Web Application Performance Testing

UNIVERSITY OF TURKU
Dept. of Information Technology
Jukka Palomäki
Master's thesis
2009

UNIVERSITY OF TURKU
Department of Information Technology / Faculty of Mathematics and Natural Sciences

JUKKA PALOMÄKI: Web Application Performance Testing

Master's thesis, 53 pages
Software Engineering
December 2009

Web application performance testing is an emerging and important field of software engineering. As web applications become more commonplace and complex, the need for performance testing will only increase.

This paper discusses common concepts, practices and tools that lie at the heart of web application performance testing. A pragmatic, hands-on approach is assumed where applicable; real-life examples of test tooling, execution and analysis are presented right next to the underpinning theory.

At the client-side, web application performance is primarily driven by the amount of data transmitted over the wire. At the server-side, selection of programming language and platform, implementation complexity and configuration are the primary contributors to web application performance.

Web application performance testing is an activity that requires delicate coordination between project stakeholders, developers, system administrators and testers in order to produce reliable and useful results. Proper test definition, execution, reporting and repeatable test results are of utmost importance.

Open-source performance analysis tools such as Apache JMeter, Firebug and YSlow can be used to realise effective web application performance tests. A sample case study using these tools is presented in this paper. The sample application was found to perform poorly even under the moderate load incurred by the sample tests.

Keywords: Web application, performance testing, HTTP, JMeter, Firebug, YSlow

TURUN YLIOPISTO
Informaatioteknologian laitos / Matemaattis-luonnontieteellinen tiedekunta

JUKKA PALOMÄKI: Web Application Performance Testing

Diplomityö, 53 sivua
Ohjelmistotekniikka
Joulukuu 2009

Suorituskykytestaus on tärkeä osa nykyaikaista ohjelmistotuotantoa. Web-sovellusten määrän ja monimutkaisuuden alati lisääntyessä, niiden suorituskyvyn testauksen ja validoinnin merkitys kasvaa varmasti. Tässä tutkielmassa tarkastellaan web-sovellusten suorituskykytestaukseen liittyviä käsitteitä ja käytäntöjä teoriassa ja esimerkkien avulla.

Asiakkaan puolella sovelluksen suorituskykyyn vaikuttaa eniten verkon yli siirrettyjen resurssien määrä ja koko. Palvelinpuolella suorituskykyyn vaikuttavat erityisesti ohjelmointialusta, sovelluksen toteutuksen monimutkaisuus ja konfiguraatio.

Tuottaakseen luotettavia ja hyödyllisiä tuloksia, web-sovelluksen suorituskykytestaus vaatii erityisen paljon koordinointia projektin eri osapuolten (projektin vetäjä, kehittäjät, testaajat ja ylläpito) kesken. Testien määrittely, suunnittelu, toteutus ja testitulosten toistettavuus ovat erityisen tärkeitä asioita testauksen onnistumisen kannalta.

Vapaan lähdekoodin, suorituskyvyn analysointiin tarkoitetut ohjelmat kuten Apache JMeter, Firebug ja Yslow mahdollistavat tehokkaiden suorituskykytestien toteutuksen. Tutkielmassa esitetään edellämainittujen työkalujen avulla yksinkertainen suorituskykytesti. Esimerkkisovelluksen suorituskyky oli testien perusteella huono.

Asiasanat: Web application, performance testing, HTTP, JMeter, Firebug, YSlow

ACKNOWLEDGEMENTS

Special thanks to Timo Knuutila (University of Turku), Tuomas Mäkilä (University of Turku) and Teemu Tasanto (ATR Soft Oy) for your patience, support and critical feedback during the long incubation of this work. I would also like to thank Eija Karsten, Sinikka Järvinen and Riikka Vuokko for providing the document template and verbal guidelines upon which to build my thesis.

Turku, December 2009

Jukka Palomäki

TABLE OF CONTENTS

1 INTRODUCTION ... 2
1.1 Purpose ... 2
1.2 Web applications ... 3
1.2.1 Client-server model ... 3
1.2.2 The HTTP protocol ... 4
1.2.3 Server-side implementation ... 6
1.2.4 Client-side implementation ... 8
1.3 Performance testing ... 10
1.3.1 Rationale ... 10
1.3.2 Test types ... 11
1.3.3 Implementation ... 12
2 WEB APPLICATION PERFORMANCE ... 13
2.1 Frontend performance factors ... 13
2.1.1 HTTP request count ... 13
2.1.2 Caching of resources ... 14
2.1.3 DNS lookups ... 14
2.1.4 Redirects ... 15
2.1.5 Compression ... 15
2.1.6 Style sheets ... 15
2.1.7 JavaScript ... 16
2.2 Backend performance factors ... 16
2.2.1 Platform ... 17
2.2.2 Implementation ... 17
2.2.3 Configuration ... 18
2.3 Remarks ... 18
3 WEB APPLICATION PERFORMANCE TESTING ... 20
3.1 Preparing for performance tests ... 20
3.1.1 Defining acceptance criteria ... 20
3.1.2 Designing the test scenarios ... 22
3.1.3 Building the test suite ... 24
3.2 Performance test execution ... 26
3.2.1 Validating the test suite ... 26
3.2.2 Creating a baseline ... 27
3.2.3 Benchmarking with multiple test runs ... 27
3.2.4 Reasons for performance test failure ... 28
3.2.5 Monitoring tests ... 29
3.3 Analysis and reporting of test results ... 29
3.3.1 Test data collection ... 29
3.3.2 Test data analysis ... 30
3.3.3 Reporting results ... 31
4 CASE STUDY: WORDNET ... 33
4.1 Setup ... 33
4.2 Tooling ... 34
4.2.1 JMeter ... 34
4.2.2 Firebug ... 35
4.2.3 YSlow ... 36
4.3 Sample frontend analysis ... 36
4.4 Sample backend load test ... 40
4.4.1 The test scenario ... 40
4.4.2 Recording the test script ... 40
4.4.3 Executing the test script ... 44
4.4.4 Analysing the test data ... 48
4.5 Remarks ... 49
5 CONCLUSIONS ... 50
BIBLIOGRAPHY ... 52

1 INTRODUCTION

1.1 Purpose

The purpose of this work is to discuss modern web application performance testing from a theoretical and a practical standpoint, with an emphasis on the latter.

The reader is first presented with a comprehensive overview of web applications and performance testing in general. Chapter 2 considers factors affecting web application performance. Chapter 3 discusses web application performance test design and implementation. Chapter 4 begins by introducing three commonly used performance testing and analysis tools (Apache JMeter [10], Firebug [11] and YSlow [12]), each of which is subsequently used in a sample case study that provides a hands-on perspective to this paper. The case study walks through typical performance testing tasks and provides some insight into common bottlenecks in web application performance via examples. Finally, all key findings are presented as conclusions.

The reader is expected to possess basic knowledge in the fields of computing, software engineering and web technology. Among other things, this means that the reader should have some knowledge of markup languages such as the *Hypertext Markup Language* (HTML) and *Extensible Markup Language* (XML), client-side browser scripting languages such as *JavaScript*, as well as *Cascading Style Sheets* (CSS). Previous knowledge of the *Hypertext Transfer Protocol* (HTTP) will surely prove beneficial.

Specific details regarding web server internals and configuration, web browser internals and support, application server technologies and computer networking are omitted from this paper. Furthermore, *Rich Internet Application* (RIA) technologies such as *Ajax* (Asynchronous JavaScript and XML) are only discussed in limited detail where applicable. This approach enables us to place an exclusive focus on the intended topic, web application performance testing.

1.2 Web applications

Web applications are *platform-independent*[1] software applications that are run on a web server and/or application server, with the user interface rendered by the client's web browser, and communication taking place over a computer network.

The application architecture that powers traditional web applications is called the *client-server* model. In this model, the client sends *requests* to a server, which in turn processes the requests and provides *responses*. This is called the request-response cycle and it lays the foundation for web applications and performance testing thereof. Most web applications utilize the aging HTTP protocol to achieve this type of communication. The following subsections explore these concepts in more detail.

1.2.1 Client-server model

A client-server application is a distributed system in which an application server processes requests from (multiple) clients in order to provide a service to those clients. There is a clear separation between the client and the server, and they are often run on separate machines (though they may also reside on the same machine), with communication between the two taking place over a computer network, such as a *LAN* (Local Area Network) or the *Internet*. [8]

Application state is persisted at the server-side, with the client only storing necessary tokens (e.g. browser *cookies*) that are used to distinguish clients from one another and transient data that is manipulated in order to provide input to the server. The server also manages application logic (excluding any logic embedded in the user interface, e.g. client-side validation), as well as interfaces to external systems (such as databases) that are often necessary for an application's operation.

The client's task is to provide input so as to change the state of the system. The client accomplishes this by composing input via the user interface and dispatching requests

[1] A web application may be accessed from any platform with a suitable web browser.

that contain the necessary input, to the server. The server in turn processes the requests, makes necessary and appropriate modifications to system state based on the input, and provides a response that describes these changes. The client then updates the user interface based on the response, to allow for the user to visualize the changes and potentially provide more input via subsequent requests.

While this type of communication may appear as *stateless* (“do this, do that”), consecutive requests are often logically inter-connected. Hence a mechanism for maintaining a context for the client is necessary. The so-called *session* serves this purpose. Sessions are often implemented by attaching tokens (such as textual session keys), that identify the client, to requests. This allows for the server to identify the source of the request and provide a *stateful* service that remembers what the client did on previous requests, while processing the next.

1.2.2 The HTTP protocol

The HTTP protocol is a stateless application-level protocol that powers the web. It is developed by the *World Wide Web Consortium* (W3C) and the *Internet Engineering Task Force* (IETF). The current version HTTP/1.1 was made publicly available in 1999. [4] As the protocol defines a request-response standard for client-server applications, it is best described in terms of the request-response cycle.

In order to dispatch an HTTP request, a client first establishes a network connection with the server, commonly a *TCP* (Transmission Control Protocol) connection on *port* 80, though any other reliable transport-level protocol and port would do. The client then sends a request, which is composed of a number of *headers* and an optional *body* to the server. The server processes the request and sends back a response, composed of a status code, a number of headers and an optional body. Finally the connection is closed. [5]

A single physical network connection may be reused for multiple request/response cycles to avoid the overhead of creating a new socket connection on each request. The *Keep-Alive* header is used to control this behavior. In addition, multiple connections

may be run in parallel (which is the usual case with most modern browsers) to improve concurrency and throughput. This is especially important since modern websites often contain a large number of resources (images, scripts, style sheets, etc.) that need to be fetched in order to fully render a single (HTML) document.

An HTTP request is always targeted to a particular server-side resource. This resource may be static, such as an image or a static HTML document, or dynamic, such as a *PHP* (Hypertext Preprocessor) script that produces dynamic content. The *Content-Type* header in the response reveals the *MIME* (Multipurpose Internet Mail Extensions) type of the response body. The HTTP status code and message are used to signal the client of the response status and possible errors.

The most common HTTP status codes are 200 OK (indicating success), 302 FOUND (indicating a redirect to another location) and 404 NOT FOUND (indicating a missing resource). In case of a redirect, a client is required to follow the redirect to the secondary *URI* (Uniform Resource Identifier). In case of an error status, it is up to the client to decide what to do; most often the simplest course of action is to display a corresponding error message to the user.

HTTP defines a number of different request methods, namely HEAD, GET, POST, OPTIONS, PUT, DELETE, TRACE and CONNECT, each of which serves a slightly different purpose. It is important to note that a particular web server may not support all of the above methods (and this is in fact the usual case). The most important methods, with respect to web applications, GET and POST, are discussed next.

GET is the most commonly used request method (such a request is indeed sent every time one types a URI to a browser's address bar and hits enter). A GET request is usually dispatched in order to retrieve (read) a particular resource, such as an HTML document or an image file, but it may also be used to submit data in order to alter system state. [9]

A GET request carries all parameters in the request URI, and hence the request size is often limited to a client-dependent maximum (as an example, Internet Explorer allows up to 2083 characters in the URI [13]), though no maximum is specified in the HTTP specification. A sample GET request and the corresponding response (body omitted for brevity) are shown below. Also note the sample *request parameter*, conveniently named as "parameter".

```
GET /index.html?parameter=value HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8

<response body omitted>
```

Unlike GET, POST is most often used to submit form data to the server in order to alter system state [9]. The target resource for such requests is often a server side script/component capable of producing a dynamic response. A POST request may contain an arbitrary number of request parameters embedded in the request body, and possibly *multipart* data to enable file uploads. POST also places no limits on the size of the request data, though servers/applications may refuse to serve requests exceeding a particular size limit.

The HTTP specification also defines the https: URI scheme, which adds a security layer via *SSL* (Secure Sockets Layer) or *TLS* (Transport Layer Security) encryption to the communication stack [4]. This scheme is commonly used for secure connections in e.g. web-based banking applications. Further details about the https: URI scheme and data encryption are, however, beyond the scope of this paper.

1.2.3 Server-side implementation

The server-side implementation of a typical web application consists of (but is not

limited to) the following components:

- HTTP server to handle incoming requests and provide responses
- Application container / scripting engine to host applications
- Application logic components
- Database backend
- Static resources

The HTTP server is responsible for forwarding incoming requests to the application container / scripting engine, and for providing corresponding HTTP responses to the client. Commonly used HTTP servers include Apache (httpd) and Microsoft IIS. Java application servers also include embedded HTTP connectors to handle HTTP traffic. Static resources (such as images and style sheets) are often served directly by the HTTP server to avoid the overhead of dispatching such requests to the application container.

The application container provides a runtime framework and a further layer of abstraction for applications that are run in it. Among other things, this involves mapping requests to application components and wrapping raw HTTP requests and responses with technology specific constructs (such as the HttpServletRequest and HttpServletResponse interfaces of Java's JSP/Servlet specification) to enable efficient processing by application components.

Currently available application container / scripting engine technologies include JSP/Servlet containers (Java-based, such as Apache Tomcat and Oracle WebLogic), PHP (most often run as a module to the Apache web server) and ASP.NET (for Microsoft IIS).

A web application is also typically backed by one or more *databases*. Databases are used to store persistent application data. An application may also include file storage, integrations to other systems via web services or message brokers such as *JMS* (Java Message Service). It is important to realize that transactions across these resources contribute to web application performance overhead.

In a *clustered* environment, multiple instances of a particular application may be running at once, possibly on multiple physical servers, with the application container infrastructure coordinating the instances. Clustering generally has a positive impact on performance and scalability, since it allows for load to be distributed across multiple application servers and/or application instances. Further details about clustering are, however, beyond the scope of this paper.

1.2.4 Client-side implementation

The client-side of a typical web application is simply composed of the client's web browser. Such a client is commonly known as a *thin* client (as opposed to a *thick* desktop-based client application). Nowadays browsers come in several varieties, but the most commonly used browsers are Mozilla Firefox, Internet Explorer, Opera and Safari. The browser is responsible for communicating with the web server over the HTTP protocol, rendering the user interface of the web application, and enabling user input.

The user interface is commonly rendered as an HTML document which may contain text, input fields, links to other documents/resources, embedded objects (such as Java *applets*) as well as references to images (e.g. *JPG* or *PNG* images), scripts (e.g. JavaScript) and style sheets (CSS). The client first retrieves the root HTML document (which may be static or dynamic) with a GET request, parses the document text (as it appears in the response body), and then resolves/retrieves any referenced resources via subsequent GETs.

Finally scripts are executed, styles are processed and the document content (styled text along with images, objects etc.) is rendered to the user. Please note that this serial description of the flow of events is a simplification at best. Browsers may (and often do) execute the steps concurrently, improving performance and, consequently, user-experience. Section 2.1 provides additional coverage of client-side rendering issues.

User input is provided using the so-called *forms*. Forms are HTML *elements* that are used to collect user input via input elements such as text fields, checkboxes, and select

boxes. Upon form submission (i.e. when the user clicks the submit button), the form data is sent to the server via an HTTP request (commonly a POST, though the exact request method is configurable via the form element's *action* attribute) and the corresponding response is rendered to the user.

A simple example form (with static HTML source code and a screenshot of the rendered output) is shown in Figure 1. This particular form only contains a single text field and the submit button. Upon submission, the name and value (as input by the user to the text field) of the parameter "parameter" are sent to "resource.php" via a POST request.

```
<html>
<head>
    <title>Sample form</title>
</head>
<body>
    <form action="resource.php" method="post">
        Label:
        <input name="parameter" type="text" />
        <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

Figure 1: Sample HTML form

An alternative/complementary technique to using forms is to simply provide links/buttons that point to a particular resource (via the "href" attribute or by utilizing JavaScript), along with necessary parameters. Such an action is always dispatched via a GET request, and with *fixed* parameter value(s). A sample HTML snippet and corresponding rendered output (a simple link or anchor) is shown in Figure 2. This example also illustrates how multiple parameters are embedded to the request URI, by using an apostrophe "&" as a delimiter.

```
<a href="resource.php?parameter1=value1&parameter2=value2">Link</a>
```

Figure 2: Sample HTML link

Upon clicking this link, a GET request, along with the parameter-value pairs, is sent to

the resource “resource.php” and a corresponding (dynamic) response is rendered to the user. An obvious disadvantage with this type of input is that the request parameter values are fixed. Hence it is most useful for representing actions that need no user input (apart from the button click), e.g. a simple delete or export action.

This concludes the introduction to web applications. For further details about HTTP, HTML, CSS, JavaScript and related web (application) technologies, please refer to [14].

1.3 Performance testing

Performance testing is an important and emerging field of software engineering that is applied in order to measure application performance under varying load, identify performance problems and bottlenecks, and to verify that an application meets set performance criteria. This commonly involves measuring system throughput and latency with a varying number of (simulated) concurrent users, over extended periods of time, and with different load profiles (usage scenarios). [3]

1.3.1 Rationale

As is the case with other types of software testing, performance testing is often overlooked or even left out all together. In part this can be attributed to common myths with regard to performance testing. One such misconception is that performance testing is done solely for the purpose of breaking a system. This is however not the case, though performance tests can also be run in order to identify the saturation point (i.e. the maximum amount of load, discussed later in this chapter) for an application. [7]

For any complex application, it is important that testing (performance testing included) is done early and often. This not only allows for identifying (performance) problems early in the development cycle, further enabling early refactoring and other corrective action, but it also helps to reduce future maintenance costs which usually make up for a large part of an application’s total cost.

Performance testing can also be used to examine the *scalability* (measure of how

effectively an application responds to added resources), *reliability* (measure of how robust and fault-tolerant an application is) and *resource usage* (processing, memory, etc. analysis through *profiling*) of an application, as well as to compare different application vendors' solutions for performance. There are in fact several distinct kinds of performance testing; these are discussed next.

1.3.2 Test types

According to [1], the four basic types (the *LESS* approach) of performance testing are:

- Load testing
- Endurance testing
- Stress testing
- Spike testing

Load testing is conducted in order to determine how an application behaves under varying load. This involves varying the number of simulated concurrent users, test duration and test steps. As the name implies, *endurance testing* is carried out to examine an application's long-term behavior under moderate load. Endurance testing is indeed often coupled with profiling in order to identify resources that may be depleted (through resource leaks) over extended periods of time.

Stress testing, on the other hand, is executed for the purpose of finding the saturation (i.e. breaking) point of an application and to examine how gracefully (throttling down, crashing, etc.) the application (and the surrounding runtime) is able to navigate such a situation. *Spike testing* is a special case of stress testing and is used to determine how well an application responds to sudden increases in load.

In addition to the basic types listed above, there are two other important types of performance testing, namely *scalability testing* and *frontend analysis*. Scalability testing is carried out to examine how an application scales to handle increased load (i.e. serve more users) with added resources. Scalability tests can be implemented by running one or more of the above types of performance test against setups with differing resources

and comparing the results. If a significant increase in application performance and/or capacity is observed, as a result of adding to available resources, then the system is said to scale well. Frontend analysis is about observing client-side rendering performance.

1.3.3 Implementation

Performance tests are commonly implemented as a set of *scripts* that are run to generate *load* to the target application instance(s). This is achieved by utilizing a number of *load injector* machines, each running a separate instance of the test set. [7] Most currently available performance testing tools allow for running such distributed tests in a coordinated fashion. For small-scale scenarios, however, it may be adequate to use a single injector, as most load testing tools can effectively simulate multiple concurrent users even from a single injector.

Test scripts may be crafted by hand or recorded by using an appropriate tool. For web applications, scripts are often recorded by simply tracking the HTTP traffic between the client and the target application with a proxy server. Depending on testing tool used, the scripts may be fitted to include conditional tests, parameterization (e.g. dynamic user credentials), loops, assertions, timers and random test elements. Scripts may be recorded or written in a number of (programming) languages, such as XML, C, Java or Python.

An essential part of performance testing is reporting. Hence performance testing tools must offer means of analyzing and/or exporting test results. The number and quality of recorded *metrics* depend on the type of test and tool used. Common result metrics for web applications include response time and the HTTP response status code. Results are commonly published in textual format, e.g. XML or *CSV* (Comma-separated values), or graphical charts (e.g. line, bar, pie or scatter charts).

This concludes the short introduction to performance testing. Factors affecting web application performance are discussed next. Further details about performance testing, including specification, design and implementation issues, appear in chapter 3.

2 WEB APPLICATION PERFORMANCE

The factors affecting web application performance (i.e. page rendering or response times, to good or bad) can be roughly divided into two categories: client-side (*frontend*) performance and server-side (*backend*) performance. These factors are discussed next, along with some concrete advice for improving application performance in the general case. The final section provides limited discussion on the relative importance of these factors, and on how they are related to web application performance testing. Material in this chapter is based on [2], unless otherwise noted.

2.1 Frontend performance factors

At the client-side, the key to good performance is to minimize network traffic[2]. Below we address some common ways to accomplish this. Furthermore, sections 2.1.6 and 2.1.7 provide guidelines on how to improve page rendering performance by optimizing style sheets and JavaScript, respectively.

2.1.1 HTTP request count

As explained in section 1.2.4, a single HTTP request is typically used to fetch the root HTML document. The root document may, however, refer to an arbitrary number of other resources, such as images, scripts or stylesheets. Each of these resources must be fetched with a subsequent HTTP request. Each HTTP request adds to performance overhead since it creates network traffic between the client and server. Thus it is immediately obvious that reducing the number of referenced resources and, consequently, the number of HTTP requests, will improve application performance.

In a related vein, Ajax allows for HTTP requests to be dispatched asynchronously with Javascript code, without reloading the entire HTML page along with all of its referenced resources (as opposed to the traditional web programming model as described in section 1.2.4). Hence, generally speaking, Ajax greatly improves frontend performance, because

[2] Because a real computer network provides only limited bandwidth, and network latency grows with the physical distance between a client and server, it makes sense to minimize the amount of transferred data.

it reduces the total number of HTTP requests. But like any other Javascript code, poorly devised Ajax code can also hinder web page rendering performance. For further information about Ajax, see e.g. [14].

2.1.2 Caching of resources

To reduce the number of HTTP requests, browsers are keen to cache resources. This means that a browser is able to store certain resources (such as images, style sheets and scripts) locally, instead of fetching them over the network each time. This behavior is controlled by a number of request and response headers. A client can perform a *conditional GET* request by supplying the *If-Modified-Since* header. In response to a conditional GET, if the resource has not changed, the application server may return a 304 Not Modified response with no body; this reduces the amount of transmitted data.

An application server may choose to supply the *Expires* and/or *Cache-Control* response headers with responses; these are used to signal the client that a resource should only be re-retrieved after a particular date or period of time has passed. Proper use of the above headers may result in a significant reduction in the number of HTTP requests. Thus caching should be used, whenever possible, to improve application performance.

2.1.3 DNS lookups

IP (Internet Protocol) addresses, such as 132.49.12.36, are used to locate servers on the Internet. These numerical addresses are, however, hard for a human to remember. Luckily, the *Domain Name System* (DNS) exists to provide a mapping between a human-readable hostname (such as “www.google.fi”, embedded in resource URIs) and the corresponding IP address.

Unfortunately this mapping comes with a cost. A typical DNS lookup (to resolve the IP address) for a particular hostname takes approximately 20-120 milliseconds to complete. Even with DNS caching, this reduces performance. Thus the number of DNS lookups should be reduced to a minimum (by minimizing the number of distinct hosts

serving resources) to improve application performance.

2.1.4 Redirects

Redirects are used for a multitude of purposes, such as tracking user movement (by proxying requests via trackers), and the *redirect-after-post* [15] technique, which is used to prevent the "double submit" problem after submission of a form that uses the POST request method. It is important to realise, however, that a redirect always requires the client to dispatch an extra HTTP request to the secondary URI, which implies reduced performance. Thus redirects should be avoided to improve application performance.

2.1.5 Compression

The body of an HTTP response can be compressed to reduce the amount of transmitted data. A client can indicate support for compression by using the *Accept-Encoding* request header with an appropriate compression method. Conversely, the application server may supply the *Content-Encoding* header to indicate a compressed response body. A commonly used compression method is gzip [16]. Compression should be applied to reasonably-sized (> 2KB) *text* responses to improve application performance.

2.1.6 Style sheets

As any other static resource, style sheets should be cached by the browser to reduce the total number of HTTP requests. Caching of style sheets is enabled by using *external* (rather than *inline* or *embedded)* style sheets, which allow for the style sheet to be requested separately from the main document, and by appending the appropriate caching headers (as discussed in section 2.1.2) to responses.

Because browsers often utilize *progressive rendering*, i.e. render whatever content is available as soon as possible, misplaced references to style sheets can delay the rendering of a web page by forcing the browser to defer rendering of the entire document until those references have been resolved. It is thus appropriate to put

references to style sheets at the top of the HTML document to allow for proper progressive rendering and, consequently, improved application performance.

Another source of poor performance with regard to style sheets are CSS *expressions*. CSS expressions are a powerful way of dynamically controlling page layout and style, because they are re-evaluated every time the page changes (upon window resize, for example). Unfortunately this evaluation requires significant processing power and adds to performance overhead. Thus CSS expressions should be avoided, whenever possible, to improve application performance.

2.1.7 JavaScript

Like style sheets, JavaScript scripts should be externalized and cached whenever possible to improve performance. But unlike style sheets, scripts should be placed at the bottom (or as near the bottom as possible) of an HTML document for best performance. This is because script execution not only blocks parallel downloads of resources, but also effectively disables progressive rendering of elements appearing after the script.

Furthermore, because JavaScript is a rich programming language that allows the developer to use arbitrary names for variables and functions, add comments, and format code with an arbitrary amount of whitespace (spaces and tabs), script files can become large, which implies reduced performance. To counter this, compression, minification (trimming comments and whitespace) and obfuscation (minifying variable, function, etc. names) should be utilized for improving application performance.

Finally, one should make sure that an external JavaScript script is never included to a single HTML document more than once. Duplicate scripts require both duplicated HTTP requests and processing effort, which implies reduced application performance.

2.2 Backend performance factors

A high-performance backend is able to to process a large number of concurrent client requests with minimal response times. Factors driving this ability are considered next.

2.2.1 Platform

Selection of programming language and platform, runtime environment and development tools all contribute to web application performance and scalability. Programming languages and platforms affect performance because they vary greatly in their implementation and runtime performance. As examples, compiled code (e.g. C++) generally performs better than interpreted code (e.g. Java), static typing (of e.g. Java) avoids the runtime overhead of dynamic typing (of e.g. Python), and dynamic semantic checks (of, say, Java) can be useful for debugging and error detection, but incur significant runtime overhead.

Updated versions of a particular platform typically include performance enhancements not found in earlier versions. Vendor-specific performance may also differ (which is often the case for e.g. application servers). Development tools affect performance by promoting particular styles of development, application frameworks, libraries, as well as deployment strategies and targets. Use of modern platform-architectural styles such as *cloud computing* [17] can have a profound (positive) effect on an application's performance and scalability.

2.2.2 Implementation

Server-side code quality, architectural complexity and selection of third-party libraries and/or modules can have a significant effect on application performance. Hence proper selection of algorithms, architectural models and libraries, use of well-established coding idioms, optimization of database queries, effective use of application frameworks and efficient modularization, among others, are vital for good performance.

It is also important to realise that high-level approaches to application development (utilizing various frameworks and a layered design) often simplify the task of the programmer by hiding unnecessary implementation details, and make the system easier to develop, comprehend and maintain due to separation of concerns. Unfortunately this adds further layers of abstraction to the application stack, which may have an adverse

effect on performance due to increased indirection (e.g. longer method invocation chains or the use of *reflection* in languages that support it). Fortunately, a modular or layered design can also have a positive effect on scalability, which in turn can be harnessed for improving application performance by adding to available server resources, such as the number of processors or the amount of memory.

2.2.3 Configuration

Proper configuration of application and database servers (and clusters thereof) is often vital for good application performance. An application server (and the application running in it) must often handle requests from multiple concurrent clients. Thus the configuration of thread pools, database connection pools, memory management (e.g. garbage collection) etc. can have a substantial effect on performance. Proper configuration of database properties, such as indexing, table spaces or caching is equally important. Other external resources, such as file servers or message brokers may require similar attention in order to achieve best performance.

2.3 Remarks

As suggested by [2], from a user's perspective, frontend performance is more important, as typically only 10–20% of the total response time is spent fetching the root HTML document, which includes any (dynamic) backend processing. The remaining 80–90% is spent rendering the response on the client-side (including the fetching of related resources such as images, style sheets and scripts with subsequent HTTP requests).

It is, however, important to realize that while frontend performance typically makes up for most of how the user perceives application performance, a poorly performing backend can bring the entire application to its knees by taking a long time to process the initial request, or by refusing to handle the request at all. It is ultimately the backend that must handle high and unpredictable concurrent load over long periods of time.

With performance testing in mind, it is thus important to analyze performance in both domains. Frontend performance should be measured in order to identify and enhance the

quality of the immediate user-experience. Backend performance should be measured in order to determine maximum concurrent load, scalability and long-term application behavior, among other things. Whereas chapter 3 focuses on backend performance (load testing), the case study of chapter 4 incorporates examples of both frontend and backend performance analysis using appropriate tools.

This concludes the introduction to web application performance aspects. The next chapter will build upon the theory presented thus far to present a thorough discussion of the intricacies of web application performance test design and implementation.

3 WEB APPLICATION PERFORMANCE TESTING

This chapter discusses practical implementation issues of web application performance testing, including test preparation, execution and reporting. The focus is on backend load testing. Performance testing is an activity that may occur concurrently with other application development tasks, or be carried out after an application has been successfully deployed into production. In general, our discussion does not assume any particular phase in application lifecycle; where it does, the phase is clearly stated. Material in this chapter is based on [1], unless otherwise noted.

3.1 Preparing for performance tests

The test preparation phase involves the definition, design and building of the test environment and scripts. These items are discussed next.

3.1.1 Defining acceptance criteria

In order to establish performance (acceptance) criteria for an application, requirements elicitation (as it appears in the initial application design phase) must include performance considerations, such as projected user base and number of concurrent users, typical usage scenarios, desired quality of service (e.g. in terms of maximum response times) and maximum server resource utilization, to name a few. In a formal process, the result of these considerations is the *performance requirement document*.

In addition, it is often necessary for the client and service provider to sign a *Service Level Agreement* (SLA). The SLA is a formal, binding document on an application's performance acceptance criteria, agreed upon by both the client and service provider. Further, established performance requirements are used to define a *performance test strategy* (document). This strategy represents a high-level roadmap for performance tests. Topics covered by the test strategy typically include (but are not limited to):

- Scope
- Metrics

- Objectives
- Load profiles
- Test environment
- Think time
- Test data

Scope defines the extent to which performance testing is conducted, including disucssion of the components to be tested and the types of test to execute (e.g. LESS). *Metrics* define the criteria by which system performance is measured. Common metrics for web applications include response time and throughput. Relevant metrics should be defined by consulting appropriate *stakeholders*.

Objectives represent the rationale for carrying out performance tests. Typical objectives include verifying an application's ability to handle a specified number of concurrent users or asserting its ability to sustain high load over a long period of time without resource leaks. Objectives must be conceived by consulting appropriate stakeholders. *Load profiles* represent typical usage scenarios for the application. Realistic load profiles should be deduced by consulting relevant business stakeholders.

Preliminary discussion of the *test environment* must be included in the strategy. The test environment should resemble the production environment as far as possible. A standalone performance test environment provides most accurate results since it is not shared by interfering testing and/or production activities. Unfortunately such an environment may be not be readily available; in this case performance test execution should be isolated from other activities to ensure reliable results.

User *think time*, i.e. the time a user typically takes to "think" before executing a particular action, such as submitting a form, must be addressed in the strategy. Think times can have a profound effect on test relevance. Too short or long think times can result in biased test results, due to unusually high or low transaction rates, respectively.

Last, but surely not least, performance test data must be addressed in the strategy. This

includes discussion of both dynamic input (user credentials, form data, etc.) as well as test database setup. Test data should resemble that of production as far as possible to enable reliable results. Hence if real production data is available, it should be used. If not, sufficient amounts of realistic test data should be generated. Unfortunately test data generation is a daunting and time consuming task, and lack of proper test data can invalidate an otherwise legitimate test setup.

3.1.2 Designing the test scenarios

The test design phase captures the performance requirements and strategy of the definition phase to produce a solid performance test design (blueprint), which is in turn realised in the building phase. Needless to say, test design is the single most important step in performance testing lifecycle. It is essentially composed of three components: *scenario*, *workload* and *tooling* design. These concepts are discussed next.

A scenario is a collection of *transactions*[3]. In practical terms, a scenario is a sequence of user actions, such as logging in to the system, clicking on a particular link, submitting a form, and finally logging out of the system. Scenario design is vital for realistic simulation of application usage in performance tests, and a prequisite for workload design. A scenario should be composed of transactions that represent typical and/or critical user actions, and have significant performance effects.

To identify frequently occurring transactions, stakeholders and e.g. application server access logs should be consulted. It is important to study application usage over a sufficiently long period of time, because usage patterns can vary greatly over time, based on time of day, day of week, week of month, or even month of year. For example, in a banking application, weekdays are likely to incur more load than weekends. Similarly, a payroll application will likely have less use during the summer months due to vacations.

Transactions should also be prioritized based on the following qualities:

[3] In this context, a transaction is a user action that results in server-side processing, typically an HTTP request to a server-side application resource. It may (but is not required to) span a database transaction.

- Concurrency
- Number of user interactions
- Computational requirements
- Resource usage

Concurrency defines the degree to which a transaction is typically executed simultaneously by concurrent users. A single transaction may be composed of multiple user interactions (requests). Transactions may also have different computational requirements in terms of required processing power and time. Resource usage refers to the I/O[4] operations incurred by a transaction, among other things.

Workload design builds on scenario design by assigning transactions (or scenarios) to specific (simulated) *user groups* (such as the "customers", "managers" and "support personnel" of a banking application), assigning relative *weights* to the user groups and to transactions within each group, and *sequencing* transactions within groups.

The relative weight of a user group or transaction denotes its relative importance and commit rate (some groups use the appplication more actively, and some transactions take place more often than others) within a test. Sequencing of transactions denotes the ordering and timing of transactions during a test run within a user group. The sequence of transactions is typically inferred from the corresponding scenario.

In *goal-oriented* workload design, performance tests are designed to assert certain system qualities, such as high system availability or graceful degradation during overload. In *transaction-oriented* design, performance tests are devised so that they focus on particular, critical transaction types. *Architecture-oriented* workload design focuses on verifying the scalability, robustness and efficiency of application architecture, and as such requires intricate knowledge of its implementation. *Growth-oriented* design places an exclusive focus on testing system scalability.

Tooling design involves the selection of performance testing tools. At a bare minimum,

[4] Input/Output, such as file access

for web application performance testing, the tool(s) should provide the following features, necessary for devising, running and analysing tests:

- HTTP protocol support
- Test script editing and recording capability
- Client-side cookie support (for session tracking)
- Ability to parameterize tests with input data (e.g. user credentials)
- Ability to run test scripts with an arbitrary number of users and iterations
- Ability to record relevant metrics (such as response time) at run time
- Test data export capability

In addition to the features listed above, for some applications, it may be necessary to support data transport encryption (via HTTPS, see the end of section 1.2.2), basic authentication or file uploads (multipart requests), among other things. For further guidelines on performance testing tool selection, please refer to appendix B of [1].

3.1.3 Building the test suite

The purpose of the build phase is to implement the test design in a way that enables successful test execution. At this stage, the target application must be deployable, and must successfully implement all of the features that are to be tested. The build phase is essentially composed of four tasks: creating a *performance test plan*, setting up a *test environment*, developing *test scripts*, and setting up a *test schedule*. These items are discussed next.

A performance test plan (document) saves the results of the test definition and design phases (we will not repeat the items dicussed in the previous two sections here), and appends a detailed plan of test execution, including plans for the remaining three tasks above. In addition, a performance test plan typically includes discussion of any assumptions, constraints and risk factors that are present in the design. A performance test plan is thus an essential tool for the project manager, test designer, and testers alike.

Test environment construction is a crucial task in performance test setup. The goal is to create an environment that most closely resembles production (otherwise the test results would not apply to production). It spans the setup of both client-side machines (load injectors, see section 1.3.3) and server-side application components (application servers, database servers, firewalls, load balancers etc.) At the same time, it is a very domain, environment and application-specific task that requires a lot of coordination between stakeholders. Hence we only outline some important considerations below.

- Is a proprietary performance testing environment available?
- If not, can external noise (interference from other users) be eliminated?
- Does the test environment resemble production (w.r.t hardware and software)?

These considerations are typically driven by budget, time and resource constraints. It is generally difficult (or next to impossible) to use a production environment for performance testing, due to inherent noise from regular use and data integrity constraints (performance tests must not modify real production data, such as the account balances in a banking application). In a similar vein, a simulated environment that *perfectly* matches production environment in both hardware and software, is generally too costly and time consuming to implement. In practical terms, test environment setup is composed of:

- Hardware and software installation
- Hardware, software and network configuration
- Application build, deployment and configuration
- Client-side (load injector) setup
- Test data(base) setup

Once a test environment has been setup, test scripts[5] can be devised for each relevant scenario (or transaction). As explained in section 1.3.3, the scripts are either manually written or recorded, depending on the complexity of the scenario and available tooling. In either case, a script must be manually edited to include dynamic, environment or user-specific input data (i.e. request parameters), and to modify think times (see relevant

[5] A test script is a sequence of programmatic HTTP requests (transactions) to application resources. It may include requests to static resources (such as images), but these are often omitted for simplicity.

part of section 3.1.1). *Form tokens*[6] are a prime example of dynamic input. A form token's value varies per request (form reload) and renders the use of a static, recorded value impossible[7]. In a similar vein, session tracking, if implemented by appending session ids (that vary per browser session) to request URIs, must be taken into account in script development. Caching of resources is another important consideration (see section 2.1.2 for rationale). Each script should be *smoke tested* (by running it against the test application) to assert correct runtime behavior and test data compatibility.

The final step in test preparation is to create a test schedule and assign testers to it. In a shared test environment, where e.g. functional testing and development activities may occur at arbitrary times, scheduling an isolated performance test might turn out to be a challenge. This concludes our discussion of the test preparation phase.

3.2 Performance test execution

Performance test execution can be divided into three distinct phases: validation, baseline creation and benchmarking. These are discussed next. Furthermore, sections 3.2.4 and 3.2.5 discuss typical reasons for test failure and test monitoring, respectively.

3.2.1 Validating the test suite

Prior to running actual tests, the entire test suite (of scripts) must be validated. For this purpose, *elaboration* and *self-satisfaction* tests are executed. Elaboration tests are run to verify that the system operates as expected during a performance test, and that the test runs produce reasonable output data (metrics). They are also useful for:

- Verifying test data integrity
- Understanding system behavior when subjected to performance tests
- Establishing a *proof-of-concept* for performance tests (to management)
- Debugging any remaining issues with the test scripts and/or environment
- Familiarizing testers with the test suite and environment

[6] Form tokens are hidden HTML form elements that are included to prevent a double form submit.
[7] A test script must parse the token value from a previous response that was used to render the form.

- Tuning application parameters

Due to resource and time constraints, elaboration tests are often run with a small number of concurrent simulated users and a limited number of runs. As the name implies, additional self-satisfaction tests complement elaboration tests by building tester confidence in the test suite and tooling (via repeated runs and peer/expert review) and by asserting system readiness for the final performance tests.

3.2.2 Creating a baseline

To establish a point of comparison (reference) for future performance test runs, a baseline is created. In other words, the results of the initial stable run(s) of the test suite are recorded for future reference. Any subsequent test run (with the same configuration and test data) can then be compared to the baseline to see whether performance has improved or declined.

In particular, a baseline allows for application performance to be tracked across builds and versions, though major application revisions may require a rebuild of the baseline due to functional, architectural or platform-induced changes that render the comparison unreliable. A baseline can also be used to identify performance deviations due to configuration changes and *tuning* (see section 3.2.5 below).

3.2.3 Benchmarking with multiple test runs

Once a baseline has been setup, an arbitrary number of test runs (benchmarking) will follow. In addition to running tests with each application revision, the need for repeated test runs may arise due to failed tests (runtime errors, see section 3.2.4 below), test data corruption, human error, or other unexpected conditions during test execution. Endurance tests are clearly most susceptible to such problems due to inherently long run-time.

Furthermore, multiple test runs provide more reliable results because they effectively

eliminate uncertainty due to transient factors (such as someone mistakenly using the target application for other purposes during a performance test run) in the test environment. In particular, *repeatable* results are likely to promote testers' and stakeholders' confidence in the test environment and application performance.

Depending on the test data and the application functionality being tested, it may be necessary to "reset" the test environment and/or database between test runs to a particular state. For instance, in a banking application, the need for an intermediate reset would arise with a test that creates an account with a particular number. If such a test were to be run again *without* resetting the database, the account creation (and the surrounding test) would fail due to a duplicate account number.

In a related vein, the word *benchmark* can also refer to the use of an external, industry-standard performance benchmark provided by a third-party organization. It is carried out by running a set of tests, that comply with the specifications of the industry-standard benchmark, to produce a score. Benchmarks complement the results of proprietary performance tests by allowing direct performance comparison (via scores) to applications from other vendors. [3] Further details about benchmarks are, however, out of the scope of this paper; refer to [3] for a more thorough discussion on the subject.

3.2.4 Reasons for performance test failure

A performance test run is considered as failed if it fails to produce reliable and useful results. Typical causes for failed performance tests include *application defects* and *network congestion*.

Whereas test scripts typically operate in a deterministic manner, i.e. execute a predefined sequence of steps (requests), application defects[8] may cause the application to respond in a way that was not expected by the test script. In this case the script cannot continue, and test execution is aborted with a runtime error.

[8] Concurrency bugs (due to the high number of concurrent users) are a common source of error in performance tests. They are also notoriously difficult to debug due to their non-deterministic nature.

Network congestion is harmful to performance test execution for two primary reasons: first, it may result in unusually high response times and poor throughput due to excess network delays; second, complete connection failures generally cause a test to abort with a network timeout error. In this respect, proper test network configuration is important.

3.2.5 Monitoring tests

We end our discussion of the test execution phase with test *monitoring*. Monitoring a performance test involves keeping track of how the test is progressing. By monitoring a test run, any errors are spotted immediately, allowing for the test to be cancelled and restarted on the spot, which saves time. Monitoring typically includes tracking test script or tool (console) output, application and database server logs, and observing realtime server load statistics such as processor and memory usage through *profiling*.

Profiling is especially important when coupled with tuning (modifying application and/or server configuration in order to find the best performing setup). Unfortunately profiling and tuning are very broad topics in themselves, and hence were not included in the scope of this paper; the interested reader may refer to [19] for further information.

3.3 Analysis and reporting of test results

Performance test result analysis typically involves three tasks: test data collection, analysis and reporting. These tasks are discussed in the following subsections.

3.3.1 Test data collection

Test data is typically collected by observing test tool output logs. The number, format and content of these logs is tool-dependent. A typical solution would be to output two distinct log files, one containing a summary of the test run(s), and another containing the raw test data. Runtime errors during a test run may also produce separate error logs. Log files are typically in either XML or CSV format to enable efficient parsing and analysis thereof. A typical[9] test output log contains the sequence of transactions with the

[9] Load testing a web application over HTTP

following columns and metrics (though this list is by no means exhaustive):

- User id
- Iteration
- Timestamp
- Transaction id
- HTTP status code
- Response time
- Error status

User id represents a tool-dependent identifier for a simulated user (a single test is typically run with multiple simulated concurrent users), e.g. a running number. *Iteration* represents the iteration number (in case of multiple iterations of the same test), which is generally a running number. *Timestamp* represents the time of execution for the transaction, which may be relative to test start time, and is typically given to millisecond precision. *Transaction id* is a transaction (request) identifier, e.g. a predefined number.

HTTP status code is the response status code. *Response time* is the time elapsed from sending a request to receiving a full response. *Error status* is a boolean that tells whether the test failed due to a runtime error (in which case an error log is typically created) or a bad HTTP status code (e.g. 404 not found, see section 1.2.2 for details).

Depending on the types of metric being collected, it may also be necessary to collect persistent profiling data from server-side machines. This is usually achieved by consulting relevant server logs and/or the output of dedicated profiler tools (that were active during the performance test run). Server-side metrics are most useful for investigating and pinpointing potential performance bottlenecks.

3.3.2 Test data analysis

Once test data has been successfully gathered, it must be analyzed. Analysis is carried out by calculating (aggregate) metrics, such as average response time per time unit or

peak number of transactions per time unit, from the data. Furthermore, the (aggregate) metrics may be compared to a baseline to produce a performance *trend*. A test tool may incorporate the necessary facilities for data analysis. Alternatively, generic spreadsheet tools (such as Microsoft Excel) may be used. At the server-side, enterprise-grade management tools often incorporate sophisticated data analysis and reporting facilities.

According to [1], test data analysis serves the following purposes (non-exhaustive, in order of relative importance):

- Checking whether all tests (transactions) were executed as planned
- Checking whether set performance criteria were met by the system under test
- Identifying performance bottlenecks and possible remedies

The first item is important because errors in a test run *may* render test data unreliable, thus preventing further analysis. It is up to the performance analyst to decide whether a particular number of failed transactions constitutes a failed test run. The second and third items are at the heart of software performance testing. Test data must be compared to specified criteria to determine whether application performance is within acceptable limits. Performance bottlenecks may be identified by focusing on particular transactions, application resources or server-side metrics.

3.3.3 Reporting results

Reports are the primary deliverable of performance tests. Hence they must convey all the relevant performance metrics in a concise and self-evident manner. Reports are typically created to serve the interests of different stakeholders, such as management, developers or system administrators. Management is most interested to see how their investment in performance testing and application development pays off. Developers require detailed performance reports to be able to enhance application performance. System administrators use their reports to determine an optimal runtime configuration.

As explained in section 1.3.3, reports may be produced in textual (tabular) format or graphs. Whereas developers typically require intricate details (i.e. “raw” figures) on

application performance, for management, a handful of high-level and colorful pie charts (along with proper justification) will likely suffice. When producing reports, standard statistical analysis methods[10] may be used, along with conventional[11] tooling. Spreadsheet tools can be used to provide dual representations (tables and charts) of data.

Within a report, it is important to highlight any anomalies in the test data. For example, sudden and/or constant increases in response time or application/database server resource usage may be indications of performance problems. Graphs are particularly effective at conveying such trends.

This concludes our discussion of web application performance testing. The interested reader may refer to e.g. [1], [3], and [7] for further information. Chapter 4 provides examples of both test tooling and implementation.

---

[10] Averaging, sampling and histograms, among others
[11] Generally available (open-source or commercial), non-specialized

4 CASE STUDY: WORDNET

The purpose of this case study is to convey key ideas presented in the previous chapters in a pragmatic manner. The web application under test is *WordNet*, a large lexical database of English, publicly available on the internet, provided by Princeton University. [21] This particular application was chosen because it couples an adequately complex frontend with search functionality that provides a good candidate for backend load testing.

We begin by describing the test setup (section 4.1) and test tooling (section 4.2). For brevity, tool installation steps are not included. Section 4.3 describes a sample frontend analysis, and section 4.4 walks through a simple load test. Finally, some generic remarks about the test results are made.

4.1 Setup

The following setup was used to drive tests:

- HP Compaq nw8240 laptop computer
- Pentium M 2GHz processor, 1.5GB RAM
- Windows XP Pro SP3 operating system
- 54 Mbit WLAN (Wireless LAN) network
- 8/1 Mbit ADSL2+ internet connection

The following application versions were used:

- Firebug 1.4.2
- Apache JMeter 2.3.2
- Mozilla Firefox 3.5.2
- Sun JRE[12] 6u15
- YSlow 2.0.0b6

[12] A Java Runtime Environment [22] is necessary to run JMeter.

All tests were run on a freshly booted machine with a minimal number of simultaneously running (background) processes to avoid interference with the tests.

4.2 Tooling

This section provides a brief introduction to the open source performance testing and analysis tools used in the case study. JMeter was selected primarily from personal experience. It is also seemingly popular (with a Google hit count of 574,000 as of December 2009). Firebug and YSlow were selected because they come highly recommended by [2]. It must be noted that a number of alternative tools (both open source and commercial) exist. For a listing of open source alternatives, refer to [18].

4.2.1 JMeter

Apache JMeter is a Java-based *load testing tool*. It can be used to load test applications over a variety of protocols and APIs (Application Programming Interfaces), including HTTP, SOAP (Simple Object Access Protocol), JDBC (Java Database Connectivity), LDAP (Lightweight Directory Access Protocol), JMS, POP3 (Post Office Protocol) and IMAP (Internet Message Access Protocol). JMeter is a lightweight desktop application with a simple graphical user interface (GUI). It can also be run from the command line, if necessary (e.g. in a headless[13] environment).

JMeter runs on the *Java Virtual Machine* (JVM), so it is fully portable across machine architectures and operating systems, and has native support for *multithreading* (i.e. tests can be run with multiple simulated concurrent users and groups within a single instance). JMeter allows for test scenarios to be recorded using a built-in proxy server. It has a fully configurable, pluggable (via plug-ins) and scriptable (using e.g. *BeanShell*[14]) test architecture, so all types of performance test (LESS and more) can be run with it. In addition, multiple test iterations (runs), input parameterization, loops, assertions and timers are supported out-of-the-box.

[13] No display device (monitor) attached
[14] BeanShell [20] is a lightweight scripting language for the Java platform

Pluggable *samplers* allow for a number of metrics to be recorded at test run-time. Basic test data analysis and reporting facilities are also included; these can be extended by installing appropriate plug-ins. JMeter uses a proprietary XML format to store test plans (steps) and test result data alike.

With respect to web application performance testing, while JMeter effectively simulates the behavior of a web browser (or, actually, multiple browsers), it is *not* a web browser. In particular, JMeter does not render HTML, evaluate style sheets or execute JavaScript. It simply dispatches the same HTTP requests (and receives the same responses) as a real browser would, with the addition of recording response time, throughput and other relevant metrics at the same time. [10]

4.2.2 Firebug

Firebug is a Firefox[15] add-on that can be used for *frontend performance analysis*. It enables a number of web application frontend development tasks (non-exhaustive list):

- HTML inspection and editing
- CSS inspection, editing and visualization
- JavaScript execution, debugging, logging and profiling
- DOM (Document Object Model) inspection
- Network traffic monitoring

For our purposes, the last item is clearly most important. Firebug's network monitoring facilities allow the user to track response times per HTTP request (i.e. per resource), monitor browser cache usage, inspect HTTP request and response headers, and visualize the entire loading of a web page (from the root HTML document to any referred resources) on a timeline. Furthermore, Firebug's JavaScript profiling capabilities may come in handy when diagnosing script-induced performance problems. [11]

[15] There is also a "lite" version of Firebug that can be used with any browser. Refer to [11] for details.

4.2.3 YSlow

YSlow is a Firefox add-on that integrates seamlessly with Firebug. YSlow analyzes and *grades* frontend performance (like a benchmark, see section 3.2.3), and suggests ways to improve it, based on a predefined set of rules (custom rule sets may also be defined). Some of these rules were introduced in chapter 2.1. Grading is done on a scale of A through F, where A denotes best performance. YSlow also provides views for inspecting individual resources (e.g. size) and overall statistics (such as total page weight). [12]

4.3 Sample frontend analysis

This section depicts a simple frontend analysis. As example, we analyze the WordNet homepage. We consider page load time, the number of HTTP requests, caching, compression, and finally the YSlow grade and performance suggestions.

The first step is to launch Firefox. Once Firefox is up and running, we must enable Firebug and YSlow for all web pages. Firebug is enabled by right-clicking on its "bug" icon in the browser's status bar (lower right corner of the screen) and selecting the "On for All Web Pages" option. In a similar fashion, YSlow is enabled by right-clicking on its "gauge" icon and selecting the "Autorun" option. These tasks are illustrated in Figures 3 and 4 below.

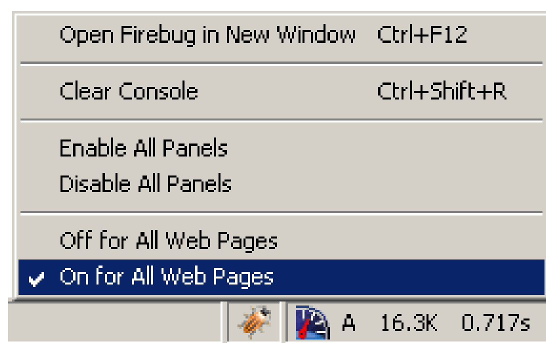

Figure 3: Enabling Firebug in Firefox

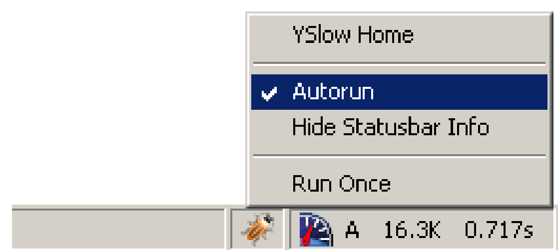

Figure 4: Enabling YSlow in Firefox

To bring up the Firebug console, left-click on the Firebug icon. Now select the "Net" (network) view by clicking on the corresponding tab. We haven't loaded a page yet, so the view is empty. The view should resemble that of Figure 5 below.

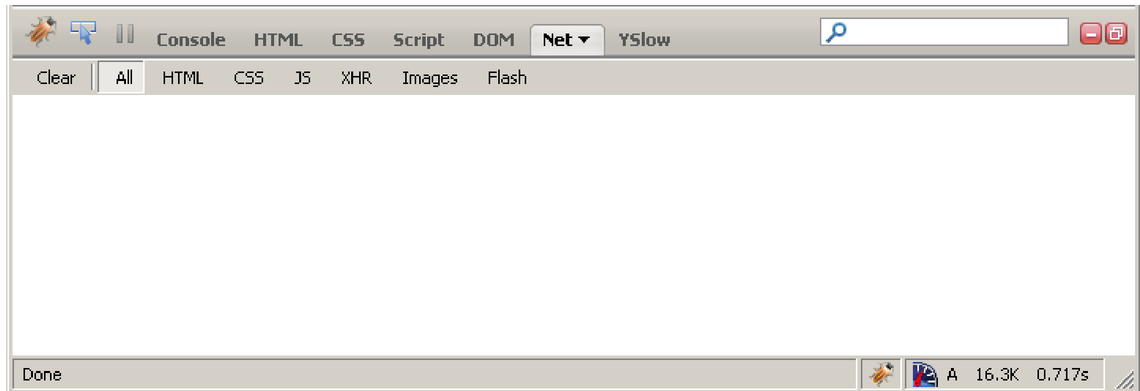

Figure 5: Empty Firebug network statistics view

The next step is to browse to the WordNet homepage. To do this, simply type the target URL (http://wordnet.princeton.edu/) into the browser's address bar and hit enter. As the page loads, Firebug should update its view. The resulting view is illustrated in Figure 6.

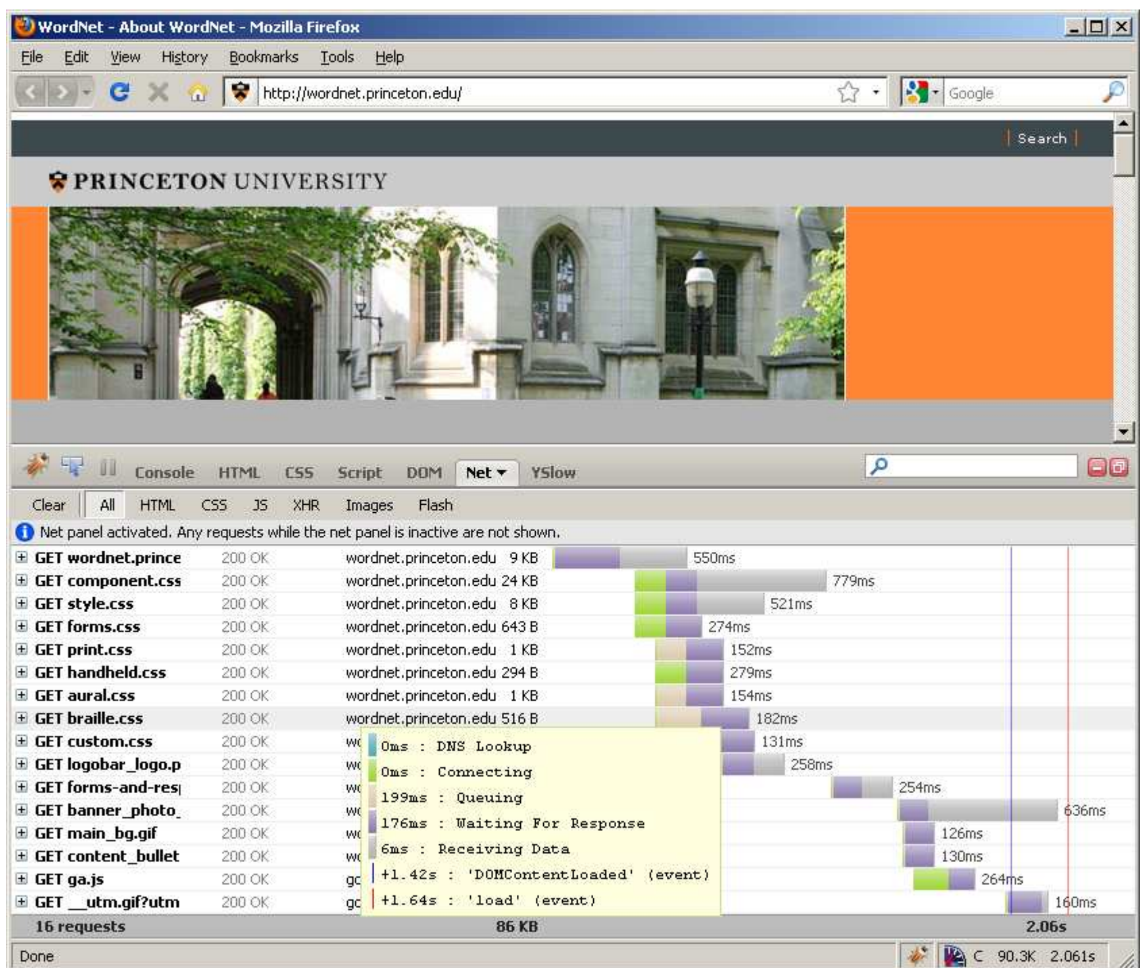

Figure 6: Firebug network statistics for the WordNet homepage, empty browser cache

Looking at the statistics, 16 GET requests were made, with a total response time of 2.061 seconds, and a combined page weight of 86 KB. The requests are displayed in the order in which they were sent, starting with the root HTML document. For each request, HTTP method and status code, target host and file size are shown. By hovering the mouse on top of a particular request, a detailed breakdown of the total response time for the request is shown (the rectangular popup box in Figure 6 above). The timeline graph on the right conveys the same data in a visual manner, with matching colors.

The page was initially loaded with an empty browser cache (see section 2.1.2). To demonstrate the effect of a *primed* (i.e. full) browser cache, we now reload the page by clicking on the browser's reload button. The resulting statistics are shown in Figure 7 below. In a nutshell, by caching resources, the total response time was reduced to 1.036 seconds (a 50 % reduction), and the total amount of transferred data decreased from 86 KB to a mere 11 KB (a 90 % reduction).

As described in section 2.1.2, the browser sends a conditional GET request to retrieve a resource that has been previously cached (even with an explicit reload of a page). In our sample, all but one of the referred resources were in fact cached, and for each of those resources the server returned a 304 status code to indicate that the cached version is valid, avoiding the need to transfer data over the wire.

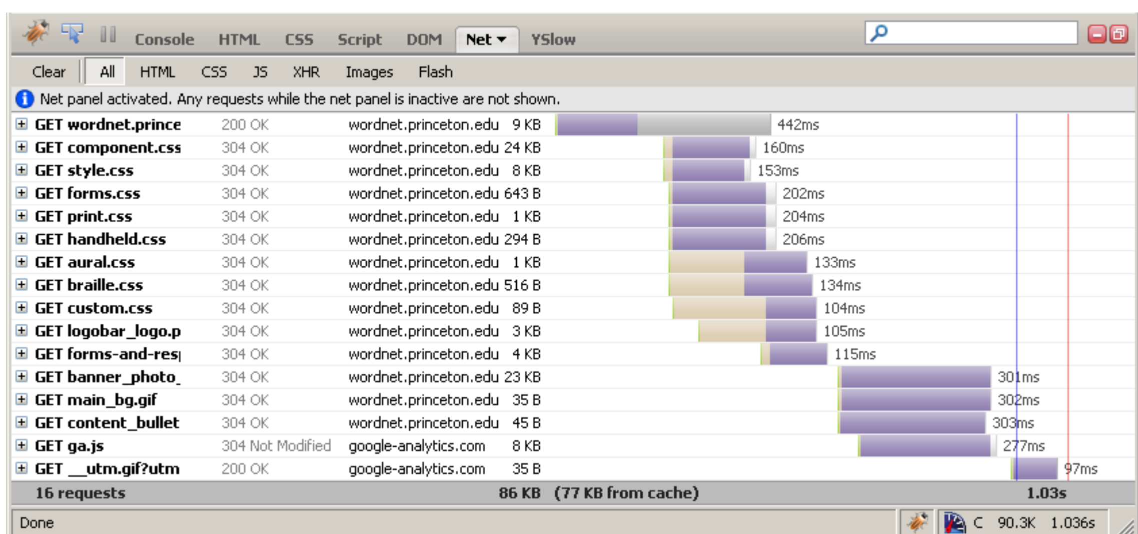

Figure 7: Firebug network statistics for the WordNet homepage, primed browser cache

We now turn our attention to YSlow. To view the YSlow grade and statistics for the WordNet homepage, simply select the YSlow tab on the Firebug console. This is illustrated in Figure 8 below. The view has been truncated for brevity. In particular, it does not show all grading criteria, though unsatisfied criteria are shown at the top.

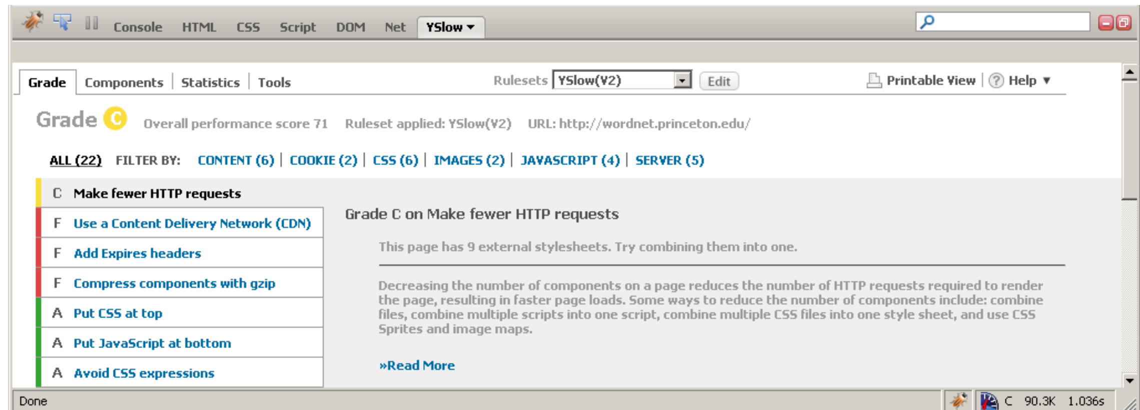

Figure 8: YSlow grade 'C' for the WordNet homepage

As can be seen from Figure 8, the WordNet homepage received a grade C, scoring 71 out of 100 points with the default ruleset (YSlow V2). While most of the tests passed with flying colors (grade A), the number of HTTP requests was considered high due to a large number of external style sheets (9). As a remedy, YSlow suggests combining the style sheets. This would in fact reduce the total number of HTTP requests by minimizing the number of referenced style sheets in the root HTML document.

Further, resources were not compressed, and none had a far-future Expires-header. As explained in section 2.1.3, compression improves performance by reducing the amount of transmitted data. A far-future Expires-header would further improve performance by eliminating the conditional GET requests that now took place. YSlow also detected that a Content Delivery Network (CDN)[16] was not used to serve static content on the WordNet homepage. Response times could be improved if these issues were addressed.

This concludes our sample performance analysis of the WordNet homepage.

[16] A content delivery network is a collection of third-party servers that host static resources (such as images and scripts) to reduce application server load. Please refer to e.g. [23] for further information.

4.4 Sample backend load test

The target of our sample load test is the WordNet online search, located at the URL http://wordnetweb.princeton.edu/perl/webwn and accessible via the WordNet homepage. The sample test is necessarily a black-box test[17], since we do not have access to the target application's implementation and/or configuration details. Section 4.4.1 describes the sample test scenario. Section 4.4.2 walks through the process of recording a test script that follows the scenario. In section 4.4.3 we run the test script with multiple concurrent simulated users. The final section provides some analysis of the test results.

4.4.1 The test scenario

The test scenario is simple: a user opens the WordNet online search page, enters a *misspelled* English word to the "Word to search for:" field and clicks on the "Search WordNet" button, only to discover that the search returns no results. The tenacious user then re-runs the search, this time with proper spelling, to obtain a non-empty result set.

4.4.2 Recording the test script

To record the test scenario, we first start up the JMeter load testing tool. Once JMeter is running, we right-click on the "Workbench" item, and select the *Add > Non-Test Elements > HTTP Proxy Server* option to enable the proxy server (pane) that is used to record a test script. This is illustrated in Figures 9 and 10 below.

[17] Black-box testing is carried out without any knowledge of the target application's implementation. [24]

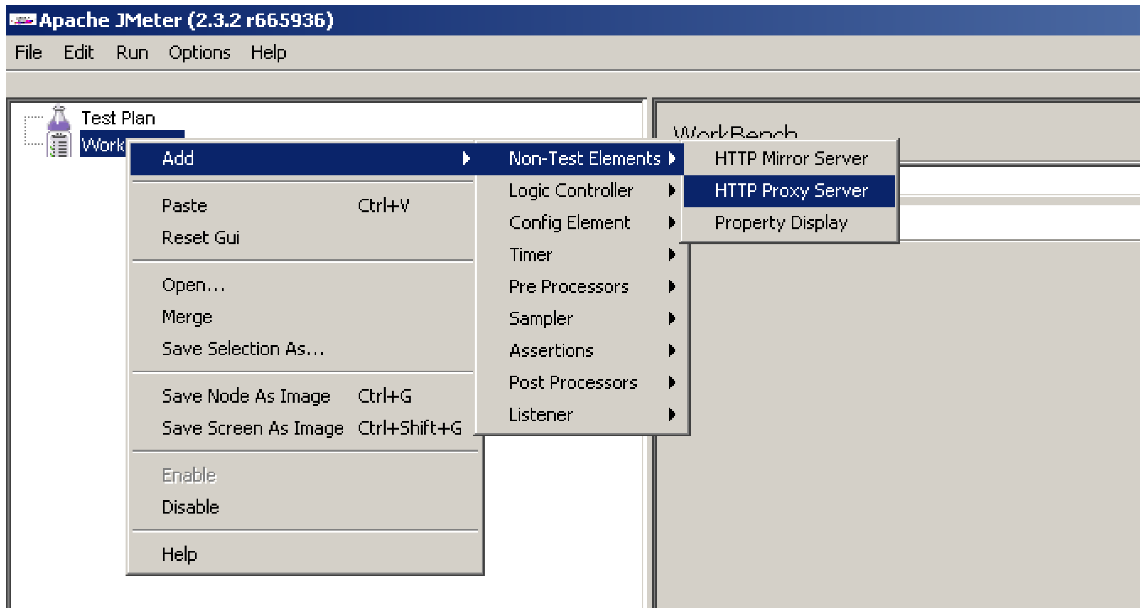

Figure 9: Enabling the HTTP proxy server pane in JMeter

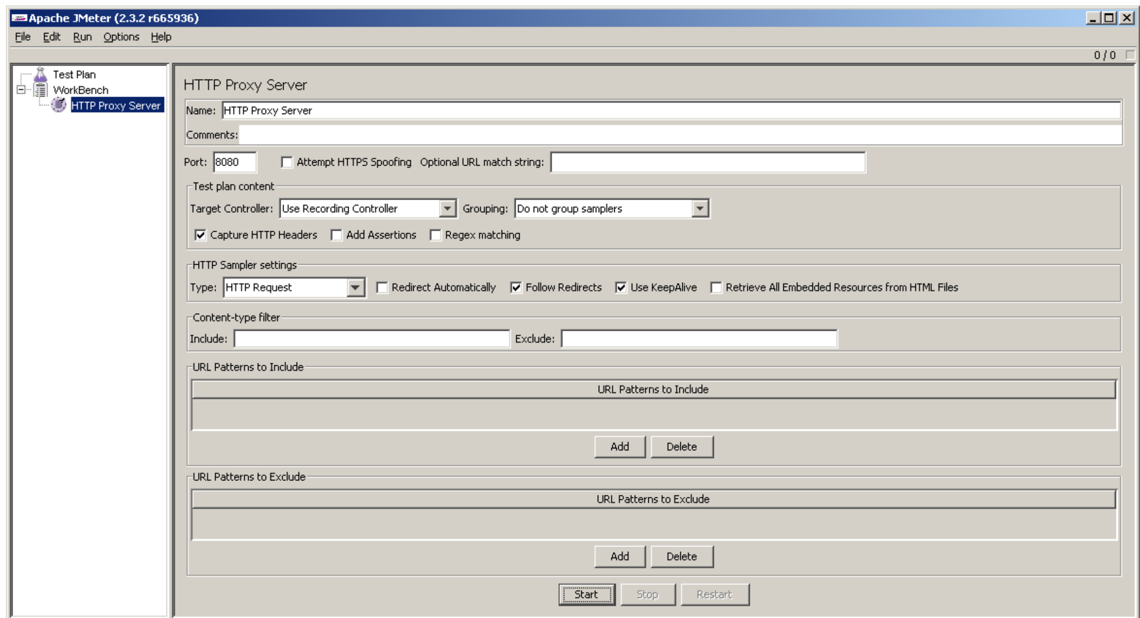

Figure 10: The HTTP proxy server settings pane, with controls at the bottom

The default settings for the proxy server (running on port 8080) are fine for our purposes. We then click on the "Start" button at the bottom of the pane to start the server. The next step is to start up the Firefox web browser. Once Firefox is up and running, we select *Tools > Options... > Advanced > Network > Connection > Settings... > Manual proxy configuration*, and enter the values "localhost" and "8080" to the "HTTP Proxy" and "Port" fields, respectively. Finally we save the settings by clicking on "OK" twice. Refer to Figure 11 for proxy configuration details.

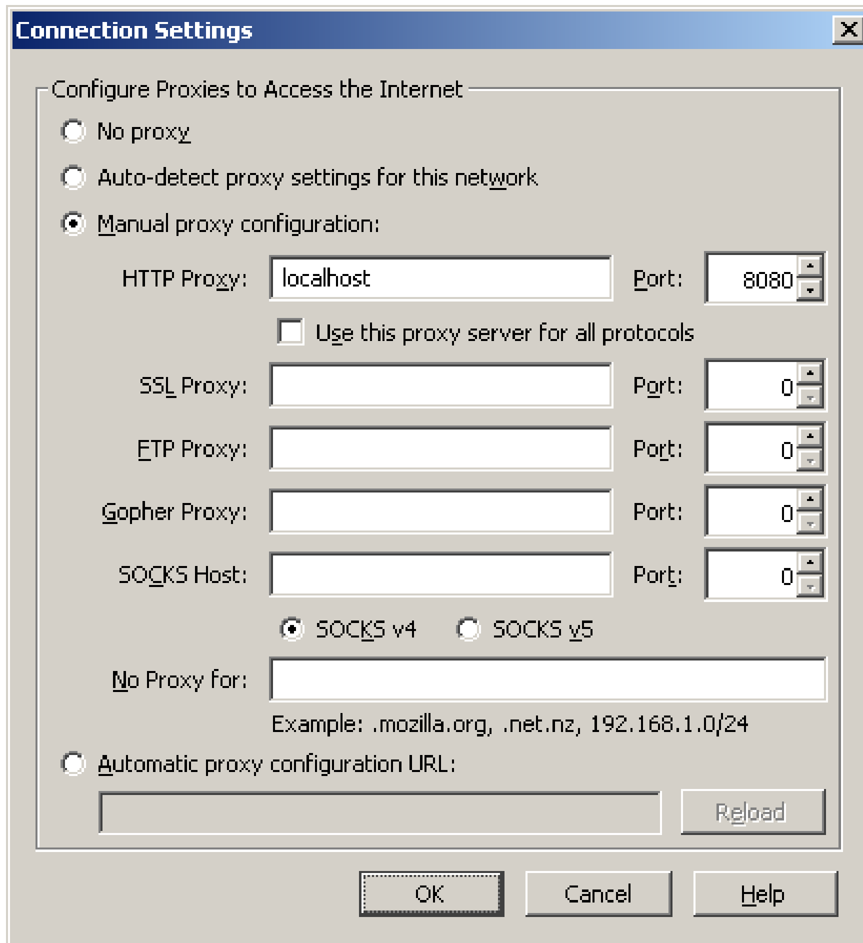

Figure 11: Configuring the JMeter HTTP proxy to use in Firefox

To record the script, we now simply execute the user actions as per the scenario, using the browser. The proxy server intercepts all HTTP requests and generates the test steps accordingly. We first open the WordNet online search form at the appropriate URL, enter a misspelled keyword ("duk") and run a search. Because the initial search returns no results, we then then correct the spelling of the keyword ("duck") and run the search again. These steps are illustrated in Figures 12, 13 and 14 below.

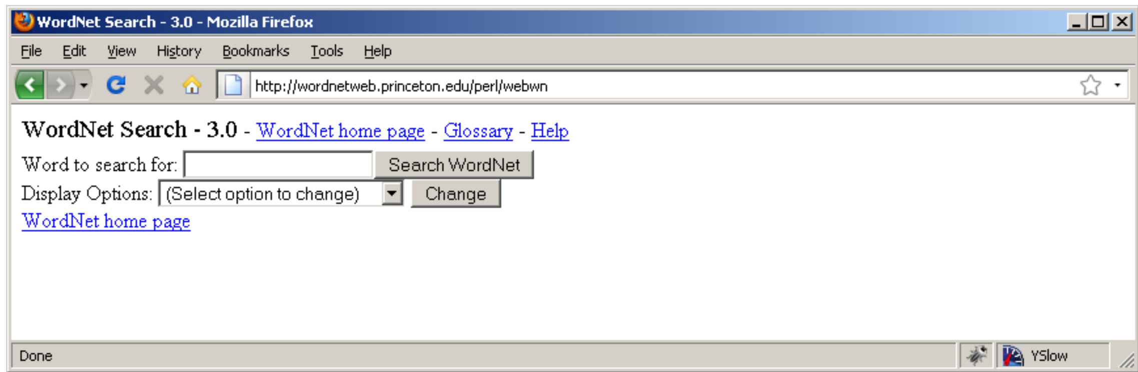

Figure 12: WordNet online search form

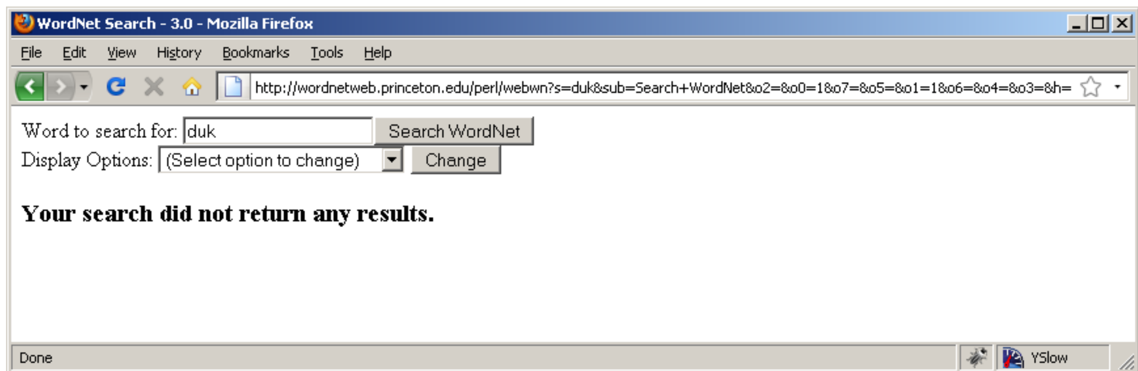

Figure 13: Empty result set for the initial query with misspelled keyword

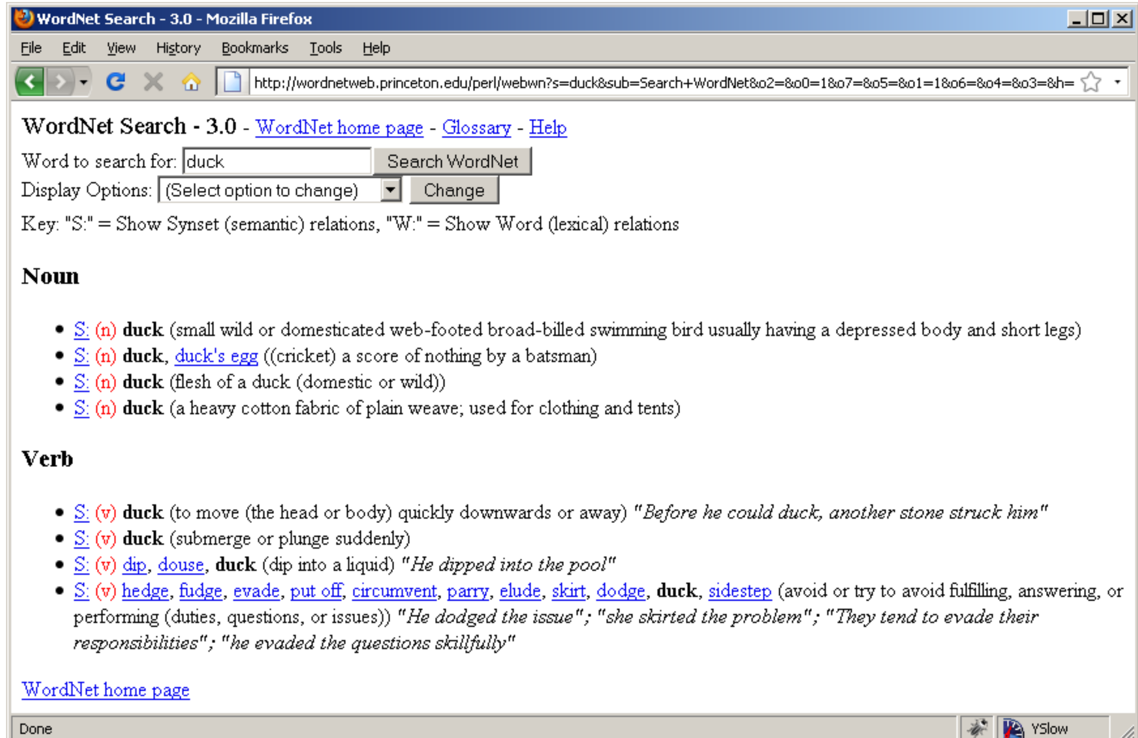

Figure 14: Non-empty search results for the second query with correctly spelled keyword

Once the above steps have been completed, it is necessary to stop the proxy. This is

accomplished by clicking on the “Stop” button in JMeter’s proxy server settings pane. As a result, three GET requests (i.e. transactions, see section 3.1.2) are generated and shown under the “Workbench” item, in the order in which they were dispatched. The first GET request was used to load the search form, whereas the second and third requests represent our failed and successful queries, respectively.

Figure 15 illustrates the resulting view with the first recorded request highlighted. As shown, JMeter allows us to modify the details of a recorded request. We will later utilize this capability to enable user-specific input.

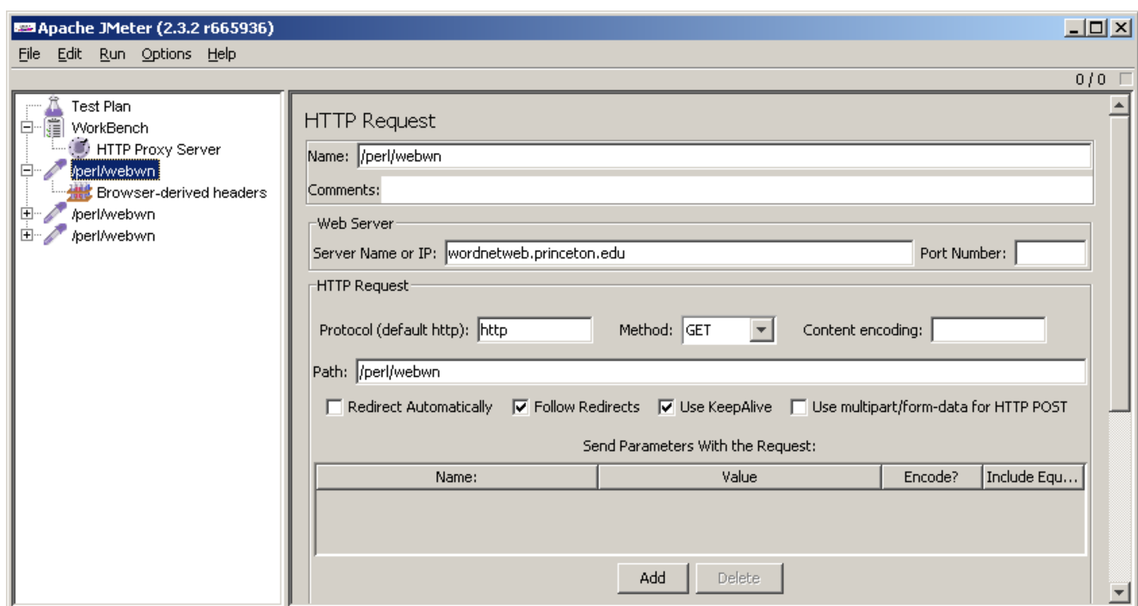

Figure 15: HTTP request settings pane for the initial request that loads the search form

Test script recording is now complete. In the next section we see how the recorded requests are configured to load test the target application with multiple simulated users.

4.4.3 Executing the test script

The sample load test is to be run with *five* concurrent users, *two* runs, and with a different keyword-pair (misspelled and correct word) for each user. To accomplish this, we must add necessary configuration items and the recorded requests to a test plan. We begin by adding a *User Parameters* element to the plan. This element is used to provide user-specific input, in this case the search keywords. To add this element, we right-click

on the “Test Plan” element and select *Add > Pre Processors > User Parameters*. This brings up the User Parameters settings pane. We then click on the the “Add Variable” button twice to add a row for each variable (misspelled word, correct word) and finally click the “Add User” button five times to add columns for each simulated user. We then enter the following values (see Table 1 below) to the fields.

Variable name	**User 1**	**User 2**	**User 3**	**User 4**	**User 5**
correct	brick	girl	moon	fight	bomb
misspelled	brik	gilr	muun	figth	bomp

Table 1: User-specific input

The variable names (in italics above) are later used to refer to the values when we setup the corresponding requests. The pane should now resemble that of Figure 16 below.

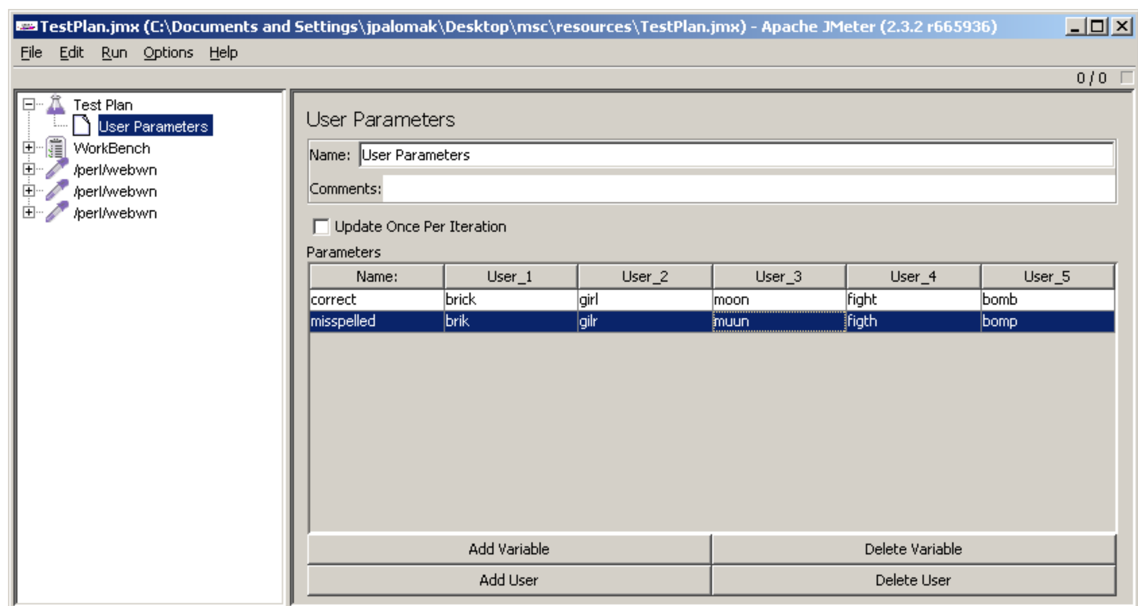

Figure 16: The User Parameters pane with user-specific input values entered

To enable test execution, and to setup the desired number of users and runs, we add a *Thread Group* element (i.e. a user group, see section 3.1.2) to the test plan. This is accomplished by selecting *Add > Thread Group* from the test plan’s context menu. We then input the desired number of concurrent users (5), runs (2), and the ramp-up period[18] (5) to corresponding fields. The resulting view is shown in Figure 17 below.

[18] The ramp-up period controls the rate at which simulated users start their scenario. In this case, users start to execute the scenario at five second intervals. This prevents a sudden increase in application load.

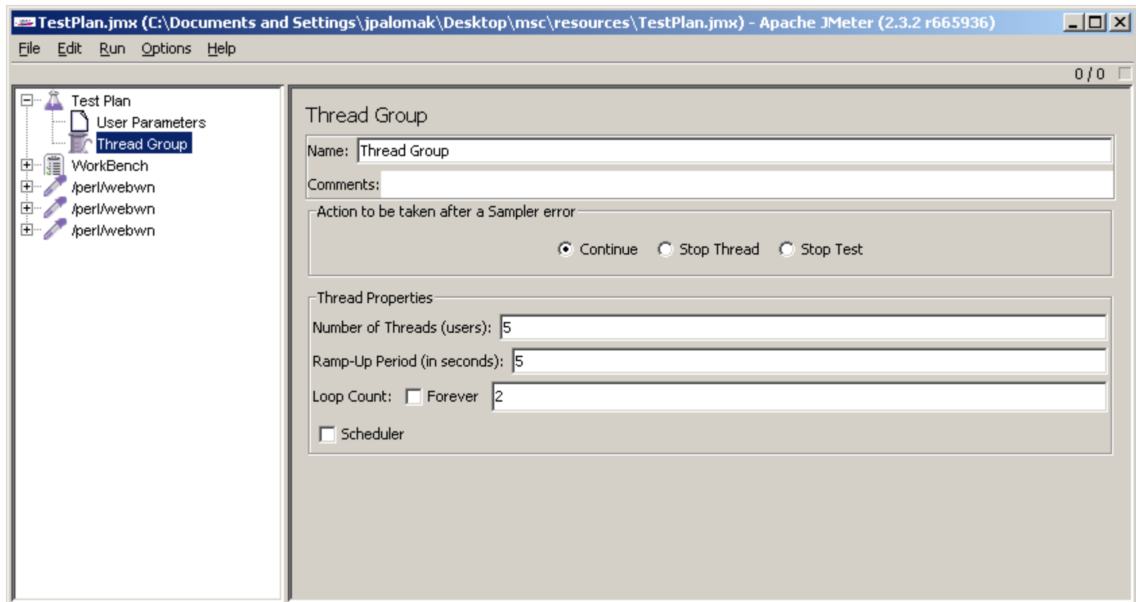

Figure 17: Thread group setup

We must now add the recorded requests to the test plan. This is done simply by dragging the requests to the thread group and selecting the "Add as Child" option. While doing this, it is important to maintain the order of the requests. As we assign the requests to the thread group, we also rename the requests to make it easier to identify them. Renaming a request simply involves changing its "Name" attribute via the corresponding request settings pane. The setup should now resemble that of Figure 18.

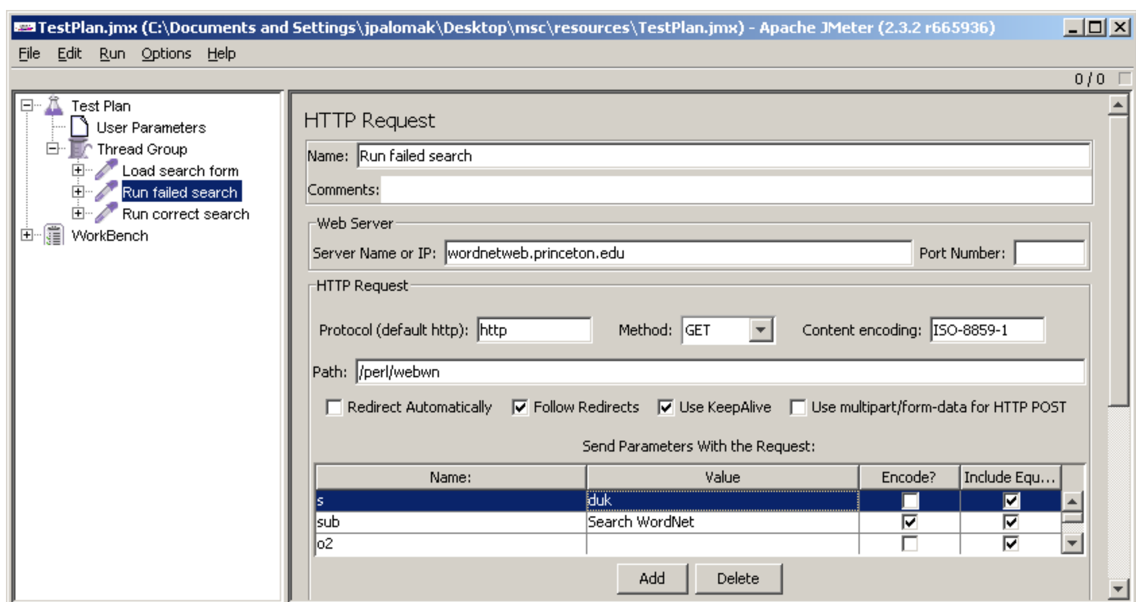

Figure 18: Request assigned to thread group, note the modified 'Name' attribute at the top

To enable dynamic, user-specific input, we must now set the value of the request

parameter, that corresponds to the search keyword, to refer to the User Parameters we defined earlier. The appropriate parameter “s” is highlighted in Figure 18 above (under the “Send Parameters With the Request” section). To pass in a dynamic search keyword at run-time, we simply replace the recorded parameter value “duk” with the expression `${misspelled}`. This expression refers to the variable defined in Table 1. We then repeat the procedure for the correct search request, replacing the parameter value “duck” with the expression `${correct}`. This is illustrated in Figure 19 below.

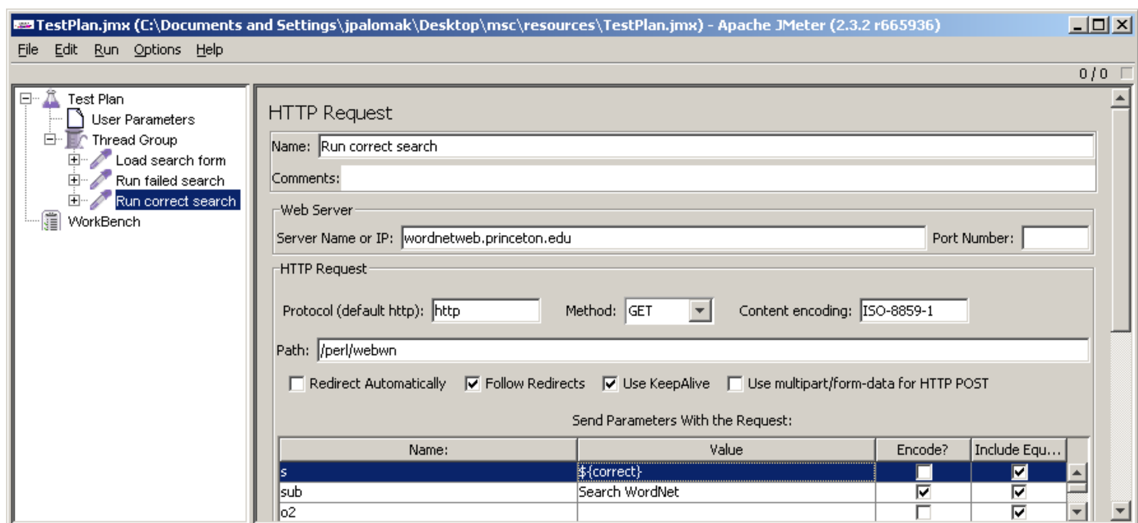

Figure 19: Modifying a request parameter value to refer to the appropriate user parameter

JMeter is now able to supply the dynamic parameter values with requests at test run-time. A final step is to add a listener to the test plan. A listener tracks test execution by saving relevant metrics, and allows us to visualize test results once the test has been run. Out of the box, JMeter provides a number of default listeners. We select the “Statistical Aggregate Graph” listener that is available as an external plugin[19]. This listener produces a nice aggregate graph with average response time and throughput.

To add the listener, we select the *Add > Listener > Statistical Aggregate Graph* option from the test plan’s context menu. The final setup is illustrated in Figure 20 below. Now that the test plan is complete, we start the load test by selecting the test plan item and pressing CTRL + R on the keyboard, or by selecting *Run > Start* from the top menu. Test execution can be tracked by monitoring the “gauge” label at the upper right corner

[19] See http://rubenlaguna.com/wp/better-jmeter-graphs/

of the screen (see Figure 20). Once the label turns grey and shows a value of “0 / 5”, the test has run to completion. The next section discusses basic result analysis with JMeter.

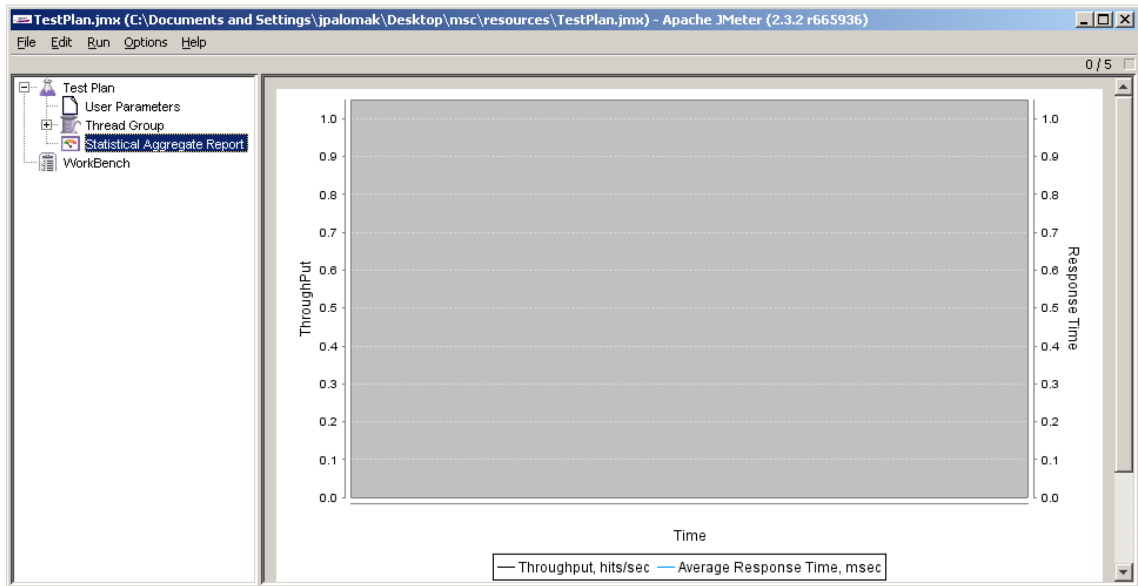

Figure 20: Final JMeter test plan with the statistical aggregate report listener added and visible

4.4.4 Analysing the test data

Once the test has run to completion, we can visualize the results by selecting the “Statistical Aggregate Report” item. The resulting view is shown in Figure 21 below.

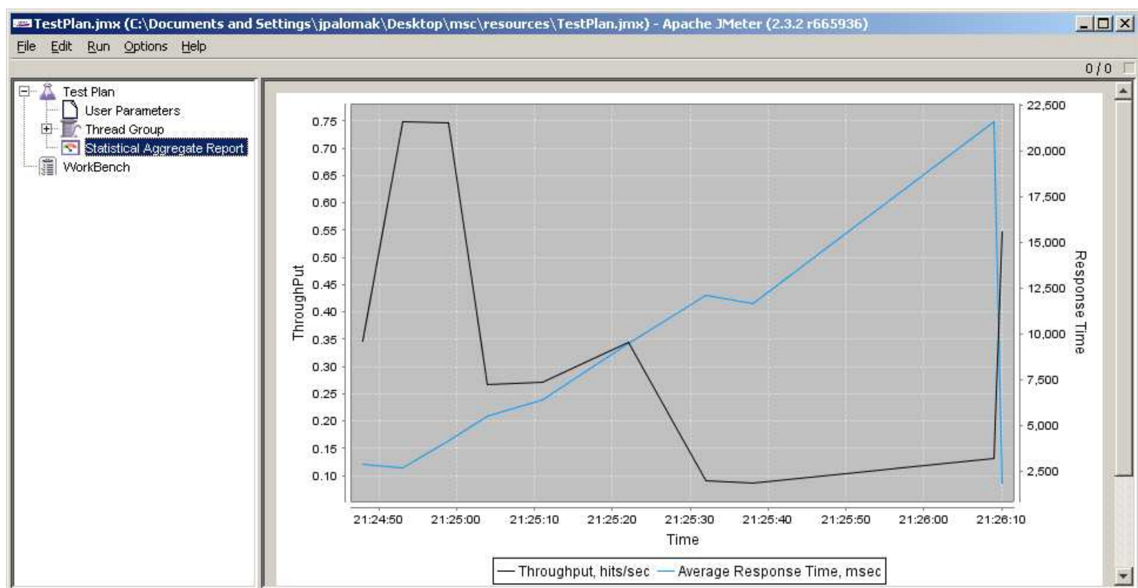

Figure 21: Test results as visualized by the statistical aggregate report

Looking at the graph, we see that the average response time was rather poor at 12

seconds, with values ranging from approximately 3 seconds to 22 seconds. Throughput (the number of requests completed per time unit) was equally poor, averaging at a mere 0.4 transactions per second (or 24 transactions per minute). These observations confirm what can be seen by using the application with a browser; the WordNet online search does not generally perform very well, especially in the face of multiple concurrent users.

This concludes our sample load test of the WordNet online search.

4.5 Remarks

The previous sections covered typical performance test implementation tasks through real-life examples. The two main aspects of web application performance testing, namely frontend analysis and backend load testing, were covered. Our discussion was of necessity simplistic, and only provided examples for a subset of the previously discussed testing practices and tool features. With these considerations in mind, the interested reader is encouraged to further evaluate these tools and practices for him/herself. For this purpose, the sample tests provide a good starting point.

5 CONCLUSIONS

Through reading the previous four chapters, the reader should now possess a basic understanding of the intricacies of web application performance testing. In the following paragraphs we summarize some key findings, and provide pointers for future study.

In chapter 2, we identified the factors that determine web application performance. At the client-side, performance is primarily driven by the amount of data transmitted over the wire. At the server-side, selection of programming language and platform, implementation and configuration are the primary contributors to application performance. The performance effects of modern platform-architectural models such as cloud computing are of particular interest, and represent a viable topic for further study.

Chapter 3 walked through the process of load testing a web application. We covered test definition, design, execution and reporting issues, with an emphasis on practical implementation. Performance testing was found to be an activity that requires delicate coordination between project stakeholders, developers, system administrators and testers, in order to produce reliable and useful results. Proper test definition and design are of utmost importance. Baselining allows for application performance to be tracked over time and across builds and versions.

Chapter 4 introduced three performance testing and analysis tools (Apache JMeter, Firebug and YSlow) that can be used to realise effective web application performance tests with minimal overhead. However, several alternative tools (commercial and open-source) also exist. A comparison between the tools presented here and their alternatives would also prove to be an interesting continuation to this paper.

Chapter 4 also provided a detailed walkthrough of typical performance testing tasks, using the tools mentioned above. Our sample frontend analysis identified some typical performance bottlenecks (lack of response compression and proper caching) in the sample application. The sample load test conveyed the key ideas and test phases of

chapter 3 in a minimal but pragmatic manner. The target application was found to perform poorly even under the moderate load incurred by our sample load test. The results must be taken with a grain of salt, however. Since the tests were run in a black-box manner, no guarantees can be made about the validity of the results, since external noise from other users (that may very well number in the thousands, for all we know) cannot be eliminated.

While this paper considered Ajax and other RIA technologies only briefly, future web applications will increasingly utilise Ajax in their implementation. This means that performance test tools and methodologies will also need to evolve. In particular, the traditional request-centered (get root HTML page, get image, get stylesheet etc.) approach to performance testing may need to evolve towards a more user-oriented approach (open search page, type text into field, click submit, etc.).

In particular, use of functional test tools such as Selenium [25] or WebDriver [26] can help the performance test designer to build more intuitive test scripts that lift the level of abstraction from the HTTP request to that of a single user. This approach can greatly reduce the complexity of the test scripts, rendering performance testing a less daunting and less time-consuming process.

BIBLIOGRAPHY

[1] Subraya, B.M. 2006. Integrated Approach to Web Performance Testing – a Practitioners Guide. IRM Press, United Kingdom.

[2] Souders, Steve 2007. High Performance Web Sites – Essential Knowledge for Frontend Engineers. O'Reilly, United States.

[3] J.D. Meier et al., 2007. Performance Testing Guidance for Web Applications. Microsoft Corporation, United States.

[4] HTTP specification (RFC2616). ftp://ftp.isi.edu/in-notes/rfc2616.txt

[5] HTTP (Wikipedia). http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[6] HTML (Wikipedia). http://en.wikipedia.org/wiki/HTML

[7] Software performance testing (Wikipedia). http://en.wikipedia.org/wiki/Software_performance_testing

[8] Client-server model (Wikipedia). http://en.wikipedia.org/wiki/Client-server

[9] W3C, 2004. URIs, Addressability, and the use of HTTP GET and POST. http://www.w3.org/2001/tag/doc/whenToUseGet.html

[10] Apache JMeter. http://jakarta.apache.org/jmeter/

[11] Firebug. http://getfirebug.com/

[12] YSlow. http://developer.yahoo.com/yslow/

[13] Microsoft support article 208427. http://support.microsoft.com/kb/208427

[14] W3Schools. http://www.w3schools.com/

[15] Redirect After Post. http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPost

[16] GZIP file format specification. http://www.faqs.org/rfcs/rfc1952.html

[17] Cloud computing (Wikipedia). http://en.wikipedia.org/wiki/Cloud_computing

[18] Open source performance testing tools. http://www.opensourcetesting.org/performance.php

[19] Software profiling (Wikipedia). http://en.wikipedia.org/wiki/Software_profiling

[20] BeanShell. http://www.beanshell.org/

[21] WordNet. http://wordnet.princeton.edu/

[22] Java. http://www.java.com

[23] Content delivery network (Wikipedia).
http://en.wikipedia.org/wiki/Content_delivery_network

[24] Black-box testing (Wikipedia). http://en.wikipedia.org/wiki/Black-box_testing

[25] Selenium. http://seleniumhq.org/

[26] WebDriver. http://code.google.com/p/selenium/