

# Tuning Dynamic Web Applications using Fine-Grain Analysis

Jordi Guitart, David Carrera, Jordi Torres, Eduard Ayguadé and Jesús Labarta  
European Center for Parallelism of Barcelona (CEPBA)  
Computer Architecture Department - Technical University of Catalonia  
C/ Jordi Girona 1-3, Campus Nord UPC, Mòdul C6, E-08034 Barcelona (Spain)  
{jguitart, dcarrera, torres, eduard, jesus}@ac.upc.es

## Abstract

*In this paper we present a methodology to analyze the behavior and performance of Java application servers using a performance analysis framework. This framework considers all levels involved in the application server execution (application, server, virtual machine and operating system), allowing a fine-grain analysis of dynamic web applications. The proposed methodology is based on the suggestion of Hypotheses that could explain the presence of certain Symptoms that lead to bad server performance, an unexplained server behavior or a server malfunction. The methodology establishes that Hypotheses must be verified (in order to confirm or discard them) by performing some Actions with the performance analysis framework. In order to show the potential of the proposed analysis methodology, we present three successful experiences where a detailed and correlated analysis of the application server behavior has allowed the detection and correction of three performance degradation situations.*

## 1. Introduction

The increasing popularity of dynamic web content in the applications currently developed for Internet demands new performance requirements to the application servers that host them. To achieve these performance requirements, fine-grain tuning of these servers is needed, but this tuning can be a hard work due to the large complexity of these environments (including the application server, distributed clients, a database server, etc.). Tuning application servers for performance requires of tools that allow an in-depth analysis of application server behavior and its interaction with the other system elements. These tools must consider all levels involved in the execution of web applications (operating system, Java Virtual Machine, application server and application) if they want to provide significant performance information

to the administrators: the origin of performance problems can reside in any of these levels or in their interaction.

Although a number of tools have been developed to monitor and analyze the performance of dynamic web applications (e.g. Introscope [15], JProbe & Performasure [12] and e-TEST & OneSight [5]), none of them allow a fine-grain analysis of the dynamic web applications behavior considering all levels involved in the application server execution. Some tools focus the analysis on the application server level, neglecting the interaction with the system. Other tools (e.g. Introscope, JProbe) incorporate the analysis of the system activity to their monitoring solution, but summarize this analysis giving general metrics (as CPU utilization or JVM memory usage) providing only a quantitative analysis of the server execution. Summarizing, existing tools base their analysis on calculating general metrics that intend to represent the system status. Although this information can be useful for the detection of some problems, it is often not sufficiently fine grained and lacks of flexibility. For this reason, in this paper we present an analysis methodology to perform a complete analysis of the application servers behavior based on providing to the user detailed information about all levels involved on the application server execution, giving him the chance to construct his own metrics, focused to the problem he needs to solve.

The proposed methodology is based on the suggestion of *Hypotheses* that try to explain the presence of *Symptoms* showing bad server performance, an unexplained server behavior or a server malfunction. The methodology establishes that *Hypotheses* must be verified (in order to confirm or discard them) performing *Actions* with the performance analysis framework. This methodology is based on previous experience and knowledge in the analysis of scientific parallel applications. As examples of the proposed analysis methodology, we present three successful experiences where a detailed and correlated analysis of the application server behavior has allowed the detection and correction of three performance degradation situations.

The rest of the paper is organized as follows: Section 2

introduces dynamic web applications. Section 3 describes the analysis methodology proposed in this paper. Section 4 describes the experimental environment used in our evaluation. Section 5 presents three analysis experiences showing the effectiveness of the proposed analysis methodology, and finally, Section 6 presents the conclusions of this paper.

## 2. Dynamic Web Applications

Dynamic web applications are a case of multi-tier application and are mainly composed of a Client tier and a Server tier, which in its turn uses to consist of a front-end web server, an application server and a back-end database. The client tier is responsible of interacting with application users and to generate requests to be attended by the server. The server tier implements the logic of the application and is responsible of serving user-generated requests. When the client sends to the web server an HTTP request for dynamic content, the web server forwards the request to the application server (as understood in this paper, a web server only serves static content), which is the dynamic content server. The application server executes the corresponding code, which may need to access the database to generate the response. The application server formats and assembles the results into an HTML page, which is returned by the web server as an HTTP response to the client. The implementation of the application logic in the application server may take various forms, including PHP [11], Microsoft Active Server pages [8], Java Servlets [14] and Enterprise Java Beans (EJB) [13]. This study focuses on Java Servlets, but the analysis methodology proposed can be applied with the other mechanisms for generating dynamic web content, with the same effectiveness.

## 3. Analysis Methodology

Our analysis methodology starts when an observation that can represent a performance lost or a server malfunction is produced when doing typical server maintenance work (for example, when examining the server log files), or when performing a study of basic metrics looking for anomalous values or behaviors. These observations showing low performance or unexplained errors are the *Symptom* that something is going wrong in our server, and motivate an in-depth analysis of the server behavior.

When we detect a *Symptom* of a server malfunction, the analysis methodology indicates that we have to suggest a *Hypothesis* to explain this *Symptom* apparition, and using the performance analysis framework, perform the necessary *Actions* to confirm or discard this

*Hypothesis*. This framework, which consists of an instrumentation tool called Java Instrumentation Suite (JIS [3]) and a visualization and analysis tool called Paraver [10], considers all levels involved in the application server execution (operating system, JVM, application server and application), allowing a fine-grain analysis of dynamic web applications. For example, the framework can provide detailed information about thread status, system calls (I/O, sockets, memory & thread management, etc), monitors, services, connections, etc. Further information about the implementation of the performance analysis framework and its use for the analysis of dynamic web applications can be found in [3] and [6].

The result of the *Actions* can confirm the *Hypothesis*, discard it, or detect another *Symptom*. The methodology establishes that we have to perform the necessary *Actions* until we are able to verify or discard the *Hypothesis*. In the first case, we have found the cause of server anomalous behavior. In the second case, we have to suggest another *Hypothesis*, and start again the verification process based on *Actions*.

## 4. Experimental Environment

We use Tomcat v4.0.6 [7] as the application server. Tomcat is an open-source servlet container developed under the Apache license. Its primary goal is to serve as a reference implementation of the Sun Servlet and JSP specifications, and also to be a quality production servlet container. Tomcat can work as a standalone server (serving both static and dynamic web content) or as a helper for a web server (serving only dynamic web content). In this paper we use Tomcat as a standalone server. Tomcat follows a connection service schema where one thread (the `HttpConnector`) is responsible of accepting new incoming connections on the server's listening port and assigning to them a socket structure. After this, the `HttpConnector` assigns the created socket structure to another thread (an `HttpProcessor`) and continues accepting new connections. The `HttpProcessor` will be responsible of attending and serving the received requests through the persistent connection established with the client. Persistent connections are a feature of HTTP 1.1 that allows serving different requests using the same connection. The pattern of a persistent connection in Tomcat is shown in Figure 1. On each "connection" we distinguish the execution of several "requests" and the time devoted to maintain the connection persistence ("connection (no request)"), where server is maintaining opened the connection waiting for another client request. A connection timeout is programmed to close the connection if no more requests are received. Each "request" consists of the server pre/post process ("request (no service)") and the "service" execution. We have

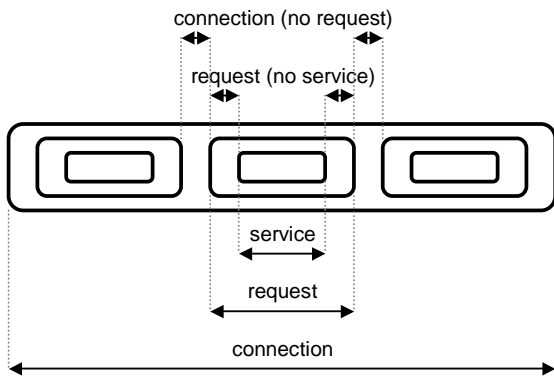


Figure 1. Tomcat persistent connection pattern

configured Tomcat setting the maximum number of `HttpProcessors` to 25 and the connection timeout to 10 seconds.

The experimental environment also includes a deployment of the RUBiS (Rice University Bidding System) [1] benchmark servlets version 1.4 on Tomcat. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. RUBiS defines 27 interactions. 5 of the 27 interactions are implemented using static HTML pages. The remaining 22 interactions require data to be generated dynamically. RUBiS supplies implementations using some mechanisms for generating dynamic web content like PHP, Servlets and several kinds of EJB. RUBiS also supplies a client emulator for generating workloads consisting of a number of

concurrent clients interacting with the server. Each emulated client opens a session with the server. Each session is a persistent HTTP connection with the server. Using this connection, the client repeatedly makes a request, parses the server's response to the request, and follows a link embedded in the response. Each emulated client waits for an amount of time, called the think time, before initiating the next interaction. The think time is generated from a negative exponential distribution with a mean of 7 seconds. RUBiS defines two workload mixes: a browsing mix made up of only read-only interactions and a bidding mix that includes 15% read-write interactions.

Tomcat runs on a 4 processor Intel XEON 1.4 GHz with 2 GB RAM running 2.5.63 Linux kernel. We use MySQL v3.23.43 [9] as our database server with the MM.MySQL v3.0.8 JDBC driver. MySQL runs on a dual processor Intel XEON 2.4 GHz with 2 GB RAM running 2.4.18 Linux kernel. We have also two dual processor Intel XEON 2.4 GHz with 2 GB RAM running Linux kernel 2.4.18 machines running the RUBiS 1.4 client emulation software. Each client emulation machine emulates 850 clients performing requests to the server during 150 seconds using the browsing mix (read-only interactions). Server machine is connected with database and client machines through independent 100 Mbps Ethernet LANs. For our experiments we use the Sun JVM 1.4.2 for Linux, using the server JVM instead of the client JVM and setting the initial and the maximum Java heap size to 512 MB.

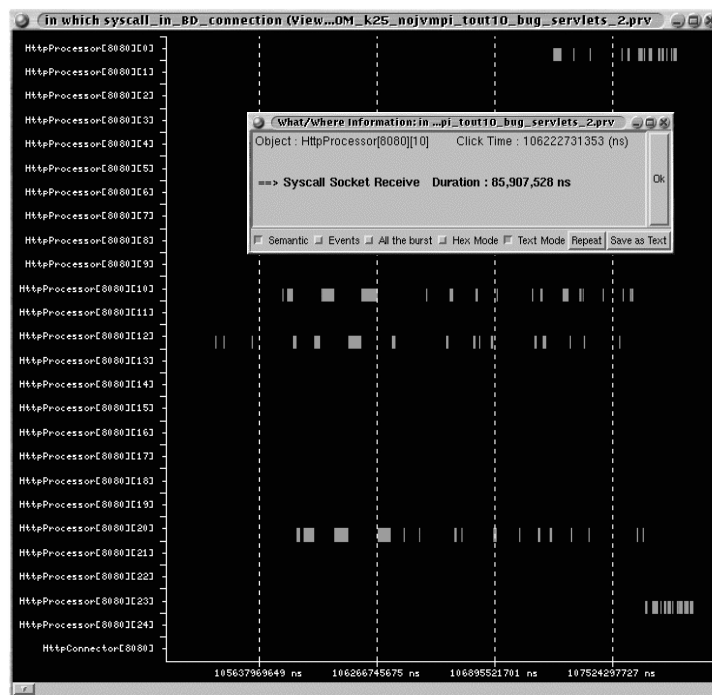


Figure 2. System calls performed by `HttpProcessors` when they have acquired a database connection

## 5. Analysis Experiences

In this section, we summarize three experiences that demonstrate the effectiveness of the proposed analysis methodology for detecting performance problems and/or errors on application servers. The complete description of the analysis performed as well as the description of other successful analysis experiences can be found in a companion report to this paper [6].

### 5.1 Case Study 1

Our first case study starts from an observation made when inspecting the Tomcat log files. Good Tomcat administrators should perform the observation of log files periodically in order to detect possible server malfunctions. When examining the RUBiS context log file of Tomcat, we find error messages like:

- Servlet.service() for servlet BrowseCategories threw exception java.lang.NullPointerException at com.mysql.jdbc.ResultSet.first(ResultSet.java:2293)
- java.sql.SQLException: Operation not allowed after ResultSet closed

The appearance of these error messages in the log file is a *Symptom* that something is going wrong, and motivated an in-depth analysis to determine the causes of this behavior. The proposed analysis methodology

establishes that we had to suggest a *Hypothesis* that explains the *Symptom* detected. Considering the messages shown before, our *Hypothesis* was that the problem was related with the database access.

At this point, we had to take the necessary *Actions* to verify the *Hypothesis* made (using the performance analysis framework). In this case, we had to verify if database access was done correctly.

Our first *Action* to verify the *Hypothesis* was to analyze which system calls were performed by HttpProcessors when they had acquired a database connection. This information is displayed in Figure 2 (horizontal axis is time and vertical axis identifies each thread), where each burst represents the execution of a system call when the corresponding HttpProcessor has acquired a database connection. As indicates the textual information in the figure, HttpProcessors get database information using socket receive calls. This *Symptom* corresponds to the expected behavior if managing correctly the database connections, thus we need more information about the database access to verify our *Hypothesis*.

Then, the next *Action* taken was to analyze the file descriptors used by the system calls performed by HttpProcessors when they had acquired a database connection. This information is displayed in Figure 3, where each burst indicates the file descriptor used by the system call performed by the corresponding HttpProcessor when it has acquired a database connection.

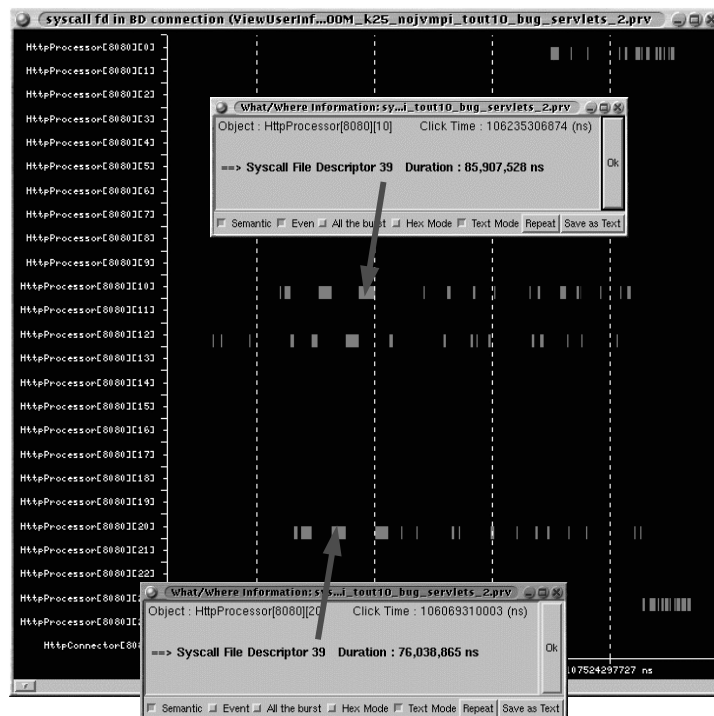


Figure 3. File descriptors used by the system calls performed by HttpProcessors when they have acquired a database connection

As indicates the textual information in the figure, several HttpProcessors are accessing the database using the same file descriptor (that is, using the same database connection). This is conceptually incorrect, and should not happen. This *Symptom* confirmed our *Hypothesis* about a wrong access to database.

At this point, we wanted to determine why several HttpProcessor used the same file descriptor to access the database, so we suggested another *Hypothesis* that located the problem in the RUBiS database connection management. The *Action* taken to verify this *Hypothesis* consisted of inspecting the RUBiS servlets source code. This inspection revealed the following bug. Each kind of RUBiS servlet declares three class variables (ServletPrinter sp, PreparedStatement stmt and Connection conn). These class variables are shared by all the servlet instances, and this can provoke multiple race conditions. For example, it is possible that two HttpProcessors access the database using the same connection conn.

This problem can be avoided declaring these three class variables as local variables in the doGet method of the servlet, and pass them as parameters when needed.

## 5.2 Case Study 2

A good practice when tuning an application server for performance is to make periodical studies of some basic metrics that indicate the performance of the application server. These metrics include for example the average service time per HttpProcessor, the overall throughput, the client requests arrivals rate, etc. The result of this basic analysis can encourage us to initiate a more detailed study to determine the causes of an anomalous value in these metrics. For example, our second case study starts from an observation made when analyzing the average service time per HttpProcessor on server.

Figure 4 shows the average service time for each HttpProcessor, calculated using the performance analysis framework. In this figure we notice one HttpProcessor with an average service time considerably higher than the others. This is a *Symptom* of an anomalous behavior of this HttpProcessor, and motivated an in-depth analysis to determine the causes of this behavior. We first analyzed the state distribution when the HttpProcessors are serving requests. Figure 5 shows the percentage of time spent by the HttpProcessors on every state (run, uninterruptible blocked, interruptible blocked, waiting in ready queue, preempted and ready). In this figure we realize that the problematic HttpProcessor is most of the time in interruptible blocked state (about 92% of time) while the other HttpProcessors are blocked about the 65% of time.

In order to explain this *Symptom*, our *Hypothesis* was to assume that the HttpProcessor could be blocked waiting

response from the database. We inferred this *Hypothesis* because the database is a typical resource that can provoke long waits when working with application servers. To verify this *Hypothesis*, the *Action* taken was to analyze the system calls performed by HttpProcessors when serving requests. This analysis revealed that the problematic HttpProcessor was not blocked in any system call, which means that it was not blocked waiting response from database, but did it have at least an open connection with the database? To answer this question, the *Action* taken consisted of analyzing when HttpProcessors acquire database connections. In this case, we observed that the problematic HttpProcessor blocked before acquiring any database connection.

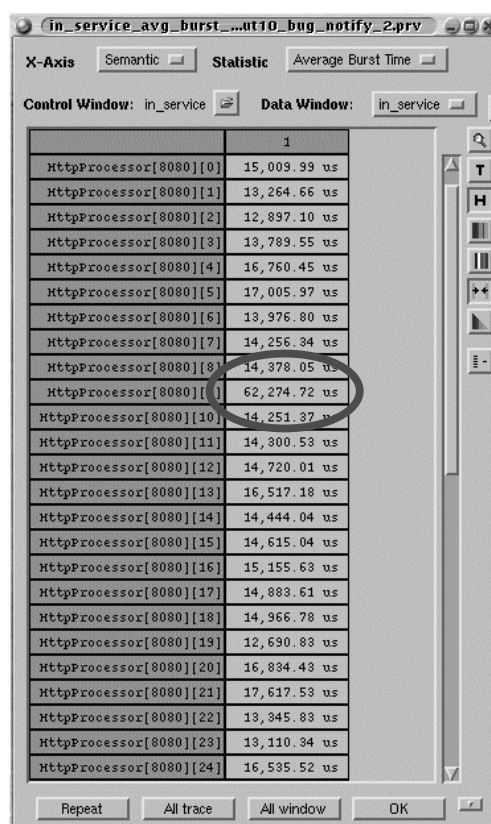


Figure 4. Average service time per HttpProcessor

With all this information we concluded that our first *Hypothesis* was wrong, that is, the problematic HttpProcessor was not waiting response from the database. Therefore, we needed a new *Hypothesis* to explain why the problematic HttpProcessor was blocked most of the time. Considering that, as we have seen before, the problematic HttpProcessor had not acquired any database connection yet, our new *Hypothesis* was that this HttpProcessor could have problems acquiring the database connection. To verify this *Hypothesis*, we

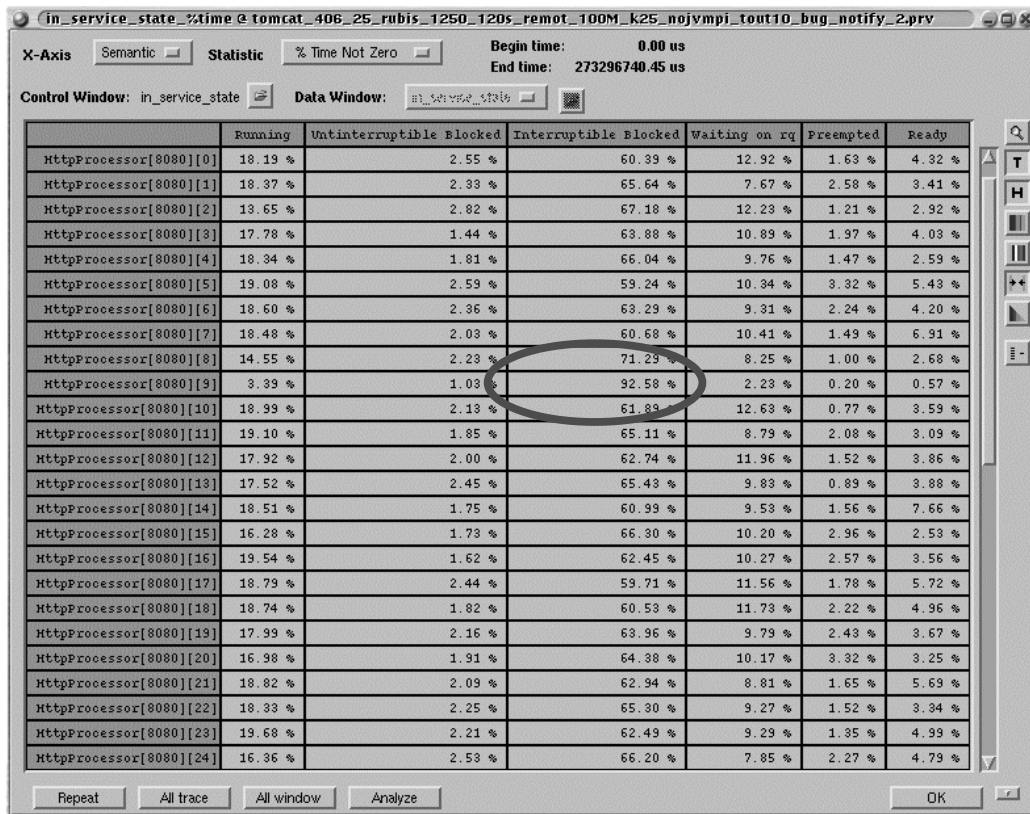


Figure 5. State distribution of HttpProcessors during service (in percentage)

displayed the database connections management, which is shown in Figure 6. Light color indicates the acquisition of a database connection and dark color indicates the wait for a free database connection. Notice that the problematic HttpProcessor (HttpProcessor 9 in the figure) is blocked waiting for a free database connection. This *Symptom* confirmed the *Hypothesis* that there could be problems acquiring database connections. In this figure we also detect the origin of the problem on the database connection management, because it can occur that a database connection is released, while there are some HttpProcessors waiting for a free database connection, but they are not notified. Notice that HttpProcessors 4 and 9 are blocked waiting for a free database connection. When HttpProcessor 14 releases its database connection, it notifies HttpProcessor 4 that can acquire this connection and continue its execution. Other HttpProcessors holding a database connection release it, but none of them notifies HttpProcessor 9.

Trying to explain this anomalous behavior, our *Hypothesis* supposed that a wrong database connection management at RUBiS is causing the problem. In order to verify this *Hypothesis*, the *Action* taken was to inspect the RUBiS servlets source code. This inspection revealed a bug. By default, in RUBiS one HttpProcessor only

notifies a connection release if free database connection stack is empty. But imagine the following situation:

There are  $N$  HttpProcessors that execute the same RUBiS servlet, which has a pool of  $M$  connections available with the database, where  $N$  is greater than  $M$ . This means that  $M$  HttpProcessors can acquire a database connection and the rest  $(N - M)$  HttpProcessors block waiting for a free database connection. Later, an HttpProcessor finishes executing the servlet and releases its database connection. The HttpProcessor puts the connection in the pool and, as the connection pool was empty, it notifies the connection release.

Due to this notification, a second HttpProcessor wakes up and tries to get a database connection. But before this second HttpProcessor can get the connection, a third HttpProcessor finishes executing the servlet and releases its database connection. The third HttpProcessor puts the connection in the pool and, as the connection pool was not empty (the second HttpProcessor has not got the connection yet), it does not notify the connection release. The second HttpProcessor finally acquires its database connection and the execution continues with a free connection in the pool, but with HttpProcessors still blocked waiting for free database connections.

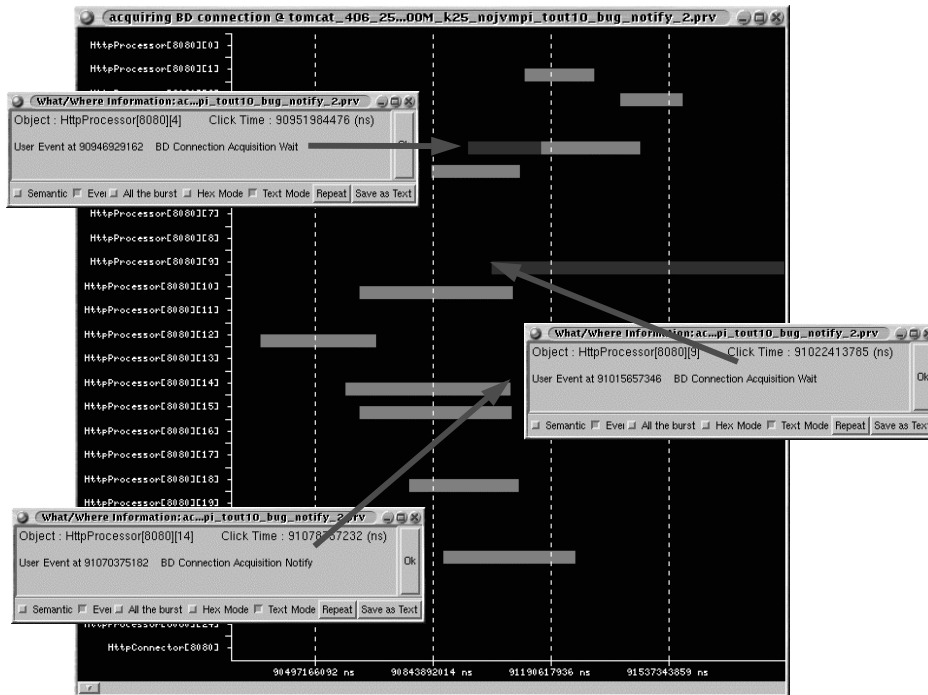


Figure 6. Database connections acquisition process

### 5.3 Case Study 3

Our third case study starts from an observation made when analyzing the overall throughput on server. We observed that when increasing the load from 550 to 800 clients, the throughput only increased from 95 to 105 replies/s, while we expected to reach 135 replies/s if scaling linearly. This is a *Symptom* indicating that the server was saturated (it could not handle all the input load). Our *Hypothesis* to explain this *Symptom* consisted of assuming that server could be saturated due to resources. In order to determine the causes of server saturation, the *Action* taken consisted of measuring the time spent by the server with opened client connections, specifying the time devoted to each phase of the connection. This information is displayed in Figure 7a, in which the two analyzed loads are displayed, showing that when increasing the number of clients, a heavy increment of time in “connection (no request)” occurs. Time spent in “request (no service)” and in “service” remains practically equal.

Going deeply into time spent in “connection (no request)”, we concluded that it consists basically of the execution of the `poll` system call (around 95% of time). This system call blocks waiting for some event in a file descriptor. Considering this behavior, it is expected the state of `HttpProcessors` (run (R), uninterruptible blocked (UB), interruptible blocked (IB), waiting in ready queue (WRQ), preempted (P) and ready (RD)) when in

“connection (no request)” to be mainly IB. To verify this behavior, the *Action* taken was to analyze the state of `HttpProcessors` when in “connection (no request)”. This information is displayed in Figure 7b. Notice that, with an input load of 800 clients, `HttpProcessors` spent more of 50% of their time in WRQ state. This means that some event has been detected on a file descriptor, but there is not available processor to handle the incoming event. This *Symptom* confirmed our *Hypothesis* that server could be saturated due to resources.

With this information, we decided to tune our server to run with two processors instead of one. With this new configuration the throughput increased to 135 replies/s. In the same way, time spent by the server in “connection (no request)” were then similar with 550 and 800 clients, as shown in Figure 7a.

## 6. Conclusions

In this paper we have presented three successful experiences where a detailed analysis has allowed the detection and correction of three performance degradation situations when executing the RUBiS benchmark with the Tomcat application server. This analysis is based on a methodology that specifies the suggestion of *Hypotheses* to explain the *Symptoms* of bad server performance or server malfunction appeared, and the verification of these *Hypotheses* performing *Actions* with a performance

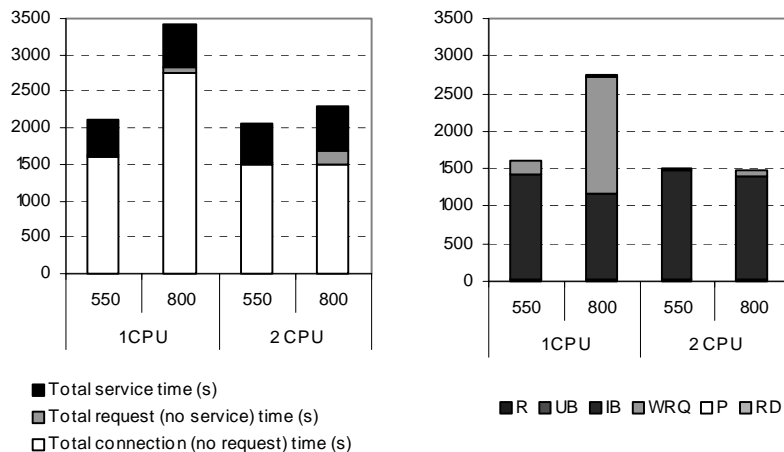


Figure 7. a) Time with client opened connections  
b) State of HttpProcessors when in "connection (no request)"

analysis framework. This framework considers all levels involved in the application server execution (operating system, JVM, application server and application). The three analysis experiences demonstrate the benefit of disposing of correlated information about all these levels to perform a fine-grain analysis of server execution.

The proposed analysis methodology intends to help the application server administrators when they detect an anomalous server behavior during the server maintenance process (log files review, analysis of basic performance metrics, etc.) providing fine-grain information that is not accessible using other analysis tools. Although the experiences presented are based on servlets to implement the server application logic for generating dynamic web content, as the proposed methodology as the performance analysis framework can be used with the same effectiveness with other dynamic web content generators, constituting a powerful mechanism for performance tuning and analysis of web applications.

## 7. Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01, by Direcció General de Recerca of the Generalitat de Catalunya under grant 2001FI 00694 UPC APTIND and by the CEPBA (European Center for Parallelism of Barcelona). For additional information about the authors, please visit the Barcelona eDragon Research Group web site [2].

## 8. References

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani and W. Zwaenepoel. *Specification and Implementation of Dynamic Web Site*

*Benchmarks*. IEEE 5th Annual Workshop on Workload Characterization (WWC-5), pp. 3-13, Austin, USA. November 25, 2002.

[2] Barcelona eDragon Research Group <http://www.cepba.upc.es/eDragon>

[3] D. Carrera, J. Guitart, J. Torres, E. Ayguade and J. Labarta. *Complete Instrumentation Requirements for Performance Analysis of Web based Technologies*. 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), pp. 166-176, Austin, USA. March 6-8, 2003.

[4] E. Cecchet, J. Marguerite and W. Zwaenepoel. *Performance and Scalability of EJB Applications*. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), pp. 246-261. Seattle, USA. November 4-8, 2002.

[5] Empirix Solutions for Web Application Performance <http://www.empirix.com>

[6] J. Guitart, D. Carrera, J. Torres, E. Ayguade and J. Labarta. *Successful Experiences Tuning Dynamic Web Applications using Fine-Grain Analysis*. Research Report UPC-DAC-2004-3 / UPC-CEPBA-2004-2. January 2004.

[7] Jakarta Tomcat Servlet Container <http://jakarta.apache.org/tomcat/>

[8] Microsoft Active Server Pages - <http://www.asp.net>

[9] MySQL - <http://www.mysql.com>

[10] Paraver - <http://www.cepba.upc.es/paraver>

[11] PHP Hypertext Preprocessor - <http://www.php.net/>

[12] Quest Software Solutions for Java/J2EE <http://www.quest.com/>

[13] Sun Microsystems. Enterprise Java Beans Technology <http://java.sun.com/products/ejb/>

[14] Sun Microsystems. Java Servlets Technology <http://java.sun.com/products/servlet/>

[15] Wily Technology Solutions for Enterprise Java Application Management <http://www.wilytech.com/solutions/index.html>