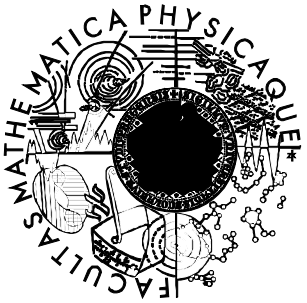
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Rastislav Wartiak

Performance Testing Tool for Web Applications

Department of Software Engineering

Supervisor: RNDr. Tomáš Kalibera, Ph.D.

Study Program: Computer Science, Programming

2007

I would like to thank my supervisor, Tomáš Kalibera, for numerous pieces of advice, corrections, and the time he spent while guiding me during the writing of this thesis.

Furthermore, I would like to thank my colleague, Zbyněk Šlégl, who gave me the idea for this thesis, and my friend Charles A. Dunbar, who reviewed the text.

I hereby declare that I wrote the thesis myself using only the referenced sources. I agree with lending the thesis.

Prague, May 29, 2007 Rastislav Wartiak

Contents

Název práce: Nástroj pro výkonnostní testování webových aplikací
Autor: Rastislav Wartiak
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Tomáš Kalibera, Ph.D.
e-mail vedoucího: tomas.kalibera@dsrg.mff.cuni.cz

Abstrakt: Během vývoje webových aplikací je splnění výkonnostních požadavků klíčové pro akceptaci aplikace zákazníkem. Výkonnostní testování je proto důležitou součástí jejich vývoje. Tato práce se věnuje implementaci nástroje pro výkonnostní testování, který by umožnil testování webových aplikací popsaného pomocí skriptů, v úrovni porovnatelné s dostupnými komerčními nástroji a s možnostmi, které nejsou u těchto aplikací běžné, jako například automatické generování hlaviček požadavků nebo různé větvení provádění skriptu. Požadovaná funkcionalita byla odvozena z analýzy vybraných webových aplikací a implementovaný nástroj umožňuje simulovat uživatele pracující s těmito aplikacemi. Toto bylo potvrzeno sérií testů na aplikaci Katastrálny portál Slovenskej republiky a testovací aplikaci RUBiS, jejichž výsledky jsou obsaženy v této práci.

Klíčová slova: výkonnostní testování, webová aplikace, zátěžové testování, měření výkonu, WebStress

Title: Performance Testing Tool for Web Applications
Author: Rastislav Wartiak
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Kalibera, Ph.D.
Supervisor's e-mail address: tomas.kalibera@dsrg.mff.cuni.cz

Abstract: When developing web applications, meeting performance requirements is vital to customer acceptance of the applications. Therefore, performance testing is an important part of their development. This thesis focuses on the implementation of a performance testing tool that would cover the functionality needed to test web applications, would be comparable with commercially available tools, would have the ability to generate load defined by scripts, and would offer functionality that is not widely offered by such tools, such as automatic request headers generation or variable script branching. The desired functionality was derived from the analysis of selected web applications, and the implemented tool is capable of simulating users interacting with these applications. This was proved by the creation of series of tests on Slovak Cadastral Portal and RUBiS benchmark site, whose results are included in the thesis.

Keywords: performance testing, web application, stress testing, benchmarking, WebStress

Preface

During the implementation of projects with large-scale web applications in Ness Czech, the company for which I work, the need arose for a performance-testing tool that would suit our needs without burdening project budgets. After reviewing some well-known open-source performance testing tools, I realized there is no such tool that would cover all the functionality we considered as important.

That is why I decided to define requirements for a tool we could use for performance testing and possibly for capacity planning in our web-application-implementation projects. All requirements and the global design were gathered in the tool specification document (47 pages), based on which I have developed a new tool. With the support of documentation (User's Guide, 48 pages, and User's Reference, 22 pages), this tool can be used as an alternative to commercial tools on some of our projects. All the mentioned documents can be found on the enclosed CD.

One of the expected uses of the tool was to test the performance of the Slovak Cadastral Portal, a web application run by the Geodetic and Cartographic Institute of Slovak Republic. This application uses modern web technologies (like AJAX), and the tool was able to simulate user load. The test results are included in the thesis.

1 Introduction

1.1 Performance testing

Webster's New Millennium Dictionary of English defines performance testing as "*testing conducted to evaluate the compliance of a system or component with specified performance requirements*." Performance testing covers a wide range of activities, and the thesis in the following text deals only with one variety of software performance-testing: web application performance testing. Other names used for similar activities are stress testing and benchmarking.

Performance testing can be carried out as a benchmark test or load test, or it can be used for capacity planning. Benchmark testing is performed with applications running on target systems (hardware and standard software) to get information about the highest load of users working with the application it can handle, to check whether it reaches the limits that were set in a contract, or it is used during the application design phase. This is a short-term test and can last up to a couple of hours.

Load tests (sometimes called soak tests) identify possible performance issues in a longer time period. Most common are memory issues that are often not detected during short-term testing. During this type of performance test, the application must be able to handle the generated load and its performance should not degrade over time.

Capacity planning depends on performance testing, and its purpose is to find out how the application can scale. Given the application and a test system, one can gather information about highest load this test system can handle and then interpolate these findings to design target system that should handle the requested target load and not spend more than is really needed.

Other types of performance testing exist, such as determining how the application can recover from a high load or a hardware failure.

1.2 Motivation

Existing web-application performance-testing tools offer a broad variety of functionality. However, none of them combines all the functionality we expected to use in our projects. This resulted in my decision to define and implement such a tool.

License fees for existing performance testing tools vary from zero to thousands of U.S. dollars. For projects that are tight on budget, the tool must be free. The only expenses allowed are for test preparation, execution, and reporting. To keep these expenses down, a test definition must be easy to maintain not only during the project, but it must be available for reuse on future projects. The tool must be able to perform as much of the common activities as possible. Such activities are request-header generation or automatic retrieval of images included in the page.

Most web applications this tool would test depend heavily on user input; therefore, the tool must offer easy manipulation of user data and requests that are sent to the application server. User input is not only data a user enters into web forms, but the selection of functionality as well. In web applications, this means clicking on links displayed on page. Not all functionality is used equally; thus, it must be possible to branch in test execution randomly, having different weights for individual actions.

Based on user input, applications generate responses, and under very high load, they can fail either by collapsing and sending nothing or only an error code or by responding with a page that has an error message inside. The desired tool must be able to parse the response and check for user-defined data. In fact, it must incorporate some functionality that covers functional testing to be able to detect the application errors. According to its finding, it must be able to branch in test execution.

Target systems running our web applications are always off-site, either at the customer's location or in hosting centers. Because clients for performance testing in such installations are mostly only servers running on Linux or UNIX, the performance-testing tool must run on such systems.

In an ideal world, everything works perfectly when the test is run. In reality, first runs often show problems in server configuration or in application itself. To identify such flaws, data gathered during the test can be helpful. In such a case, pre-defined reports and graphs might be not enough. If the tool were able to generate data for statistics software, it would be possible to mine the data and help to find the application bottleneck.

2 Project objectives

2.1 Web-application performance-testing

The primary task of a web-application performance-testing tool is to generate load. Load is represented by number of users accessing the tested application simultaneously. Because the testing tool only simulates these users, they are called virtual users. Each virtual user represents one individual user working with the application.

The interaction of such a user with the application consists of actions like clicking on a link or submitting data in a web form. The description of the user interaction that should be reproduced during the test is called a test scenario. Test scenarios can combine all actions a user would perform during the visit to the application, e.g., displaying the main page, browsing on the site, and possibly login and logout, or only small part of it, e.g., entering data into a multi-page web form.

The whole test session consists of many virtual users performing one or more test scenarios and possibly submitting some data to the application. The number of users is usually increased during the test session, and the session finishes after a defined time elapses or the application behavior will become unacceptable according to criteria defined in the test.

Another important task of a web-application performance-testing tool is to create a report of what happened during the test session after the session finishes. This report should contain information about the load the tool generated and information about the application's behavior. At a minimum, the load is described by the number of virtual users used for the generation or the number of requests sent to the application server. The application behavior is at least described by response times and number of errors encountered.

2.2 Requirements

The implemented tool will be used to simulate users working with web applications, to measure the limits of tested applications, and to test their behavior under extreme loads. The results of such tests will be used to improve the performance of the application and to prove that the implementation projects met required limits.

As this is a performance-testing tool, it has to be able to generate load and to create a report of the test session afterwards. Generating load to a web application means to simulate user browsers requesting files from the application server(s) and submitting data back to the application. What files will be requested, what data will be submitted and how the response should be handled is described in test scenarios. These scenarios must be in a form that allows versioning, i.e., storage in a development repository with the possibility to compare and merge versions.

As users have different ways of using tested applications, the tool must allow for branching in the test execution depending on the test data and the responses retrieved from the tested application. Test scenarios must allow for random branching as well.

The results of the performance testing of web applications that depend on a database are sensitive to submitted data because of caching in the database. Good load, therefore, has to contain as much various data as possible in order to force the database to use a different part of data files. Because data are submitted to the

application via web forms, the tool must be able to easily manipulate individual fields and load test data into them. This will allow the shortening of the time needed to prepare test scenarios and make them easier to read and maintain.

Reports created after the test session finishes contain information about generated load and the application behavior. This usually means couple of pre-defined graphs or lists of values. To analyze irregular patterns in test results, it will help to load gathered data into statistical software and mine the data.

Apart from general requirements, it was expected that this tool would be used to test specific web applications. One of those applications is described and tested in chapter 6.1 – Slovak Cadastral Portal. The possibility to test this web application was one of the main goals for the implementation of this tool.

A complete and detailed list of requirements can be found in the specification document in Appendix A on the enclosed CD. Functional and non-functional requirements are formulated in chapter 3 of the specification.

2.3 Interaction with web application

Even though the process of web browsing might seem easy and straightforward for the user, it covers various communications between the browser and the server.

Web browsers interact with web application servers using requests and responses. To display a page in the browser, one request and response is usually not enough. The browser has to retrieve at least one (X)HTML[1] page (or more, if frames are used) and then to retrieve attached files, as images, scripts, or styles. To speed up the process and to save bandwidth, browsers often cache files in the client's cache and store their modification dates and times.

Modern web applications go even further. When the page is displayed and user is reading the text, it is possible to send asynchronous requests in the background, without user's explicit activity. Data retrieved can be used to update dynamically the content of displayed page without the need for a complete page refresh, as was the practice in the past.

A performance-testing tool for web applications must be able to simulate such communications while keeping test scenarios simple and readable.

2.4 Overview of the following document

In chapter 3, a tool driven web application testing is analyzed, discussing possible approaches to fulfill the abovementioned requirements and defining the functionality that the implemented tool will have. Next, chapter 4 presents a simple test session, using the implemented tool, starting from the test definition up to the execution and test result presentation. The defined tool-implementation details are described in chapter 5, and the tool itself is then evaluated in chapter 6. Chapter 7 is dedicated to the description of selected other test tools and their comparison with the implemented tool. Finally, chapter 8 wraps up this thesis.

[1] (Extensible) Hypertext Markup Language ((X)HTML) – language used for the creation of web pages.

3 Analysis: Tool-Driven Web Testing

Web applications are usually designed to sustain load of hundreds or even thousands of simultaneous users. Performance testing of such applications cannot be made by real users; such tests would be very hard to manage and the results would not be reproducible. In such a situation, tool-driven tests are an option. This chapter examines what has to be done to implement such a tool and the approaches used in the implementation of the WebStress tool.

3.1 Load generation

Primary task of a web-application performance-testing tool is to generate load. Load to a web application is generated by simulating network interaction between the client browser and the application server.

Connection

Communication between a user – client browser – and an application – application server – is through a TCP network connection. The client browser connects to the application server defined in requested URL[2].

Connection closing. When connection is made, the client browser sends a request and waits for a response. The application server processes the request and sends back the response to the client browser. After the response is transferred, the connection can be closed or left open for subsequent requests. Creating a connection consumes network and application resources. If the client browser expects to send more requests to the application server, it can be faster to reuse an existing open connection for that. The most common situation for connection reuse is to request files referenced by a downloaded HTML page (called attached files in this text), such as images or styles.

The possible approaches are (1) close the connection after each request, (2) keep the connection open to request attached files and close immediately afterwards, or (3) keep the connection open for a longer time and reuse it if a user requests a new page from the same server during this time. Web browsers that implement persistent connections in HTTP[3]/1.1 or the Keep-Alive extension to HTTP/1.0, i.e. all common browsers nowadays, keep connections open. As this has an impact on the application server's performance, a testing tool should be able to keep and reuse connections to the application server. The same approach is used by default in WebStress with the possibility to close connections automatically after each request or manually as a command in the test definition.

Web application servers have limits set for the maximum time between requests and for the maximum number of requests sent using one connection to avoid problems with high number of server processes waiting for idle clients. For example, the default values for the Apache web server (version 2.0) are 15 seconds and 100

[2] Uniform Resource Locator (URL) – address of a web page, in this context

[3] Hypertext Transfer Protocol (HTTP) – communication protocol between web browsers and web servers, defined in RFC 2616 [1]

requests, respectively. If connections are kept open, the testing tool must be able to detect a closed connection after such a limit is met without reporting an error.

Multiple network connections. As another way of speeding up the process of presenting (or rendering, as it is often called) a web page to the user, web browsers can open multiple network connections. A browser can parse the response on the fly, and when only partial content is retrieved, it can request attached files. Browsers then open simultaneous connections and combine this with multiple requests using one connection, as mentioned earlier. According to RFC[4] 2616 [1], clients that use persistent connections should limit the number of simultaneous connections that they maintain to any server – not more than 2 connections. This limit is not always obeyed, and it can be usually changed in the browser configuration. The number of open connections can affect the application server's performance, so testing tools that allow multiple connections for one virtual user can better simulate real user load. Because of more complicated response parsing and connection pool management, WebStress uses only one network connection for a virtual user.

Traffic shaping. Users working with the application are connected to it either using the Internet or intranet network lines. Especially in case of the Internet, available network bandwidth may differ between real users and the testing environment. The slower the connection, the longer the response time, noticeably for larger files. As this may have an impact on measured application response times, the testing tool may limit the network transfer for individual virtual users. This is called traffic shaping. This can be implemented by measuring the network throughput for individual virtual users in regular intervals, e.g., one second. The test definition can contain the specification of the network bandwidth for individual users, e.g., as a number of kilobytes per second. When the measured throughput for newly received data would be higher than the specified bandwidth, only part of the data should be read from the network leaving the rest for later, i.e., the next measured interval.

Without traffic shaping, the testing tool connecting to the tested application by local or another high-speed network may report shorter response times than the same load generated by real users spread out on the Internet. WebStress does not support traffic shaping because the applications that it was used to test so far have short responses; thus, the impact would be low.

Request

As mentioned earlier, communication between the client browser and the application server consists of requests sent from the browser to the server and responses that are sent in the opposite direction. This communication uses the HTTP protocol, current version 1.1. This version was first published by RFC 2068 in January 1997 and then updated by RFC 2616 in June 1999. Since then, it has been implemented widely in web browsers. Version 1.0, which still is used sometimes, was defined by RFC 1945 in May 1996. As most web browsers use version 1.1, the testing tool should implement this version. In fact, not supporting this version might result in impossibility to test web applications that are set up as virtual hosts[5]. WebStress implements version 1.1.

[4] Request for Comments (RFC) – formalized memoranda addressing Internet standards
[5] Virtual hosts – multiple web applications that share the same IP address

Structure. A request is a plain-text message, finished by one empty line (CRLF) and optionally with extra user data that are sent to the application. The first line of the request specifies the method to be performed, URL, and the version of the protocol. Possible methods are GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE, and CONNECT, where only first two (or three) are used in regular communication between users and applications. The GET and POST methods are required for almost all web applications; therefore, the testing tool must implement them. The difference between GET and POST is that the first sends the user data as a part of URL, whereas POST adds the data to the end of the request. Apart from these two methods, WebStress implements the HEAD method too, which has the same header as GET.

Headers. Starting with the second line, requests can contain header fields. The request header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers. The current HTTP version defines more than 40 various header fields that control the connection, sessions, content, caching and authorization. Individual fields are discussed in the appropriate part of the text in this chapter.

User data. Optional data that can be sent with the GET or POST requests are created from the user input entered into web forms. This data consist of name=value pairs, where the names are taken from the web form in the response HTML page and the values are appropriate data entered by the user. How this data can be prepared is discussed later in this chapter.

Response

After the request is processed by the application, the response is sent back. It is similar to the request – status code, headers, empty line and the data (response body). The status code defines whether the request was processed successfully, further activity is required from the client, e.g., redirection, or an error occurred. As a minimum, the testing tool must be able to detect status code 200 (OK) to allow the test to continue in its execution, as specified in test definition. Furthermore, it should process 301 (Moved Permanently), 302 (Found), and 303 (See Other) codes to automatically request the page these redirects point to because these codes are often used in web applications. The 4xx (client errors) and 5xx (server errors) codes indicate that there is a problem with the test definition, with the application, or with the tool itself, and they should be reported to the user and/or counted as test errors.

Error detection. Correct responses (with 200 status code) contain the body with requested file, either an HTML page or an attached file. Even though on an HTTP level this means success, the application might fail in such a case. If the application caught an error or an exception, it would send a correct response that would contain a description of that problem. Deciding only on the status code, whether the request was processed successfully, can be considered only as a very basic testing tool functionality.

A more advanced level is to store the response to be available for the following test commands to inspect its content. Such an inspection can search for specific data in the response, like expected result or for a sign of an error. This search does not have to understand the meaning of the data; it can simply search for a substring or a regular expression. WebStress allows searching for a substring in the response.

Compression. To save the network bandwidth, HTTP allows compression of the response for the transfer. The client browser can inform the application server in the request headers about the methods with which it can decompress (gzip, compress, or deflate), and the server can then compress the response using one of those methods prior sending it back to the client. Web browsers usually support at least one of the compression methods and if the application server is configured to compress the responses, it will be used in communication with the real users. The compression process can have an impact on the application server performance, so the testing tool should support it. This means sending appropriate request-header fields and decompressing the data if they are offered to the test commands. WebStress supports the gzip and deflate methods.

Parsing. Responses that are HTML, JavaScript[6] or CSS[7] files can reference other (attached) files. Web browsers automatically request these attached files, unless explicitly configured not to do so. Therefore, during the performance tests, this should be done the same. This can be achieved either by including all requests of such files in the test definition or by instructing the testing tool to parse the response and request them automatically. The first possibility does not require any extra functionality from the testing tool, but the second one requires that the tool understand the data. As it can be predicted what files will be attached to the response, a testing tool that cannot parse the response will still be able to test web applications. Parsing and automatically requesting can simplify test definition creation, but the test execution will require more resources from the client – parsing the response consumes some resources. Therefore, parsing should be done only when needed and requested by the test definition. WebStress is able to parse automatically HTML responses and request attached files, if instructed to do so.

Parsing HTML responses can go even further – the testing tool can allow the test commands to access whole response as a DOM structured document. This can be useful to use specific data from the response in subsequent test execution. WebStress does not allow this.

Caching

Web applications consist of two types of content – static and dynamic. Static content is usually stored as files on the application server, and dynamic content is represented by scripts (CGI, PHP, ASP, JSP, etc.). The application server sends the modification time of static content with each response, and because it does not change between requests, it can be cached on the client. Web browsers usually do so, and when cached content if requested, the request-header field is included with the cached modification time. If the server does not have newer content, it responds with 304 (Not Modified) status code. This saves the server performance and the network bandwidth.

Caching is very common for attached files – images, scripts and styles. HTML pages are often generated, as it the core of interactive web applications. In such a case, web browsers of real users retrieve the static content only once, and subsequent responses do not contain the content again. As this may affect the application server's

[6] JavaScript – a scripting language most often used for client-side web development

[7] Cascading Style Sheets (CSS) – a stylesheet language used to describe the presentation of a document written in a markup language (e.g. HTML)

performance, the testing tool will generate a more realistic load with the ability to cache static content. Unfortunately, the combination of response parsing and caching can pose a problem: while for non-parsed content it is enough to store the modification time, a parsed response must be stored in its entirety (raw or parsed). Otherwise, it would be impossible to process a test command accessing data that were retrieved only for the first time. The basic approach would be to store data separately for each virtual user. However, as they all work with the same application, they should retrieve the same static content. This content then can be shared among them, thus limiting the amount of data stored. Each of the virtual users would still have its own cache with a list of files that it has already cached and references to the files stored in the global cache to be able to get the content of those files.

WebStress does not store the data, and because of this it supports only caching of non-parsed data, i.e., all the non-HTML data. The applications it was used to test so far generate HTML pages on the fly, so none of them can be cached. With the implementation of the parsing of JavaScript or CSS in the future, storing this data would be reasonable, but as they are not parsed yet, there is no need to store them. Yet, their modification times are stored, and during the load generation, they are sent to the application server. The tool, with respect to the application, therefore behaves as real users' browsers.

Sessions

HTTP communication is state-less by definition. Because many web applications need to track the users working with the application (sessions), three main approaches were developed: (1) hidden form fields, (2) URL rewriting, and (3) cookies[8]. This allows maintaining a relation between successive requests made to the application server by one user. The first time a user requests data from the application, a unique session identifier is assigned to him/her.

Hidden form fields. The hidden-form-fields method is used for simple web applications and servers that do not offer other methods. It requires that to keep track of a user, s/he has to navigate between pages using web forms. Each time a user submits the form, the application generates a new page containing another form (apart from regular content), with a hidden field containing the session identifier. Usage of this method is very limited and is used only to track a few consecutive requests, not the whole session. If the testing tool is able to process the web form and use the data in the following request (as WebStress is), it supports this type of session tracking.

URL rewriting. Another method for session tracking is called URL rewriting. In this method, session identifier is transferred as a parameter of GET requests. Each URL in the response that requires session tracking contains the session identifier, e.g., `page.jsp?sessionid=xxx`. This allows tracking the session without the need of web forms, even though the session identified is transferred as a web form parameter. To support URL rewriting, the test tool must either identify the session identifier automatically and append it to all requests or the navigation has to be done using parsed links from the response. If the test definition contains the name of the session identifier, automatic detection is not complicated to implement. If the testing

[8] cookie – text sent by a server to a web browser and then sent back unchanged by the browser each time it accesses that server

tool is able to parse the HTML response and allow the test definition to access the list of links in the response, URL rewriting can be supported too. WebStress uses this approach.

Cookies. A very common method of session tracking nowadays is to use cookies. This does not require any changes to the generated HTML and usually is implemented in the server code. Once a user requests the data from the application server, a cookie containing the session identified is sent in the response header. The web browser then stores it and sends it in all subsequent requests to the server. The application can then access the cookie value and identify the session and the user. A testing tool without the support of cookies will not be able to test the majority of web applications that depend on session tracking. WebStress supports cookies.

Authentication

Many web applications offer or even require the user to authenticate (log in) to work with the application. Authentication can be implemented in HTTP or at the application level. HTTP authentication is set up in the application server's configuration. If a protected URL is requested, the server responds with 401 (Unauthorized) status code and the web browser ask the user to enter the credentials. The browser will then add a request-header field with it. There are different ways of HTTP authentication – Basic, Digest, and NTLM, to name a few. Because the browser caches the credentials until closed, preventing users from logging out, the application has no control over the username and password input and optional authentication and single sign-on is not possible, so this method is not found often in interactive web applications [20].

Application level authentication is in fact a regular web form that is submitted to the application and the access is then tracked with a session identifier. If the testing tool supports web forms and the session tracking type used by the application, it can authenticate to it.

As there are different types of authentication and session tracking, the testing tool must be able to simulate the same types as the application uses. The most common type used is application level authentication, and this type is supported by WebStress.

Virtual users

A web-application testing tool simulates real users simultaneously running many such simulations. Each of them is called virtual user. These virtual users share a common test definition with each bunch of virtual users working as specified in one of the test scenarios. Because the testing tool must execute the test for all virtual users in parallel, it can depend on the multitasking capabilities of the operating system or it can implement its own "task switching."

Processes and threads. Operating systems usually offer two ways of multitasking – multiple processes and multiple threads in one process. The processes approach is easier to port between platforms, but the synchronization and communication between processes is more complicated. Multiple processes can be easier to distribute to more computers. The threads approach can save resources, e.g., by sharing memory, but its implementation usually requires extra libraries that can limit the tool portability. Communication between threads limits its usage to only one computer, but it is faster than many inter-process communication methods.

One process, one thread. Implementation using one process and one thread requires that the tool keeps track of all network connections and time-outs together. This complicates the implementation, but it can be faster than other approaches because it can wait on all events using one system call and needs no inter-process or inter-thread communication and synchronization. This approach has an important drawback: non-blocking system calls must be used and all longer-lasting actions must be cut into smaller pieces and planned separately. Otherwise the measured response times might not be accurate because the tool will process the responses later then they arrived. Another problem is that the tool will not be able to use more than one CPU core, which limits its ability to generate higher loads.

From all three possible approaches, WebStress uses threads. Multiple processes require more complicated inter-process communication and consume more resources, and the one-process-one-thread approach would be very complicated to implement, so the implementation would in fact not focus on performance testing, but on task switching and planning.

Distributed load. To generate very high loads, one computer may not be enough. Load then has to be generated using more computers and this implies more processes. All the processes have to be synchronized and the results have to be completed together. In distributed testing, one computer is usually dedicated to control the execution of other so-called agents that generate the load. The test definition can be processed at one place, e.g., user desktop computer, and then distributed to the agents, e.g., remote computers. To save the network bandwidth and reduce the CPU utilization of the agents, it may be better to store the results during the test execution locally and transfer them to the controlling process after the load generation finishes. This requires that the computer times be synchronized, e.g., using NTP[9].

Think-times. During the test execution, requests are not generated all the time – real users read the content of the pages displayed or enter the data into web forms. These pauses are called think-times. These think-times can be defined in the test definition as the same between all requests or used as specific commands in the test definition, with the user deciding when the think-time should be inserted. This definition can be static with the same think-time for each virtual user and test execution or it can be dynamic with a specified probability distribution. Distributions used by the test tools evaluated in chapter 7 are uniform, exponential, and Gaussian. Apart from these, a normal distribution can be used, especially by the tools that record the test definition by monitoring real user activity. Measured think-times can then be used as the means. Which of the probability distributions describes best the real user and should be used in the test definition is up the creator of the test. The testing tool must have a way of defining the think-time and should implement at least dynamic think-times. Otherwise, all the virtual users will keep the same pace and the load will not vary, as it would with real users.

Ramp-up period. When generating the load of hundreds or thousands virtual users, they cannot be started all at the same time. This could cause a performance problems for the tool itself, but more importantly, for the tested application itself. In

[9] Network Time Protocol (NTP) – protocol for synchronizing the clocks of computer systems over packet-switched, variable-latency data networks

real life, the load increases steadily, as new users are connecting to the application. The testing tool must be therefore able to reach the desired number of virtual users over a period of time. This controlled increase of the number of virtual users is called ramp-up period. It can be implemented by defining the final number of virtual users combined either with the length of the ramp-up period or by the increase in the number of virtual users in time, e.g., three new virtual users every two seconds. The tool would then continuously increase the number of virtual users until reaching the final number. If the test definition allows stopping the test execution after pre-defined time or when number of errors or the response time reaches a certain limit, the test can be started without setting the upper limit on the number of virtual users. This approach can be used to find the maximum number of users the application can handle, while still keeping reasonable response time or error rate.

3.2 Reporting

Another important task of a web-application performance tool is to create a report of what happened during the test session. Based on this report, it is possible to evaluate the test session and the behavior of the tested application under the load.

Measurements

A test report is created from values measured during the test session. These values either describe the load generated or the behavior of the application. While the first is controlled by the tool and the test definition, the second depends on the application and the test environment, e.g., network lines. A combination of values of both types and their possible dependency describes the application performance.

Generated load. Load generated by a performance-testing tool can be described with elapsed time since the start of the test, number of virtual users, number of open connections, number of requests sent, or number of requests simultaneously waiting for a response.

Application behavior. The basic value that describes the behavior of the tested application is response time. How it can be measured is analyzed in the following paragraph. Other values that can be measured are number of errors that occurred during the test session or size of a response. Errors can be counted by checking the status codes of the responses and by checking the response body, as described in the previous text.

Time measurement

As described in the previous text, communication between the client browser and the application server consists of network connection opening, request, response and possibly network connection closing. As the connection closing cannot be perceived by the user and is not important for him/her, it does not have to be measured. When going into more details, the following important points in time are interesting:

- **Name-lookup time:** The time it takes from the start (the moment the tool is going to request a file) until the host name of the application server is resolved.
- **Connect time:** The time it takes until the connection to the remote host is created.

- **Pre-transfer time:** The time it takes until the request is sent completely to the server.
- **Start-of-transfer time:** The time it takes until the first byte of response arrives.
- **Total time:** The time it takes until the whole response is transferred.

All these times are represented graphically in Figure 3.1. As web browsers start to display the content of the response already during the transfer when they have enough information, a user can perceive the start-of-transfer time and the total time. These two times affect the user's perception of the application and, therefore, should be measured by the testing tool. If a connection is reused, name-lookup time and connect time are zero because this was already done by a previous request.

Name-lookup time, connect time, and pre-transfer time are part of start-of-transfer time for the user, and when of one these three times increases, it will affect the times that follow afterwards. Measuring these times can help find problems in the underlying layers of the application, such as the server or operating system configuration. For example, a long connect time might indicate the test is hitting limits for open network connections or number of processes. Measuring these times is an added value for the testing tool. WebStress measures all the five mentioned times.

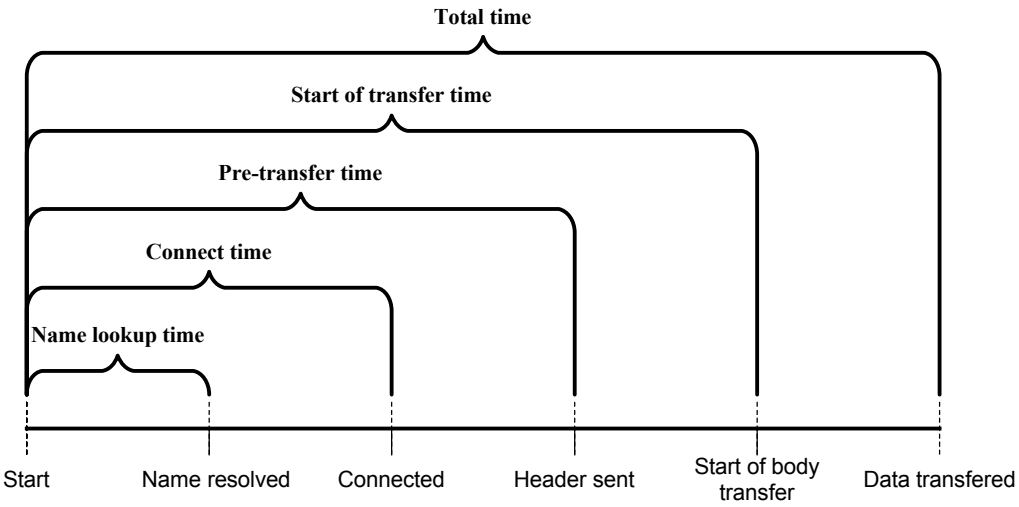


Figure 3.1: Important points in time during the network communication

Page-load time. In the previous text, time measurements were described for individual pairs of requests and responses, i.e., times from opening a connection to retrieving one response. When a web browser displays a page, it uses not only the file with the page content itself, but the attached files too. Time to retrieve the page and all attached files is called page-load or transaction time. Testing tools that parse the response and automatically request the attached files can measure this time, because of the knowledge that the subsequent requests are related to the request for the page itself. Other tools have to provide the user with specific commands that can be used to define the range of requests related together – a transaction, to be able to measure it. Another option is to use a different command to request manually attached files and the tool can then combine the request for the page and the subsequent requests for the attached files. This can, in fact, automatically define a transaction. WebStress does not measure page-load times, but automatic requesting of attached files or separate command to request them manually, which WebStress

offers, can be used to implement automatic definition of transactions. In case of need, a separate command defining a transaction can be implemented too.

Exact time measurement. Some of the times measured can be only a few milliseconds long. Even though the regular system call for time measurement, `gettimeofday`, should have the precision of 1 microsecond, at least on Windows it is about 10 milliseconds, as explained in [14] and [15]. Better accuracy can be achieved by using processor ticks – time stamp counters. With processors running at gigahertz speeds, the achievable accuracy is in nanoseconds. Many architectures offer the possibility to read these counters, as can be seen in the list of platforms supported by PAPI library – one of libraries that can be used to read the counters. WebStress uses RDTSC instruction of Intel/AMD x86 systems to read the counter or regular system call on other architectures. As many operating systems on other platforms offer accuracy of at least 1 millisecond, it is enough to use system calls to measure the time on such systems.

Time is measured internally by the operating system and system libraries for application sleeps too. When the testing tool waits a think-time period, it may sleep using a system call. Sleep-time resolution is based on the system timer resolution, e.g., 10 milliseconds for Linux 2.4 kernel and 1 millisecond for Linux 2.6 kernel (see [25]). Extra time has to be counted for the call to return, according to [26] it can last about 10-30 milliseconds for the x86 architecture. Tools that use a separate thread can easily work with a resolution of 10 milliseconds or even worse, as the think-times will probably in a rank of seconds or more. Tools that execute more virtual users in one thread have shorter sleeps, as they will wait for the time period between two consecutive requests of two different virtual users. With a higher number of virtual users, there can be a time in milliseconds, so the tool must use high-resolution timer. Possible implementations are described in [26].

Resource monitoring

The response times of the tested application include the time needed to transfer the requests and responses through the network and to process them by the tool. When the network is not congested and the computer running the tool is responding quickly, this overhead creates only a very small part of the response time. As the generated load increases, so does the network, CPU, and memory usage on the client. To avoid affecting the results of the test with the performance of the network or the client, these resources should be monitored. If the CPU, memory, or network bandwidth usage is reaching its limits, e.g., is consuming more than 80 percent of the resource, the test results may be affected and the test should be repeated with a lower load or in an environment with a faster CPU, more CPUs or memory, or a faster network connection.

The application server(s) is monitored by response times. However, if the application does not perform as expected, it can be helpful to monitor server resources too. Server monitoring can help to identify the hardware bottlenecks of the application server(s), such as the lack of the system memory. If the implementation team decides to improve application performance by a hardware upgrade, server monitoring can advise which part of the system should be upgraded.

Ways of monitoring. Monitoring can be done by the user running the test or by the tool itself. User can monitor the client or the server by operating system tools like `sar` [17] or `taskmgr`. Common way of monitoring network-attached devices is to

use SNMP[10]. SNMP exposes management data in the form of variables on the managed systems, which describe the system configuration. These variables, e.g., CPU utilization, physical and virtual memory allocation or network usage can then be queried by managing applications. As there exists an SNMP implementation for a lot of platforms (see the list of supported operating systems for Net-SNMP [18]), it can be used to monitor most web applications and test clients. Net-SNMP offers a complex library that can be used in the testing tools. WebStress does not monitor client or server resources, but its architecture is prepared to do so in the coming releases.

Content of the reports

Measured values have to be presented to the user. This can be done in a form of statistical numbers, such as mean and standard deviation, or in a form of a graph (also called plot or chart). 2D graphs can show a measured value changing in respect to another one, e.g., how the response time was changing with an increasing number of virtual users. Measured values and their division into two groups are described on page 18.

Meaningful combinations of values to display are:

- one of application behavior values versus one of generated load values, e.g., response time versus number of virtual users
- one of application behavior values versus one of resource monitoring values, e.g., number of errors (application time-outs) versus server CPU utilization
- one of resource monitoring values versus one of generated load values, e.g., server memory usage versus number of open connections
- one of generated load values versus another generated load value, e.g., number of virtual users versus test execution time

Values describing a possible effect versus a possible cause should be combined because as the report is looking for relationships and correlations between measured values.

Graph types. Common graph charts (according to [19]) are a scatter plot, a histogram, a bar chart, a pie chart, and a line chart. Another graph useful in statistics is a box plot. Relationships that change continuously, e.g., the number of virtual users versus test-session time, can be displayed using a line graph. Such a visualization can reveal physical relationships, and one can quickly and intuitively understand the behavior described by the data. In contrast, discrete measurements are better displayed using a scatter plot, histogram, or box plot. Box plots in particular are useful for displaying statistical information about the data, such as smallest observation, lower quartile, median, upper quartile, and largest observation, as for example for the response time versus test session time.

3.3 Test definition

Environment and language

The load to the tested web application is generated based on a test definition. A test definition is a formalized test scenario, i.e., a sequence of actions and decisions that

[10] Simple Network Management Protocol (SNMP) – protocol used by network management systems to monitor network-attached devices for conditions that warrant administrative attention

describe the user working with the application. This formalization transforms a test scenario into a form readable by the testing tool; it creates a test definition.

A test definition can be created explicitly by the user using a language or edited graphically, created by monitoring sample user activity as a proxy, or created from a stored application server log. Explicit definition can leverage all the load-generation functionality a testing tool can have: it can define requests, check the response, branch, use variable test data, and others. However, preparing a test definition this way can take more time because the user has to learn the syntax of the language or the usage of the test-definition part of the tool.

Monitoring user activity or using the server log from the past creates a test definition that merely replays a fixed test with no possibilities to vary. Usage of such a tool is simpler and can be used for basic performance tests. To create a test definition based on log of an application using URL rewrite mechanism, the tool has to identify the session identifier in the URLs. Another problem is that servers do not log the user data for POST requests by default because this may reduce the server's performance and pose a security risk.

To overcome the limitations of both approaches, the testing tool can combine the functionality of the both. The user is then able to create the draft of the test definition by monitoring the real activity or using the log and then manually completing it with other commands, as a response content check. The problems of such an approach can be that the tool does not know which of the data gathered is important for the test and the created test definition can then be very detailed and thus complicated to read and maintain.

Language or a graphical test definition interface. A test definition can be represented as a text in a defined language (script) or as commands entered and displayed graphically by the tool. The text form can be maintained in any text editor and easily stored and versioned in a software repository, but it is harder to learn to work with the tool. The language for the script can created specifically for the testing tool itself or extensions (libraries) to an existing language can be created. When using an existing language, it is enough to implement the libraries offering all the testing tool commands, and, when combined with internal or other external routines, the created test definition can be compiled with the language's native compiler into separate executable or dynamically linked to the tool. As this approach is very complicated to implement, testing tools usually use their own language and process the test definition internally (see Table 7.1: Web test tools comparison matrix).

Creating and maintaining the test definition graphically by the tool can offer the user support during the process – he/she can browse in a list of available commands and their parameters. The disadvantage is that the user does not see all the details of the test definition in one piece, but has to go through individual commands. Such test definitions can be edited only by the tool itself and it is hard to version them or merge two test definitions together. Again, it is possible to take the good parts of both approaches, and the graphical interface can read and write the test definition from/to a script.

Testing tools that can only generate the load from a server log do not have to have any specific test definition because the log itself is the definition. Tools that can only monitor the activity and replay the actions without any possibility for the user to alter the created test definition can use internal (binary) formats.

WebStress defines the test definition by a script with its own language. One of the goals before the implementation was the possibility to store the test definitions in

a software repository and to allow concurrent changes to it with merging. A graphical interface that reads/writes the script can be implemented that would be compatible with current format and syntax.

Test data

Users working with web applications read the data retrieved from the application and often send back user data too. This data can be for example login information or queries. To send (or submit) the data, users enter them into input boxes or use list boxes, check buttons, and radio buttons. All these elements are grouped together in web forms.

Therefore, the testing tool has to send user data. During the test, the tool can either send the same (static) data all the time or send different (dynamic) data with every request. Static data can be defined directly in the test definition, but not dynamic data. Static data can be used in cases when the application performance does not depend on the data (user name, comments, etc.). Using static data for queries could result in irrelevant test results because caching would probably apply on either a database or a file system level. The tested application could then perform very well during the tests, but not in the production with real users.

Dynamic data. To get relevant results in performance tests, dynamic data is needed in most cases. Therefore, data for queries must be prepared in advance, for example, from the application data, i.e., its database. This preparation can be in form of a text, such as CSV or XML, a binary file that is read by the tool during the test execution, or an SQL query to the database. If a database is used, it should not be the same database the application uses or running on the same system because it may affect the performance of the application and, therefore, the test results.

Sometimes, the applications performance does not depend on the data submitted, but static data cannot be used. This can happen when the application requires unique input from the user every time. Prepared data, as in the previous paragraph, can be used, but they have to be prepared for every test again. In such situations, the testing tool may generate the data automatically. A user can define a template and the tool will generate the input based on it. The simplest form of a template is a sequence of numbers to be generated. When combined with a static string, the tool can, for example, generate user names to be registered in the tested application.

WebStress supports static and dynamic data in CSV format. Parsing XML files takes more time and can slow down the tool performance when reading the external file during the test execution. In the test definition, it is possible to specify, whether the data should be loaded into memory or read from the file all the time, and in the second case, this could cause problems. Using a binary format can cause problems for the users to create it, so it is not used. Large sets of data probably would be probably from the application database, and it is not possible to create binary output with standard SQL queries.

Application-generated data. One type of data sent from the application to the user and then sent back to the application has been previously mentioned, i.e., the session identifier. A web application can send other data to the user that it expects to be returned back, e.g., when the user enters data into one page of a web form and then moves to the next page. In such a case, the application can use hidden form fields and temporarily store the data until all the required data are entered. A testing tool that is not able to parse the response from the application and retrieve those data

to be used in subsequent request cannot be used to test such an application. The testing tool must either allow the user to get the data during the test to prepare the request or automatically prepare the request using the hidden fields and default values of fields that are displayed. This second approach, implemented in WebStress, allows the user to concentrate on user-entered data only and leave the work that is normally done by a web browser to the testing tool.

4 Basic Usage Explained on Example

4.1 Description

The following is a very simple test to show the usage of WebStress' basic commands for the test definition. Detailed usage of WebStress is described in User's Guide in Appendix B and User's Reference in Appendix C, both on the enclosed CD.

Test consists of one scenario and this is intended to do the following:

- Test the performance of the Google web server, by continuously fetching its main page and then performing a query. Queries use a list of predefined words.
- Images, scripts, and external styles are fetched too, if referenced by the retrieved page.
- Test starts with three virtual users, and each second one new virtual user is added.
- When the virtual user execution finishes, it starts over again.
- Test execution finishes after 10 seconds or after 60 requests sent to the server.
- Test report contains two graphs – server response time in test execution time and distribution of response times of the main page and the query.

Test scenario is divided into two states. First state requests the main page and the second state performs the query and checks the response.

State	Description	Next state
START	Wait for 0-1 seconds, get main page.	SEARCHFOR
SEARCHFOR	Wait for 0-2 seconds, randomly select from a list of prepared words and perform a query. Check that response is correct.	

The description of the test is then transformed into the test script and the test configuration. The test script contains commands for the virtual users, e.g., requests that should be made. The test configuration defines the test limits, e.g., the number of virtual users and how long should be the test executed, and the graphs that should be included in the test report.

4.2 Test script

The test script is a test scenario transformed into a language that can be processed by the tool.

```
// file name: test-google.script

// all URLs in this script will be prepended with Google address
BASE_URL "http://www.google.com/"
// images, scripts and external styles will be fetched automatically
SGET TYPE AUTO

// test data, randomly selected for each query
DATA QUERY RANDOM
{
```

```
  Q

  "music"
  "sport"
  "nature"
}

// every script execution starts in state START
STATE START
{
  // wait for random time between 0 and 1 seconds
  WAIT RANGE 0 1
  // fetch main page, English version
  GET "/intl/en/"
  // when page is retrieved, continue in state SEARCHFOR
  NEXT STATE SEARCHFOR
}

// will perform query in this state
STATE SEARCHFOR
{
  // wait for random time between 0 and 2 seconds
  WAIT RANGE 0 2

  // get data for the query
  LOAD DATA QUERY INTO d_query
  // parse last retrieved page and prepare web form to submit
  LOAD FORM
  // enter query into the form
  ENTER FROM d_query USING q
  // submit the form to the server
  SUBMIT FORM

  // check that response contains specified string
  // execute code in the block if not
  NSEARCH " seconds)" {
    // log server response, can be useful to check what happened
    LOGDATA
    // finish the test and report an error
    FAIL
  }
  // finish the test and report success
}
```

4.3 Test configuration

The test configuration defines test-session parameters, i.e., test scripts used, number of virtual users, how long the test should be performed, and what should be included in the test report. Generally, more scenarios can be tested simultaneously and the test configuration would contain a reference for each of the test scripts describing its scenario. Each of the scripts must have its own definition of the number of virtual users.

```
// file name: test-google.conf

// use defined file as a script for test
SCRIPT[]=test-google.script
// start with 3 virtual users and add one more every second
LOAD[]=START 3 INCREASE 1 IN 1
// restart finished scripts
```

```
RESTART=YES

// stop after 60 requests sent
STOP_REQUESTS=60
// stop after 10 seconds
STOP_TIME=10

// create a test report with specified name
PAGE=test-google.html

// create a graph and include it in the test report
// use results from all scripts used in this test
TESTS[]=ALL
// horizontal axis will contain test execution time
AXIS_X[]=TIME
// vertical axis will contain server response time
AXIS_Y[]=RESPONSE_TIME_ALL

// create a graph and include it in the test report
// use results from first scripts only
TESTS[]=1
// horizontal axis will contain test script states
AXIS_X[]=STATES
// vertical axis will contain server response time
AXIS_Y[]=RESPONSE_TIME_ALL
```

4.4 Test run

Once the test configuration and the test scripts are prepared, the test can be executed. This example is started as:

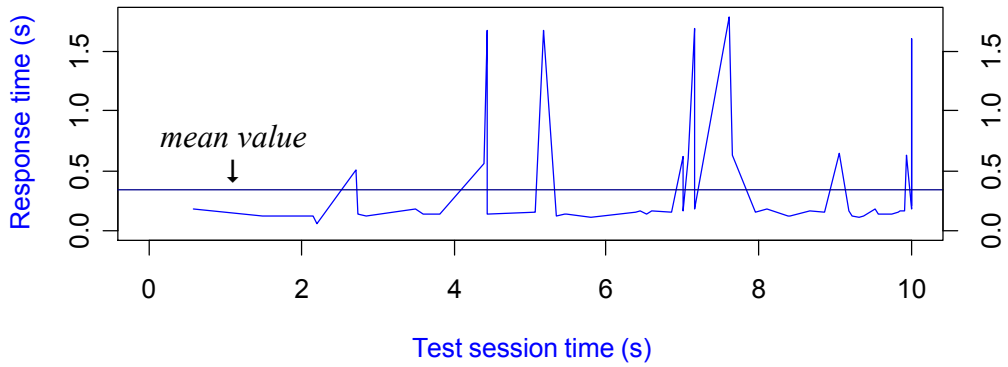
```
./webstress -s test-google
```

The tool displays the information about the test definition, the test scripts, and the test data it uses, and if everything is correct, the test execution starts. During the test, the tool displays information about the virtual users that started and finished and reports all errors. All this information is stored in the log too.

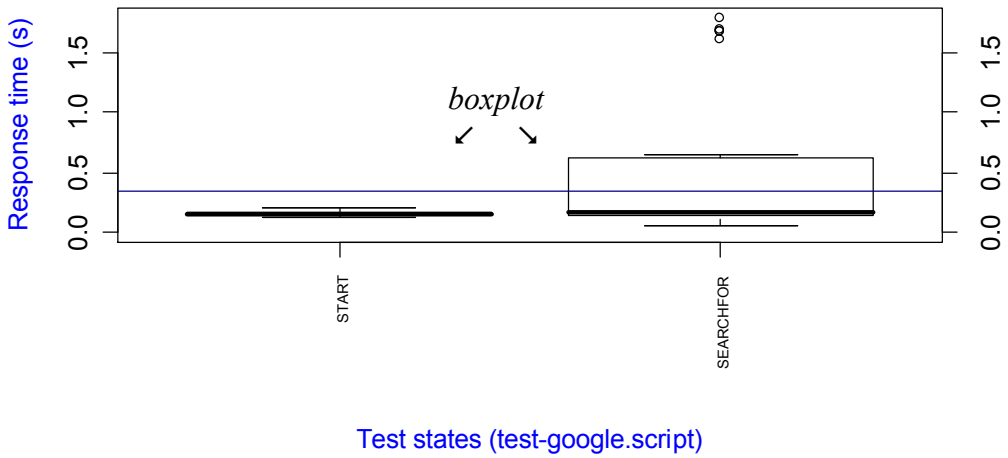
4.5 Test report

At the end of the test session, the tool creates input files for the R statistical tool and runs it. The created graphs are then automatically included in the test report. Once the tool finishes, it is possible to open the HTML test report in the browser. The content of the report from this example test session follows.

Configuration file: test-google.conf ← *configuration used for the test*
Test start: Sat May 12 12:40:23 2007 ← *test start time*
Test finish: Sat May 12 12:40:35 2007 ← *test finish time*
Finish reason: TIME ← *reason, why the test finished*
Running time before stop: 10 seconds ← *time before stopping the test*
Total requests: 43 ← *number of requests sent to the server*
Total errors: 0 (0 %) ← *errors encountered (+ ratio to requests)*
Total runs: 28 ← *number of the script executions*
Total users: 12 ← *maximum number of virtual users reached*



Mean value: 0.344625



Mean value: 0.344625

Created by WebStress 0.3

This sample test report contains two graphs: (1) a line graph of the response time versus the test session time, and (2) a box plot of the response time versus test script states. The first graph depicts all measured response times of the tested application during the test, and individual points are connected with lines. The horizontal line on the graph represents the mean value of all measured response times, i.e., "average" response time. The test-report header says that 43 requests were sent, and according to the graph, 11 of them took longer than the mean.

The second graph depicts, for each of the states used in the test script, the five-number summary, which consists of the smallest observation, lower (first) quartile[11], median[12], upper (third) quartile, and largest observation. The box plot is sometimes called box-and-whisker diagram, because the vertical lines that extend from the box are called whiskers. The lower and upper ends of the box represent the lower and

[11] a quartile is any of the three values which divide the sorted data set into four equal parts, so that each part represents ¼th of the sample or population; 1st quartile cuts off lowest 25% of data and 3rd quartile cuts of highest 25% of data

[12] a median is a number dividing the higher half of a sample, a population, or a probability distribution, from the lower half; it is equal to the second quartile

upper quartiles respectively, and the thick line in between represents the median. Whiskers represent the smallest and largest non-outlier[13] observations. In addition, the box plot indicates which observations, if any, are considered unusual, or outliers. From the graph, it can be seen that the medians of response times to requests in both states are almost equal. However, while the maximum response time for the START state is about 250 milliseconds, more than a third of the responses to requests in the state SEARCHFOR took longer than half a second. That is because the third quartile is about 600 milliseconds. A few responses took even more than 1.5 second.

[13] an outlier is an observation that is numerically distant from the rest of the data

5 Implementation

5.1 Design

WebStress is implemented in C++ and using STL[14], compliant with the ISO/ANSI C++ 1998 standard [29], with the exception of the use of a `va_copy` macro [30], which was included later in ISO/IEC 9899:1999 C standard [31]. The tool links the libcurl library [11] for the network communication and depends on the R statistical tool for the graph generation [12]. During the implementation, Code::Blocks IDE, GNU C++ compiler, and its MinGW port to Windows were used.

The tool consists of one binary executable. The architecture comprises the following components: controller, test configuration processor, test script executor, and graph generator. The components access the following external objects: test configuration, test script, test data, log, report, graph, web server, and statistics. This logical model is depicted in Figure 5.1, and individual components are described further in section 5.2. These components are mapped to the classes described in section 5.3 and Figure 3.1. Communication between the classes is captured in section 5.4 and Figure 5.3.

A detailed description of the original architecture prepared before the implementation of WebStress is included in Appendix A on the enclosed CD. The implemented tool very closely follows the specification. Only minor changes to the design were needed.

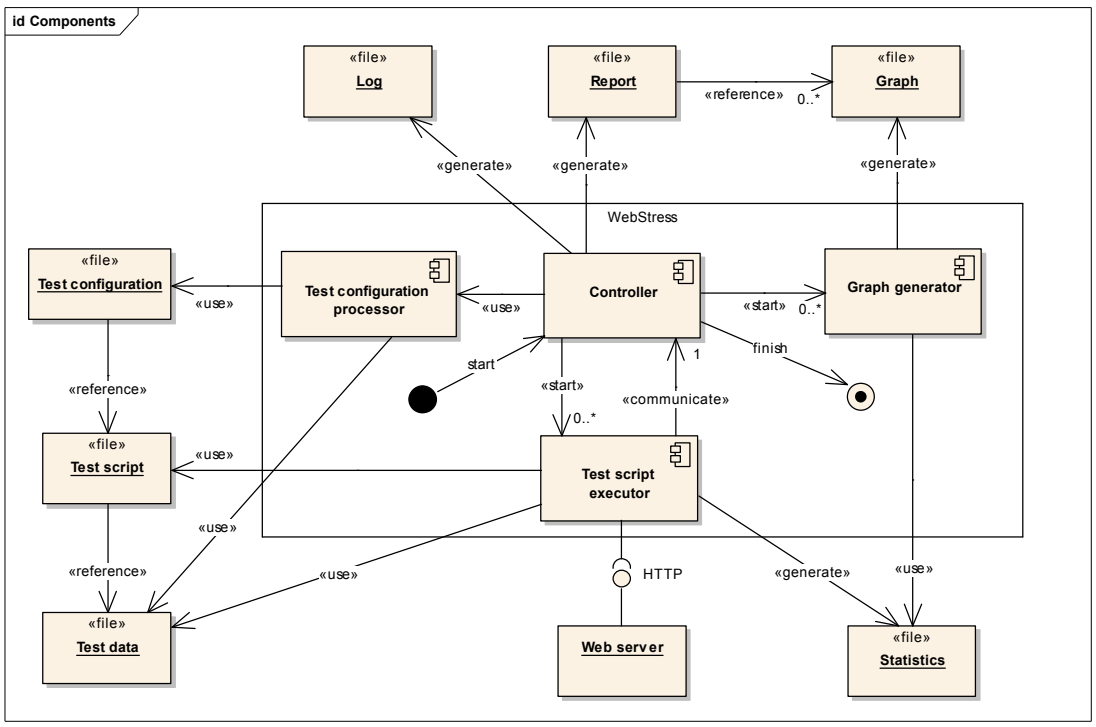


Figure 5.1: Logical model of WebStress implementation components

[14] Standard Template Library (STL) is a software library included in the C++, providing containers, iterators, and algorithms.

5.2 Components

Controller. This component prepares the test session, e.g., by calling the test configuration processor and initializing the log, and starts the test script processors for individual virtual users. It gathers all the information about the test during its execution and checks for the limit conditions. When at least one of them is met, it finishes the test session. In the end, it generates the report, with the graphs generated by the graph generator included in it.

Test configuration processor. This component processes the test configuration including the test scripts and the test data. It reads all these files, compiles the individual test scripts, and stores the test data or references to them into the memory.

Test script executor. This component executes the compiled test scripts simultaneously for each of the virtual users. The test data are either used from the memory or read directly from the external file, depending on the test scripts. Load generation, HTTP communication, and HTML processing are the tasks of this component. It communicates with the tested application, creating connections, sending requests, and receiving responses. It generates the test statistics for the graph generator from information about this communication. When needed, it parses the received responses and provides the test scripts with the gathered data.

Graph generator. This component generates the graphs used in the test report. Graphs are defined in the test configuration and are created based on statistics generated by the test script executor. Statistics are transformed into CSV files, and the R statistical tool is then run to process these files.

5.3 Classes

The mapping of the components defined in section 5.1 to the appropriate classes is in Table 5.1. The relationships between the classes are depicted in Figure 5.2.

Module	Classes
Controller	Dispatcher Logger
Test configuration processor	CommandFile TestScript
Test script executor	TestScript VirtualUser WebConnector HTMLParser ScriptState ScriptData
Graph generator	Chart

Table 5.1: Mapping of the modules to the classes

CommandFile parses the test configuration and fills in the internal structures of the Dispatcher. Only one instance of this object exists and is initialized by the Dispatcher after the tool is started.

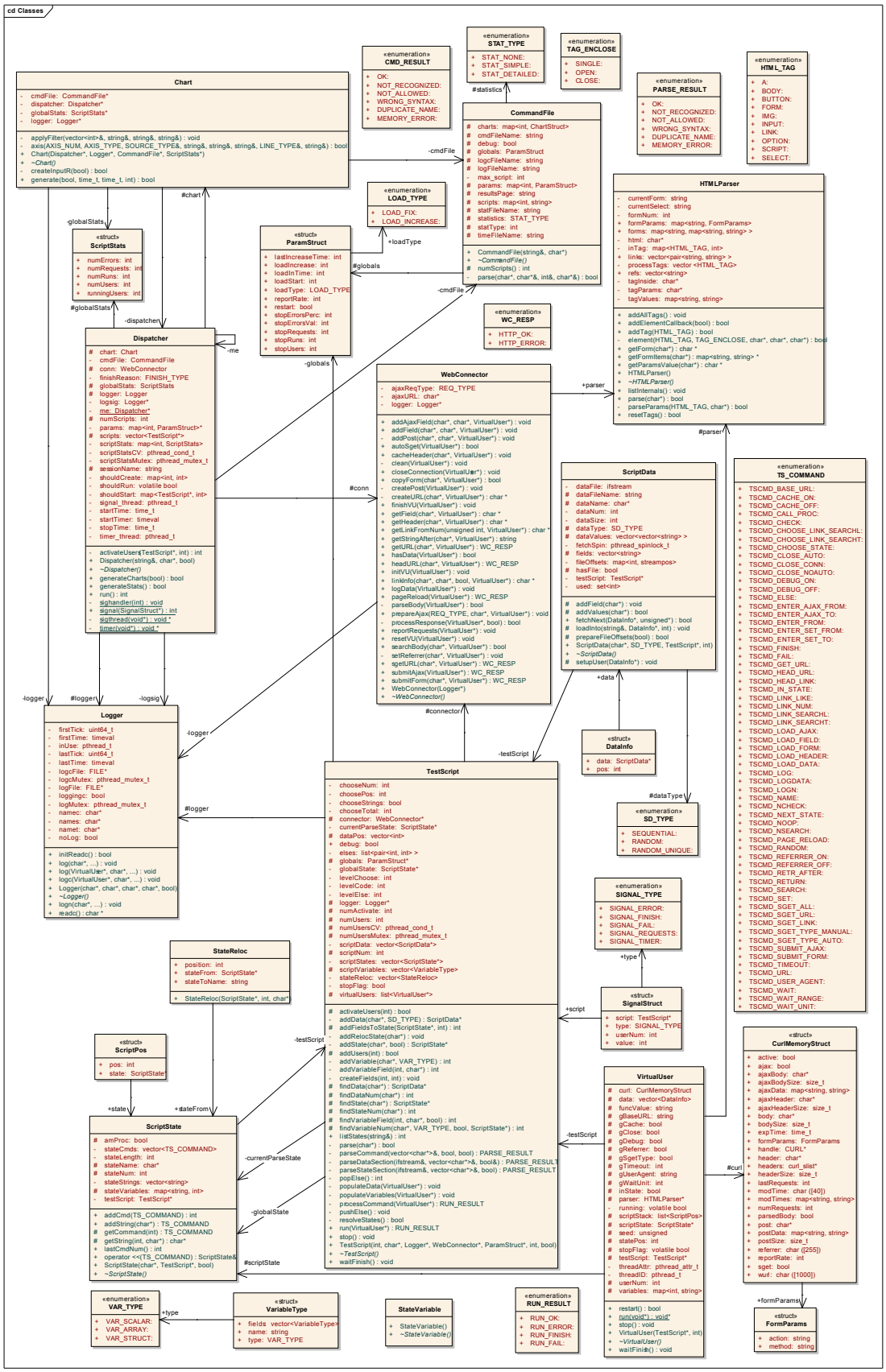


Figure 5.2: WebStress class diagram

Dispatcher is the main class of the tool. It performs the following actions:

1. Parse the test configuration – create an instance of the CommandFile.
2. Create the Logger, the WebConnector, and the Chart.
3. Parse and compile the test scripts – create an instance of the TestScript for each of the test scripts mentioned in the test configuration.
4. According to the test configuration, create multiple instances of the VirtualUser. The first virtual users are created just when the test execution starts, and the rest of them are created later during the test.
5. Process the information from the virtual users about the test execution, such as the number of requests or errors.
6. Check the limit conditions and finalize the test session when at least one of them is met.
7. Instruct the Chart to process the statistics generated by the virtual users and create the graphs based on this information.
8. Generate the report that includes the test-session information and the generated graphs.

Logger is responsible for creating the log files. All instances of all the classes, executed in multiple threads use only one common instance of the Logger. Therefore, concurrent access is managed by this class.

TestScript is created for each of the test scripts. Upon initialization, it parses the given test script and creates a compiled version of it in the memory. Individual commands are represented as byte codes, and variables and constants are defined and referenced by indexes in the created arrays. The test script is divided into test states, with each state stored separately in an instance of the ScriptState. The TestScript class uses the WebConnector class to communicate with the tested application and to access the data in retrieved responses.

ScriptState is merely a container for the compiled test script's commands and the constants used in the script, i.e., numbers and strings.

ScriptData stores the prepared test data. Each set of the data is stored in a separate instance, and these data are shared between virtual users executing the same test script that contains this data. Sharing allows offering sequential access to the data by all the virtual users without repeating the same data.

VirtualUser holds all the data specific to the virtual users – test script variables, statistics, current state in the script, the data about communication with the application, and the last response retrieved by that particular virtual user. When initialized, it starts its own thread and returns back to the caller (Dispatcher). All the test execution and load generation is performed by these threads.

WebConnector is responsible for the HTTP communication. All requests to the application web server(s) are made through a single instance of this class, but in the threads of virtual users. The VirtualUser instances store all the data. The most important tasks of this class are the connection management, request generation including headers, file caching, and the preparation of the data that is submitted to the tested application.

HTMLParser is responsible for parsing the HTML files retrieved as the responses from the tested application. It supports only the HTML tags that are used for web-form generation or to reference the attached files, such as images and styles. Parsing is event-driven and the internal callback function is called for the recognized tags. The information about the forms, links, and images is stored in arrays accessible by the test script.

Chart is responsible for the generation of the graphs included in the test report. Based on the combinations of the measurements specified in the test configuration, the source file for the R statistical tool is created, and then the R is executed on the statistics gathered during the test execution. Graphs are generated as PNG images.

5.4 Process flow

Important parts of the execution inside of the tool are depicted in the UML sequence diagram in Figure 5.3. Individual calls are accompanied by a short description.

5.5 Decisions made

As the implementation was focusing on performance testing, parts of the tool that were not related to it were implemented using libraries and external applications. This sped the implementation and (hopefully) decreased the occurrence of bugs.

Network communication was implemented using libcurl communication library [11]. This library offers very flexible HTTP communication with automatic redirects processing, compression, cookies support, header generation, and it is thread-safe. It even supports SSL[15], which can be implemented in WebStress in the future.

To parse HTML response, libwww library was tested during WebStress' implementation. However, its usage was complicated and the results were not satisfactory, so simple SAX parsed was developed for the usage in WebStress.

The preferred way of implementing multiple virtual users were threads. When talking about threads and portability, pthread [13] is usually the choice. There is no full implementation for Windows, so the usage was limited only to routines available there.

One of the goals for WebStress' implementation was to generate data about the test session to be readable by a sophisticated statistics tool. One such tool is R [12], and WebStress uses it to generate graphs for the report. At the end of the session, data is prepared in CSV format and a source script is prepared for R. Based on this, R generates graph images that are then included in the report.

5.6 Problems encountered

During later stages of the tool implementation, when the tool prototype was tested to generate higher loads, there were problems with resource consumption – CPU and memory.

Spin lock

Because the tool is implemented as a multi-threaded application, access to the log file and statistics generated during the test execution must be synchronized. From the

[15] Secure Sockets Layer (SSL) – cryptographic protocols which provide secure communications on the Internet

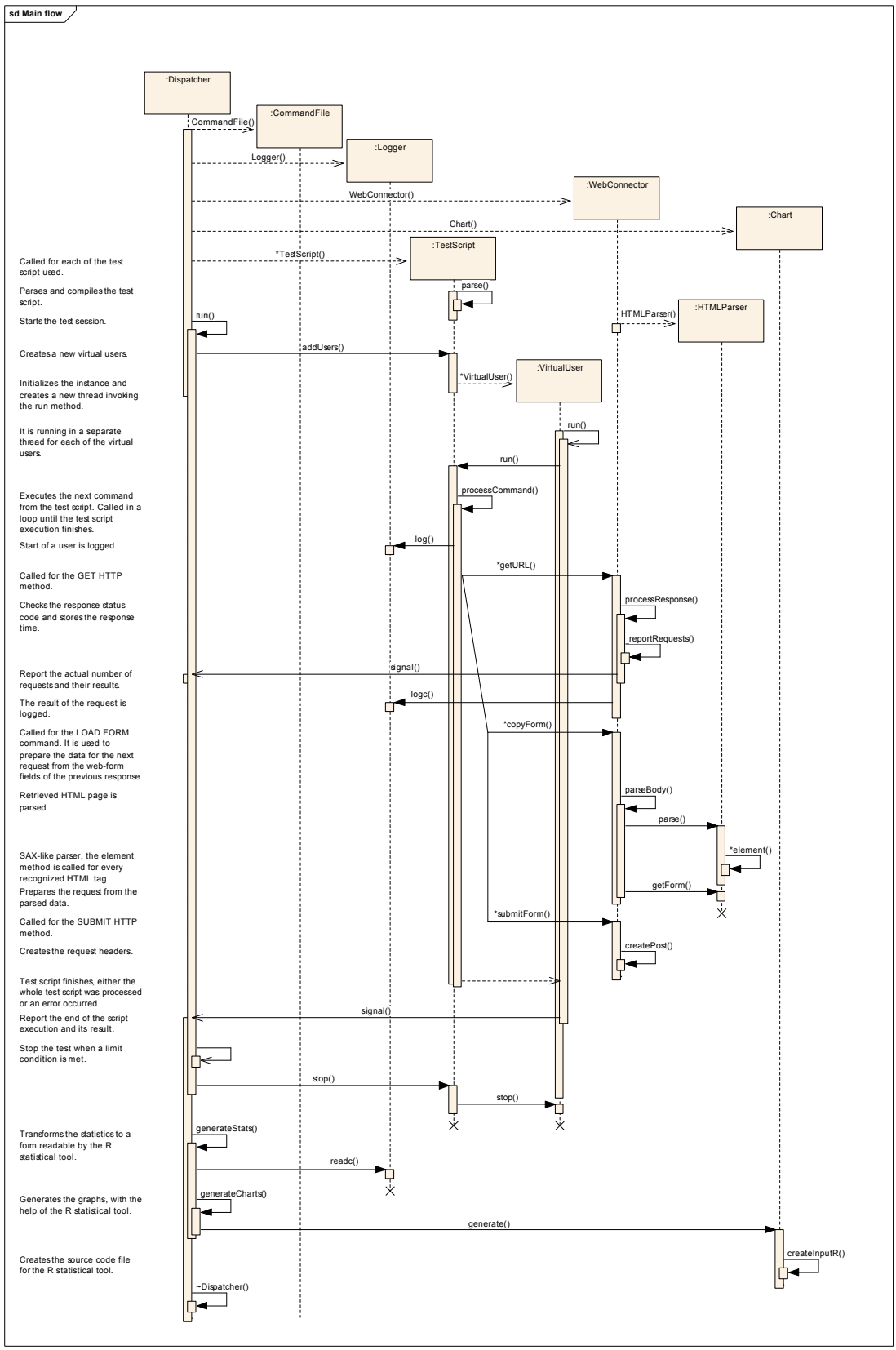


Figure 5.3: WebStress process flow (important parts)

analysis of the problem, it seemed that using a spin lock (active waiting) would minimize the time spent waiting for a shared resource.

When the tool was using hundreds of threads or when no think-times were used in the test scripts, threads wanted to lock the resource very often. Because of the design of a spin lock, this resulted in high CPU usage. When replaced by regular mutexes (passive waiting), CPU usage dropped noticeably.

Stack size

To implement threads in the tool, the pthread (POSIX threads) library was used. Each thread has its own stack that is allocated when the thread is created. Its size can be adjusted prior its creation, but the default value varies between platforms. The Linux default value is 10 MB. With up to a few hundred threads, this posses no problem because this memory is only virtual and it is physically allocated only when needed. However, this setting does not allow more than approximately 400 threads on 32-bit architectures because of the limitation of the size of the virtual addressing space. After decreasing the stack size to 1 MB, it is possible to create a couple thousand threads, if the operating system will allow it.

5.7 Code Metrics

Table 5.2 contains CCCC Software Metrics Report. CCCC [28] is a tool which analyzes C++ files and generates a report on various metrics of the code. Metrics supported include lines of code, McCabe's complexity, and metrics proposed by Chidamber and Kemerer and Henry and Kafura.

5.8 License

WebStress is implemented using the libcurl [11] and pthread [13] libraries. libcurl is distributed under an MIT/X derivate license [21], and pthread library is distributed under various licenses, usually with the operating system. The Windows implementation of pthread library called pthreads-win32 [24], which is used to build the binary in the Windows distribution of the tool, is distributed under GNU Lesser General Public License (LGPL) [22]. The R statistical tool that WebStress uses for the graph generation, but is not built with it, is distributed under GNU General Public License (GPL) [23].

WebStress itself is distributed under GNU General Public License (GPL) [23]. Source code and binary distribution(s) are available for free at the project website: `http://webstress.tereus.eu/`.

Metric	Tag	Overall	Per Module
Number of modules Number of non-trivial modules identified by the analyzer. Non-trivial modules include all classes, and any other module for which member functions are identified.	NOM	31	
Lines of Code Number of non-blank, non-comment lines of source code counted by the analyzer.	LOC	3106	100.194
McCabe's Cyclomatic Number A measure of the decision complexity of the functions which make up the program. The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram. The analyzer counts this by recording the number of distinct decision outcomes contained within each function, which yields a good approximation to the formally defined version of the measure.	MVG	1200	38.710
Lines of Comment Number of lines of comment identified by the analyzer.	COM	960	30.968
Lines of code per line of comment Indicates density of comments with respect to textual size of program.	L_C	3.235	
Cyclomatic Complexity per line of comment Indicates density of comments with respect to logical complexity of program.	M_C	1.250	
Information Flow measure (inclusive) Measure of information flow between modules suggested by Henry and Kafura. The analyzer makes an approximate count of this by counting inter-module couplings identified in the module interfaces.	IF4	594	19.161
Information Flow measure (visible) IF4 calculated using only relationships in the visible part of the module interface.	IF4v	594	19.161
Information Flow measure (concrete) IF4 calculated using only those relationships which imply that changes to the client must be recompiled of the supplier's definition changes.	IF4c	0	0.000

Table 5.2: Code metrics

6 Evaluation

To evaluate the functionality of WebStress, I tested several web applications using this tool. This chapter provides a description of two of the tests, the Slovak Cadastral Portal and RUBiS benchmark site. The test results are included. These applications cover many of the possible architectures used in today's web applications. In fact, WebStress was specifically designed to test Slovak Cadastral Portal. Tests of other applications, specifically the Czech Cadastral Data Look-up and OpenSTA demo site, are included as examples in the WebStress distribution, in Appendix E on the enclosed CD.

The applications described in this chapter use web forms with input boxes, lists, radio buttons, simple buttons, and buttons represented by an image, and they sometimes even perform background queries after a user enters the data. Test data with only a couple of records are stored inline in the script, and data with hundreds of records are stored in separate files. They require users to log in for most if not all actions and use cookies or URL parameters to track a session.

6.1 Slovak Cadastral Portal

Description

The Slovak Cadastral Portal is a web application used by citizens and private and public organizations to access the data of the Slovak cadastral offices (real estate register). It has been in operation since 2004, but it currently is undergoing a massive software and hardware upgrade. The application allows querying for information about land, buildings, apartments, and owners. It consists of two parts: (1) descriptive (text) data access, and (2) cadastral map browsing. Users can either display the results of their queries inside the browser or generate PDF documents that copy paper documents available at cadastral offices. The list of possible queries is based on user authorization, e.g., some government agencies can query on the social security number of a real estate owner. A snapshot of a web page used in the application to query information about land is in Figure 6.1.

The Slovak Cadastral Portal is available only in Slovak, but the following test uses English terminology for queries and data. Only the descriptive data part of the application is tested because map browsing is implemented as an ActiveX component, i.e., application code that is executed on the client, similar to Java applets. Maps can be tested by specific software from the browsing component vendor.

The test was performed on the target hardware that was being prepared to go into production. The whole application runs on an IBM p590 machine with 8 dual-core 2.1 GHz Power5 CPUs and two additional dual-core Intel Xeon 3 GHz machines. Oracle 10g database software on AIX and Oracle J2EE application servers on Red Hat Enterprise Linux were used to run the descriptive part of the application.

Tests

This test simulates users with basic user rights, which are the vast majority of users working with the application. It covers querying based on the identification of a plot

Figure 6.1: Preview of Slovak Cadastral Portal

of land, the title[16], the real estate owner, building, apartment, and any proceedings (registration of a change). It does not cover working with generated PDF documents. These documents are processed by separate servers and are benchmarked differently.

Each query specifies a region in which it should be performed, the cadastral unit for real estate, and the district for proceedings. Because most users do not know the exact names of cadastral units, it is possible to select a province using a dropdown list and then continue by selecting a district, city/village, and finally the name of the desired cadastral unit. Each query is performed in the background using AJAX[17], and the page is updated without the need to reload it all.

After a region is selected, the identification of a piece of real estate or a proceeding is entered. Generally, it is a combination of numbers and letters. The whole query is then sent to the application server, and the response is checked. Test data were generated from the application database, and they consist of approximately 600 different records for each type of query. Specific data that should be contained in the response were enclosed with each record to be able to check that the response was correct. This, in fact, represents simple functional testing performed together with performance testing.

The application contains built-in restrictions for fast queries. One of them is to limit the number of queries sent by one user in one minute. Therefore, more accounts were pre-created, and virtual users use them randomly. Another restriction is that

[16] title – owner's interest in a piece of property, represented by a document

[17] Asynchronous JavaScript and XML (AJAX) – a development technique for creating interactive web applications by exchanging small amounts of data with the server behind the scenes, so that the entire web page does not have to be reloaded each time the user requests a change.

each request is delayed by at least half a second. Real users will not notice, but it slows down robots rapidly. This delay is not applied to background (AJAX) queries, making them faster. This is because only the public code lists, e.g., list of districts or cadastral units, can be requested this way. If the application does not respond in 10 seconds, it is considered as an error.

The test starts with 15 virtual users, and then three new virtual users are started each second, until 800 virtual users is started. This is the ramp-up period.

The description of the actions performed in this test is in Table 6.1. All the think-times are randomly generated from the specified range using a uniform distribution. In some of states, more requests are sent and different think-times are used in between them. The complete test configuration and test script can be found in Appendix D on the enclosed CD.

State	Description	Think-time	Next state
START	Display the home page, welcome information, and login form.	none	LOGIN
LOGIN	Log into the application, use one of the pre-created accounts.	6-20 s	WORK
WORK	Randomly decide what information to query. One of the options is to log out and finish.		LAND, TITLE, OWNER, BUILDING, APARTMENT, PROCEEDING, LOGOUT
LAND	Query information about a plot of land.	0-5/1-3/ 1-3/2-5/ 3-12 s	WORK
TITLE	Query information about a specific title.	0-5/1-3/ 1-3/2-5/ 3-10 s	WORK
OWNER	Query information about real estate owned by a specific person.	0-5/1-3/ 1-3/2-5/ 4-15 s	WORK
BUILDING	Query information about a building.	0-5/1-3/ 1-3/2-5/ 3-10 s	WORK
APARTMENT	Query information about an apartment.	0-5/1-3/ 1-3/2-5/ 3-17 s	WORK
PROCEEDING	Query information about a specific proceeding.	0-5/1-3/ 1-3/ 3-10 s	WORK
LOGOUT	Log out of the application and finish.	10-40 s	

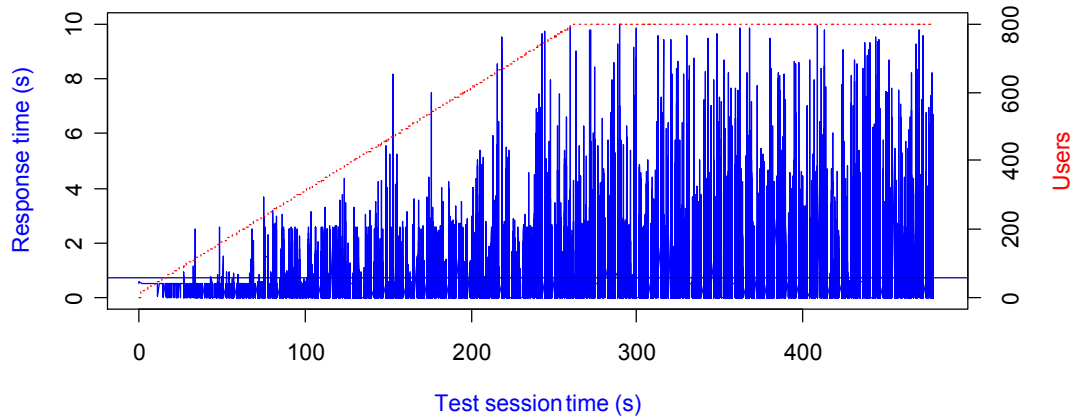
Table 6.1: List of states used in Slovak Cadastral Portal test script

Results

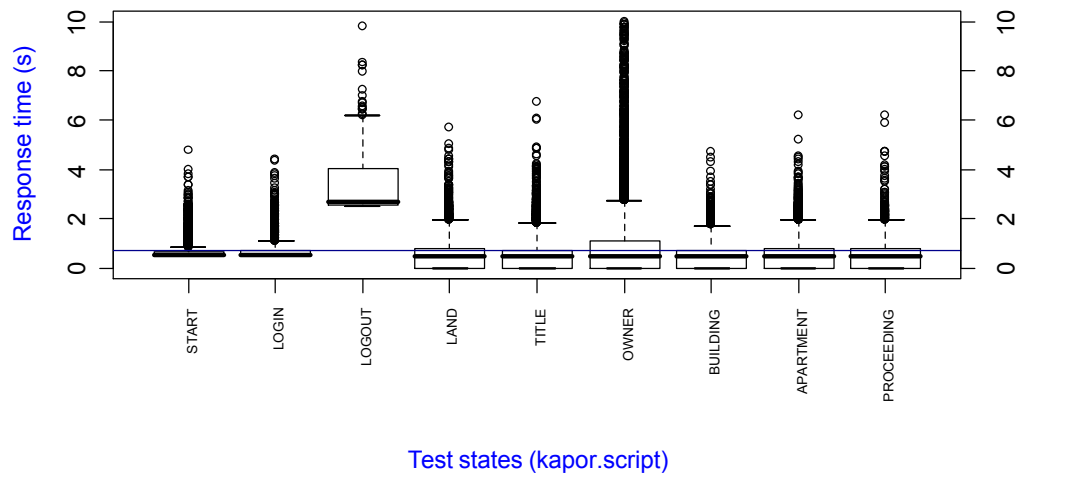
The test was performed on a live system before entering production mode. Thus, it was possible to measure peak performance without interfering with real users'

activity. To generate the load, one separate computer connected to the system's internal LAN was used.

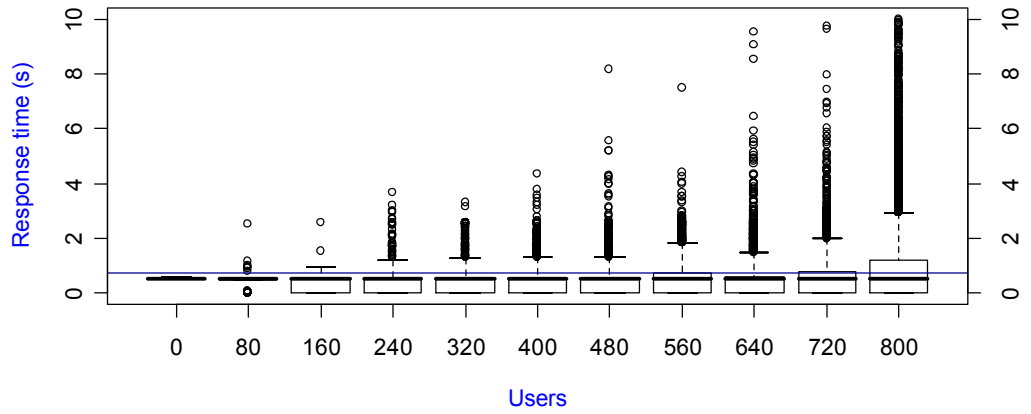
Configuration file: kapor.conf
Test start: Sun May 13 21:39:30 2007
Test finish: Sun May 13 21:48:07 2007
Finish reason: TIME
Running time before stop: 480 seconds
Total requests: 27654
Total errors: 288 (1 %)
Total runs: 1437
Total users: 800



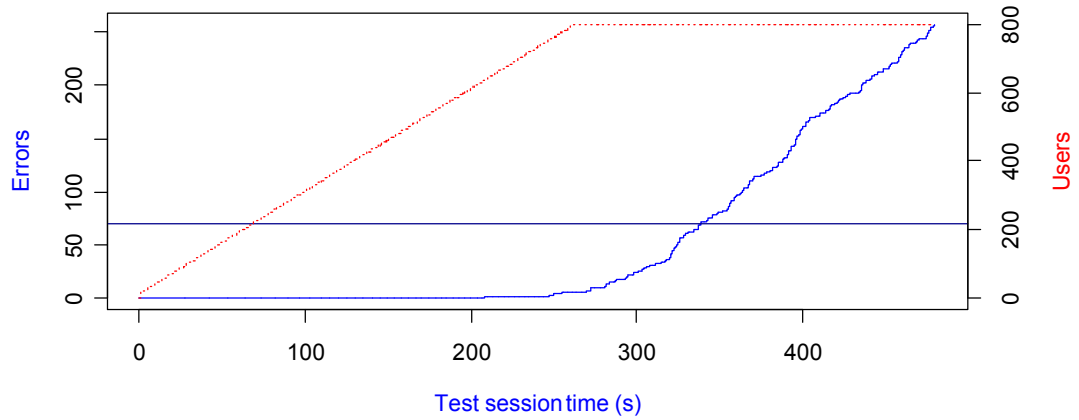
Mean value: 0.7452555



Mean value: 0.7452555



Mean value: 0.7452555



Mean value: 69.4392

Created by WebStress 0.3

The test report consists of a header and four diagrams: (1) a line graph of the response time versus the test session time, (2) a box plot of the response time versus test script states, (3) a box plot of the response time versus number of virtual users divided into 10 groups, and (4) a line graph of the number of errors and number of virtual users versus the test session time. Notation in these graphs and terms used in the following text are defined in chapter 4.5.

In the first graph, the number of virtual users starts at zero and steadily increases up to 800 virtual users, with the ramp-up period lasting 4 minutes and 20 seconds. The response times measured were increasing with increasing load. Combining the results with the third graph reveals that the median of response times for different numbers of virtual users is about the same. However, the third quartile and the number of outliers were increasing. Adding information from the fourth graph that some of the requests were timing-out with higher load shows that the application appears not to handle the load of 800 users very well.

This test was set to slowly raise the load up to 800 virtual users and then keep a steady load for a few minutes. Before the tool achieved this load limit, the application did not respond to some requests before the specified time-out (errors in the last graph). The error rate was approximately one error per second. Apart from these time-outs, 75 percent of the responses under a load of 800 virtual users were retrieved in less than three seconds (upper-rightmost whisker in the third graph).

The test results in the second graph show that requesting the main page, logging in, and especially logging out of the application takes more time compared with other requests. This is because the application servers are connected to a cluster that is distributing information about sessions to allow seamless take-over in case of a malfunction of one of them. All the other queries take approximately the same time, with slightly longer times for querying information based on real-estate owner.

6.2 RUBiS benchmark site

Description

RUBiS [9] is an auction site application modeled after eBay.com that is used for performance-testing of application servers and their performance scalability. It implements the core functionality of an auction site: selling, browsing, and bidding. It distinguishes between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register, but they are only allowed to browse. Buyer and seller sessions require registration. Snapshot of a web page used in the application to display information about an item for sale is in Figure 6.2.

This application defines 26 interactions that can be performed from the client's Web browser. Among the most important ones are browsing items by category or region, bidding, buying or selling items, leaving comments on other users and consulting one's own user page. Browsing items also includes consulting the bid history and the seller's information. The test definition in this test uses all 26 interactions; the only difference from the original tests is that it uses different think-times and probabilities of transitions between interactions.

The test was performed on the PHP version of RUBiS, installed on AMD Duron 1.1 GHz machine running Gentoo Linux. Apache with the PHP module was used as a web server, and a MySQL database was used as back-end storage for the application data.

Tests

This test simulates users that are browsing items on sale, bidding information, information about sellers, users bidding on items, and users selling their items. It covers all the functionality of the application. To log in, pre-created accounts are used. If the application does not respond in 20 seconds, it is considered an error.

A description of actions performed in this test is in Table 6.2. All the think-times are randomly generated from the specified range using a uniform distribution. The complete test configuration and test script can be found in Appendix D on the enclosed CD.

Results

The test was performed using an old and slow server and one faster client computer. To better simulate real application behavior, the application database was loaded

Figure 6.2: Preview of RUBiS benchmark site

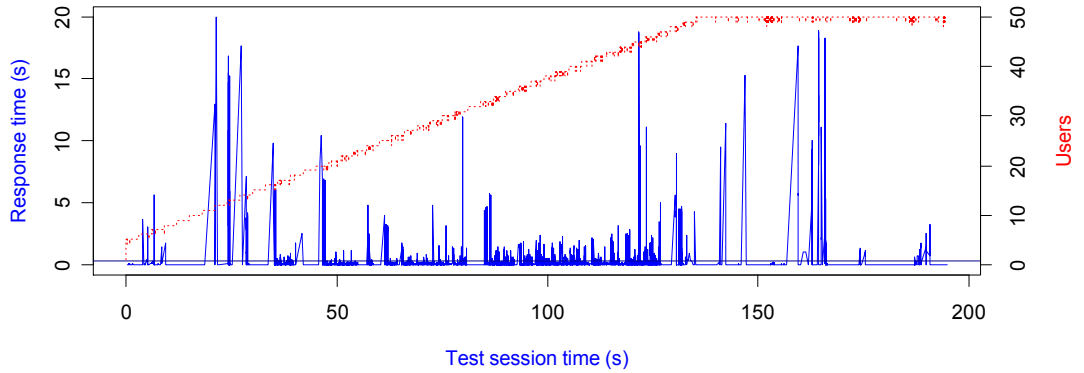
State	Description	Think-time	Next state
START	Retrieve the home page; decide what to do with the application.	0-5 s	BROWSE, LOGIN_SELL, LOGIN_ABOUT
BROWSE	Browse items, either by category or by region.	1-3 s	BROWSE_CAT, BROWSE_REGION
BROWSE_CAT	Browse by categories, select randomly from a list.	3-10 s	LIST_ITEMS
BROWSE_REGION	Browse by regions, select randomly from a list.	2-8 s	BROWSE_CAT
LIST_ITEMS	List items in specified category/region.	5-20 s	ITEM_DET, LOGIN_BID, LIST_ITEMS_PREV, LIST_ITEMS_NEXT
LIST_ITEMS_PREV	Display previous page of list of items.	5-20 s	LIST_ITEMS
LIST_ITEMS_NEXT	Display next page of list of items.	5-20 s	LIST_ITEMS
ITEM_DET	Display specific item description.	6-30 s	BID_HIST, USER_DET, LOGIN_COMM, LOGIN_BUY, LOGIN_BID

State	Description	Think-time	Next state
BID_HIST	Display bid history for specified item.	4-12 s	USER_DET
USER_DET	Display user details and comments; decide whether to display information about users that entered a comment.	7-18 s	USER_DET, START
LOGIN_COMM	Log in to enter user comment.	2-4 s	ENTER_COMM, LOGIN_ERR
LOGIN_ERR	Login error, fail.	none	
ENTER_COMM	Enter user comment.	5-10 s	START
LOGIN_BUY	Log in to buy an item directly.	2-4 s	ENTER_BUY, LOGIN_ERR
ENTER_BUY	Buy an item.	1-3 s	START
LOGIN_BID	Log in to bid on an item.	2-4 s	ENTER_BID, LOGIN_ERR
ENTER_BID	Bid on an item.	2-5 s	START
LOGIN_SELL	Log in to sell an item.	2-4 s	SELL_CAT, LOGIN_ERR
SELL_CAT	Select category in which to sell an item.	3-10 s	ENTER_SELL
ENTER_SELL	Sell an item, enter all required details.	5-10 s	START
LOGIN_ABOUT	Log in to see own user information.	2-4 s	ABOUT, LOGIN_ERR
ABOUT	Display own user information, list of items on sale, bids, etc.	4-20 s	ITEM_DET, USER_DET, ENTER_BID
REGISTER	Enter registration data and create a new user.	7-15 s	START

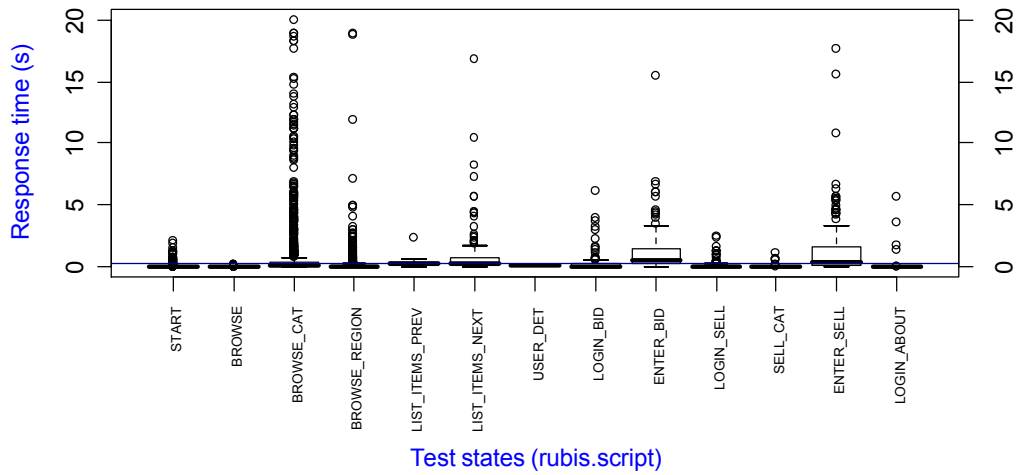
Table 6.2: List of states used in RUBiS test script

with test data available at the application web site [27] prior the test. I had to update the loaded data, because the application creates auctions lasting only up to 7 days, and they were created in year 2001. This amount of data allowed load generation that was beyond the server's capabilities.

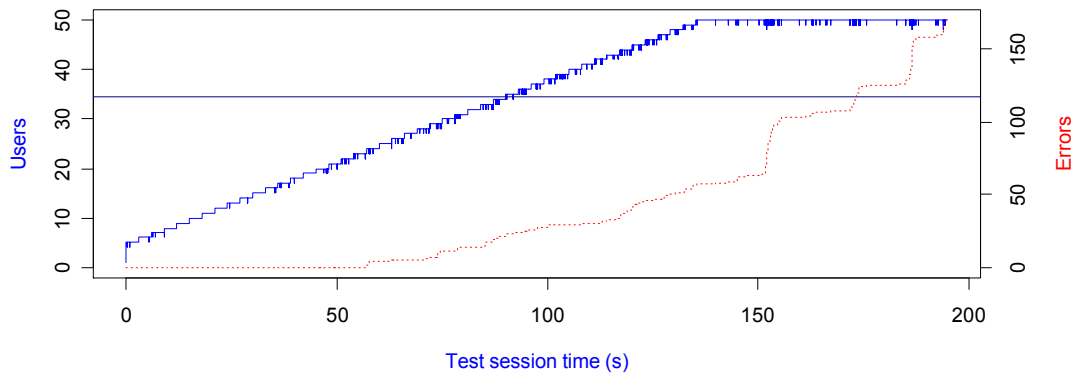
Configuration file: rubis.conf
Test start: Sun May 13 11:28:05 2007
Test finish: Sun May 13 11:31:40 2007
Finish reason: TIME
Running time before stop: 195 seconds
Total requests: 5096
Total errors: 214 (4 %)
Total runs: 350
Total users: 50



Mean value: 0.323407



Mean value: 0.323407



Mean value: 34.4724

Created by WebStress 0.3

The test report consists of a header and three diagrams: (1) a line graph of the response time versus the test session time, (2) a box plot of the response time versus test script states, and (3) a line graph of the number of errors and number of virtual users versus the test session time. The notation in these graphs and the terms used in the following text are defined in chapter 4.5.

From the combination of the results in all three graphs, it can be seen that the increased load had only a marginal impact on most of the responses of the application. Most of the requests were either processed in a short time, up to one or two seconds, or took longer than 20 seconds and were considered as time-outs. Only a small part of the requests was processed in the time between 5 and 20 seconds. The number of time-outs started to increase after one minute, or when the load reached 25 virtual users. With the load of 50 virtual users, the error rate was about two errors per second, which is equal to one error in 25 seconds for each virtual user.

The hardware used for the test was able to process from 30 to 40 users working with the application as specified in the test script. With a higher load, the number of errors increases. Most of the responses are sent back to the client in tenths of second. On average, the longest responses are for actions that write data into the application database – bidding or selling an item.

6.3 Summary

The implemented tool was used to test four web applications, and the results of two of such tests were presented in this chapter. A scripting language allowed the description of scenarios of users working with these applications, and test execution showed that increased load had an impact on tested applications. RUBiS was developed to be tested by web-application performance-testing tools, and this tool was able to simulate all the interaction of the application as the standard tests. These are included in the RUBiS distribution and are used to performance-test underlying software, i.e., the application server.

During these tests, tested applications were limiting the load generation and not the tool itself. Even during the test of Slovak Cadastral Portal, where 800 virtual users were used to interact with the application running on the target hardware, client CPU (Power5, 2.1 GHz, dual-core) utilization was only a few percent.

7 Related projects

In this chapter, four open-source and one commercial web-application performance-testing tools are compared to WebStress. Each tool is presented by short description and with one column in a detailed comparison matrix in Table 7.1.

7.1 OpenSTA

OpenSTA stands for Open System Testing Architecture. It is a toolset for distributed software testing, currently with capability of generating load in HTTP(S) protocol. As a first step in test definition, user sessions are recorded using an internal proxy server and the tool generates static script. This script is usually modified by user in order to use different test data. Apart from load generation, the tool is able to monitor clients and servers running Microsoft Windows or an SNMP daemon.

OpenSTA offers very precise control over connections to the server and supports multiple connections with the possibility to define which requests will use individual connections. At the same time, this complicates script maintenance because of most the actions have to be defined manually.

7.2 JMeter

Apache JMeter is a Java desktop application designed to load test functional behavior and measure performance. Apart from web applications, it allows the testing of databases (using JDBC), FTP and LDAP servers, and JMS services. A test scenario is defined as a tree of elements edited in a GUI that then can be executed either in the GUI or using command-line utilities. Its functionality can be extended with plug-ins that communicate via the documented interface.

JMeter offers easy-to-understand user interface and the fast definition of simple test scenarios. However, when it comes to dynamic text execution and web-form processing, it gets more complicated.

7.3 Web Application Stress Tool

Microsoft Web Application Stress (WAS) Tool is a very simple tool, offering only static tests, still with some possibilities to use prepared test data. A test scenario is defined by a list of requests and a delay between them. This can be created manually, recorded using a built-in proxy or generated for an Internet Information Server log file. When submitting data to the application, it is possible to work with individual form fields and these can be imported from an HTML file. Client and server can be monitored using Windows NT/2K/XP performance counters.

WAS Tool offers simple record/replay functionality that can be used as a first round of performance testing. Bandwidth for individual virtual users can be limited, but only to analog or ISDN modem speeds. The last and only version of this tool is from December 2002.

7.4 Tsung

Tsung (formerly IDX-Tsunami) is a distributed load-testing tool that currently supports the HTTP, PostgreSQL, and Jabber protocols. Tsung is developed in Erlang,

a concurrency-oriented programming language that gives it powerful distributed capabilities. A test scenario is defined with XML and can be extended by Erlang modules. Servers can be monitored by Erlang agents or SNMP.

Tsung offers load generation of thousands of virtual users, running on many computers simultaneously under a single control. Above common response time reporting, it is possible to measure time of user-defined groups of actions, e.g., to process a multi-page web form. As a drawback, test definition does not offer many possibilities to branch test execution. Another handicap is that version for Windows is not publicly available, but it seems it might be possible to compile it from source.

7.5 LoadRunner

Mercury LoadRunner is the only commercial product in this comparison. It is very mature, supports more than 40 protocols, and can be used not only to test performance, but to analyze problems in tested applications too. With the help of its monitors, the performance of all layers of the application can be analyzed and it is possible to drill down to such details as SQL statements. Test scenarios are defined using C-style syntax, and C libraries can be linked to the test, which allows every possible variation in the test execution.

LoadRunner offers a lot of the functionality a performance-testing tool might offer – powerful language for test definition, detailed monitoring of all components of tested systems, and highly customizable reporting. According to [10], it has market share of 77 percent in load testing. The license for such a tool can, depending on the protocols and virtual users used, cost tens of thousands of U.S. dollars.

7.6 Comparison matrix

In chapter 3, the requirements for a web-application performance-testing tool were outlined and the possible approaches discussed. These criteria are used in Table 7.1 to compare presented tools with each other and with WebStress.

WebStress loses to other tools in some of the criteria. However, the intention was not implement a tool that would be better in all aspects. The purpose was to offer functionality that would be useful during the tests of the web applications with which I work, and the available tools do not cover them.

	WebStress	OpenSTA	JMeter	WAS Tool	Tsung	LoadRunner
Cost / License	Free	Free	Free	Free	Free	Thousands of U.S. dollars
Supported platforms	Unix/Linux, Windows	Windows	Unix/Linux, Windows, OpenVMS, requires JVM	Windows	Linux, Solaris, *BSD, MacOS X	Windows for administration, Unix/Linux for load generation
Load generation						
Connection closing	Can be left open, or closed automatically or manually.	Controlled manually.	Can be left open or closed automatically.	Can be left open, with extra request header.	Kept open during the execution of a virtual user.	Can be left open or closed automatically.
Simultaneous connections	No	Yes, user defined.	No	No	No	Yes, automatically or user defined.
Proxy server support	Yes	Yes	Yes	No	No	Yes
Traffic shaping	No	No	Number of requests.	Fixed modem speeds or T1 line.	No	Arbitrary bandwidth.
HTTP methods	GET, POST, HEAD	GET, HEAD, POST	GET, POST	GET, HEAD, POST	GET, POST	GET, POST, and custom methods
Request headers generated automatically	If-Modified-Since, Referer, and User-Agent	None	Default values for all requests can be set.	User-Agent	User-Agent	Accept, Accept-Encoding, If-Modified-Since, Referer, and User-Agent
Compressed response	Yes	No	Yes	N/A, response is not processed.	N/A, response is not processed.	Yes

	WebStress	OpenSTA	JMeter	WAS Tool	Tsung	LoadRunner
Parsing of HTML response	Images, style sheets, external scripts, background, images, and web-forms.	Accessible as DOM.	Images, applets, style sheets, external scripts, frames, background images, and sounds.	No	No	Images, applets, external scripts, and frames.
Files caching	Only attached files and no HTML data.	No	No	No	If-Modified-Since header can be set manually.	Yes, with various settings. The cache can be stored and then loaded later.
Pre-created response from form fields	Yes, prepared from default values.	No, query string has to be created manually.	No, but fields can be entered separately.	No, but fields can be entered separately.	No, query string has to be created manually.	No, but fields can be entered separately.
Cookies support	Processed automatically.	Manually	Processed automatically.	Processed automatically.	Processed automatically.	Processed automatically, script can manipulate them.
Authentication	No	Basic method header can be created manually.	Basic	Yes	Basic	Basic, NTLM, Digest
Virtual users implementation	Single user per thread.	Single user pre thread and a thread pool.	Single user per thread.	Multiple users per thread, multiple threads.	Erlang internal threads, no OS threads.	Single user per thread, or per process.
Distributed load	No	Yes	Yes	No	Yes	Yes

	WebStress	OpenSTA	JMeter	WAS Tool	Tsung	LoadRunner
Think-time	Constant or uniform random.	Constant	Constant, Gaussian, or uniform random.	Constant	Exponential	Constant or uniform random.
Reporting						
Page-load or transaction time	No	Manually defined transactions.	No	No	Page-load time, pages are divided with think-times.	Manually defined transactions.
Resource monitoring	No	SNMP, Windows NT	Tomcat 5 status servlet	Windows NT	SNMP, Erlang agents	Yes
Report format	HTML, with PNG images. Results can be further analyzed with a statistical tool.	Displayed in GUI.	Displayed in GUI, graphs can be saved to file.	Displayed in GUI.	HTML	Displayed in GUI, and can be stored as HTML or MS Word DOC.
Test definition						
Test definition	Script, Own language	Script, Own language (SCL)	Graphical, Tree, JavaScript	List of requests.	Script, XML, Erlang extra modules	Script, C, Java, Visual Basic
Recording capability	No	Yes	Yes	Yes	Yes	Yes
Log sampling	No	No	Access log format (e.g. Apache)	IIS format	No	No
Searching in response	Substring	Only explicit addressing using DOM.	Yes	No	Regular expression	Substring

	WebStress	OpenSTA	JMeter	WAS Tool	Tsung	LoadRunner
Branching	Random (with weights) or based on response.	According to a variable value, e.g., loaded from the response.	Random link selection and loops.	No	Only to loop requests or abort test session.	Yes, it is a fully featured programming language.
User-prepared data	Scalars and structures, inline, or in external CSV file.	Scalars, inline. Structures have to use mutexes.	Scalars and structures, in external CSV or XML file.	Scalars, inline, or structures from a database.	Yes, inline using Erlang or one external file.	Scalars and structures, in external CSV file or from a database query.
Tool-generated data	No	Integer range.	Number counter, random radio button selection.	No	Unique ID generation.	What is allowed in the programming language used.

Table 7.1: Web test tools comparison matrix

8 Conclusion

WebStress, the web-application performance-testing tool implemented, fulfils the functional and non-functional requirements set out in the specification. These requirements were to generate HTTP load with many virtual users, specified by a text script and to be able to generate common headers in requests, including referrers and cookies, process web forms, allow for branching of test execution, parse the retrieved response, request attached files, and cache the files once retrieved. With this range of functionality, it can be used for performance testing of many web applications.

Because one of the main goals was to use this tool for performance tests of Slovak Cadastral Portal, the implementation was successful – the tool was able to test the application on the target hardware and the results helped to improve its performance before going into production. During the tests, the tool generated the load of up to 1,000 virtual users and did not fully utilize the resources of the computer on which it ran. The application consists of a cluster of three application servers and used software load-balancing during the tests. The load-balancing process with such a number of users was slowing down the application, so the tests will be repeated later when a hardware load-balancer is installed.

The implemented tool automatically pre-creates a request to the tested application from the web form fields contained in the previous response. Creating a test definition to test an application that uses many web forms or stores the application data in hidden form fields is then much easier and the user can concentrate on the test preparation without bothering with details. Tools publicly available on the Internet do not offer such functionality.

Even though one computer to generate the load was enough so far for the tests, one of the possible future improvements is to offer distributed load generation. This cannot only allow generating higher loads, but it can make the load more realistic because it could come from more sources. Another improvement would be to offer a more sophisticated grammar for the test definition language and new commands, such as expression evaluation. This probably would result in a formalization of the grammar and the use of a lexical analysis.

Tests performed so far show that the tool is stable, and it is expected that it will be used to test other web applications developed in other projects. Apart from these plans, the driving force for the new development will be the tool usage, and new features will be added as needed by the tested applications.

9 Bibliography

[1] RFC2616, http://www.faqs.org/rfcs/rfc2616.html

[2] OpenSTA – Open Systems Testing Architecture, Performance testing of web applications, open source, http://www.opensta.org/

[3] Apache JMeter, Java desktop application designed to load test functional behavior and measure performance, open source, http://jakarta.apache.org/jmeter/

[4] Microsoft Web Application Stress Tool, Simulation of multiple browsers requesting pages from a Web site, Microsoft Corporation, http://support.microsoft.com/kb/231282/

[5] Tsung, Multi-protocol distributed load testing tool, open source, http://tsung.erlang-projects.org/

[6] Mercury LoadRunner, Enterprise-class performance testing tool, supporting many various protocols, Hewlett-Packard, http://www.mercury.com/us/products/performance-center/loadrunner/

[7] Webster's New Millennium Dictionary of English, Preview Edition, Lexico Publishing Group, LLC, Long Beach, California, U.S.A.

[8] Google search engine, http://www.google.com/

[9] RUBiS, Auction site prototype, PHP, Java servlets or EJB, http://rubis.objectweb.org/

[10] Application Load Testing Report, Yankee Group, July 2005

[11] libcurl - the multi-protocol file transfer library, http://curl.haxx.se/libcurl/

[12] The R Project for Statistical Computing, http://www.r-project.org/

[13] POSIX Threads, http://wikipedia.org/wiki/POSIX_Threads

[14] Windows Time, Microsoft Developer Network, http://msdn2.microsoft.com/en-us/library/ms725496.aspx

[15] Time is the Simplest thing..., The Joseph M. Newcomer Co., http://www.flounder.com/time.htm

[16] Performance Application Programming Interface, http://icl.cs.utk.edu/papi/

[17] Linux / Unix sar command, http://www.computerhope.com/unix/usar.htm

[18] Net-SNMP library, http://www.net-snmp.org/

[19] Chart (or graph), http://en.wikipedia.org/wiki/Chart

[20] HTTP Authentication Woes, Bill Venners, April 6, 2006, http://www.artima.com/weblogs/viewpost.jsp?thread=155252

[21] libcurl license, a MIT/X derivate, http://curl.haxx.se/docs/copyright.html

[22] GNU Lesser General Public License (LGPL), http://www.gnu.org/licenses/lgpl.html

[23] GNU General Public License (GPL), http://www.gnu.org/licenses/gpl.html

[24] POSIX Threads for Win32, http://sourceware.org/pthreads-win32/
[25] A Peek Inside the Clock, The Linux Magazine, September 15th, 2001, http://www.linux-mag.com/id/866/
[26] High-resolution timing, Linux I/O port programming mini-HOWTO, Riku Saikkonen, http://tldp.org/HOWTO/IO-Port-Programming-4.html
[27] The database dump used for the experimental performance tests of RUBiS, 113 MB, http://rubis.objectweb.org/results.html
[28] CCCC - C and C++ Code Counter, http://cccc.sourceforge.net/
[29] International Standard 14882 - Programming Language C++, September 28, 1998, http://www.ncits.org/cplusplus.htm
[30] va_copy macro, manual page, http://www.g00gle.net/man/man3/va_copy.3.html
[31] ISO/IEC 9899 - Programming languages – C, http://www.open-std.org/jtc1/sc22/wg14/www/standards

10 Appendices

All the appendices can be found on the enclosed CD, in their respective directories.

Appendix A WebStress specification

`webstress_specs.pdf` – version 1.0, May 9, 2006

Appendix B User's Guide

`users_guide.pdf` – version 0.3, May 28, 2007

Appendix C User's Reference

`users_reference.pdf` – version 0.3, May 28, 2007

Appendix D Test configurations and scripts used

Slovak Cadastral Portal

```
kapor.conf
kapor.script
apartment.txt
building.txt
land.txt
owner.txt
proceeding.txt
title.txt
```

RUBiS

```
rubis.conf
rubis.script
```

Appendix E WebStress distribution

`webstress_0.3.tar.gz` – version 0.3