JTRACER: A FRAMEWORK FOR AUTOMATIC TEST GENERATION FOR SECURE WEB APPLICATIONS

by

Edward Javier Herrera Aguirre

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER ENGINEERING

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS
2009

Approved by:

Bienvenido Vélez, PhD President, Graduate Committee	Date
Manuel Rodriguez, PhD Member, Graduate Committee	Date
Jaime Seguel, PhD Member, Graduate Committee	Date
Pedro Vásquez, PhD Representative of Graduate Studies	Date
Isidoro Couvertier, PhD Chairperson of the Department	Date

ABSTRACT

JTRACER: A FRAMEWORK FOR AUTOMATIC TEST GENERATION

FOR SECURE WEB APPLICATIONS

By

Edward Herrera Aguirre

Web application systems are one of the most ubiquitous software systems in use today. In contrast to traditional software systems, web application systems can evolve rapidly due to changes usage demands. Currently many tools and techniques has been developed for testing of Web applications however, the session generation data is still the more significant aspect for Web application testing. In this thesis we introduce JTracer, a framework for automatic test generation for secure Web applications. We have developed tools and techniques for automatically generating testing traces that could be used to measure and thus improve the tolerance of Web applications to sudden increases in load. Several algorithms that generate session data from logs files have been characterized showing the scenarios where they can suite better and finally, an algorithm for generating artificial session data is implemented.

RESUMEN

JTRACER: UN MARCO DE TRABAJO PARA LA GENERACION AUTOMATICA DE PRUEBAS PARA APLICACIONES WEB SEGURAS

Por

Edward Herrera Aguirre

Las aplicaciones Web son los sistemas de software más extendidos y utilizados hoy en día. Contrariamente a los sistemas de software tradicionales, las aplicaciones Web pueden evolucionar rápidamente debido a cambios en las demandas de uso. Actualmente muchas herramientas y técnicas han sido desarrolladas para hacer pruebas a las aplicaciones Web sin embargo, la generación de datos de sesiones de usuario aún es el aspecto más significativo para las pruebas a las aplicaciones Web. En este trabajo de tesis introducimos JTracer un marco de trabajo para la generación automática de pruebas a aplicaciones Web seguras. Hemos desarrollado herramientas y técnicas para generar automáticamente guías que pueden ser usadas para medir y mejorar la tolerancia de las aplicaciones Web a incrementos inesperados en la carga que reciben. Una variedad de algoritmos que generan pruebas servidores han sido caracterizados para mostrar los escenarios donde dichas pruebas puedan desempeñarse óptimamente y finalmente, un algoritmo para generar sesiones artificiales ha sido implementado.

To Nadia my wife, the great love of my life. To my parents Edward and Julia for the love, patience and guidance. To my sister Yubelly, brother Mijail and for Peti.

ACKNOWLEDGEMENTS

I would like to thank God for giving me the strength and the faith to accomplish this task. I thank to my wife Nadia for their unconditional love, patience, support and understanding. Thanks to my parents Edward and Julia for give me the life, their infinite love and their advices, thanks to my sister Yubelly and brother Mijail for accept me as I am. I wish to thank to my advisor, Dr. Bienvenido Vélez for his support, advice, and patience, Dr. Manuel Rodríguez, and Dr. Jaime Seguel to accept be part of my committee and thrust in me. Special thanks to Dr. Ismael Pagán Trinidad for the economical support, his friendship and understanding.

Table of Contents

Table List

Figure List

1 INTRODUCTION

1.1 Overview

Web application systems are one of the most ubiquitous software systems in use today. Since they appeared they have grown quickly and have evolved faster than other software systems. Day to day more information systems are being supported by this technology and most of the information systems are likely to be supported by this technology in the future. As they become adopted by more and more companies, they have become more complex and sophisticated. In many cases their success is crucial for the success of the company. Thus ensuring the reliability and robustness of the Web application systems is a big concern for companies.

Although traditional techniques for testing software systems have been used and proved for a long time, these techniques are no longer adequate for testing web application systems. In contrast to traditional software systems, web application systems can evolve rapidly due to changes usage demands. They also require more complex maintenance due to their heterogeneous, distributed, concurrent, and platform-independent nature. All those factors demand more complex techniques for web application testing.

We are particularly interested in developing tools and techniques for automatically generating testing traces that could be used to measure and thus improve the tolerance of a Web application to sudden increases in load.

As example, one web application designed to manage a few hundred users could suddenly find itself managing millions of users. This increment may trigger many scalability problems in the web application that couldn’t possibly be predicted unless an appropriate tool is available.

1.2 Problem Statement

Sudden increments in the number of concurrent users that Web applications systems can support have raised their size and complexity. Such complexity has forced developers to spend more time testing and validating.

No existing technique or method has proven to uncover all errors or bugs in web application systems. Most testing methods serve a specific testing purpose. Manually test case generation is a common technique for validation and verification of software systems when used for testing Web applications. However these techniques are very labor intensive and often can result in test suites that are not representative of real usage patterns. Techniques for automatically generating realistic testing suites are thus mighty desirable.

Web applications service uses interactions in the form of sessions comprising several requests. A session typically starts when the user logs in and ends when he/she logs out. This type of interaction requires a protocol for maintaining the relationship among multiple requests belonging to a given session. The Web application must be able to associate each request that it receives with a particular session and a particular user even when those request may arrive at the server in any order.

Manually creating a testing user session is a complex task. Tools that allow to accomplish this task automatically requires the use of proxies, capture-replay tools or scripts written by the software testing engineer. Using proxies or capture-replay tools requires simulating the user interaction with the Web application, hence the user session is created in a virtual fashion. That is each user session is created following a set of steps according to an artificial script which may necessarily reflect the real user interaction with the web application.

Our research is an attempt to automate the process of user session test creation and execution by automatically instrumenting a Web application to collect real user interaction traces even when the communication channel is encrypted.

1.3 Proposed Solution

The proposed solution is to log the user session from inside the Web application. This is accomplished by automatically injecting code after a static analysis of the Web application source code. The produced log is then processed to get the real world user session. The log file will be translated to a .jmx file which is a XML file used to describe test cases for the JMeter [18] testing tool. This test generation process is transparent to the user of the Web application system as well as to the software testing engineer.

Once the log file is captured the log data is used to create various types of test suites that can simulate various types of access patterns. Also the log data can be analyzed by the engineer in order to learn about user behavior in order to create completely new artificial sessions that mimic such behavior.

The thesis relies on the JMeter testing tool for simulating the interaction between users and Web application systems. JMeter [18] is an easy to use open source solution used broadly for load testing of Web applications; it has an active community that maintains the application up to date. This tool allows us to validate and compare the sessions generated from the real traffic with the obtained from the Web application itself. However many other tools exist in the market [32] that can serve to this purpose, many of them are open source too. However those don’t have the support that JMeter has while the commercial tools are expensive.

1.4 Contributions

The main contribution of this research is simplifying the process of creating user session test suites and catching real world scenario user sessions, even when the protocol is encrypted. As explained before, creating a user session manually is a cumbersome task and creating the user session relying on proxies or spies generate security risks. A better way to create a trustworthy user session is by generating it from inside the Web applications secure domain. We implement and test various algorithms to generate user sessions test suites from the log data captured. We present the results of various experiments that validate the correctness of our approach and evidence its advantages using other testing techniques.

1.5 Thesis Structure

The remainder of this document is organized as follows: In Chapter 2 we present a general overview of available articles related to this thesis, along with the theoretical background for the reader to better understand of the terminology and scope of this work. Chapter 3 introduces the JTracer framework and describes their phases and components. Further the functionality of each component is described in order to show their importance in the proposed solution to the problem stated above. Chapter 4 presents the different experiments that were carried out in order to test the proposed thesis and to evaluate and analyze their results. Finally Chapter 5 presents a summary of the conclusions and directions for future work.

2 RELATED WORK

Through this chapter some concepts and previous works related to the thesis work will be shown.

2.1 Web application

Is an application stored on a server and accessed mainly using a web browser, composed by web pages logically connected which can be delivered over a network like the internet or an intranet. A web application is commonly structured as a three-tier application comprising a User Service tier to access to the application, a Business Service tier to carry out complex activities and a Data Service tier which allows data storage and retrieval.

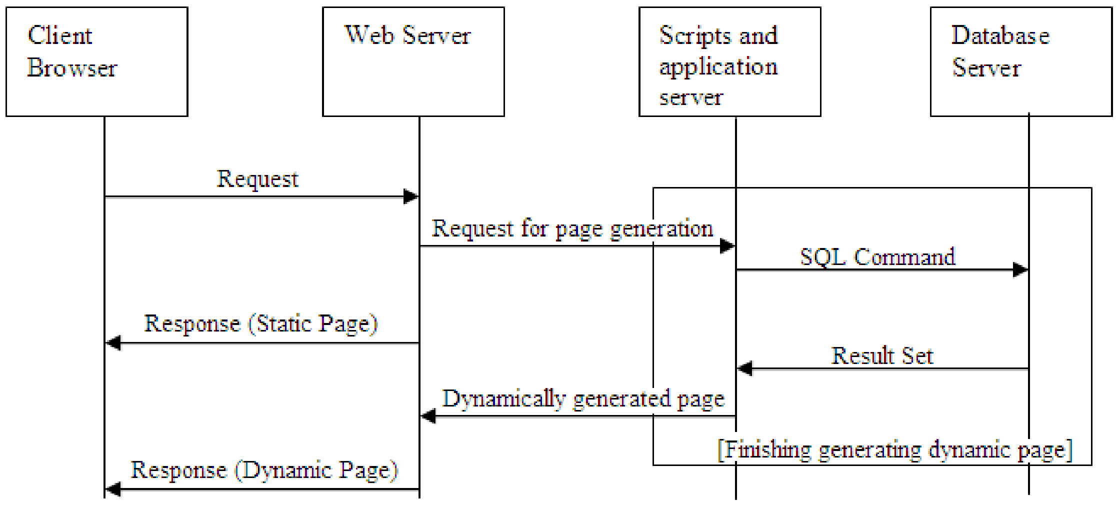

Client Browser
Web Server
Scripts and application server
Database Server
Request
Request for page generation
SQL Command
Response (Static Page)
Result Set
Dynamically generated page
[Finishing generating dynamic page]
Response (Dynamic Page)

Figure 2.1 Sequence diagram representing the interaction with a web application

In the Figure 2.1 Sidat [25] shows how a simple web application works, the client sends a request to the server through a web browser. The server process the request and delivers contents to the client. The contents are usually delivered in a markup-language form as HTML, these are interpreted by the client's web browser and shown like a web page [21]. The server's processing of the request sent by the client is a complex task, specially if the client's request include data provided by the user, in such case, the application server, the database management system and a set of scripts have to collaborate to generate a dynamic web page that fits the client's request.

2.2 Web application testing

Web application testing groups a set of tests over Web application systems which include functionality, usability, interface, compatibility, performance and security testing. For a web application system be robust, faster, reliable and eye-catching have to be successful over these tests. Although those tests are used for testing traditional software systems too, those can't be used in the same way on Web applications systems because of their characteristics [17] [19]; Testing Web applications are more complicated than testing traditional programs because of their heterogeneous, distributed, and concurrent nature along with their capacity to support hypermedia and be accessed by hundred or even thousand of users at the same time. Due to the previous aspects, testing Web applications requires to increase the complexity and to specialize the techniques used by the traditional software testing.

2.2.1 *Models based methods for Web application testing*

Many techniques have been developed for Web application testing; some of them rely on models for high level representation of Web applications.

Di Lucca et. al. [10] propose a test model for a web application representing at a coarse grained view, the web pages and a finer grain level, the inner components, scripts, links and applets. Based on that model, functional testing is carried out in two phases. In a first phase functional tests are created for the web pages of the web application and a model of the functional requirements is used to determine the expected behavior of the pages. In a second phase for each use case of the application, functional and test cases are executed on the web pages that implements such uses case. A Web application analyzer, test case generator, tests executor also have been developed to validate their solution.

As Di Lucca et. al. [10], the work of Ricca and Tonella [27] is based on models of Web applications where the definition of the testing criteria and the generation of the test cases rely on the internal structure and data flows of the Web application. Hence the model allows them first realize a static analysis where unreachable pages can be detected, data dependencies are identified, navigation correct order are validated and finally shorter paths are determined. Later, given a test criterion, a set of tests consistent of URL and input values are compared against the internal structure of the web application to determine if every page is visited at least once, if each hyperlink of each page is visited at least once and finally if every path in the site is visited at least once. This ensures that all paths that satisfies a determinate criterion are validated before the Web application is deployed.

The work of Conallen [4], Baresi et. al. [2] and Li et. al. [20] are focused at propose models for Web Applications by extending the Unified Modeling Language UML using its formal extension mechanism, hence web-specific components can be integrated with the model, concepts borrowed for other methodologies are added and finally Web applications are described from different levels of abstraction.

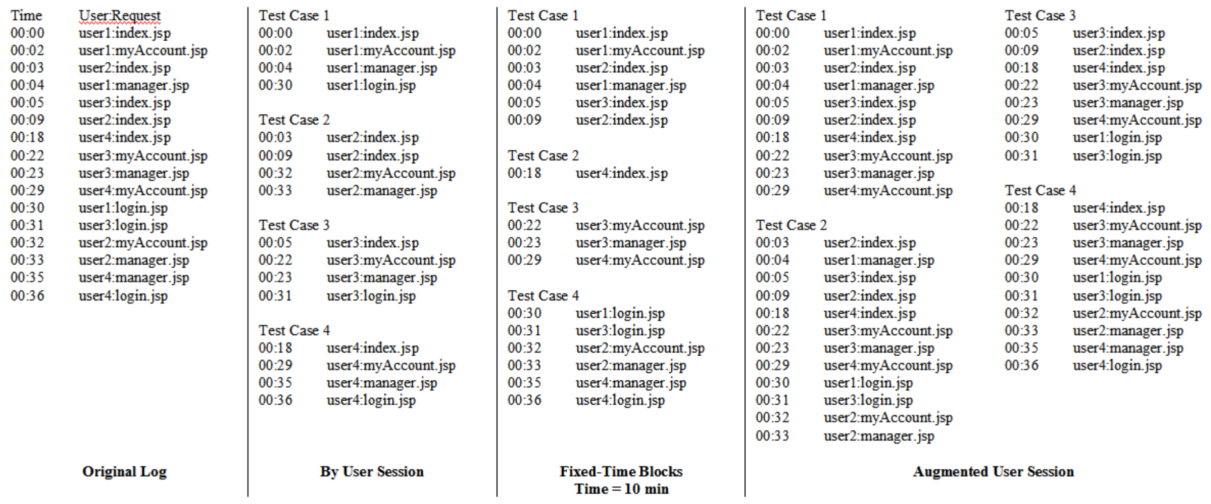
Chien-Hung et. al [6] [5] extends data flow analysis techniques for testing traditional software systems to the elements that compose a Web application, they proposed a model called WATM that capture Web applications tests artifacts so data flow test cases can be derived from them. Similarly [26] propose a methodology of Model-Driven Testing for Web applications, they proposed a model called WANM that describe the relations among web pages and the links and forms in each web page. Additionally deployment and test control models are proposed. All these models are used for applying the MDT process to Web application testing. A test engine executes test cases based on the models defined previously. The drawback of both models is the complexity for creating test data.

As we have seen so far, the tools developed by those researches allow us to generate semi-automatically the test cases by analyzing the internal structure of the Web application or by reverse engineering of the Web application. However such test cases are not derived from a real world scenario, it means they not necessary reflect the interaction of the users with the web application system. Those techniques require that the web testing engineer has knowledge of the internal structure of the application as well as the documentation of the system functionality.

2.2.2 User session based methods for Web application testing

Although the techniques mentioned so far have shown promising and encouraging results, such techniques are expensive to implement and to program moreover require much intervention from the web testing engineer. Elbaum et. al [11][12] proposed web application testing using data gathered as the user operates the Web application, their work showed that user session data gathered as users operate web browsers can be used to produce test suites more effectively than with the model based techniques and with less effort, they conclude that both techniques are complementary because the faults detected for both techniques differ.

Due to the sequential nature of the logs files created after the capture of the user interaction with the web application, tests using those data are executed in a replay fashion, it means in the same order than appears in the log file. The work of Sprenkle et al. [34] showed that creating test cases that test various levels of multi-user interaction and state dependencies provide more coverage than the user-session based technique. They propose three techniques for partitioning the log file, Fixed by block that strictly partition the log into fixed-time-length block, Server inactivity threshold where the first request of each block differs a time “t” of the last request of the previous one and, Augmented user session where each group include the request whose time is between the initial and end time of each user session. The figure 2.2 illustrates an example of the session classification according to these algorithms.



Original Log

Time	User:Request
00:00	user1:index.jsp
00:02	user1:myAccount.jsp
00:03	user2:index.jsp
00:04	user1:manager.jsp
00:05	user3:index.jsp
00:09	user2:index.jsp
00:18	user4:index.jsp
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:29	user4:myAccount.jsp
00:30	user1:login.jsp
00:31	user3:login.jsp
00:32	user2:myAccount.jsp
00:33	user2:manager.jsp
00:35	user4:manager.jsp
00:36	user4:login.jsp

By User Session

Test Case 1	
00:00	user1:index.jsp
00:02	user1:myAccount.jsp
00:04	user1:manager.jsp
00:30	user1:login.jsp
Test Case 2	
00:03	user2:index.jsp
00:09	user2:index.jsp
00:32	user2:myAccount.jsp
00:33	user2:manager.jsp
Test Case 3	
00:05	user3:index.jsp
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:31	user3:login.jsp
Test Case 4	
00:18	user4:index.jsp
00:29	user4:myAccount.jsp
00:35	user4:manager.jsp
00:36	user4:login.jsp

Fixed-Time Blocks
Time = 10 min

Test Case 1	
00:00	user1:index.jsp
00:02	user1:myAccount.jsp
00:03	user2:index.jsp
00:04	user1:manager.jsp
00:05	user3:index.jsp
00:09	user2:index.jsp
Test Case 2	
00:18	user4:index.jsp
Test Case 3	
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:29	user4:myAccount.jsp
Test Case 4	
00:30	user1:login.jsp
00:31	user3:login.jsp
00:32	user2:myAccount.jsp
00:33	user2:manager.jsp
00:35	user4:manager.jsp
00:36	user4:login.jsp

Augmented User Session

Test Case 1	
00:00	user1:index.jsp
00:02	user1:myAccount.jsp
00:03	user2:index.jsp
00:04	user1:manager.jsp
00:05	user3:index.jsp
00:09	user2:index.jsp
00:18	user4:index.jsp
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:29	user4:myAccount.jsp
Test Case 2	
00:03	user2:index.jsp
00:04	user1:manager.jsp
00:05	user3:index.jsp
00:09	user2:index.jsp
00:18	user4:index.jsp
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:29	user4:myAccount.jsp
00:30	user1:login.jsp
00:31	user3:login.jsp
00:32	user2:myAccount.jsp
00:33	user2:manager.jsp
Test Case 3	
00:05	user3:index.jsp
00:09	user2:index.jsp
00:18	user4:index.jsp
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:29	user4:myAccount.jsp
00:30	user1:login.jsp
00:31	user3:login.jsp
Test Case 4	
00:18	user4:index.jsp
00:22	user3:myAccount.jsp
00:23	user3:manager.jsp
00:29	user4:myAccount.jsp
00:30	user1:login.jsp
00:31	user3:login.jsp
00:32	user2:myAccount.jsp
00:33	user2:manager.jsp
00:35	user4:manager.jsp
00:36	user4:login.jsp

Figure 2.2 Sessions classification according to algorithms presented by Sprenkle et al.

The main drawback of the User session based technique is the cost and effort in collecting, analyzing and replaying the vast number of user sessions; therefore redundant test cases should be removed without losing fault detection and coverage properties. Techniques that reduce the set of user session gathered [14][29] try to represent with fewer sessions the behavior of many sessions.

The Harrold et al. [14] technique first define each URL of the web application as a requirement, then an optimal set of user sessions that meets the fewer requirements is determined, in each iteration a new user session is added to the optimal set until the set covers all the requirements. The cardinality of each requirement is the number of user sessions that cover such requirement. For each iteration the new session added to the optimal set is the one that covers the most requirements with lower cardinality. The main drawback of this technique is the increment in the time of processing as the number of sessions grows.

To avoid large processing as the number of sessions grows, the work of Sampath et. al. [30] uses incremental concept analysis techniques to address the test case reduction problem. Their technique is based on incremental concept formation algorithms which created a reduced set of test cases as the test cases are added to the set. The algorithm takes as input the session to be added and a table where the columns are the URLs of the Web application and the rows are the user sessions, each entry in the table is true if the session requests the URL. The table is represented as a graph called Latice that has in their high levels the URLs requested by almost all the sessions and in their low levels the URLs requested by the fewer or none sessions. As output, the algorithm returns the Latice with nodes added and deleted according to the concept analysis approach. Each iteration of the algorithm always maintains the optimal reduced set.

Sampth et al. [29] make an interesting contribution by analyzing the user session clustering, in their research they shown that choosing a user session from a cluster will not result in loss of the attributes represented or covered by other user session belonging to the same cluster, the research shown that clustering could be done based on URLs as attributes and based on the URL and name-value as attribute. This research will be useful to formulate new techniques for reduction of test sets.

Other researches have tried to synthetically create user session data starting with the already existent data, Elbaum et al. [12] showed than merging and splitting user session to generate additional test session data, were not as effective as the primarily user session technique proposed early by themself. Sant et al. [31] automatically builds statistical models of user sessions and automatically derives test cases from these models, the study demonstrated that those tests achieve high coverage and accurately model user behavior. They build Markov models starting with web log data; this is done by using statistical language learning algorithms to construct Control models and Data models that address the user session generation. The Control models represent the possible sequence of URLs that are visited when the user navigates the application and the Data models represent the set of name-values values in a request for a specific URL like this User session are created according to the distribution learned from the web log.

2.2.3 Others methods for Web application testing

Others methods for Web application testing rely on static code analysis as proposed by Deng et al. [8] who after an analysis of the source code of the Web application create and execute dynamically test sessions relying on graphs that represents the URLs and the links between them. Halfond and Orso [13] get value-domain pairs by analyzing the source code of a Web application, those pairs could be used by software testing engineers to supply data that exercises the system.

Xiaoping and Hongming [39] use formal specifications for testing Web applications, the test process, the security, the functionality and the performance are specified using a formal language defined by themselves. A test engine reads test specifications from a XML file and generates test cases based on such specifications; the results are compared with the expected results which are specified using the same formal language. Although the research has not been proved on large and complex Web applications, the results obtained are promising.

Yu et al. [40] proposes use agents for Web application testing, where specific test agents are generated from abstract classes. Each test agent takes charge of testing a particular type of Web document or object by certain testing methods. Each test agent of high level is able to create test agents of low levels to execute the test in lower levels. The Web application testing is performed based on four levels: function, cluster, object and web application levels. The authors specify the structure and items of each agent level. The approach is flexible and extensible because any new function could be added through a new test agent.

As we can see so far, many research and many promising results has been accomplished, and the web application testing field is still being exploited and researched, day-to-day new tools are created and proposed [32]. No matter which technique is used, all of them rely on the user session data. The user session data is the element that finally exercises and validates the web application system. The thesis is aimed at catch and replicate such user sessions making it a simple and inexpensive task for the Web application testing engineer. Later the Web application testing engineer could use them as the main ingredient along with the different techniques to test the web application systems.

2.3 Performance testing

Performance tests are usually described as belonging to one of the following three categories:

- **Performance testing**. Meier et. al [22] defines performance testing as the type of test for determining and validating the speed, scalability and/or stability characteristics of the system or application under test. The main objective is to meet the response times, throughput and resource-utilization levels expected for the software or product. The results are useful to estimate the resources needed to support the application operation.

- **Load testing**. According to Meier et. al [22] this type of testing "determines and validates the performance characteristics of the system or application under test when subjected to workloads and load volumes anticipated during production operations."

- **Stress testing**. After the application or system under test has been successful on load testing, a set of tests that simulates conditions beyond those anticipated during production operations are executed, stressful conditions as limited memory, server crash, hardware fault, among others are simulated. Then is possible determine how the application will fail and the indicators that can be monitored to warm and avoid failures [22].

To predict performance of a Web application, Benchmarks or Performance models can be used.

Benchmarks are standards workload models used by the industry to test existing architectures with expected traffic, some tools as SpecWeb99 [36], TPC-W [23], WebStone [38] and WebBench [37] use file-list as supply for the workload characterization, while Surge [3] use mathematical distributions to represent the main characteristics of the system under test. As can be noticed, benchmarks don't provide realistic workload because of the vast types of Web applications. Although TPC_W [23] provide a realistic workload characterization by simulating an e-commerce Web application, is far from simulating all e-commerce Web applications systems. Hence using a Benchmark for stress or load testing could produce non-reliable results if the Web application would not adjust accurately to some existing architecture with expected traffic.

Performance models use analytical or simulative models to predict the performance of a Web application. By replaying set of sessions against the server under evaluation measures such as throughput, response times, disk storage and computational resource, can be derived, for this many tools with diverse features have been implemented, a complete list can be found on [32].

Only accurate predictions about Web application performance in a production environment is achieved when realistic workload models are simulated [22]. Andreolini et al. [1] presents some methods for design and testing of Web applications and for improvements to do for the already deployed Web applications to satisfy performance constraints. However, a Web application is considered well performed only after passing a realistic workload.

Relying on information extracted from logs files Ruffo et al. [28] proposed WALTy a set of tools for performance analysis of Web applications. Their research shows that representative traffic can be simulated using Customer Behavior Model Graphs extracted from log files. Customer Behavior Model Graphs first proposed my Menasce et al. [24] are session-based representation of user navigational patterns and proposed for workload characterization of e-Commerce sites. Using those models trace-based synthetic workloads were performed. After testing their tool against other tools, they found different measures concluding that the results depend on the chosen virtual user behavior. Hence the choice of adequate user sessions could perform a valid workload generation.

2.4 Scalability of Web servers

Is the ability of the web server to maintain the site operable, available, reliable, and efficient as the number of simultaneous requests increases. “Scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations. Increased capacity should be in proportion to the cost, and quality of service should be maintained” [37].

To determine the scalability of a Web application is not necessary to perform a load testing by levels of numbers of users until the performance desired is met. Do it in that way is time-consuming and very expensive. By using load testing along with analytic or simulation performance models the scalability of a Web application can be predicted.

3 DESIGN AND IMPLEMENTATION OF JTRACER

3.1 Overall architecture of JTracer

This chapter presents the overall JTracer architecture. Each component and its roles is explained in detail. The JTracer incorporates tools and components to automate the creation of load testing suites, their execution against the application server and the reporting of errors produced by such execution. JTracer is capable of feedback of the response of a Web application under different load levels and user behaviors.

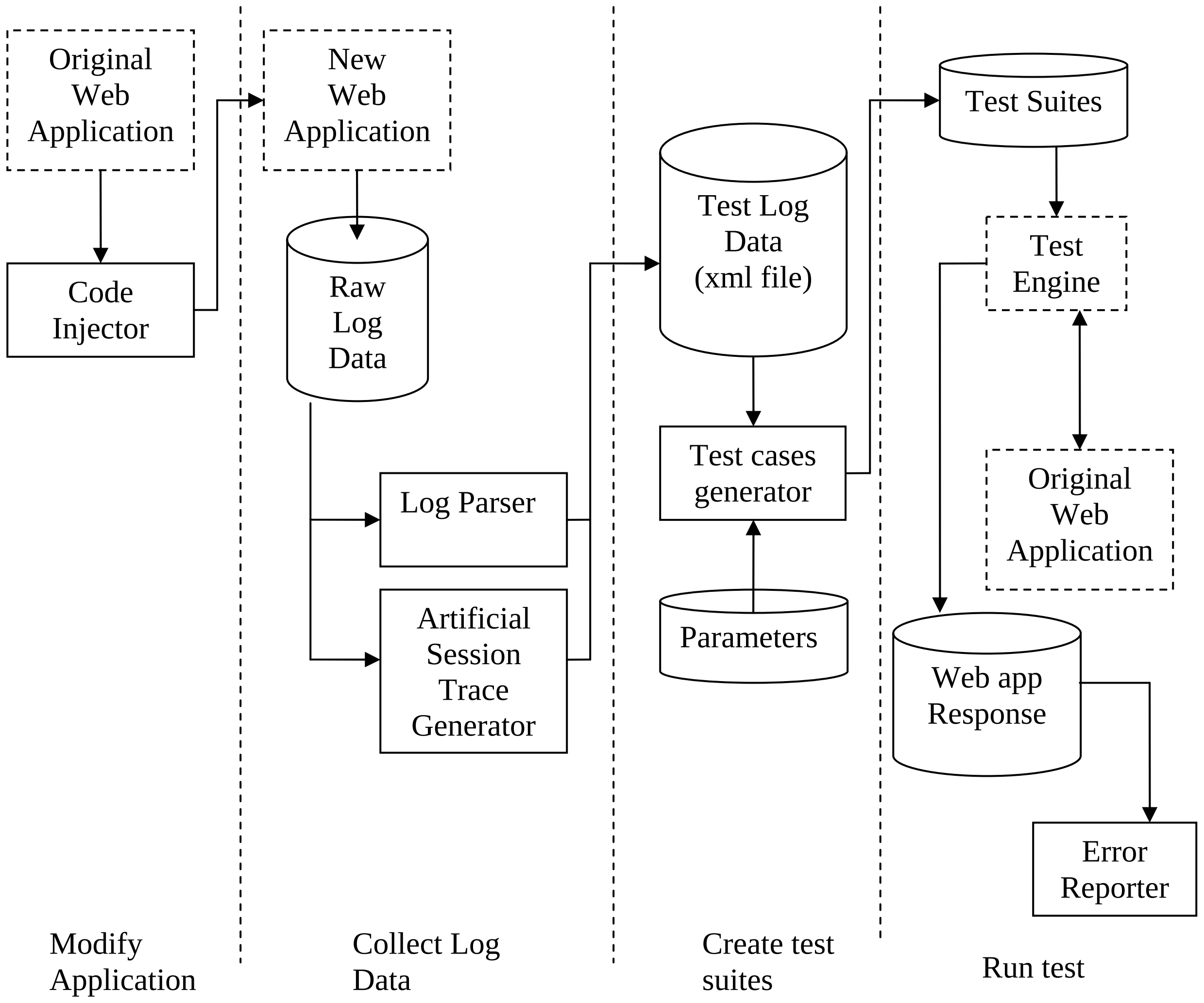


Figure 3.1 Overall System Architecture

The process of generating a test suite starts with the capture of the real session data. This is done either by spying the communication port on non-encrypted protocols or by generating the session data directly from the Web application. Spying the port is an easy task and software exists in the market that allows to do it. If we want the session data would generated directly by the Web application, it has to be modified to provide such functionality. This functionality can be automatically attached to the Web application by inserting the source code necessary to log the user interactions.

The real session data captured earlier (Raw Log Data), have to be translated to a more structured file, this is done by using a parser implemented into the Log Parser module. Also the real session data could be the supply for algorithms that generate artificial session data implemented into the Artificial Session Trace Generator. The data produced after processing the raw log data will be translated according to a set of parameters to a XML file representing a test plan for the Test Engine in charge of exercising the Web application.

As the Test Engine exercises the Web application its responses are captured. Then these responses are analyzed and summarized by the Error Reporting module.

3.2 The Code Injector Module

The Code Injector Module (CIM) instruments a Web application with code to log session request as real users interacts with the Web application. The first mission of JTracer is designed to work with any Web application implemented under the J2EE Java platform and using JavaServer Faces technology. However support for other technologies could be implemented in a similar fashion.

The CIM performs a static analysis of the Web Application source code to determine the exact location where additional source code that logs the user interaction with the Web Application will be inserted. The source code injection process is divided into two phases. In the first phase, the algorithm takes as input a set of files that implements the Web Application user interfaces (.jsp files) and returns the set of managed beans components and the methods which would trigger some action caused by an HTTP request. In the second phase, the source code that logs the user interaction is injected into the user interface files as well as into the correspondent managed beans methods. A general overview is shown in Figure 3.2.

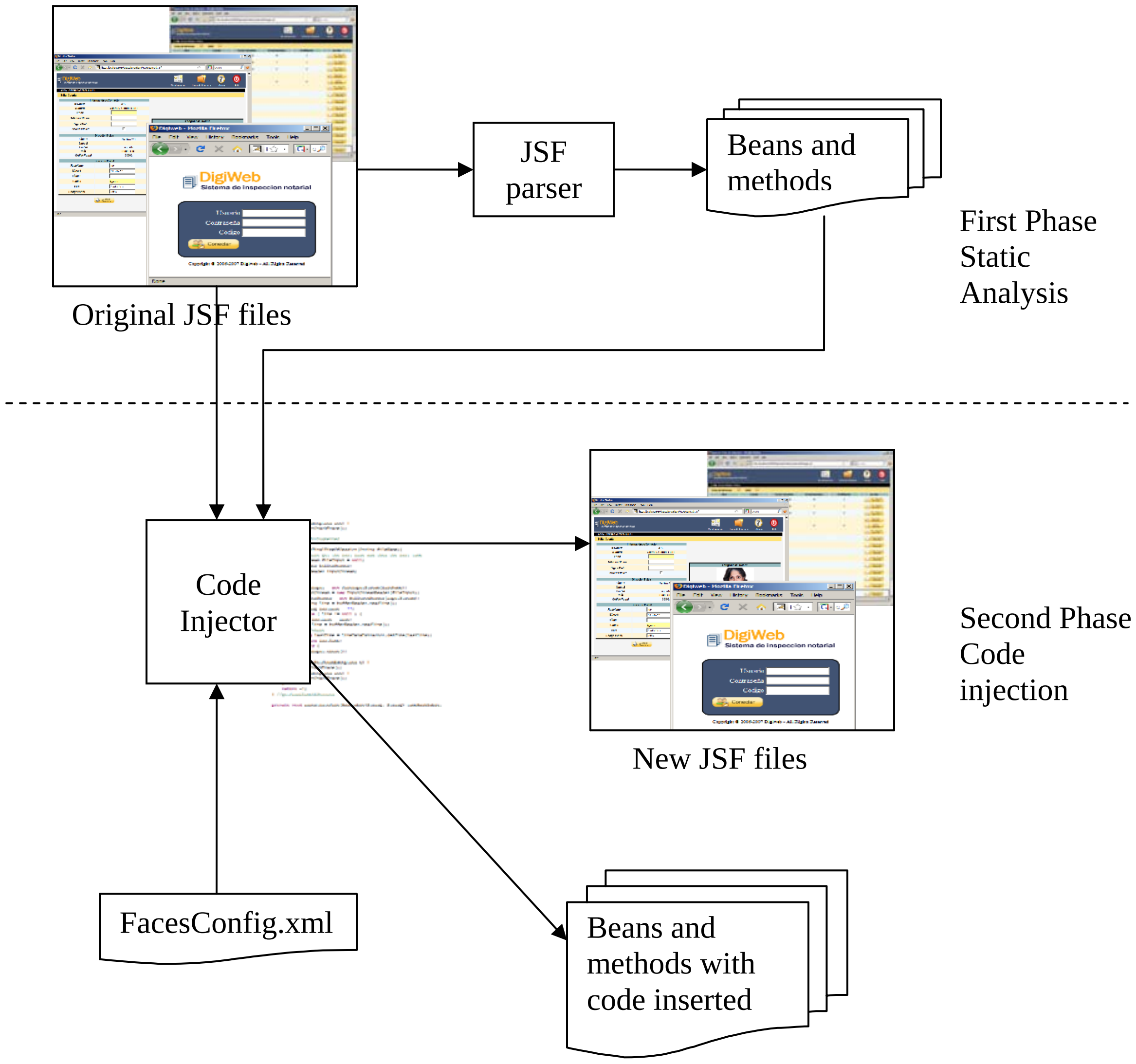

Original JSF files
JSF parser
Beans and methods
First Phase Static Analysis
Code Injector
New JSF files
Second Phase Code injection
FacesConfig.xml
Beans and methods with code inserted

Figure 3.2 Code Injector algorithm phases

To determine which beans and methods should be instrumented it is necessary to determine which user actions make a call to a method or a forward to another user interface file. The user interacts with the Web application by selecting a visual controller as a link, a menu command or a button, which triggers a new HTTP Request to the server. Such controllers and their actions are specified inside the user interface or JSP file as marks:

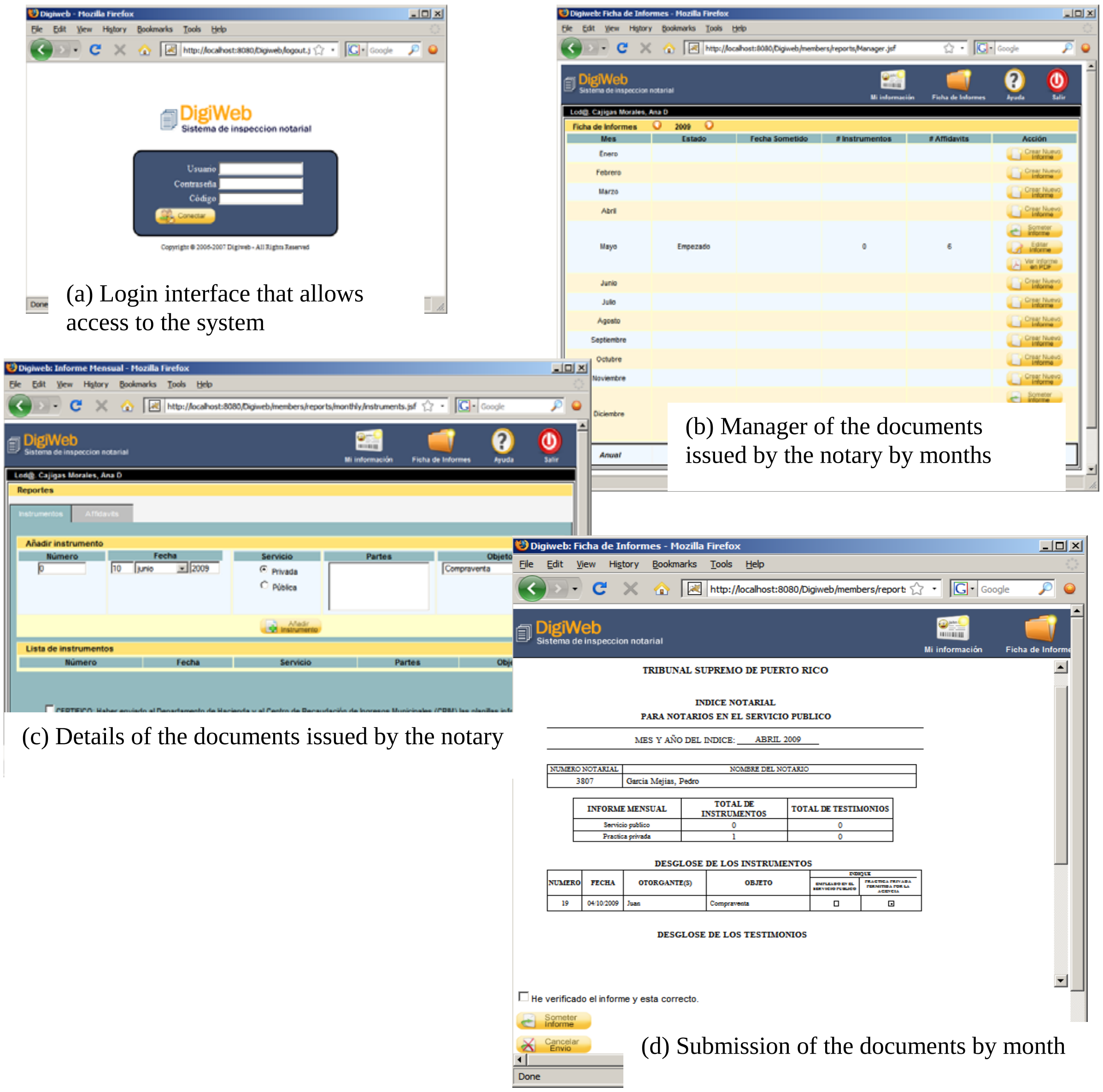
```
[<h: [component_name] action="cad">
```

Where “cad” could reference a Bean and a method name or a URL link. Thus “cad” has to be parsed to determine whether it is a method or an URL. In case “cad” references a method then the bean and the method name are added to the set mentioned previously along with their location in the source code. The bean location is extracted from the FacesConfig.xml file. In case “cad” references an URL location then the JSP implementing such URL will be added to a set of JSP elements to be processed further.

The algorithm traverses the set of BEANS, and for each method that implements a user action, source codes that implements the logging of the action as an HTTP Request is inserted. The source code inserted relies on the Faces Context class to get the current request, their headers and their parameters. The Faces Context class is a general Java Server Faces class that contains JSF-related request information among other request parameters.

Once all the managed beans have been processed, appropriate codes are inserted into the headers of the JSP files. However, no all the JSP files have to log user session data neither have to log the session data only one time. Later in this section will detail conditions that determine what user session data must be logged.

Before explaining the logging process we introduce to Digiweb [9] Web application system which we will use to illustrate the examples. This Web application is designed to manage and control documents (instruments and affidavits) issued by the notaries as well as their personal information. The system also controls the submission of the documents registered by month for no further modification. Figure 3.3 shows some interfaces that implement the Web application.


DigiWeb
Sistema de inspeccion notarial
Usuario
Contraseña
Código
Conectar
Mi información
Ficha de Informes
Ayuda
Salir
Reportes
Instrumentos
Affidavits
Añadir instrumento
Lista de instrumentos
TRIBUNAL SUPREMO DE PUERTO RICO
INDICE NOTARIAL
PARA NOTARIOS EN EL SERVICIO PUBLICO
DESGLOSE DE LOS INSTRUMENTOS
DESGLOSE DE LOS TESTIMONIOS
He verificado el informe y esta correcto.
Someter Informe
Cancelar Envío

(a) Login interface that allows access to the system

(b) Manager of the documents issued by the notary by months

(c) Details of the documents issued by the notary

(d) Submission of the documents by month

Figure 3.3 Digiweb Web application interfaces

When a notary wants to issue a document, he/she logs into the system (Figure 3.3 (a)), then select “Ficha de Informes” and then chose the month he/she wants to register the document in (Figure 3.3 (b)). Finally inserts the details of the documents as date of emission, the participants and the subject (Figure 3.3 (c)). All those documents are summarized by month and can be submitted for no further modification (Figure 3.3 (d)).

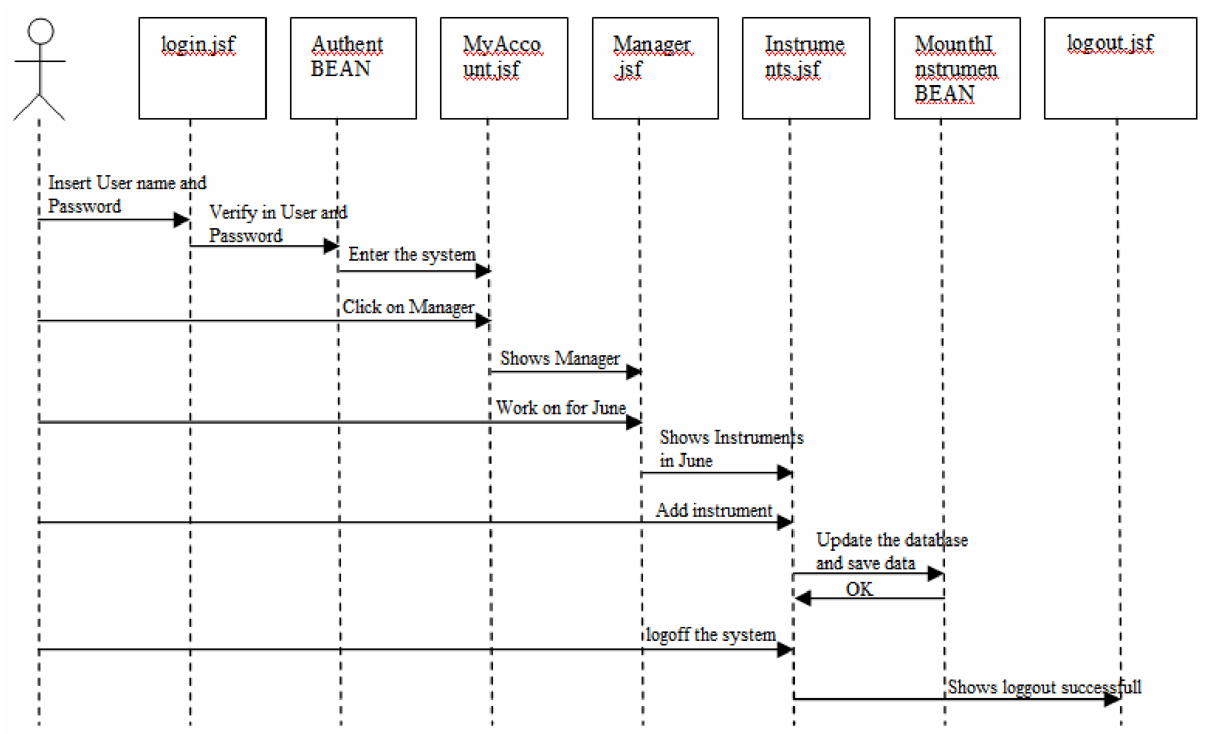


Figure 3.4 User inserting a new instrument in Digiweb Web application

Figure 3.4 illustrates how the user interaction is logged by JTracer. The diagram shows step by step the actions performed by a Web application and when the User inserts a new document into the system. Although many software components work together to implement the operations that complete a transaction, the Web interfaces are the only ones visible to the user.

Once the user accesses the Web Application, the login page is displayed. The user inserts his name and password and the Authentication bean authenticates his/her credentials. If both user and password are correct then the bean redirects the navigation to MyAccount page. Once MyAccount is shown the user pushes the link to go to the Manager page. In response the system shows the Manager interface. The user decides create a new instrument, and displays the Instruments page. Once the user completes the data requested for a new report the MonthlyInstrument bean stores the data in the database and redirects the navigation again to the instrument page after letting the user know the status of his operation. Finally the user logs out of the system.

Figure 3.5 shows the user interaction with the Digiweb, the corresponding pages and beans that work together as the user navigates the Web application and how the user actions are logged.

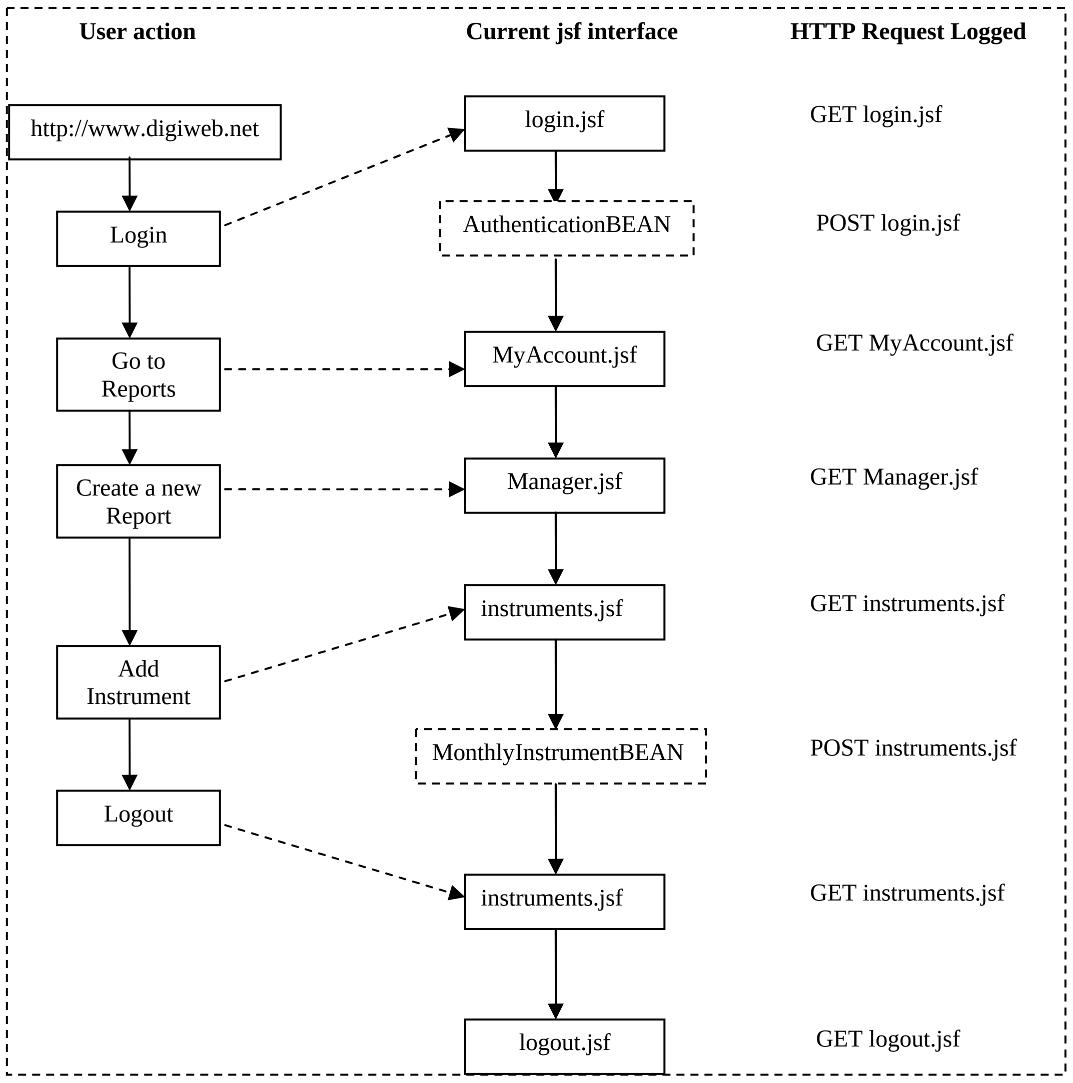


Figure 3.5 Log data as the user navigates the Web application

Table 3.1 shows at the right the HTTP Request logged by the JTracer logging algorithm. However, such logged session data doesn't reflect precisely the user interaction with the Web Application according to the JSF specification. As can be seen, each page and bean file log itself. However, according to the JSF specification pushing a button or a link have to trigger a POST request. For example, when the user is on the MyAccount.jsp page and clicks the link to go to Manager.jsp, the logged data will be “GET Manager.jsp”, however no logged data is registered for the action of clicking the link. We should remember that according to Java Server Faces specification a “POST MyAccount.jsp” must be registered in response to a click of a button or link. A more detailed explanation of the sequence of operations performed as the user navigates the Web application is presented in Table 3.2, in blank the requests that can’t be logged are shown, this happen because the interface log itself when is shown.

TABLE 3.1 Requests logged according to the user navigation

Action	Current Interface or BEAN	Request logged
User access to the application and insert user name and password	login.jsf	GET login.jsf
Bean authenticated the user and password and redirect to MyAccount.jsp	AuthenticationBEAN	POST login.jsf
The application shows the interface MyAccount.jsp	MyAccount.jsf	GET MyAccount.jsf
The user push a button to go to Manager	MyAccount.jsf	
The application shows the interface Manager.jsp	Manager.jsf	GET Manager.jsf
The user push a button to go to instruments	Manager.jsf	
The application shows the interface instruments.jsp	instruments.jsf	GET instruments.jsf
The user insert some data and push save button	instruments.jsf	
Bean stores the data into the database	MonthlyInstrumentBEAN	POST instruments.jsf
The user push button to logout the application	instruments.jsf	
The application logout	logout.jsf	GET logout.jsf

As can be noticed in Table 3.1 accessing the next page displayed must log two requests, one for the previous page and one for itself when a link or a button is clicked. However, if the request has already logged by a bean and, as mentioned above, the next page logs the request of the previous page and itself then such request will be logged twice. Also, some requests don’t have to be logged as when a page redirects the navigational flow. We summarize those cases and instrument them into the original algorithm as follows:

- If a JSP file redirects the navigation flow then such file does not have to be processed because it was not produced by a direct user interaction with the system.
- If a JSP file has a directive that finishes the session and redirects to another page then the next page does not have to be processed because it does not belong to the current session.
- If the method of the current request is GET and the previous request does not come from a bean, log along with the current request the previous request registered as POST request. It because this page has been called by clicking a link, a button or other visual component and according to the JSF specification this should produce a POST call in such page, and must be logged by the next page called.
- If the method of the current request is POST and the previous request comes from a bean, it would not log the current request because it has already been logged by the bean.

Finally Table 3.2 and Figure 3.6 show the pages and the beans and how they work together logging the user interaction with the Web application.

TABLE 3.2 Navigation flow logged according to the real user navigation

Action	Current interface or BEAN	Request logged
User access to the application and insert user name and password	login.jsf	GET login.jsf
Bean authenticated the user and password and redirect to MyAccount.jsf	AuthenticationBEAN	POST login.jsf
The application shows the interface MyAccount.jsf	MyAccount.jsf	GET MyAccount.jsf

The user push a button to go to Manager	MyAccount.jsf	Will Be Registered In The Next Request
The application shows the interface Manager.jsf	Manager.jsf	POST MyAccount.jsf GET Manager.jsf
The user push a button to go to instruments	Manager.jsf	Will Be Registered In The Next Request
The application shows the interface Instruments.jsf	instruments.jsf	POST Manager.jsf GET instruments.jsf
The user insert some data and push to save	instruments.jsf	Will Be Registered In The Next Request
Bean stores the data into the database	MonthlyInstrumentBEAN	POST instruments.jsf
The application shows the interface Instruments.jsf	instruments.jsf	GET instruments.jsf
The user push button to logout the application	instruments.jsf	Will Be Registered In The Next Request
The application logout	logout.jsf	POST instruments.jsf GET logout.jsf

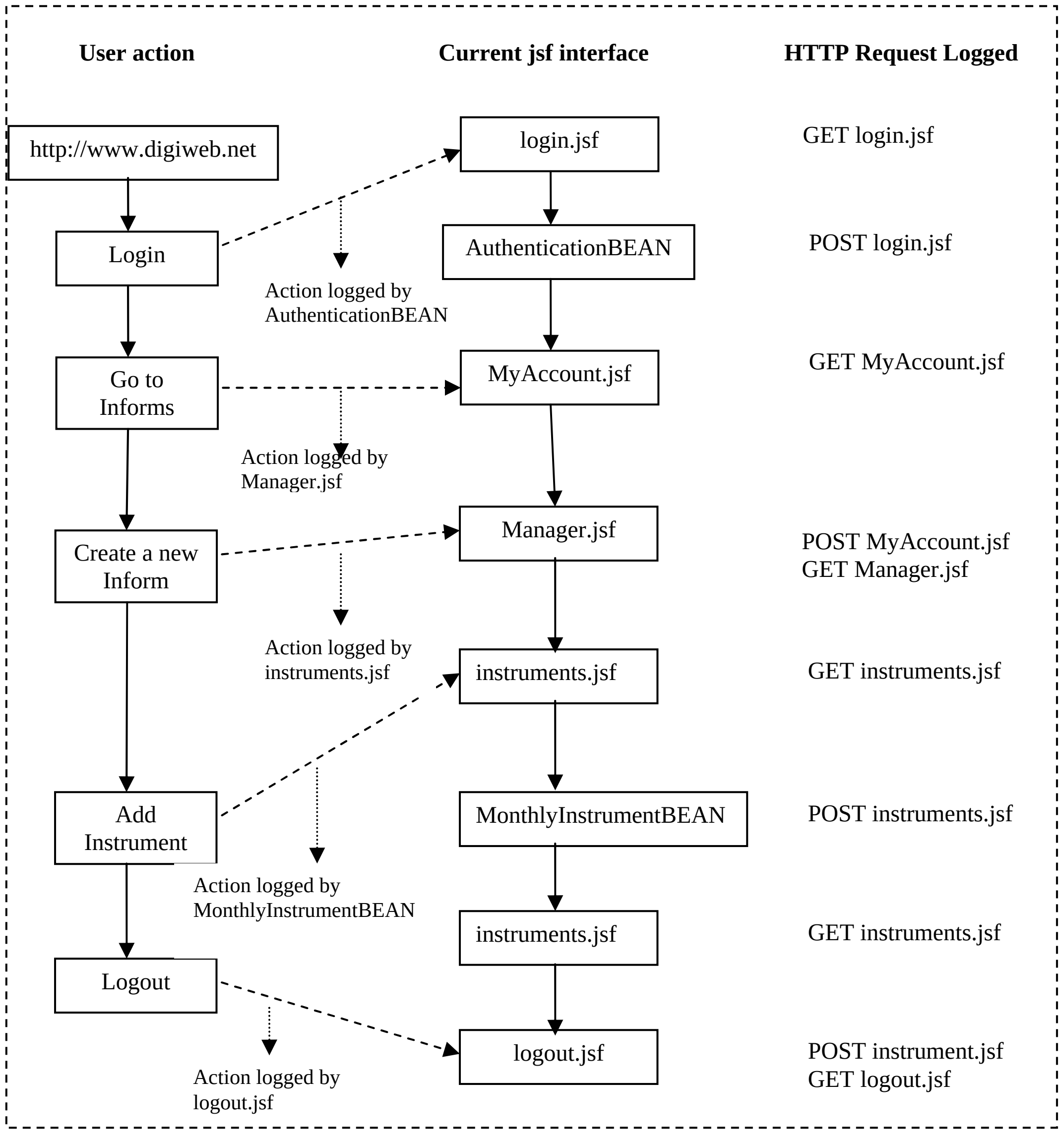


Figure 3.6 Log data according to JSF specification

The log file generated by the instrumented web application reflects the real user interaction with the Web application according to the JSF specification. This file will be the basis for the rest of the test suite generation process.

3.3 The Log Parser Module

The Log Parser (LP) extracts from the log file the data useful for load test generation, receives as input the raw log data and produces a new more readable and compact file that will be come the input to the following test case generation algorithms. The process is divided into two phases as shown in Figure 3.7. In the first phase, the tokenization phase, the list define, give example are extracted from the log file. During the second phase, the Reduction phase, the list of tokens produced in the first phase is organized in such a way that each HTTP Request is inserted into a line in a new file. Furthermore the HTTP Responses are excluded from the new log file. The new resulting log file is up to 12 times smaller than the original.

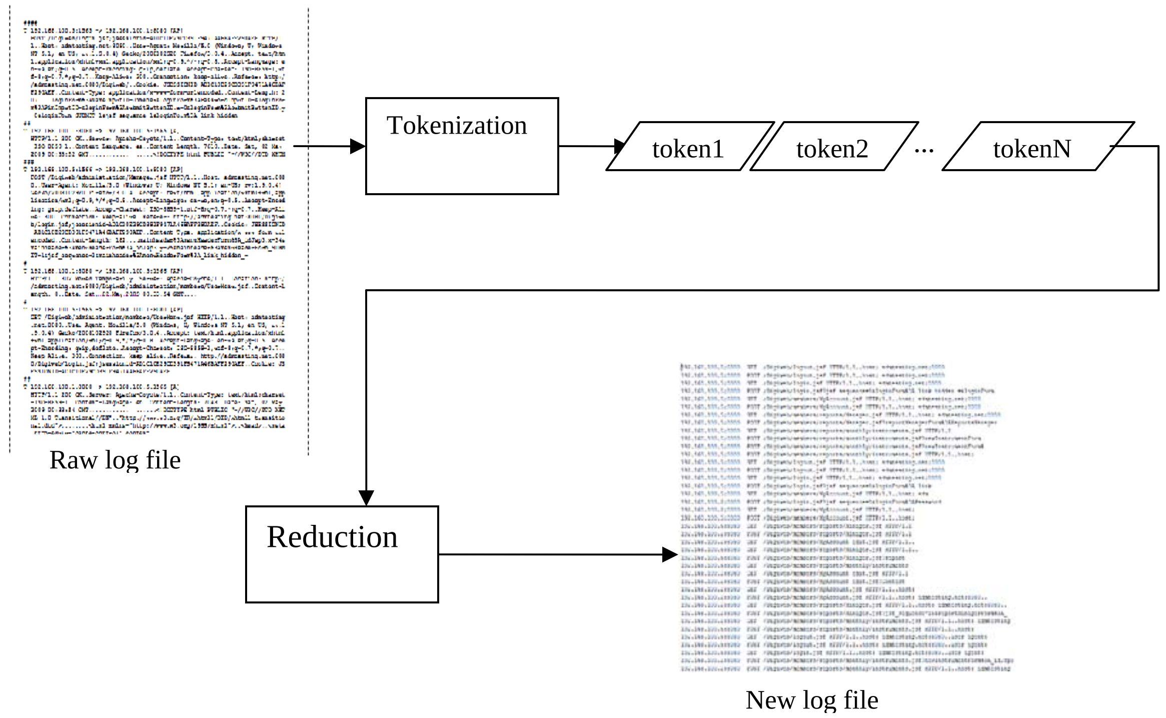


Figure 3.7 Log Parser process phases

Figure 3.8 shows a fragment of the log file. As can be seen the data in each request of the original log file is hard to read because is contains several lines, the URLs are not followed by their parameters but by the JSESSIONID and HTTP Request and HTTP Responses are intermixed in the same file. The goal of the LP module is to create a new log file more readable and flexible that can be used as input for the test cases generation algorithms and can be analyzed by a testing engineer to learn user behavior.

```
####
T 192.168.100.5:1565 -> 192.168.100.1:8080 [AP]
  POST /Digiweb/login.jsf;jsessionid=AD1C10E29CD391F9471A46BAFF290AEF HTTP/1.
  1..Host: admtesting.net:8080..User-Agent: Mozilla/5.0 (Windows; U; Windows
  NT 5.1; en-US; rv:1.9.0.4) Gecko/2008102920 Firefox/3.0.4..Accept: text/htm
  l,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8..Accept-Language: e
  n-us,en;q=0.5..Accept-Encoding: gzip,deflate..Accept-Charset: ISO-8859-1,ut
  f-8;q=0.7,*;q=0.7..Keep-Alive: 300..Connection: keep-alive..Referer: http:/
  /admtesting.net:8080/Digiweb/..Cookie: JSESSIONID=AD1C10E29CD391F9471A46BAF
  F290AEF..Content-Type: application/x-www-form-urlencoded..Content-Length: 2
  07....loginForm%3ANameInputID=Jimenez&loginForm%3APasswordInputID=&loginFor
  m%3APinInputID=&loginForm%3AsubmitButtonID.x=0&loginForm%3AsubmitButtonID.y
  =0&loginForm_SUBMIT=1&jsf_sequence=1&loginForm%3A_link_hidden_=
##
T 192.168.100.1:8080 -> 192.168.100.5:1565 [A]
  HTTP/1.1 200 OK..Server: Apache-Coyote/1.1..Content-Type: text/html;charset
  =ISO-8859-1..Content-Language: es..Content-Length: 7610..Date: Sat, 02 May
  2009 00:33:52 GMT...........  .....<!DOCTYPE html PUBLIC "-//W3C//DTD XHTM
###
T 192.168.100.5:1566 -> 192.168.100.1:8080 [AP]
  POST /Digiweb/administration/Manager.jsf HTTP/1.1..Host: admtesting.net:808
  0..User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.4)
  Gecko/2008102920 Firefox/3.0.4..Accept: text/html,application/xhtml+xml,app
  lication/xml;q=0.9,*/*;q=0.8..Accept-Language: en-us,en;q=0.5..Accept-Encod
  ing: gzip,deflate..Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7..Keep-Ali
  ve: 300..Connection: keep-alive..Referer: http://admtesting.net:8080/Digiwe
  b/login.jsf;jsessionid=AD1C10E29CD391F9471A46BAFF290AEF..Cookie: JSESSIONID
  =AD1C10E29CD391F9471A46BAFF290AEF..Content-Type: application/x-www-form-url
  encoded..Content-Length: 182....mainheader%3AmenuHeaderForm%3A_idJsp3.x=34&
  mainheader%3AmenuHeaderForm%3A_idJsp3.y=25&mainheader%3AmenuHeaderForm_SUBM
  IT=1&jsf_sequence=2&mainheader%3AmenuHeaderForm%3A_link_hidden_=
#
T 192.168.100.1:8080 -> 192.168.100.5:1566 [AP]
  HTTP/1.1 302 Moved Temporarily..Server: Apache-Coyote/1.1..Location: http:/
  /admtesting.net:8080/Digiweb/administration/members/UserMenu.jsf..Content-L
  ength: 0..Date: Sat, 02 May 2009 00:33:54 GMT....
#
T 192.168.100.5:1565 -> 192.168.100.1:8080 [AP]
  GET /Digiweb/administration/members/UserMenu.jsf HTTP/1.1..Host: admtesting
  .net:8080..User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1
  .9.0.4) Gecko/2008102920 Firefox/3.0.4..Accept: text/html,application/xhtml
  +xml,application/xml;q=0.9,*/*;q=0.8..Accept-Language: en-us,en;q=0.5..Acce
  pt-Encoding: gzip,deflate..Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7..
  Keep-Alive: 300..Connection: keep-alive..Referer: http://admtesting.net:808
  0/Digiweb/login.jsf;jsessionid=AD1C10E29CD391F9471A46BAFF290AEF..Cookie: JS
  ESSIONID=AD1C10E29CD391F9471A46BAFF290AEF....
##
T 192.168.100.1:8080 -> 192.168.100.5:1565 [A]
  HTTP/1.1 200 OK..Server: Apache-Coyote/1.1..Content-Type: text/html;charset
  =ISO-8859-1..Content-Language: en..Content-Length: 7048..Date: Sat, 02 May
  2009 00:33:54 GMT...........  ......<!DOCTYPE html PUBLIC "-//W3C//DTD XHT
  ML 1.0 Transitional//EN".."http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitio
  nal.dtd">.......<html xmlns="http://www.w3.org/1999/xhtml">..<head>..<meta
   http-equiv="cache-control" content
```

Figure 3.8 Typical user session gotten by using a spy or a proxy

In the Tokenization phase, the LP must identify the sequences of characters in the raw log file in order to construct the new readable log file. The process starts with the tokenization of the stream of characters, a token is a character, number, word, punctuation mark or sequence of characters treated as a single unit. We have considered the following.

1. Space, tab and new line characters are considered as delimiters.
2. The hash char followed by a new line char represent the start of a new HTTP Request.
3. The semicolon, colon, plus, equals chars are always delimiters.
4. The hyphen char followed by the major sign char represents the direction of the HTTP Request thus is considered a delimiter.
5. The hyphen joining words or numbers form a single token. (eg. ISO-8859-1,utf-8)
6. Web URL can have numeric and alphanumeric chars.
7. IP addresses are considered as tokens.
8. IP addresses followed by colon and a port number are considered as tokens.
9. A number could be an integer or a real and could have embedded numeric chars, the dot and the comma characters.
10. The tokenizer has been customized to identify the following keyword tokens: Host, User-Agent, GET, POST, Accept, Accept-Language, Accept-Encoding, Accept-Charset, Keep-Alive, Connection, Referer, Cookie, Content-Type, Content-Length, Connection and JSESSIONID.

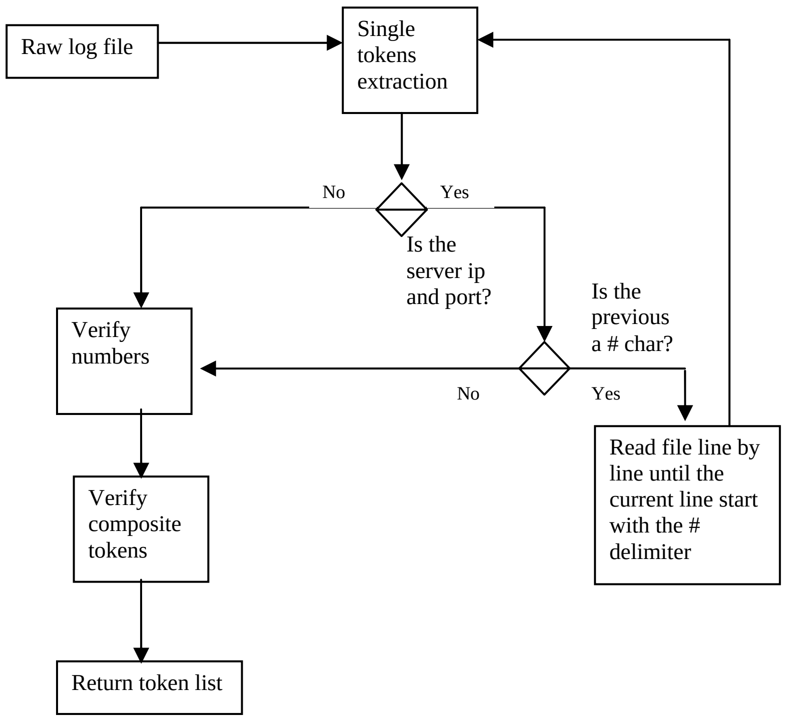

Raw log file
Single tokens extraction
No
Yes
Is the server ip and port?
Is the previous a # char?
Verify numbers
No
Yes
Read file line by line until the current line start with the # delimiter
Verify composite tokens
Return token list

Figure 3.9 The Tokenization proceess

As the raw log data contains both the HTTP requests from the user as well as the HTTP responses from the server, the tokenization process is optimized by filtering all the HTTP responses tokens. This is accomplished by comparing each new token with the value IP:PORT where IP is the IP address of the server and PORT is the port at which the server is providing the service. If the token is the same then the previous token is verified. If the previous token is the hash (#) token, then the tokenization process read the file line by line until the current line starts with a hash (#) character token followed by a newline char, then the tokenization process starts again. Figure 3.9 summarizes the tokenization process.

In the reduction phase, the original log file will be reduced and simplified, starting using the token list generated by the tokenization phase. The new log file will be constructed according to the grammar shown in the Figure 3.10 and will contains one HTTP requests per line.

```
LogFile -> (HTTP_Request)*

HTTP Request → IP:port..[Request]..[Host]..[User-
Agent]..[Accept]..[Accept-Language]..[Accept-Encoding]..[Accept-
Charset]..[Keep-
Alive]..[Connection]..[Referer]..[Cookie]..[JSESSIONID]

Request → [Method] [url] [httpVer]| [Method] [url][params] [httpVer]

Host → host : [url]:port | host : IP:port

User-Agent →  (*)

Accept → (*)

Accept-Language → (*)

Accept-Encoding → (*)

Accept-Charset → (*)

Keep-Alive → (*)

Connection → (*)

Referer → (*)

Cookie → (*)

JSESSIONID → JSESSIONID = [alfanumeric]

(*) According to the RFC2616 specifications [33]
```

Figure 3.10 Grammar specification for the new log file

The grammar for User-Agent, Accept, Accept-Language, Accept-Encoding, Accept-Charset, Keep-Alive, Connection, Referer, Cookie are specified in [33].

3.4 Artificial Session Trace Generator module

As explained in the Related Work chapter, many techniques have been proposed for Web Application test generation based on real session data.

The Artificial Session Trace Generator (ASTG) module is in charge of producing artificial logs that mimic realistic behaviors captured by real logs; and it can incorporate any algorithm for creating artificial log data; currently JTracer implements one algorithm for artificial trace generation.

3.4.1 Statistical based user data creation algorithm

The main drawback of User session testing based techniques is the lack of scalability, redundancy and production impact. D. Menasce et al. [24], J.D. Meier et al. [22] highlight the importance of the real user session data for load and stress testing. JTracer includes one algorithm for user artificial trace session generation based on statistical user behavior learned from real session traces.

The algorithm is similar to the one by Sant et al. [31] but does not rely on the same data model. We consider as a request the combination of URL and parameters. Moreover the data associated to such combination doesn’t intersect with the data associated to other URL parameters combinations even when both URL’s could be the same. Figure 3.11 shows an overall view of the algorithm.

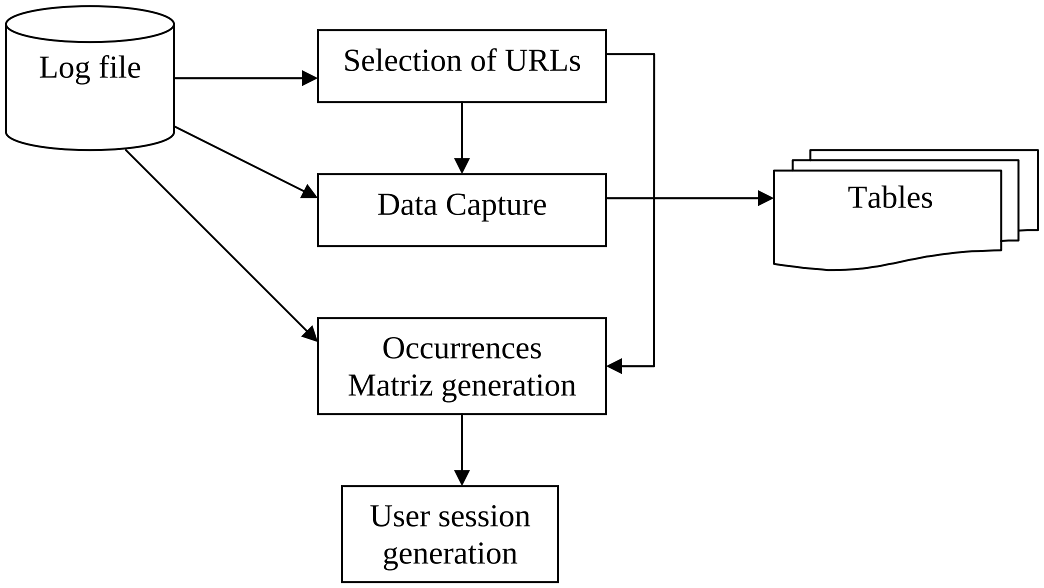


Figure 3.11 Statistical based user data creation algorithm

Our algorithm analyzes a real world session trace by extracting the behavior of the users accessing the Web application. This is accomplished by creating a square matrix where rows and columns represent the URL requests received by the Web application and cells in the matrix count the times that the request on column “j” is followed by the request in row “i”. The algorithm is divided into four phases. The first phase summarizes the requests composed by each unique URL and parameter combination. In the second phase, the data from the parameters captured in the first phase is inserted into a database. The third phase constructs the occurrences matrix representing the user behavior and finally the fourth phase generates the virtual user session trace based on the occurrences matrix.

3.4.1.1 Selection of the set of URLs

Starting with an empty set, the algorithm traverses the log file and, using the same tokenization process specified in section 2.1, gets from each HTTP Request the tokens corresponding to the URL and the request parameter names and inserts those values into the set. If the combination URL/parameters already exists then it is not inserted again. In this phase each combination URL/parameters is assigned a unique key and inserted into a list where each key corresponds uniquely to one URL/parameters combination. Hence, the cardinality of the set is always less than the number of HTTP requests in the log file. Figure 3.12 shows the pseudo-code for the selection algorithm.

```
SET = NULL;
While not END_OF_FILE
        URL = getURL(Current Line )
        ParamsNames = getParamsNames( CurrentLine )
        If URL+ParamsNames no exists in SET
                AddToSET(URL+Params)
```

Figure 3.12 Selection process

3.4.1.2 Data capture

We can avoid supplying new arguments for the session parameters by using the same argument data used in the original request if and only if this data is used in another new random session. In this phase, the algorithm creates a table for each request of the set containing a unique URL/parameter combination. This table fields will be the parameter names and their values. Once all the tables have been created, the algorithm traverses the log file again and extracts the values corresponding to each parameter of the URL/parameter combination. Then it inserts those values into the table entry with name equal to the key corresponding to the URL/parameter combination. Figure 3.13 shows the Data Capture process

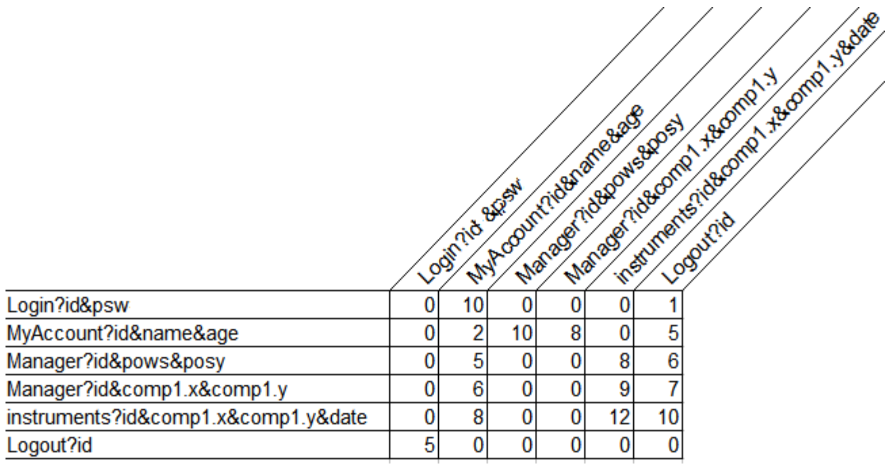
```
For each item in SET
    Create table with name URL+ParamsName
    For each item into ParamsName
        Create field Varchar with length (MAX (field Value in the log file))

While not END_OF_FILE
    ValueList = getParameterValues( CurrentLine )
    Insert into URL+ParamsName values (ValueList)
```

Figure 3.13 Data capture algorithm

3.4.1.3 Occurrences Matrix construction

The Occurrences Matrix is a square matrix with URL/parameter combination in the rows and in the columns. The cells of the matrix are integer values representing the times that the URL/parameter in the row is followed by the URL/parameter in the column. We chose the URL/parameter combination as the indexes of the Occurrences matrix because the URL alone can’t represent accurately the user request.



	Login?id &psw	MyAccount?id&name&age	Manager?id&pows&posy	Manager?id&comp1.x&comp1.y	instruments?id&comp1.x&comp1.y&date	Logout?id
Login?id&psw	0	10	0	0	0	1
MyAccount?id&name&age	0	2	10	8	0	5
Manager?id&pows&posy	0	5	0	0	8	6
Manager?id&comp1.x&comp1.y	0	6	0	0	9	7
instruments?id&comp1.x&comp1.y&date	0	8	0	0	12	10
Logout?id	5	0	0	0	0	0

Figure 3.14 Occurrences matrix

For example, Figure 3.14 shows an Occurrences matrix with six URL/parameter combinations as indexes. Cells with value cero means that the request of the row is never followed by the request of the column. This is clearly shown in the first column (Login?id&psw) where almost all the rows are cero and this clearly reflects the system flow logic where no request is prior to the login of the application. In the other hand, the cells of the last column (Logout?id) almost all have a value greater than cero and this because is more likely to log out of the Web application after any request.

More formally the Occurrences matrix is defined as follows:

O_{ij} = {count | request "i" is followed by request "j" }

Where

O = Occurrences Matrix

i, j = URL/parameter combination

The algorithm traverses the log file and for each HTTP Request inspects the URL/parameter combination of the previous HTTP Request, using the key of both current and previous HTTP Request URL/parameters combination seeks the correspondent cell into the Occurrences Matrix, then increments the current count. To construct the Occurrences Matrix the algorithm needs to traverse the log file only once. Figure 3.15 shows the algorithm.

```
currentLine = Get first line;
While (not END_OF_FILE)
        i = index of URL+ Parameters in currentLine
        j = index of URL+ Parameters in previousLine
        Oij = Oij + 1;
        previousLine = currentLine
        currentLine = Get next line
```

Figure 3.15 Algorithm for updating the Occurrences Matrix

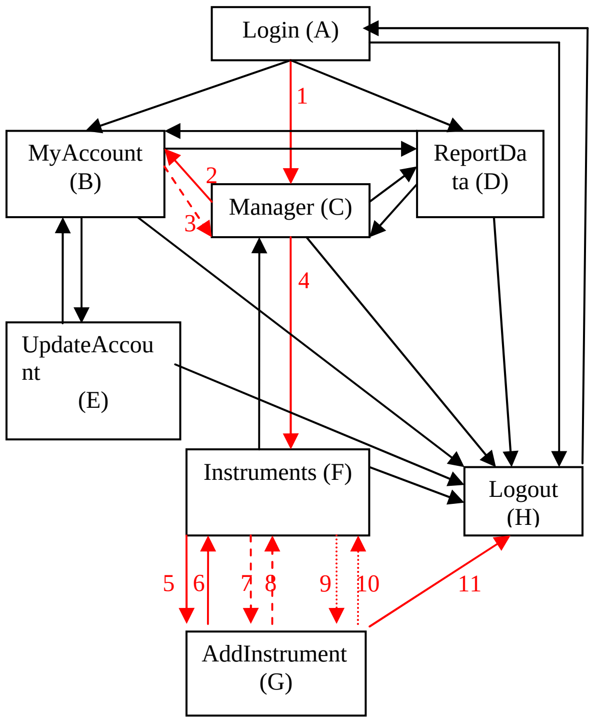
3.4.1.4 User session generation

Finally in the fourth phase, user sessions are generated relying on the Occurrences Matrix; the process of user session generation is explained in the Figure 3.16.

```
currentKey = Choose randomly an HTTP Request;
While (not END_SESSION)
        create request ( currenKey );
        currentKey = next HTTP Request($O_{ij}$);
```

Figure 3.16 User session generation algorithm

The algorithm shown in Figure 3.16 first chose randomly an HTTP Request from the log file. Then seeks the request in the rows of the Occurrences matrix. Finally chose one of the cells of the row corresponding to that request, the election of the cell is a random process influenced by the value of the cell. Hence the next more probable request to be generated would be the one with more weight in the row referencing the current key. The column corresponding to the cell chosen will be the next request to be generated. The process is repeated until a request previously defined as the end of a session is found.



Steps		A	B	C	D	E	F	G	H
1	A	0	2	10	5	0	0	0	1
3	B	0	1	5	6	10	0	0	4
2,4	C	0	5	1	7	0	10	0	8
	D	0	5	9	10	0	0	0	9
	E	0	10	0	0	0	0	0	20
5,7,9	F	0	0	10	0	0	5	20	10
6,8,10,11	G	0	0	0	0	2	30	2	10
	H	40	0	0	0	0	0	0	0

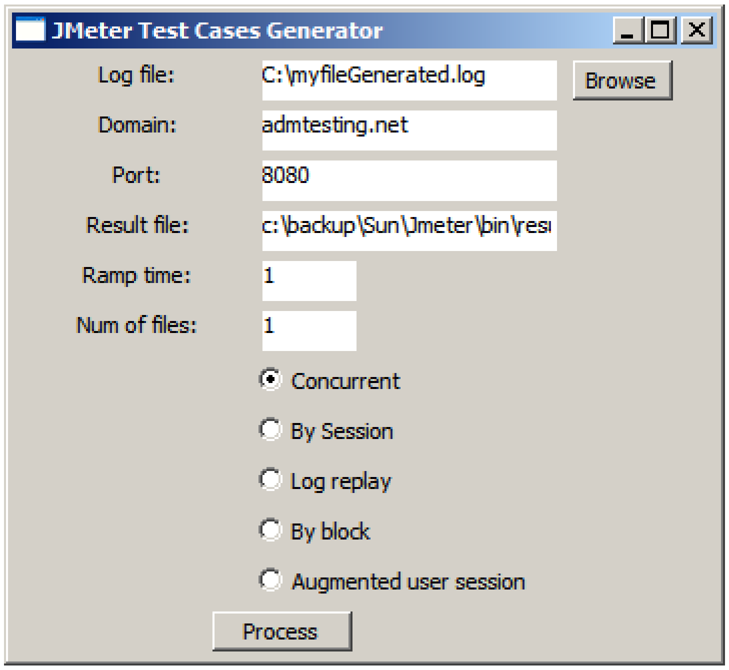
For each step a new random column is chosen. The cells with greater value are more likely to be chosen.

Figure 3.17 Navigation path created using the Occurrences Matrix

Figure 3.17 shows the navigation path chosen for creating a new user session relying in the Occurrences Matrix. Only the paths present in the log file will appear in the graph. A cero value in a cell of the Occurrences Matrix means that the URL/Parameter in the row is never followed by the URL/Parameter in the column. Therefore, any step must choose such path; this is done by excluding the cells with value cero of the process.

3.5 Test Cases Generator module

The Test Cases Generator (TCG) takes as input the Test Log Data file and a set of parameters and generates several load testing suites. This is accomplished by translating the log file data to a XML file representing a test plan for the JMeter [18] load testing tool. Variations on a few parameters, the way the sessions are grouped and the execution order of each session are used to create several different load testing suites. Each suite could be used to test a specific aspect of the Web Application according to the needs of the testing engineer. Figure 3.18 shows the parameters required to setup to create a test case.


JMeter Test Cases Generator

Log file:	C:\myfileGenerated.log
Domain:	admtesting.net
Port:	8080
Result file:	c:\backup\Sun\Jmeter\bin\resı
Ramp time:	1
Num of files:	1

[✓ Concurrent] ○ By Session ○ Log replay ○ By block ○ Augmented user session

Figure 3.18 Form used by JTracer to setup the parameters for test cases generation

The first parameter, Log file, is the path name of the log file (raw or processed) to be translated to a .jmx file. The Domain parameter and the Port parameter are the URL relative for all the HTTP Requests into the log file. The jmx file is stored in the Result file parameter. Ramp time indicates how long it will take to define all the threads or concurrent sessions if any. The Num files parameter is useful when distributed testing is needed due to capacity of the client or tester. Finally four types of tests can be generated using the same input log file. The following load tests generation algorithms have been implemented:

- Concurrent. This algorithm simulates multiple clients accessing the Web Application at the same time, each client executes one session. This test is useful to measure the maximum number of simultaneous connections that the server can support.

- By Session. This scenario simulates a set of sequential clients accessing the Web application one after another. This test is useful to verify the response time of the server for each type of session and to verify that the sessions created manually or artificially are working as expected.

- Log Replay. This scenario replays the request in the log file in the same order than it appear. The test is useful to recreate errors as happen or to replay an old scenario or historic scenario.

- By Blocks. Divides the tests by blocks of time each block contains many sessions inside. Then all the blocks can be sent concurrently. This allows testing the Web server for longer periods of time. A test with only one session by block is the same than the Concurrent scenario.

- Augmented User Session. Because partition the sessions by blocks cut some sessions, this scenario guarantees that every session in the log file is completed at least one time. For further information see [34].

3.5.1 Translating the log file to JMX file

To test a Web Application using JMeter must create a test plan. A test plan is a descriptor file that provides data and execution parameters which instruct JMeter how the test has to be conducted. Starting with a set of sessions, even slight variations on the structure of the jmx files can produce significantly distinct test cases scenarios.

Mainly three JMeter components are combined to create several test cases scenarios, The Thread Group, The Simple Controller and the Constant Timer. The rest of components supply settings that don't influence the way the sessions are sent to the server. Table 3.3 summarizes all the components used by JTracer to generate JMX files. More detailed information about these components can be found in the JMeter [18] official site.

TABLE 3.3 Brief description of the JMeter load testing components

Component	Description
ThreadGroup	Controls the number of threads used to execute the test plan. Multiple threads can be used to simulate concurrent connections.
HTTP Request	Sends an HTTP request to a web server
Simple Controller	Organizes and groups the samples and controllers
Constant Timer	Causes a delay of a constant amount of time before a request is sent to the Web server
HTTP Request Defaults	Specifies the default settings for all the HTTP Request belonging to the same group.
HTTP Cookie Manager	Ensure that each thread or simple controller gets its own cookies but shared across all the HTTP Request components in the session.

The process of creating a JMX plan test is divided into three phases. In the first phase the HTTP requests with the same JSESSIONID's are grouped as belonging to the same session. Additionally the HTTP requests without JSESSIONID but that are in a predetermined time are added to the nearest JSESSIONID group. For each session one text file containing all its HTTP Requests is created and named by its JSESSIONID. In the second phase, according to the type of test case scenario, the files generated in the first phase are organized into directories. In the third phase according to the test case scenario a Thread Group component is created by file or a Simple Controller is created by file; the HTTP request into each session file are inserted as HTTP Request component into a Thread Group or a Simple Controller, again, depending of the type of test case scenario. We now explain the process of creating each test case scenario:

- **Concurrent Test Case Scenario.** Each session file generated in the first phase is represented with a Thread Group Component. A HTTP Cookie Manager is attached to the Thread Group Component to be shared by all the HTTP Requests. The HTTP requests corresponding to the session files are added one by one to their Thread Group.

Figure 3.19 explains the creation process. Each box in the left represents a session. Inside each box are the HTTP Requests. The session is represented in JMeter as a Thread Group component and their request by HTTP Request component.

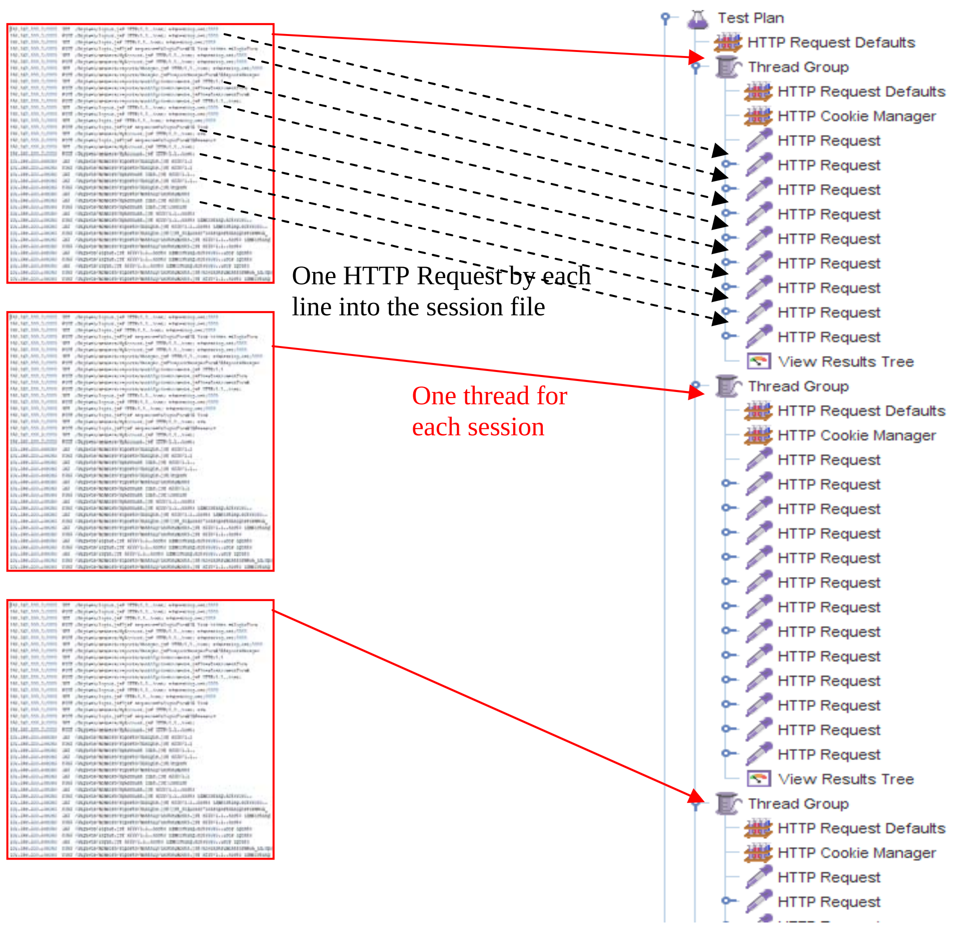


Figure 3.19 Concurrent Test Case Scenario translated to JMeter test plan

- **By Session Test Case Scenario**. Each session file generated in the first phase is represented by a Simple Controller component, only one Thread Group is used and it will group all the Simple Controllers created for each session file. An HTTP Cookie Manager is attached to each Simple Controller. The HTTP Requests in all the session files are added one by one to each Simple Controller.

Figure 3.20 explains the creation process. Each box in the left represents a session. Inside each box are the HTTP Requests. The session is represented in JMeter as a Simple Controller component and their request by HTTP Request component. As this scenario represents a sequential execution, only one thread will be necessary. A Thread Group component contains all the sessions.

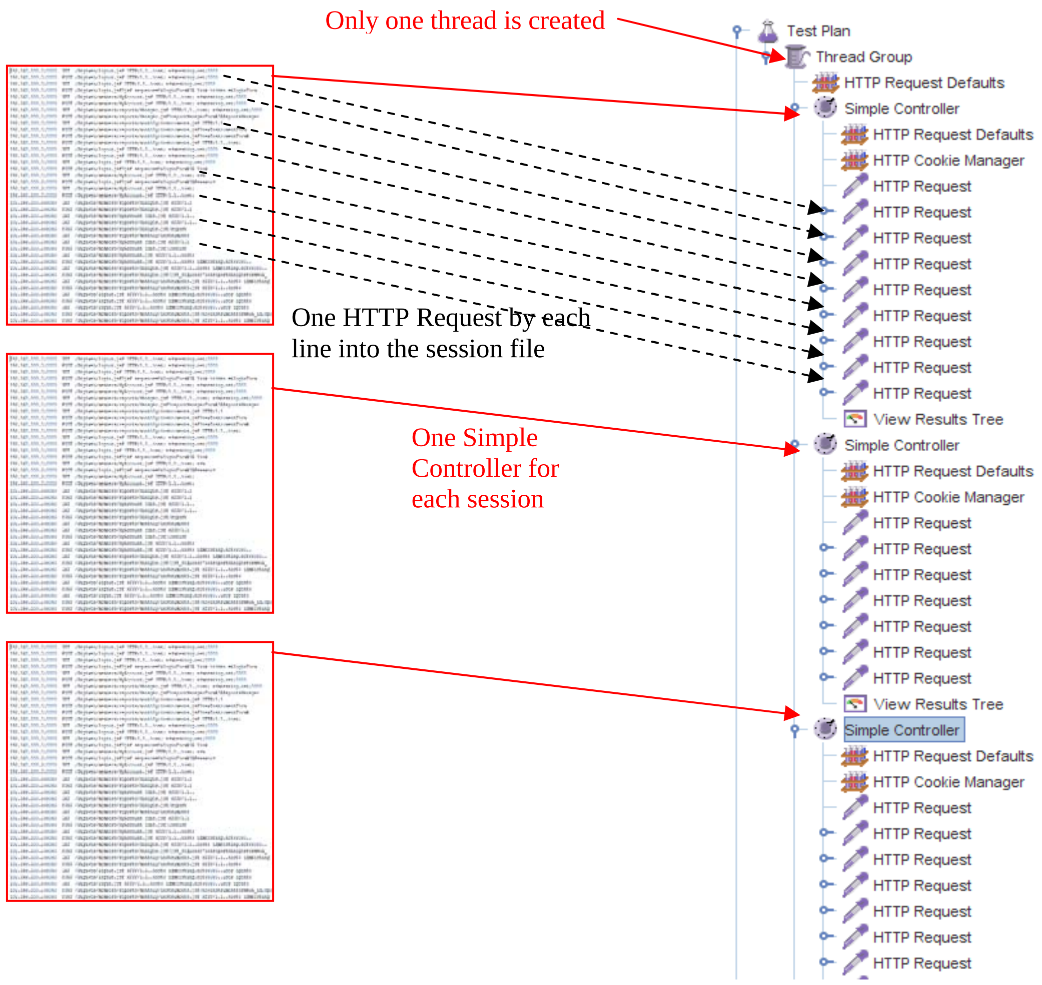


Figure 3.20 By Session Test Case Scenario translated to JMeter test plan

- **Log Replay Test Case Scenario**. For test execution of a Log Replay, the threads and HTTP Requests are created in the same way as the Concurrent Test Case Scenario. In addition to those components a Timer component is attached to each HTTP Request. This Timer will induce delay of a few milliseconds before sending the HTTP Request to the server. The requests are synchronized so as to force them to be sent in the same order as they appeared in the log file. The delay time attached to each HTTP Request is calculated as follows:

  Ti0 = Ti0 – Tfirst_HTTP , i >= 0

  Tij = Tij – Ti0 , i > 1, j > 0

  Where

  i = number of session

  j = number of HTTP Requests in the session “i”

  Tfirst_HTTP = Time of the first request of the first session logged.

  Tij = Time of the HTTP Request “j” in the session “i”

- **By Blocks Test Case Scenario**. When a larger log file is divided into blocks each block contains many sessions inside. To create a By Blocks Test Case Scenario each block is represented as a Thread Group and each session inside those blocks is represented by a Simple Controller. The Cookie Manager is attached to each Simple Controller. In this way several blocks can run concurrently.

Figure 3.21 explains the creation process. Each box in the left represents a session. Inside each box are the HTTP Requests. The session is represented in JMeter as a Simple Controller component and their request by HTTP Request component. For each block of time a Thread Group is created. The requests inside the session that are into the block interval time are attached as part of the block as a new session, no matter that this new session results incomplete.

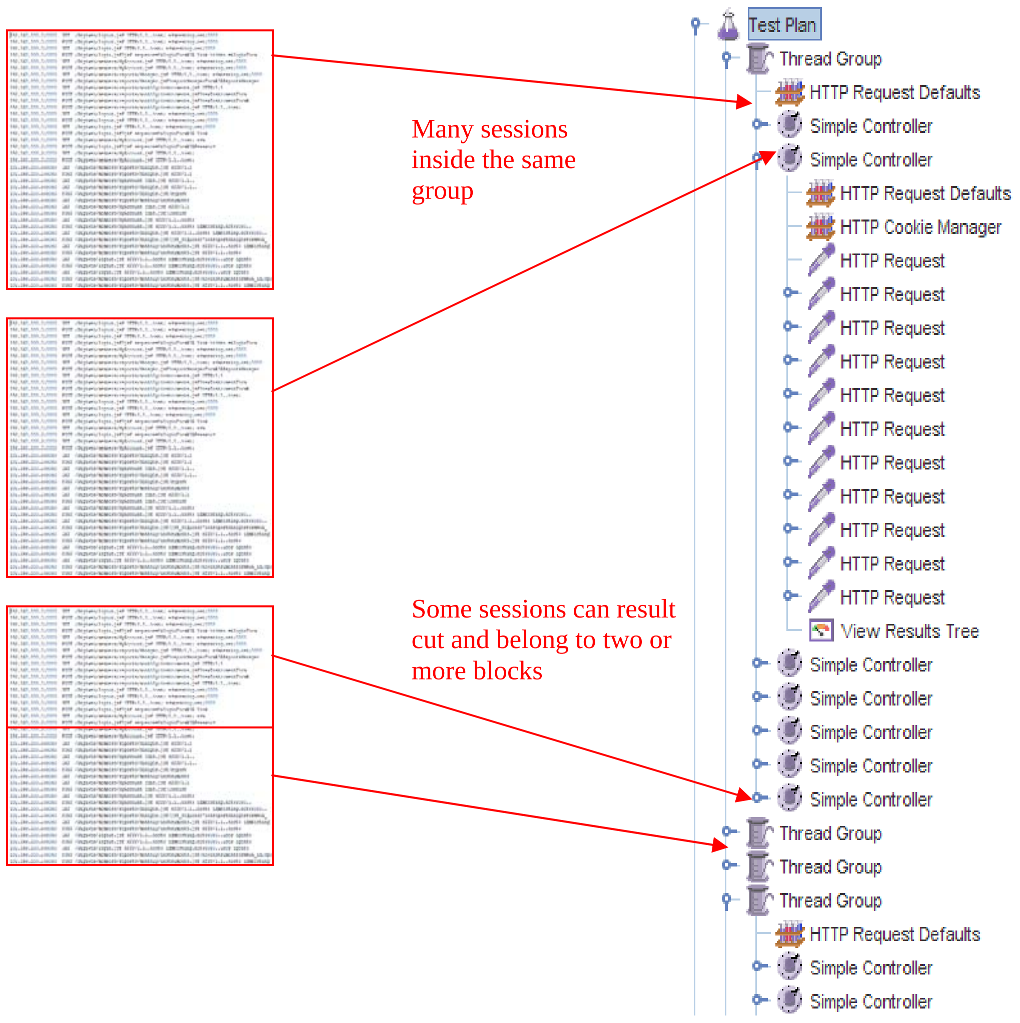


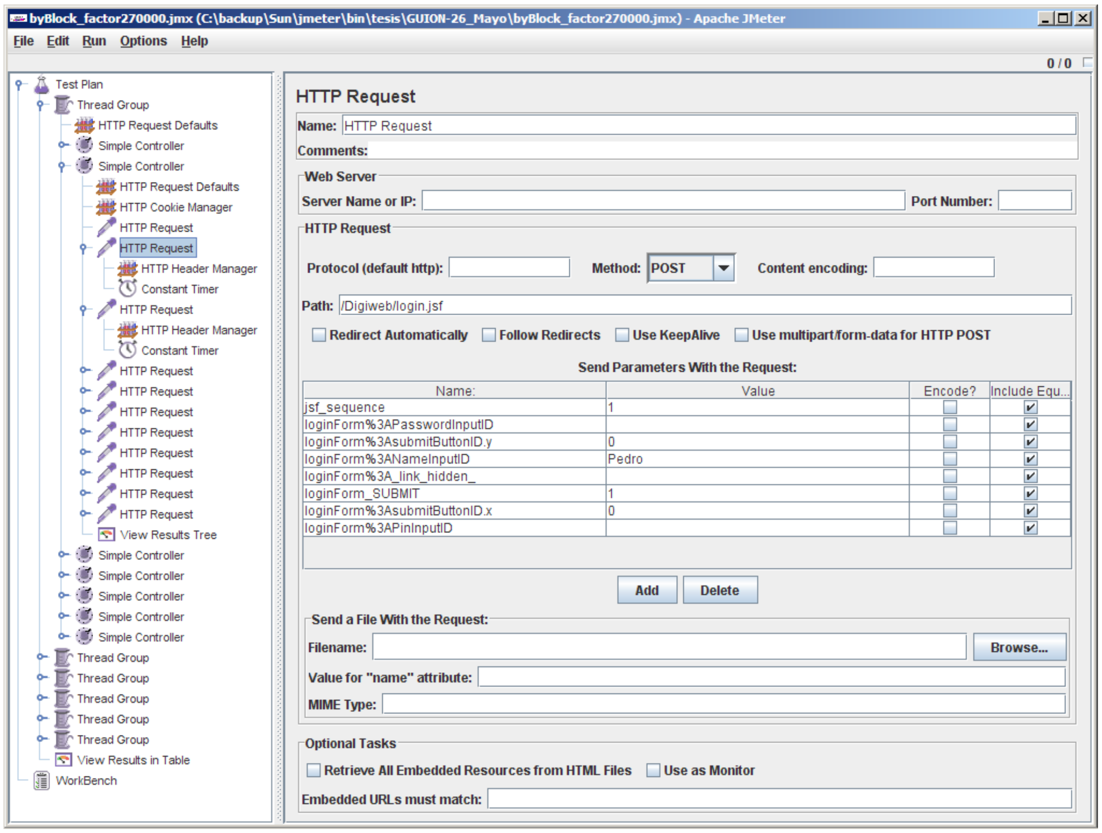
Figure 3.21 By Block Test Case Scenario translated to JMeter test plan

- **Augmented User Session Test Case Scenario**. For each session a block is created. Each block is represented by a Thread Group. The HTTP Requests with system time between the first and the last system time of the session in progress are added to the group. Each of those HTTP Requests are simultaneously grouped in the session that they belong to. Two or more sessions whose time intervals intersect its time ten this scenario is equivalent to the Concurrent Test Case scenario.

3.6 Test Engine

The Test Engine (TE) module is in charge of carrying out the test plan by sending requests and receiving responses from the Web Application server. This module also interprets the files generated by the TCG module to perform different types of load and stress testing. The current version of JTracer uses JMeter as its TE module. We chose this tool because it is an Open Source solution broadly used and with a big community that maintains it up to date. Further information can be found in the official Web site of the Jakarta JMeter project [18].

Figure 3.22 shows a snapshot of a JMeter testing tool window showing a test plan and the elements that work together for creating a load testing to the Web server. The elements are organized hierarchically being the Test Plan the top element followed by the Thread Group, the Simple Controller and the HTTP Request elements. The window at the left side shows the components and in the right side the parameters that setup such component. Currently the HTTP Request component is selected, the parameters that are send in the request are shown in the right, also the Method of the request, the URL, the protocol among others parameters.


byBlock_factor270000.jmx (C:\backup\Sun\jmeter\bin\tesis\GUION-26_Mayo\byBlock_factor270000.jmx) - Apache JMeter
File Edit Run Options Help
0 / 0

```
Test Plan
  Thread Group
    HTTP Request Defaults
    Simple Controller
    Simple Controller
      HTTP Request Defaults
      HTTP Cookie Manager
      HTTP Request
      HTTP Request
        HTTP Header Manager
        Constant Timer
      HTTP Request
        HTTP Header Manager
        Constant Timer
      HTTP Request
      HTTP Request
      HTTP Request
      HTTP Request
      HTTP Request
      HTTP Request
      HTTP Request
      HTTP Request
      View Results Tree
    Simple Controller
    Simple Controller
    Simple Controller
    Simple Controller
    Simple Controller
  Thread Group
  Thread Group
  Thread Group
  Thread Group
  Thread Group
  View Results in Table
WorkBench
```

HTTP Request

Name: HTTP Request

Comments:

Web Server

Server Name or IP: Port Number:

HTTP Request

Protocol (default http): Method: POST Content encoding:

Path: /Digiweb/login.jsf

☐ Redirect Automatically ☐ Follow Redirects ☐ Use KeepAlive ☐ Use multipart/form-data for HTTP POST

Send Parameters With the Request:

Name:	Value	Encode?	Include Equ...
jsf_sequence	1	☐	✓
loginForm%3APasswordInputID		☐	✓
loginForm%3AsubmitButtonID.y	0	☐	✓
loginForm%3ANameInputID	Pedro	☐	✓
loginForm%3A_link_hidden_		☐	✓
loginForm_SUBMIT	1	☐	✓
loginForm%3AsubmitButtonID.x	0	☐	✓
loginForm%3APinInputID		☐	✓

Send a File With the Request:

Filename: Browse...

Value for "name" attribute:

MIME Type:

Optional Tasks

☐ Retrieve All Embedded Resources from HTML Files ☐ Use as Monitor

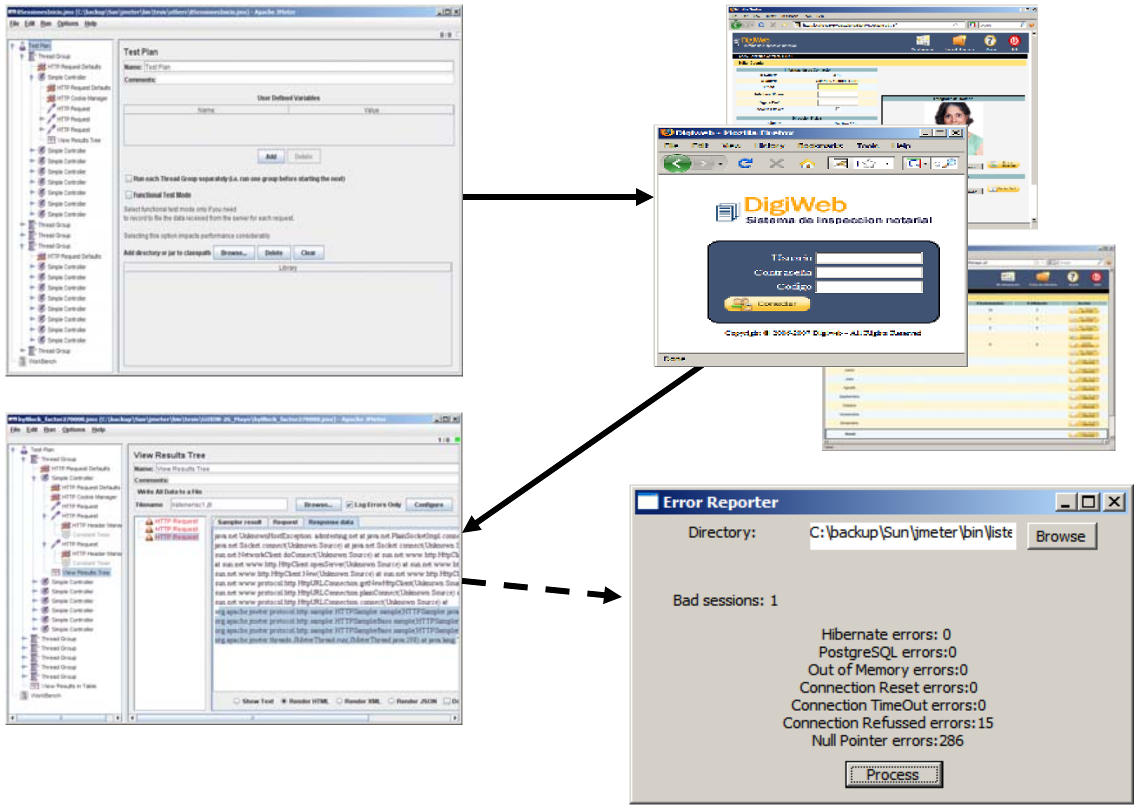
Embedded URLs must match:

Figure 3.22 JMeter testing tool interface

3.7 Error Reporting

The Error Reporting (ER) module is responsible for summarizing the errors that the TE module identified from the Web server responses. Before reporting the errors the first step consists of creating a large load for the server and classifying the errors detected by type. Then for the following tests the responses returned by the server are analyzed and summarized according a predetermined classification.

Figure 3.23 shows a test plan and their corresponding Listener component. A Listener is an element that captures the Web application responses. In the right side the error responses are shown, the text responses are analyzed and summarized by the Error Reporting module and the number of errors by error type is shown.



Directory: C:\backup\Sun\jmeter\bin\liste

Bad sessions: 1

Hibernate errors: 0
PostgreSQL errors:0
Out of Memory errors:0
Connection Reset errors:0
Connection TimeOut errors:0
Connection Refussed errors:15
Null Pointer errors:286

Figure 3.23 JMeter testing tool and Error Reporting

Table 3.4 summarizes the errors captured by the Error Reporting module.

TABLE 3.4 Errors reported by the Error Reporting Module

Component	Description
Hibernate	Reports that other request is blocking the current database record
PostgreSQL	Reports that the maximum number of concurrent connections have been exceeded
Out of Memory	Reports that the Web application system is out of memory
Connection Reset	Reports that the Web server is rejecting the connection
Connection Time Out	Reports that the Web server can respond to the request in a reasonable interval of time
Connection Refused	Reports that the Web server is not longer providing the service requested
Null Pointer Error	Reports that some component has not still been created and is been referenced as it would exist, for example a dynamic button that is generated only when the user has accomplished some process

4 EXPERIMENTAL ANALYSIS

4.1 Introduction

This chapter describes and presents the results of experiments that were carried out in order to validate our proposals. The specific objectives of the experiments were to test:

1. If the Code Injector module correctly instruments a Web application to log the HTTP Requests of the user interaction as the user navigates the Web application. Even when the Web application requests are encrypted.

2. If the algorithms implemented to generate the test cases scenarios exercise the Web server according to the characteristics of each test case scenario.

3. If the Statistically based user data creation algorithm presented in section 3.4.1 generates sessions that offer substantial improvements over replicating many times a single set of sessions.

4.1.1 Computing infrastructure

The experiments were carry out using “Digiweb”[9] a Web Application system developed by the ADMG Group of the University of Puerto Rico Mayaguez, and four commodity hardware systems.

- A Dell Precision PWS 380, Intel Pentium D 2.0 GHz, 1 GB of RAM, Nvidia Quadro 540 with 128MB with Linux Ubuntu operating system, web application server Apache Tomcat 5.5.1 and running PostgreSQL 8.2 as database system. This system was used as the server for the Web application.

- Three computer systems with Windows operating system, as the clients of the Web application. Each client was equiped with the JMeter [18] testing tool.

4.2 User navigation sequence

The objective of this experiment is to verify that the log file created after the code injection process reflects the exact sequence of actions performed by users when they accessed the Web application. Two main aspects deserve validation:

- Verify if the sequence of user actions is logged in the same order than the real navigation sequence produced by the interaction between the Web server and the users.

- Verify if the data logged produce the same Web application state than the produced by the original user navigation.

The experiments methodology was as follows:

1. The Web application is undeployed and a backup of the database is taken.
2. Source code is inserted by the Code Injector into the original Web application.
3. The new instrumented Web application is deployed.
4. A sniffer is run to catch all the traffic between the user and the Web server.
5. A set of pre-scripted sessions were executed in the new Web application. At the same time, the log file is generated by the Web application.
6. Again a backup of the resulting database is taken.
7. The log file produced by the sniffer in step 4 is compared with the log produced by the Web application in step 5.
8. The Web application is restored to its original state in step 1.
9. The Web application is exercised with the request of the log file produced in step 5.
10. The database resulting is compared with the database got in step 6.

Table 4.1 shows as example the differences between the log produced by the sniffer and the log produced by the instrumented Web application.

TABLE 4.1 Example of logged data by the sniffer and by the Web application

	Requests caught using the sniffer	**Requests produced by the code injection**
1	GET /Digiweb/	GET /Digiweb/
2	GET /Digiweb/images/logodigiweb.gif	
3	GET /Digiweb/images/newIcons/conectar.png	
4	POST /Digiweb/login.jsf	POST /Digiweb/login.jsf
5	GET /Digiweb/members/MyAccount.jsf	GET /Digiweb/members/MyAccount.jsf
6	GET /Digiweb/members/Sandbox___Download.jsf	
7	GET /Digiweb/members/Sandbox___Download.jsf	
8	POST /Digiweb/members/MyAccount.jsf	POST /Digiweb/members/MyAccount.jsf
9	GET /Digiweb/members/MyAccount-edit.jsf	GET /Digiweb/members/MyAccount-edit.jsf
10	GET /Digiweb/members/Sandbox___Download.jsf	
11	GET /Digiweb/members/Sandbox___Download.jsf	
12	POST /Digiweb/members/MyAccount-edit.jsf	POST /Digiweb/members/MyAccount-edit.jsf
13	GET /Digiweb/members/MyAccount.jsf	GET /Digiweb/members/MyAccount.jsf
14	GET /Digiweb/members/Sandbox___Download.jsf	
15	GET /Digiweb/members/Sandbox___Download.jsf	
16	POST /Digiweb/members/MyAccount.jsf	POST /Digiweb/members/MyAccount.jsf
17	GET /Digiweb/logout.jsf	GET /Digiweb/logout.jsf

Although both logs look different, the user interactions with the system are quite similar. The small differences arise because the web browser requests additional visual elements as a button images, icons, etc. As can be noticed, the requests that the Web application can not log are those that are not produced directly by explicit user action such as a mouse click on a button or hyperlink.

However, we have to verify that this fact doesn't influence the Web application final state. That is, starting with the same Web application state, the replay of the log produced by the Web application have to produce the same final Web Application state than that of the original trace. To ensure this, an algorithm that compares the database produced after the load testing of the log captured by the Web application was developed, the scenario is showed in Figure 4.1. After comparing both databases, no differences were found therefore the results of this test were successful.

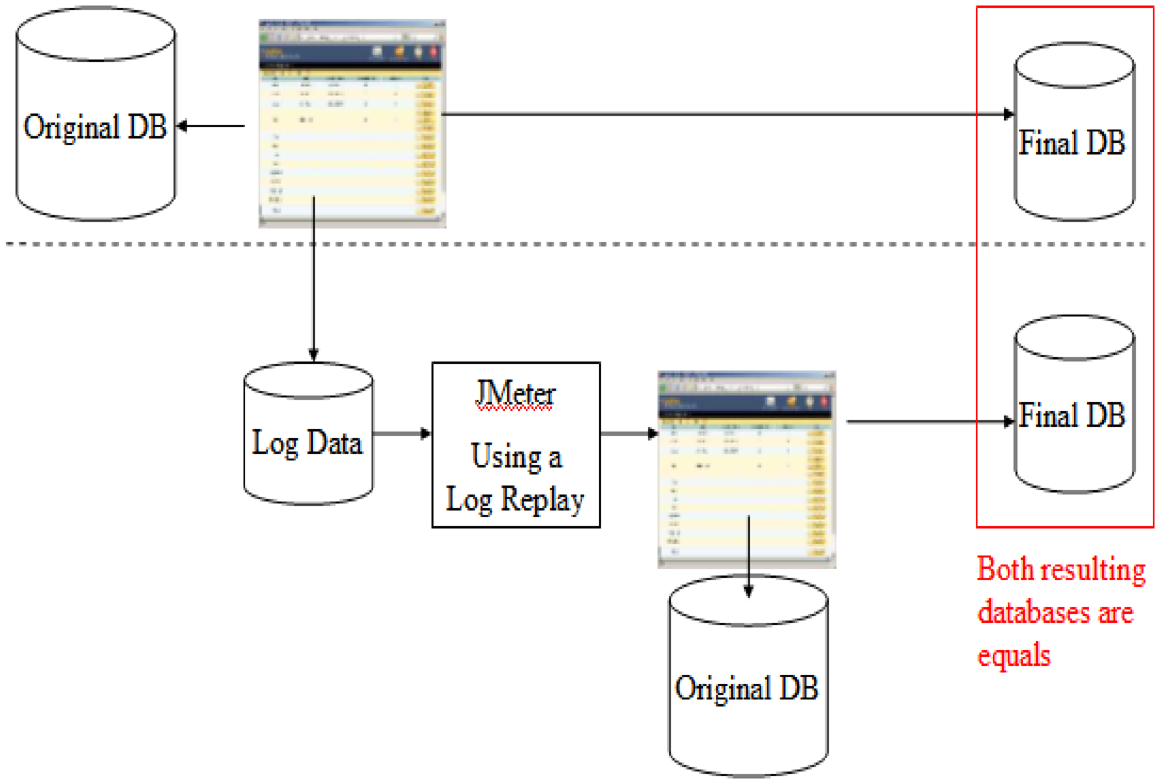


Figure 4.1 Experimental Set-up to validate the Code Injector functionality

Both experiments show that the log data produced by the Web application follow the user navigation flow. Furthermore the final Web application state produced by the log data captured is the same than the produced originally. It demonstrates that the sequence of navigation of the user is logged correctly.

4.3 Analysis of the test algorithms

A series of experiments were carried out for characterizing each test algorithm and to verify that the test cases scenarios generated by the tool produce the results according to the characteristic to the test suite.

4.3.1 Analysis of the Concurrent Algorithm

We expect a Concurrent Test Case to overwhelm the Web application server for a big load. In this way the number of concurrent connections that the Web server can attend at the same time can be estimated. Other uses for this scenario are for example to compare the results got from simulating a set of sessions sent many times (replicated) and a set of sessions generated using the algorithm presented in Section 3.4.1. In this way the second and third experimental objective can be validated. The methodology was as follows:

1. A backup of the current Web application state is taken to restart the original state for each test that is performed.

2. Starting with a set of "N" sessions, loads testing replicating the set two, three, six, nine and twelve times are executed. For each load testing the Web application is re-started to its original state. Additionally the same number of sessions than the

resulting number after each replication of the original set is created using the algorithm presented in the Section 3.4.1. Again for each load the Web application is re-started to its original state.

3. After each test the Web server responses are summarized by using the Error Reporting module and the results are compared between both types of session data.

According to the characteristics of the Web application under test and of the sessions chosen, the number of good and bad sessions reported by the Web application could vary. In this experiment we considered a session good if all their HTTP Requests are successfully executed, that is, the Web server responds successfully to all HTTP request. We expect the number of bad sessions to increment as the number of concurrent sessions increments. Table 4.2 shows the number of good and bad sessions according to the number of concurrent sessions sent. The experiments were carried out starting with 40 sessions, for each next experiment the 40 sessions were replicated two, three, six, nine and twelve times. At the same time, the same number of sessions were created using the algorithm presented in Section 3.4.1

TABLE 4.2 Comparative of errors generated by replicated and artificially sessions

	Replicating Sessions				Sessions Statistically Created		
Total Sessions	Num Times Replicated	Avg. Bad Session	Avg. Good Session	% Good Sessions	Avg. Bad Session	Avg. Good Session	% Good Sessions
40	1	10.75	29.25	73.13	4.5	35.5	88.75
80	2	21.25	58.75	73.44	9.5	70.5	88.13
120	3	40.25	79.75	66.46	33.75	86.25	71.88
240	6	138.25	101.75	42.40	114.25	125.75	52.40
360	9	269.75	90.25	25.07	231.00	129	35.83
480	12	407.25	72.75	15.16	344.00	136	28.33

Table 4.2 shows that there exists an optimal point where the number of good sessions start to decrease compared to the number of bad sessions. This can be noticed more clearly in Figure 4.2 and Figure 4.3.

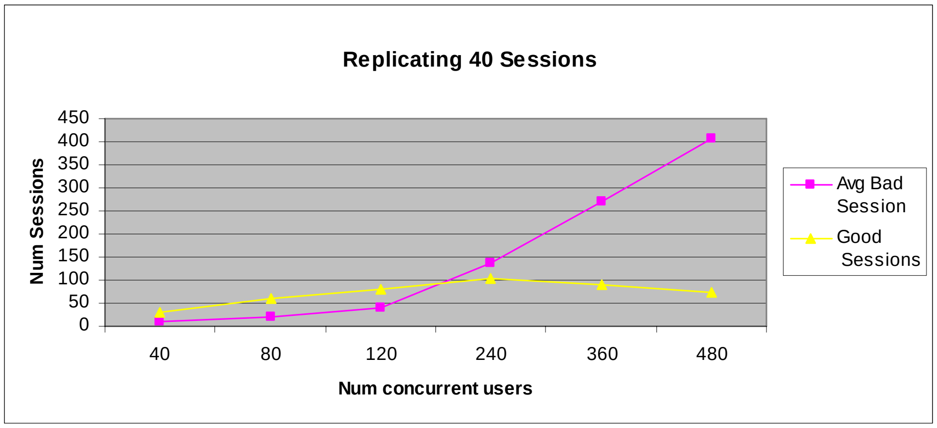


Figure 4.2 Good sessions versus bad sessions with set replicated

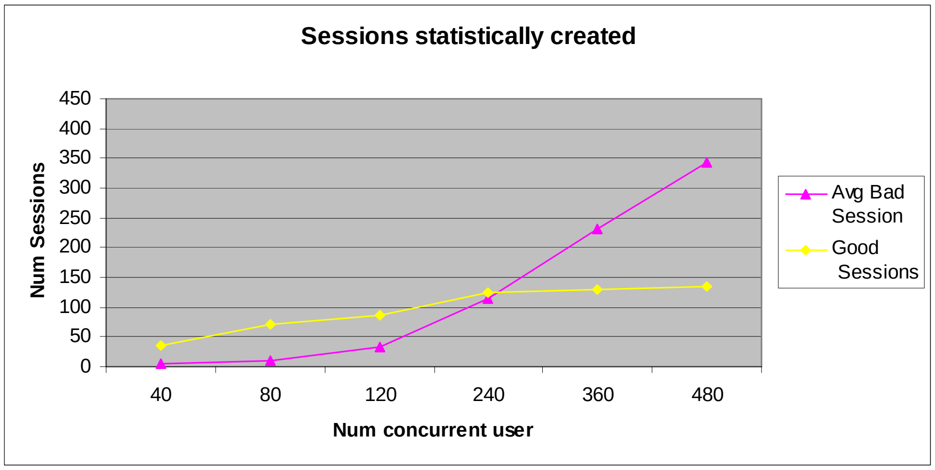


Figure 4.3 Good sessions versus bad sessions with set created artificially

The experiment allows us to collect detailed information about the errors produced during the test using the two types of sessions (replicated and statistically created). Six types of errors were found, those errors were classified using the ER module and the results are shown in Tables 4.3 and 4.4. From these tables we can see that Connection Time Out and Null Pointer errors increment faster than the others types of errors, being Connection Time Out the one that registered the bigger increment rate. Again both types of sessions discovered more types of errors in the same load testing level (240 sessions), also they registered the biggest increment in the same type of error, more than 400% for Connection Time Out in the same load testing level (360 sessions). This behavior deflects the fact that statistically created sessions have as basis the same 40 sessions that are being replicated.

TABLE 4.3 Errors detailed for sessions replicated

Total Sessions	Avg. Hibernate Error	Avg. Postgres SQL Error	Avg. Connection Reset	Avg. Connection Time Out	Avg. Connection Refused	Avg. Null Pointer
40	3.25	0	0	0	0	38.25
80	4.50	0	0	0	0	77.00
120	15.00	0	0	0	0	159.50
240	21.50	43.75	1.00	56.50	0	319.50
360	14.25	95.75	13.50	285.50	7.75	488.50
480	14.50	86.75	112.50	884.25	57.00	518.00

TABLE 4.4 Errors detailed for sessions created artificially

Total Sessions	Avg. Hibernate Error	Avg. Postgres SQL Error	Avg. Connection Reset	Avg. Connection Time Out	Avg. Connection Refused	Avg. Null Pointer
40	0	0	0	0	0	31.25
80	1.25	0	0	0	0	48.00
120	10.50	0	0	0	0	204.00
240	11.25	40.00	0	11.25	0	330.25
360	18.25	83.25	12.00	155.00	0	407.25
480	14.00	101.25	10.00	326.00	3	684.75

Figures 4.4 and 4.5 show more clearly the differences of the errors produced by using the set of sessions replicated and the set of sessions created artificially. Notice that Connection Time Out and Null Pointer exceptions are the bigger errors showed because of the test.

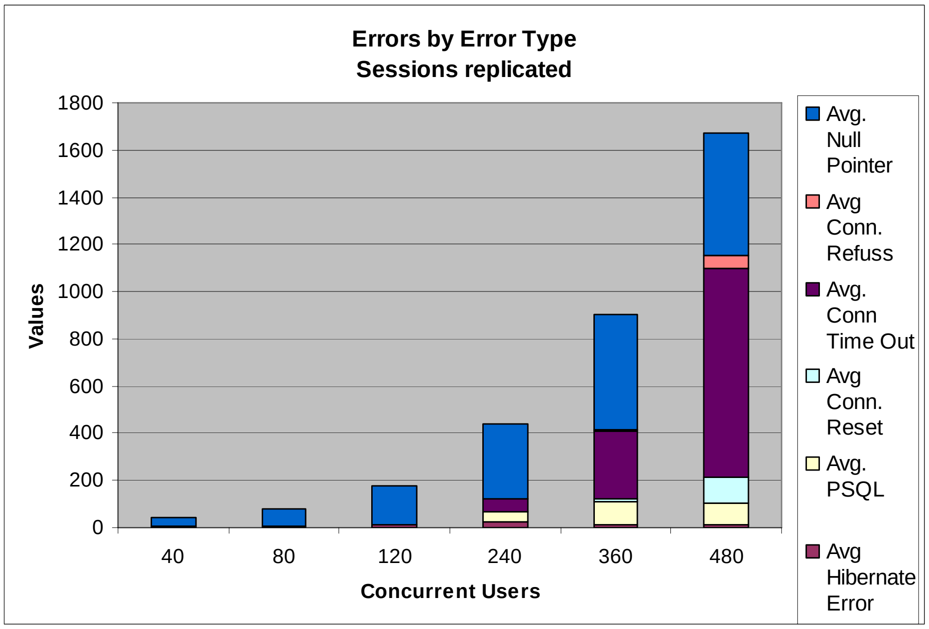


Figure 4.4 Error by error type using sessions replicated

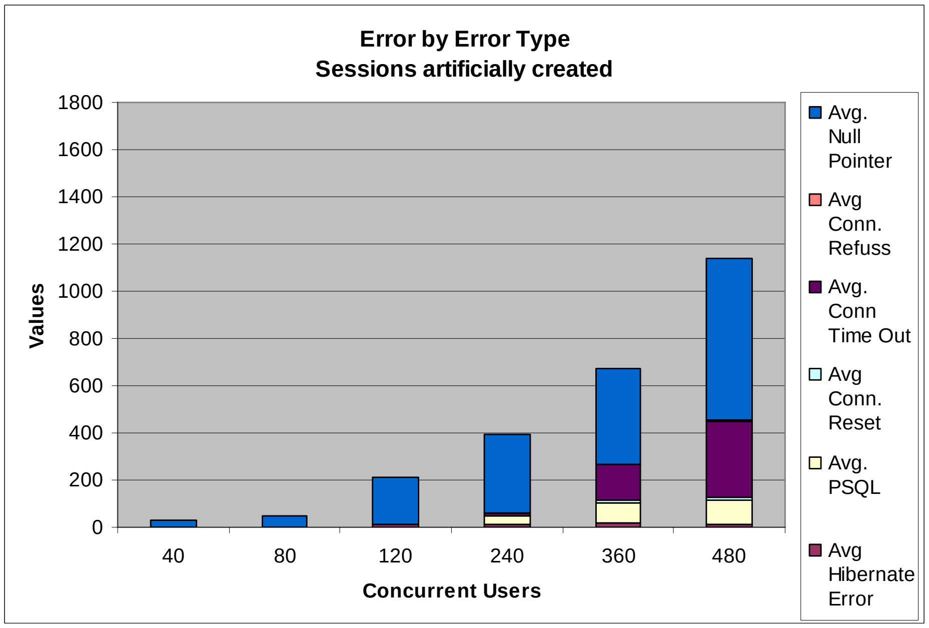


Figure 4.5 Error by error type using sessions artificially created

Despite of the sessions created statistically and the session replicated start with the same set of sessions the differences in errors reported specially for Connection Reset, Connection Time Out and Connection Refused are substantial. Therefore our results confirm that the set of sessions chosen for the test will influence substantially the final error reports. Also confirm that a bad election of the set of sessions for the testing could address to erroneous conclusions.

4.3.2 Analysis of the By Session Algorithm

The scenario produced by this algorithm allow us to verify the correctness of the sessions created, verify session interdependence and measure response time of a set of sessions. To validate this test scenario the methodology was as follows:

1. A snapshot of the Web application state is taken.
2. Many sessions are created one after another, the Web application state resulting is saved again.
3. Using the log generated by the Web application after the code injection and starting with the original Web application state the sessions were executed again.
4. The resulting Web application state is compared with the resulting Web application state from step 3.

As the sessions originally were created sequentially, the By Session Test Case Algorithm doesn't generate any error and this is validated by executing the steps 1 to 4 many times and getting always the same results as shown Table 4.5.

TABLE 4.5 Errors detected using By Session Test Case scenario

Total Sessions	**Num Test group**	**Total HTTP Request**	**Bad Sessions**	**Hibernate Errors**	**PSQL**	**Conn. Reset**	**Conn. Time Out**	**Conn. Refuss**	**Null Pointer**
46	**1**	736	0	0	0	0	0	0	0
46	**2**	736	0	0	0	0	0	0	0
46	**3**	736	0	0	0	0	0	0	0
46	**4**	736	0	0	0	0	0	0	0
46	**5**	736	0	0	0	0	0	0	0
46	**6**	736	0	0	0	0	0	0	0
46	**7**	736	0	0	0	0	0	0	0

However using By Session Algorithm with sessions created by using the algorithm presented in the Section 3.4.1 produce bad sessions as is shown in Table 4.6, this is due to the random nature of these sessions; the Null Pointer Error is caused by some component that has not been created and is been referenced as if it existed. For example a dynamic button that is generated only when the user has accomplished some process (close a new fiscal year for example), if some randomly created session tries to close a year that has not been processed this error will be produced. The Hibernate error is produced because an update or delete of some record that still not exists attempted. Table 4.6 also shows that the percent of Bad Sessions doesn't increase as the number of sessions increases, this evidence that session generation is a completely random process. However, these values can give us a relative measure of the quality of a given set of sessions.

TABLE 4.6 Errors reported by sessions executed sequentially

Total Sessions	Num Test Group	Bad Sessions	Hibernate Errors	PSQL Errors	Conn. Reset Errors	Conn. Time Out Errors	Conn. Refus. Errors	Null Pointer Errors	% Bad Sessions
46	1	23	0	0	0	0	0	149	0.50
46	2	23	0	0	0	0	0	149	0.50
46	3	23	0	0	0	0	0	149	0.50
92	1	32	1	0	0	0	0	194	0.35
92	2	32	1	0	0	0	0	194	0.35
92	3	32	1	0	0	0	0	194	0.35
138	1	56	0	0	0	0	0	386	0.41
138	2	56	0	0	0	0	0	386	0.41
138	3	56	0	0	0	0	0	386	0.41
184	1	78	0	0	0	0	0	441	0.42
184	2	78	0	0	0	0	0	441	0.42
184	3	78	0	0	0	0	0	441	0.42
230	1	85	0	0	0	0	0	596	0.37
230	2	85	0	0	0	0	0	596	0.37
230	3	85	0	0	0	0	0	596	0.37

Therefore this algorithm generates test suites that validate the correctness of the sessions created. If those sessions are created manually and under a predefined script the errors must be cero. If the sessions are created using an artificially generation algorithm the errors produced will report us the quality of the session set created.

4.3.3 *Analysis of the Log Replay Algorithm*

The main ingredient in the Log Replay is the time element, the time element determine the exact moment when an HTTP request has been sent, the big challenge here is to try to re-create one historical or error condition of the past. To test this scenario the following methodology is applied:

1. A snapshot of the Web application state is taken.
2. Under a predefined script of sequences of actions many sessions are created simulating concurrency, some of the sessions are created as dependent of the values inserted or deleted by other sessions executed in parallel. Finally the Web application state resulting is saved again.
3. Using the log generated by the Web application after the code injection and starting with the original Web application state the Log Replay is executed with several compression factors.
4. The resulting Web application state is compared with the resulting Web application state taken on step 3.

We expect not to see any errors after an accurate log replay of a previously generated log file. This is validated by following the steps 1 to 4 specified above. A total of 27 sessions were executed in parallel following a script where the sequence of execution is detailed. Step 4 of the methodology was verified and after exercising the system with several of the compression factor the results were satisfactory. Table 4.7 shows the variation in the number of errors as the compression factor varies. Furthermore, the original 27 sessions required more than 30 minutes to be created manually. However the time needed by the Log Replay to generate a successful log replay was of 46 seconds. This is caused by two factors: the time required by the user using a Web browser to send a HTTP Request always is greater than the time required by a testing tool and, the thinking time of the user could be compressed in such a way that the correspondent HTTP Request doesn't intersect with some previous HTTP Request. Table 4.7 and Figure 4.6 show the resulting errors as the compression factor is varied.

TABLE 4.7 Errors in Log Replay by compression factor

Total Sessions	Total HTTP Requests	Compression Factor	Bad Sessions	Hibernate Errors	PSQL Errors	Conn. Reset Errors	Conn. Time Out Errors	Conn. Refuss Errors	Null Pointer Errors
27	342	1	0	0	0	0	0	0	0
27	342	32	0	0	0	0	0	0	0
27	342	64	0	0	0	0	0	0	0
27	342	125	0	0	0	0	0	0	0
27	342	250	1	1	0	0	0	0	0
27	342	500	5	5	0	0	0	0	0
27	342	1000	6	6	0	0	0	0	0
27	342	2000	9	7	0	0	0	0	9
27	342	4000	7	6	0	0	0	0	4
27	342	8000	7	7	0	0	0	0	0
27	342	16000	5	2	0	0	0	0	21

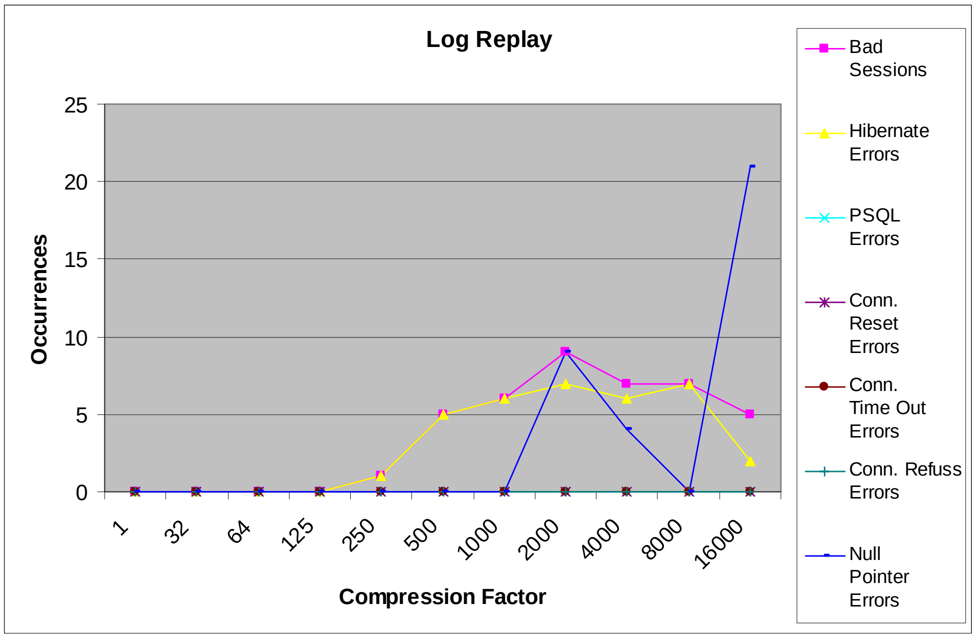


Figure 4.6 Errors in Log Replay by compression factor

The compression factor is the divisor of the time the original HTTP Request was sent, it means a compression factor of 1 will replay the log in the same time than the original load testing was executed. As can be seen from Table 4.7 a bigger compression factor doesn't necessary generates bigger number of bad sessions, although appears contradictory this happens because of two factors: the minimum difference time between two requests as the compression factor grows will always be of 1 millisecond, and the response time of the server is independent to the compression factor. For example two requests that generated an interlock blocking could not generate it under other compression factor because one of them could be nullified by other previous request.

The results in Table 4.7 confirms the expected behavior of the algorithm, starting with a compression factor of 125 the Web application doesn't show any errors as the original historic event replayed.

.

4.3.4 Analysis of the By Block User Session Test Case Algorithms

Although the Concurrent Test Case Algorithm allows determining the number of concurrent users that a Web server can attend, the By Block Test Case algorithm combines concurrency with sequential session execution. This allows us to evaluate the max number of connections supported and the number of errors produced during the concurrent execution. Although the By Block algorithm can better simulate the concurrency on a real scenario, it is more expensive to execute and to validate, requires more data and requires the selection of an optimal block length, which is often a difficult task.

The optimal block length is vital in this type of test because many sessions can be truncated and this can be interpreted as a user leaving the Web site in the middle of the session. As the block length increments, the number of sessions decrements and also the concurrency levels. To validate this algorithm the following experimental methodology was applied:

1. A snapshot of the Web application state is taken.

2. An interval of time is defined and a set of concurrent sessions are created. The sessions are managed into groups representing periods of time equal to the predefined interval.
3. The sessions inside each session group must be present in other groups too so when the test is performed the concurrent group execution will create conflicts between those sessions.
4. The Web application is restored to the state on step 1.

This test has the objective of validating the session grouping by blocks of time. It will report conflicts when interdependent sessions in separate blocks are sent in parallel to the server. For the test, 27 sessions where created according to a previously defined script which determines the execution order and the concurrency level. This script is manually created introducing interdependent sessions at different points of time. The errors reported after the By Block Test is performed will validate that interdependent sessions in different points of times conflict if they are sent in parallel. Although the Concurrent Test Case algorithm can produce the same effects, the results show that the Concurrent Test Case scenario concurrency level is greater than the By Block Test Case algorithm. This result appears discouraging however, the By Block Test Case algorithm exercise the system for longer periods of time because of his time duration characteristic.

Figure 4.7 shows a comparative of the bad sessions generated after using Concurrent and By Block test cases scenarios. As expected using the same number of sessions By Block generates fewer bad sessions than Concurrent test case because of the smaller concurrency level.

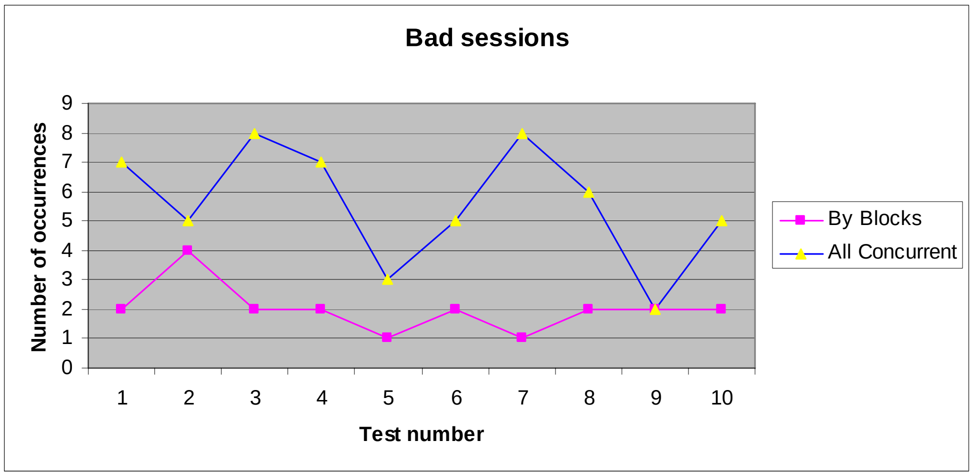


Figure 4.7 Bad sessions using a concurrent and a by block test

The results showed in Figure 4.7 demonstrates that the concurrence level is influenced depending how the sessions are sent to the Web application.

After the experiments we have concluded that using the several algorithms we can take advantage of the same set of sessions to evaluate the Web application in diverse ways. Despite of a deep analysis it can not be possible determine if the statistically based session creation algorithm produce sessions that exercise better the Web application than replicate the original set. Getting the same conclusions than Ruffo et al. [28] and Meier et al. [22], however after an analysis of the data created after each testing, we found that the data inserted by the statistically created sessions has more variety than the inserted by replicating. This is because replicating the same set will exercise the same operations on the same data.

5 CONCLUSIONS AND FUTURE WORK

In this thesis we have presented JTracer, a framework for Web application test suites creation focused on leverage the user session data. A prototype tool relying on an open source tool have been implemented and tested. Several algorithms that characterize tests to the Web application have been implemented and studied. A technique for capture the user session data from the Web application also have been implemented by using a Code Injector program. Furthermore an algorithm that generates artificially user sessions using information learned from log files have been implemented and the sessions produced by it have been compared with a set of replicated sessions.

5.1 Conclusions

In our research we have demonstrated that test session suites can be created automatically for Secure Web applications developed using frameworks such as Java Server Faces. The JTracer Code Injector module offers advantages over proxies and ad-hoc modification of Application Servers for test log data collection.

The proposed framework can reduce error introduction due to repetition of transactions to generate large loads. The framework can be implemented with a variety of algorithms that create artificial session data. Diverse session data can get diverse web application response; it brings variety of data to the test engineer.

Finally JTracer can work with any Test Engine in the market, although we used JMeter testing tool as Test Engine, other testing tools can be used only by generating a test suite compatible file with the chosen Test Engine.

5.2 Future Work

Future work can be dedicated towards:

- Using request parameter databases in order to generate test suites much larger than the log collected.
- Investigate if JTracer could suggest a schema for specifying request parameters.
- Investigate how JTracer can be modified to evaluate how well web applications comply with framework guidelines.

REFERENCES

[1] Mauro Andreolini, Michele Colajanni, Paolo Valente, "Design and testing of scalable Web-based systems with performance constraints", Proceedings of the 2005 Workshop on Techniques, Methodologies and Tools for Performance Evaluation of Complex Systems (FIRB-PERF'05), 2005.

[2] Baresi, L., Garzotto, F., and Paolini, P., "Extending UML for Modeling Web Applications". Proceedings of the 34th Annual Hawaii International Conference on System Sciences. IEEE Computer Society, Maui, Hawaii, 2001.

[3] P. Barford and M. E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In Proc. of ACM SIGMETRICS '98, pages 151–160,1998.

[4] Conallen, J., "Modeling Web Application Architectures with UML", Communications of the ACM, Vol.42, No.10,1999, pp. 63-70.

[5] Chien-Hung, Liu David C, Kung Pei Hsia, "An object-oriented web test model for testing Web applications", Quality Software, 2000. Proceedings. First Asia-Pacific Conference on. IEEE 2000.

[6] Chien-Hung, Liu David C, Kung Pei Hsia, Chih-Tung Hsu, "Structural Testing of Web Applications", Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on. IEEE 2000.

[7] Deng Yuetang, Wang Jiong, "Testing Web Database Applications", ACM SIGSOFT Software Engineering Notes Volume 29, Issue 5, 2004.

[8] Deng Yuetang, Phyllis Frankl, JiongWang, "Testing Web Database Applications", Workshop on testing, analysis and verification of web services (TAV-WEB) papers. 2004.

[9] ADMG Group Digiweb Web application system.

[10] Di Lucca, G., Fasolino A., Faralli F., "Testing Web Applications", 18th IEEE International Conference on Software Maintenance (ICSM'02), 2002.

[11] Elbaum, S., Karre S., and Rothermel G.. "Improving web application testing with user session data". In Int Conf on Soft Eng, 2003.

[12] Elbaum Sebastian, Rothermel Gregg, Karre Srikanth, "Leveraging User-Session Data to Support Web Application Testing", IEEE Transactions on Software Engineering, 2005.

[13] Halfond William G.J., and Orso Alessandro, “Improving Test Case Generation for Web Applications Using Automated Interface Discovery”, Foundations of Software Engineering, Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.

[14] Harrold, M. J., Gupta, R., and Soffa, M. L.. A methodology for controlling the size of a test suite. ACM Trans on Soft Eng Meth, 2(3):270–285, 1993.

[15] http://htmlunit.sourceforge.net/

[16] http://httpunit.sourceforge.net/

[17] Ji-Tzay Yang, Jiun-Long Huang, Feng-Jian Wang, William. C. Chu, An Object-Oriented Architecture Supporting Web Application Testing, 23rd International Computer Software and Applications Conference, IEEE 1999.

[18] http://jakarta.apache.org/jmeter/

[19] Lei Xu, Baowen Xu, A framework for Web Application Testing, Proceedings of the 2004 International Conference on Cyberworlds, IEEE 2004.

[20] Li J., Chen J., and Chen P.. “Modeling Web Application Architecture with UML”, Proceedings of the 36th International Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society, Xi’an, 2000, pp. 265-274.

[21] Maruyama, H., Tamura K., and Uramato, N, “XML and Java – Developing Web Applications”. *Addison Wesley*, 1999.

[22] Meier J.D., Farre Carlos, Bansode Prashant, Barber Scott, Rea Dennis, “Performance Testing Guidance for Web Applications”, Microsoft Coporation 2007.

[23] D. A. Menasce. Tpc-w: a benchmark for e-commerce. IEEE Internet Computing, May-June 2002.

[24] D. A. Menasce, A. V. Almeida, R. Fonseca, and M. A. Mendes. "Scaling for E-business: Technologies, Models and Performance and Capacity Planning". Prentice Hall, 2000.

[25] Mohammed Sidat. “Automated Stress Testing of Web Applications using User Session Data”. Department of Computer Science, King’s College London 2005.

[26] Nuo Li, Qin-qin Ma, Ji Wu, Mao-zhong Jin, Chao Liu, A Framework of Model-Driven Web Application Testing, Proceedings of the 30th Annual International Computer Software and Applications Conference, ACM 2006

[27] Ricca Filippo and Tonella Paolo, “Analysis and Testing of Web Applications”, Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on, IEEE 2002.

[28] G. Ruffo, R. Schifanella, and M. Sereno, R. Politi, "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance". Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA’04), 2004

[29] Sampath Sreedevi, Sprenkle Sara, Gibson Emily, Pollock Lori, Souter Amie, “Analyzing clusters of web application user sessions”, International Conference on Software Engineering, Proceedings of the third international workshop on Dynamic analysis, 2005.

[30] Sampath Sreedevi, Mihaylov Valentin, Souter Amie, Pollock Lori, "A Scalable Approach to User-session based Testing of Web Applications through Concept Analysis". Proceedings of the 19th IEEE international conference on Automated software engineering. 2004

[31] Sant Jessica, Souter Amie, Greenwald Lloyd, “An Exploration of Statistical Models for Automated Test Case Generation” Workshop on Dynamic Analysis - WODA 2005.

[32] http://www.softwareqatest.com/qatweb1.html

[33] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

[34] Sprenkle Sara, Gibson Emily, Sampath Sreedevi, and Pollock Lori, "A Case Study of Automatically Creating Test Suites from Web Application Field Data", Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications, 2006.

[35] Sprenkle Sara, Sampath Sreedevi, Gibson Emily, Pollock Lori, Souter Amie, "An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications". Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005.

[36] Standard Performance Evaluation Corporation. http://www.specbench.org.

[37] Webbench. www.veritest.com/benchmarks/webbench/webbench.asp.

[38] Webstone. http://www.mindcraft.com/webstone/

[39] Xiaoping Jia and Hongming Liu, "Rigorous and Automatic Testing of Web Applications". 6th IASTED International Conference on Software Engineering and Applications, pages 280--285, Nov. 2002.

[40] Yu Qi, David Kung and Eric Wong, "An Agent-based Testing Approach for Web Applications".