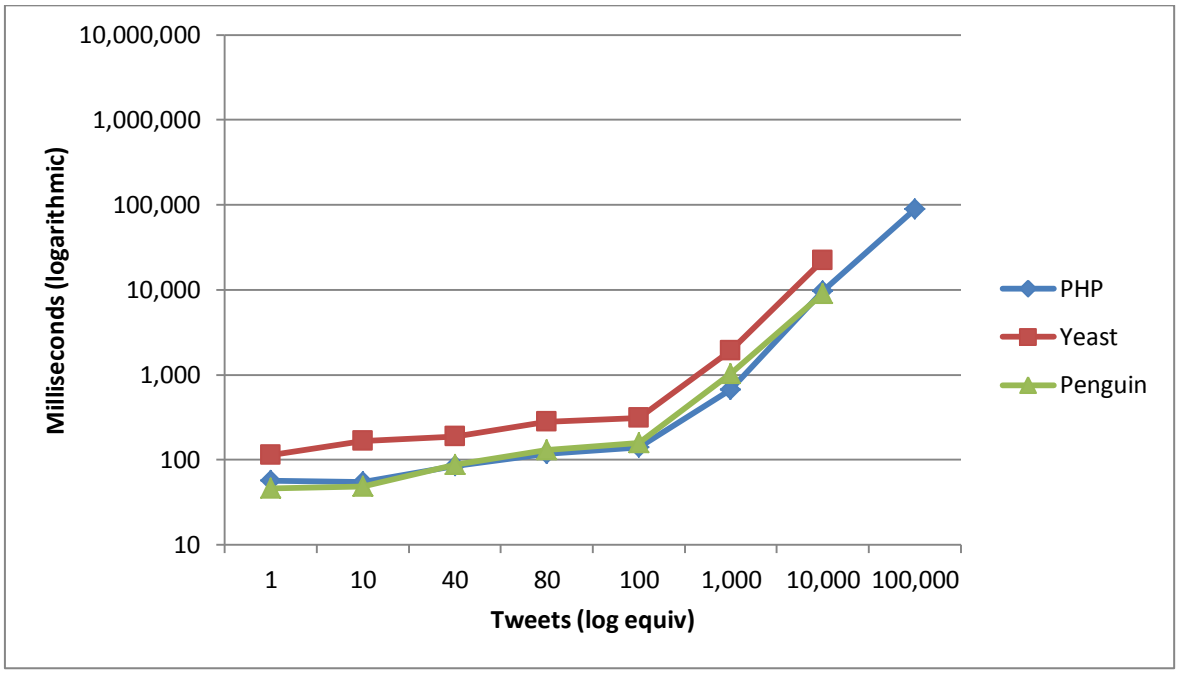

Figure 5.5: Timing results for PC2 with Firefox (wired)

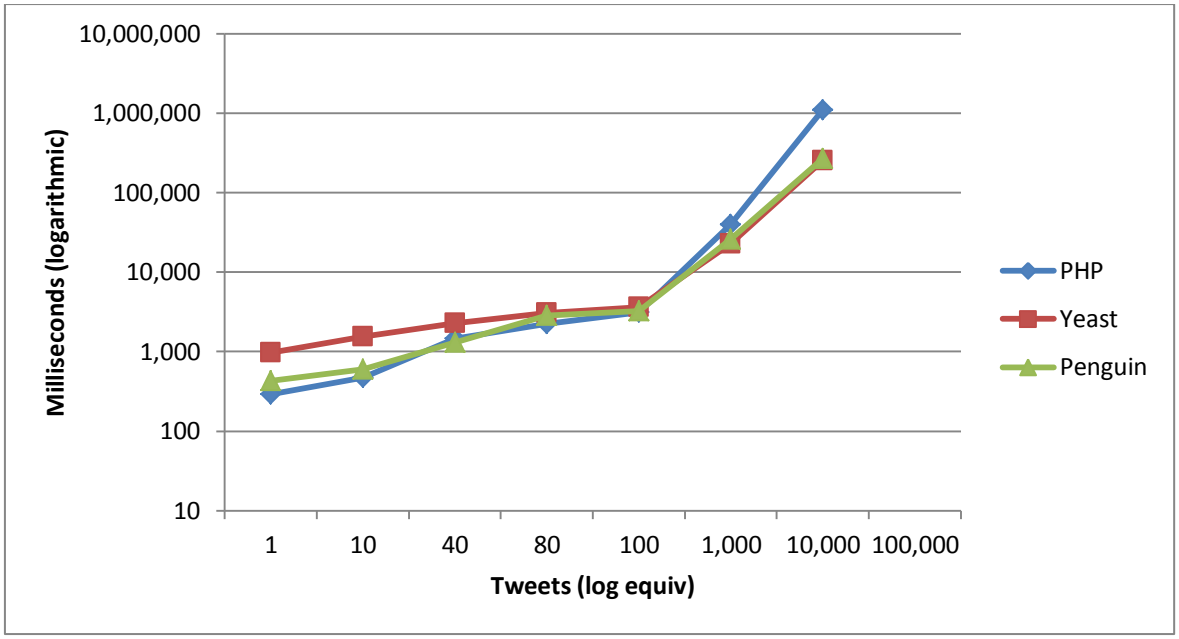

Figure 5.6: Timing results for GS4 with Firefox (wireless)

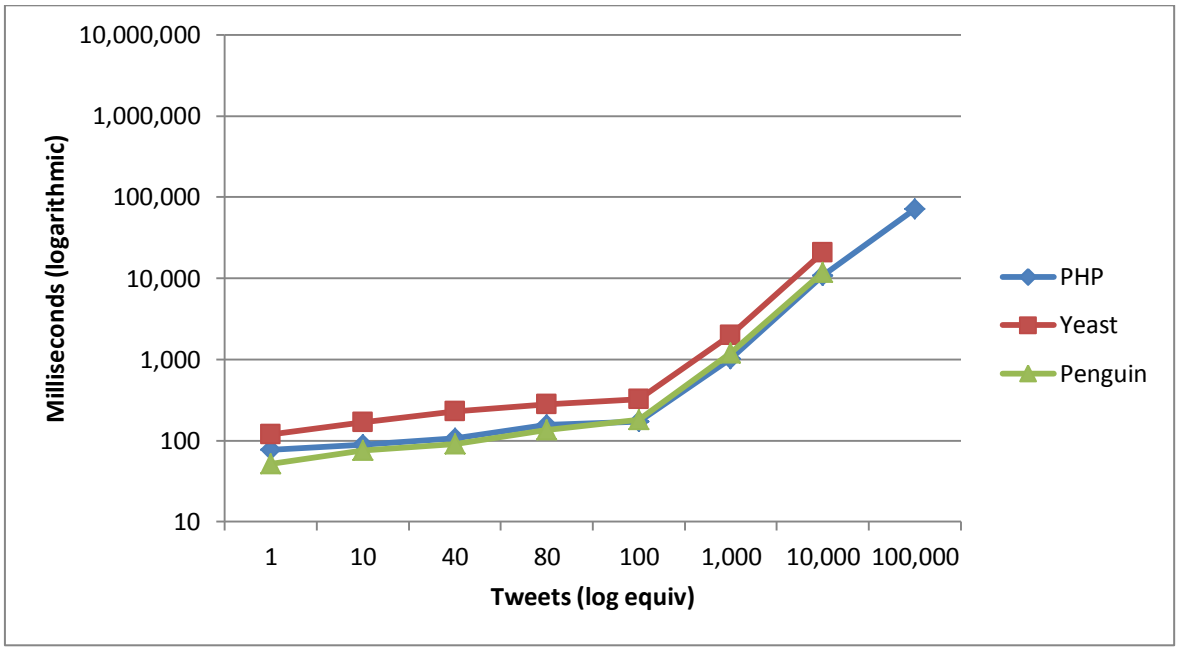

Figure 5.7: Timing results for the Laptop with Firefox (wireless)

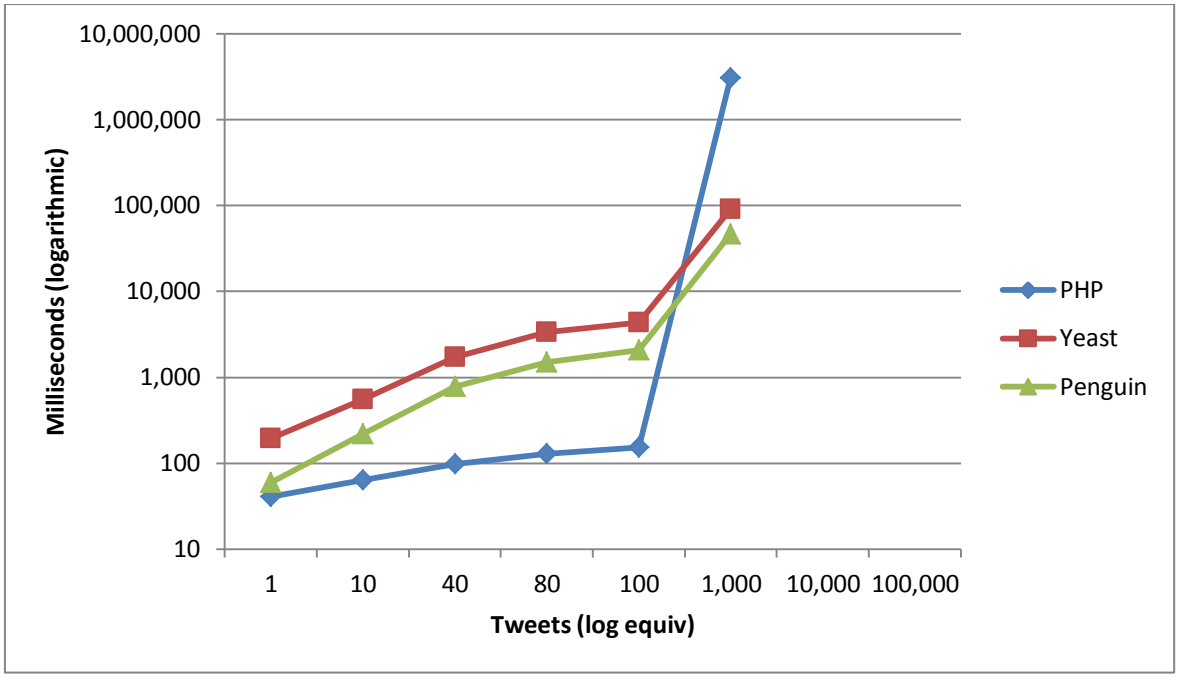

Figure 5.8: Timing results for Xbox B with Internet Explorer (wireless)

Timing tests of client-side processing latency show that under only two of the tested devices – both game consoles (Xbox A and Xbox B) – did the traditional method significantly outperform both template implementations for most tests. However, under tests of 1,000 Tweets, both devices' performance degraded, with one device (Xbox A) reaching similar times to that of the template implementations' results, while the other was significantly slower. The remaining devices exhibited very similar results concerning each implementation, but with two notable exceptions. Firstly, neither Penguin nor Yeast was able to successfully process pages with more than 10,000 Tweets, whereas the traditional method was able to handle 100,000 Tweets without crashing. Although the exceptions to this were the mobile device and both game consoles, which was likely the result of reduced hardware capabilities when compared to the other devices. Secondly, the Yeast implementation performed consistently worse across all devices when compared to Penguin. This was likely due to the way in which both engines work. The Penguin engine begins processing at the earliest possible time when DOMContentLoaded is called, whereas Yeast begins processing when OnLoad is called. This is a significant effect that may be exacerbated in applications where a large number of external resource requests are made or if they are long-running, which was discussed as part of this study's methodology of timing tests.

5.5 Experiment 3: Web Server Throughput

Apache JMeter 2.9 was used on PC2 to calculate the average throughput of accessing each web page via the web server for each implementation until the test completed. The parameters used were 20 threads, 60 loops, and a 2-second ramp-up period. For each test the achieved throughput reported by JMeter was recorded and is shown in Figure 5.9 below.

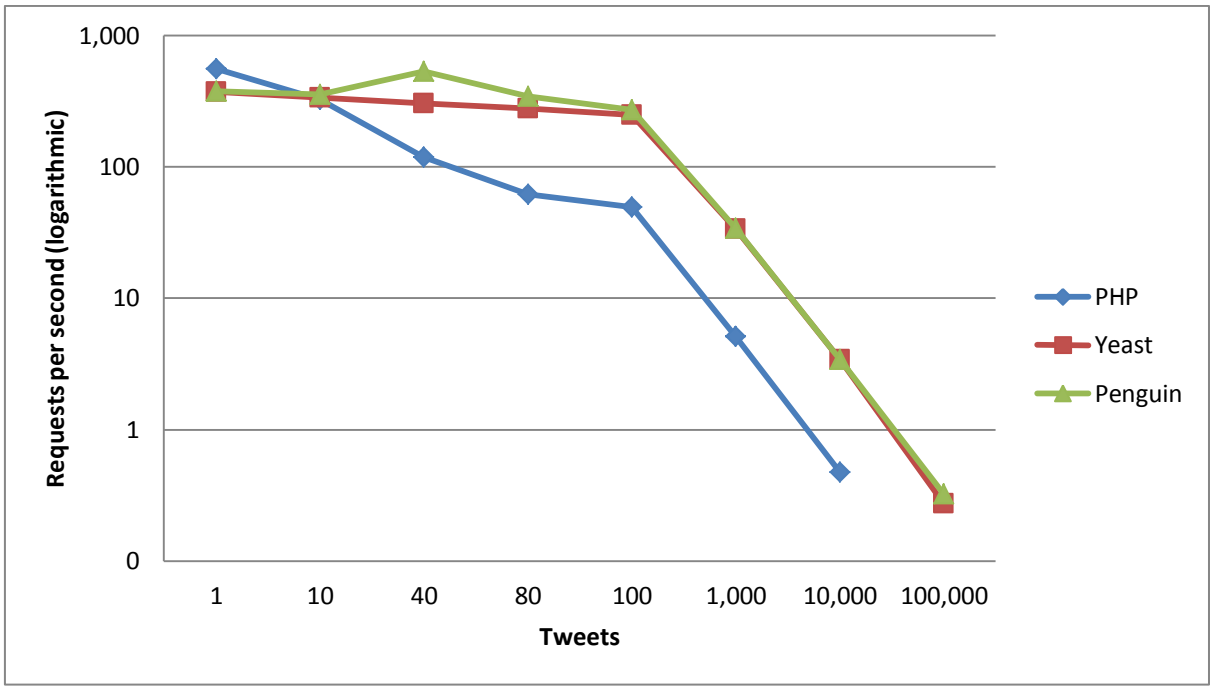

Figure 5.9: Achieved throughput via JMeter

The results appear to be highly correlated with the size of each web page, the reason being that if the web server does not need to spend as long pushing data out onto the network, it can more quickly satisfy a request. Consequently, the effect mentioned above in Section 5.3 regarding web page size can be seen here as well; the web page with only one Tweet on it using the traditional method was able to be served slightly faster than both template implementations' for the same single Tweet. However, the throughput quickly decreases after 10 Tweets when compared to the other two implementations. It also failed to return a result for pages with 100,000 Tweets, possibly because either the web server or JMeter began to fail as the amount of data increased beyond levels it was able to handle. As both the Penguin and Yeast implementations were using the same Tweet data source to construct pages, the only variable influencing the changes in size should be the number of Tweets embedded within each page. Figure 5.9 above shows the effect of web page size on throughput for both Yeast and Penguin implementations. However, the increase in

throughput for the page with the Penguin implementation and 40 Tweets may be an anomaly, as its size should be very similar to the size of the page with the Yeast implementation and 40 Tweets, which had a lower throughput.

6 Conclusion

Web pages can contain many repeating HTML structures, all of which are output by a web server as a normal part of constructing a web page. This has led to such pages contributing up to half of all data on the web (Gibson, Punera & Tomkins 2005). In this paper, an alternative methodology for determining web page processing and rendering latency was presented, and a confirmation of results in previous studies of the comparisons between two implementation methods (traditional and template) performed for web pages. The results of the experiments performed show that the implementation of a template engine can substantially increase a web server's throughput and reduce the size of a web page. However, the latter only appears to be true where the content to be combined with templates outweighs the overhead of including the necessary components for the template implementation to operate. In circumstances where only a small amount of content is to be output, it may be more effective to use the traditional approach. Assessments of browser processing and rendering latency show that the latency experienced by browsers for each implementation is comparable. However, care should be taken by developers of template engines to ensure they execute as early as possible in the browser and can still operate under heavy workloads and reduced platform capabilities.

6.1 Future Work

As one of the conclusions of this research indicated that it may be effective to use the traditional construction method where small amounts of data are to be output but templates for larger amounts, the development of a hybrid solution to handle both cases may be a worthwhile endeavour. Secondly, the results of this study show that template engines are not as stable as the traditional method when constructing web pages. It would be interesting to know whether this was caused by the template engines executing as normal JavaScript embedded in a web page as opposed to inbuilt browser code responsible for processing the traditional method. Lastly, current research on web templates has focused on implementing them within the existing architecture of web pages as JavaScript code, as has been done in this study as well. With research already established that formally defines template engines (Parr 2004), perhaps a new model of web pages can be proposed that integrates templates into them with a new HTML <template> tag, for example.

References

Al-Darwish, N 2003, 'PageGen: an effective scheme for dynamic generation of web pages', *Information and Software Technology*, vol. 45, no. 10, pp. 651-62.

Anand, A, Muthukrishnan, C, Akella, A & Ramjee, R 2009, 'Redundancy in network traffic: findings and implications', paper presented to Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems, Seattle, WA, USA.

Apache Software Foundation 2013a, *Apache JMeter*, retrieved 12th October 2013, <http://jmeter.apache.org/>.

Apache Software Foundation 2013b, *Caching Guide*, retrieved 3rd November 2013, <http://httpd.apache.org/docs/current/caching.html>.

Basavaraj, V 2012, *The client-side templating throwdown: mustache, handlebars, dust.js, and more*, retrieved 3rd November 2013, <http://engineering.linkedin.com/frontend/client-side-templating-throwdown-mustache-handlebars-dustjs-and-more>.

Benson, E, Marcus, A, Karger, D & Madden, S 2010, 'Sync kit: a persistent client-side database caching toolkit for data intensive websites', paper presented to Proceedings of the 19th international conference on World wide web, Raleigh, North Carolina, USA.

Bent, L, Rabinovich, M, Voelker, GM & Xiao, Z 2004, 'Characterization of a large web site population with implications for content delivery', paper presented to Proceedings of the 13th international conference on World Wide Web, New York, NY, USA.

Boodman, A 2011, *Stopping the Gears*, Google, retrieved 20th October 2013, <http://gearsblog.blogspot.com.au/2011/03/stopping-gears.html>.

Bower, D 2011, *What is Browser Cache and How Do I Clear / Reset My Browser's cache?*, retrieved 20th October 2013, <http://www.bowerwebsolutions.com/wp/2011/what-is-browser-cache/>.

Datta, A, Dutta, K, Thomas, H, VanderMeer, D, Ramamritham, K & Fishman, D 2001, 'A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration', in *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pp. 667-70.

Erman, J, Gerber, A, Hajiaghayi, MT, Pei, D & Spatscheck, O 2009, 'Network-aware forward caching', paper presented to Proceedings of the 18th international conference on World wide web, Madrid, Spain.

Garcia, F & Castanedo, R 2012, *Yeast-templates benchmarking*, retrieved 22nd February 2013, <http://yeasttemplates.org/bench/>.

Garcia, FJ & Izquierdo, R 2012, 'Is the Browser the Side for Templating?', *Internet Computing, IEEE*, vol. 16, no. 1, pp. 61-8.

Gibson, D, Punera, K & Tomkins, A 2005, 'The volume and evolution of web page templates', paper presented to Special interest tracks and posters of the 14th international conference on World Wide Web, Chiba, Japan.

Heng, C 2008, *What is Favicon.ico and How to Create a Favicon Icon for Your Website*, retrieved 11th October 2013, <http://www.thesitewizard.com/archive/favicon.shtml>.

Ihm, S & Pai, VS 2011, 'Towards understanding modern web traffic', paper presented to Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference, Berlin, Germany.

Johnson, R 2010, *Scaling Facebook to 500 Million Users and Beyond*, retrieved 20th October, 2013 2013, <http://www.facebook.com/note.php?note_id=409881258919>.

Key, J 1997, *Experimental Research and Design*, retrieved 10th October 2013, <http://www.okstate.edu/ag/agedcm4h/academic/aged5980a/5980/newpage2.htm>.

Koch, P-P 2013, *Reading out the end time in browser speed tests*, retrieved 11th October 2013, <http://www.quirksmode.org/blog/archives/2009/08/when_to_read_ou.html>.

Labovitz, C, Iekel-Johnson, S, McPherson, D, Oberheide, J & Jahanian, F 2010, 'Internet inter-domain traffic', *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 75-86.

Lamping, U, Sharpe, R & Warnicke, E 2013, *Wireshark User's Guide*, retrieved 3rd November 2013, <http://www.wireshark.org/docs/wsug_html_chunked/>.

Microsoft 2012, *Number and size limits of a cookie in Internet Explorer*, retrieved 3rd November 2013, <http://support.microsoft.com/kb/306070>.

Microsoft 2013, *Saving files locally using Web Storage*, retrieved 20th October 2013, <http://msdn.microsoft.com/en-us/library/ie/hh580308(v=vs.85).aspx>.

Mohan, C 2001, 'Caching technologies for web applications', in *Proceedings of the 27th International Conference on Very Large Data Bases*, p. 726.

Mozilla 2013a, *DOMContentLoaded*, retrieved 11th October 2013, <https://developer.mozilla.org/en-US/docs/Web/Reference/Events/DOMContentLoaded>.

Mozilla 2013b, *GlobalEventHandlers.onload*, retrieved 11th October 2013, <https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers.onload>.

Mozilla 2013c, *How to clear the Firefox cache*, retrieved 20th October 2013, <https://support.mozilla.org/en-US/kb/how-clear-firefox-cache>.

Mozilla 2013d, *JavaScript technologies overview*, retrieved 20th October 2013, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview>.

Mozilla 2013e, *PerformanceTiming*, retrieved 3rd November 2013, <https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming>.

Mozilla 2013f, *PerformanceTiming.domLoading*, retrieved 11th October 2013, <https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming.domLoading>.

Nottingham, M 2012, *Caching Tutorial*, retrieved 23rd March 2013, <http://www.mnot.net/cache_docs/>.

Padmanabhan, VN & Qiu, L 2000, 'The content and access dynamics of a busy Web site: findings and implications', paper presented to Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Stockholm, Sweden.

Parr, TJ 2004, 'Enforcing strict model-view separation in template engines', paper presented to Proceedings of the 13th international conference on World Wide Web, New York, NY, USA.

Pilgrim, M 2011, *The Past, Present & Future of Local Storage for Web Applications*, retrieved 20th October 2013, <http://diveintohtml5.info/storage.html>.

Richardson, DJ 2010, 'Filling the light pipe', *Science*, vol. 330, no. 6002, pp. 327-8.

Rielding, R, Irvine, U, Gettys, J, Mogul, J, Compaq, Frystyk, H, Masinter, L, Leach, P & Berners-Lee, T 1999, *Hypertext Transport Protocol -- HTTP/1.1*, IETF, <http://www.ietf.org/rfc/rfc2616.txt>.

Schulze, H & Mochalski, K 2009, 'Internet Study 2008/2009', *IPOQUE Report*.

Shuttleworth, M 2010, *Scientific Control Group*, retrieved 10th October 2013, <http://explorable.com/scientific-control-group>.

Simms, C 2011, *HTML5 Storage Wars - localStorage vs. IndexedDB vs. Web SQL*, retrieved 20th October 2013, <http://csimms.botonomy.com/2011/05/html5-storage-wars-localstorage-vs-indexeddb-vs-web-sql.html>.

Spring, NT & Wetherall, D 2000, 'A protocol-independent technique for eliminating redundant network traffic', paper presented to Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Stockholm, Sweden.

Tatsubori, M & Suzumura, T 2009, 'HTML templates that fly: a template engine approach to automated offloading from server to client', paper presented to Proceedings of the 18th international conference on World wide web, Madrid, Spain.

Telerik 2013, *HTTP/HTTPS traffic recording*, retrieved 3rd November 2013, <http://fiddler2.com/Features/http-https-traffic-recording>.

W3C 2012, *Navigation Timing*, retrieved 11th October 2013, <http://www.w3.org/TR/2012/REC-navigation-timing-20121217/#processing-model>.

WordPress 2013a, *Get Started*, retrieved 15th June 2013, <http://en.support.wordpress.com/get-started/>.

WordPress 2013b, *Template Hierarchy*, retrieved 15th June 2013, <http://codex.wordpress.org/Template_Hierarchy>.

Zhang, K, Wang, L, Pan, A & Zhu, BB 2010, 'Smart caching for web browsers', paper presented to Proceedings of the 19th international conference on World wide web, Raleigh, North Carolina, USA.