vrije Universiteit amsterdam

Resource Provisioning for AJAX Web applications

Master's thesis

in PARALLEL AND DISTRIBUTED COMPUTER SYSTEMS

Supervisor

Dr. Guillaume Pierre

Author

Andreea Alexandra Vişan
(Student no. 212803)

July 12, 2011

Abstract

Most of the nowadays popular web applications are enriched with AJAX techniques that offer a better experience than classical applications. However, these techniques greatly impact the server's performance because of their complex features. Regardless their complexity, web applications should ensure the achievement of the imposed performance target even in the presence of workload variations or flash crowds. For this objective to be reached, they should be able to reprovision on-the-fly and with minimal costs their resources. This thesis explores the challenges that AJAX features create for resource provisioning by identifying the aspects that greatly impact the web server's performance in this situation. Moreover, this thesis designs performance models for AJAX web applications that can be further used by resource provisioning algorithms to determine the optimal configuration of the web server according to the workload.

Keywords: AJAX web applications, persistent connections, dynamic content, dynamic resource provisioning

Contents

List of Figures

Chapter 1

Introduction

Web applications have become an essential part of everyday life as they offer online access to all essential services such as shopping, banking, information, and entertainment. They were originally used only to share static pages, but rapidly evolved into a more dynamic medium and today are composed of extremely interactive pages. However, they still rely heavily on the paradigm based on full-page retransmissions that presents a series of negative consequences, such as slow page refreshes and flickering. Frequently, full-page retransmission is leading to a poor user experience.

By adding AJAX features, a web application becomes more interactive, more responsive, faster and friendlier. The user is encouraged to interact more with the page as these interactions and the responses generated by the web server are handled more elegantly by the AJAX engine placed on the user's machine. The overall experience of the user is highly improved. For instance, all major Google products (Orkut, Gmail, Google Groups, Google Suggest and Google Maps) use AJAX. Moreover, many of the features that people like in Flickr depend on AJAX, and eBay.com and Amazon's A9.com search engine applies similar techniques. These successful projects demonstrate that AJAX is practical for real-world applications as AJAX web applications can be any size, from the very simple, single-function Google Suggest to the very complex and sophisticated Google Maps.

Taking a look at Google Suggest we observe that the suggested terms update almost instantly as we type the request. Moreover, while zooming in, grabbing or scrolling Google Maps, everything happens almost instantly, without waiting for pages to reload and giving the feeling of a desktop application. Moreover, the same happens regardless of the number of users who do similar actions on the same page (and there are millions of users browsing concurrently Google Map at every moment of time). Furthermore, the user has the same good experience and receives fast responses even if the workload seen by the web servers evolves, for

instance because of time changes. In order to be able, at any moment of time, to offer a good experience to their users, these web applications are adjusting their processing capacity according to the current workload.

For example, amazon.com, the world's largest online retailer, was overwhelmed by the flash crowd caused by the launching of a new gadget in 2008. Then, all US servers crashed for about two-hours. Because of that, the estimated loss was $14.835 billion which calculates to nearly $29,000 per minute [13], a situation to avoid for any application or business.

As a consequence, a mandatory requirement of all web applications is to be capable of on-the-fly processing capacity adjustment when facing request load modifications generated by either workload amount variations or workload mix variations. This resource adjustment has the objective to maintain acceptable end-to-end performance and to respect the imposed Service Level Agreement (SLA) with minimal costs. The SLA may define, for instance, the maximum average response time that the application should offer.

Dynamic resource provisioning supposes that web applications are capable to request more resources on-the-fly when they need more processing capacity because the SLA target is violated. In this situation, one has to decide which component of the web application should be re-provisioned for optimal effect. New resources can be requested from grids or from clouds, as they prove to be an attractive platform to host web applications. Moreover, when some resources are no longer necessary, web applications should be able to release them.

Currently, modern web applications involve a big number of components having complex relationships between them such that their dynamic resource provisioning is a very challenging, but difficult task.

Even if the problem of dynamic resource provisioning has been intensively studied, unfortunately, all current research focuses only on traditional web applications assuming that applications use the classical request/response/display workflow. As AJAX web applications are widely used and they suppose a much more complex interaction than the traditional workflow, special attention must be paid to this situation.

This thesis explores the challenges that AJAX web applications create for dynamic resource provisioning because to their specific characteristics: asynchronous requests sent over connections between users and the web server that are kept persistent over long periods of time. Moreover, our research addresses only the more interesting case of applications serving highly dynamic content. The aim is to identify the aspects that differentiate resource provisioning of classical web applications from those enriched with AJAX techniques.

The first contribution of this thesis consists of demonstrating the fact that the AJAX technology greatly impacts the web server's performance and leads to a completely different performance profile such that it deserves special research in

the context of resource provisioning. Furthermore, the study focused on AJAX applications hosted by process-based web servers and identifies for this scenario the most important system resources that influence the user perceived performance. This thesis identified that the amount of memory allocated to the web server is very important because it directly impacts the number of active connections that can be successfully served concurrently. This result is determined by the fact that AJAX applications maintain long-lived persistent connections such that, in this case, a bigger number of concurrent connections have to be handled by the web server, compared to classical applications. Moreover, this thesis presents a performance model for AJAX applications capable of predicting the future performance obtained with a different provisioning or when facing different workload. According to the output of the application's performance model, whenever the SLA is no more respected, one can determine what is the component that brings the most benefit if it would be reprovisioned and reprovisions it. To the best of our knowledge, this is the very first study on the particularities of AJAX applications in the context of resource provisioning.

This thesis is organized as follows. Chapter 2 discusses the current state of the research on resource provisioning for web applications and examines the aspects that differences AJAX applications we are focusing on from the classical cases that were already studied. Chapter 3 discusses the methodology followed by our study on resource provisioning for AJAX web applications and then details the characteristics and features of the applications considered as case study. Furthermore, Chapter 4 makes a comparative analysis of the server's performance of the AJAX and non-AJAX scenarios and emphasizes the differences between the two performance profiles. Chapter 5 presents the performance model of the web application considered as proof of concept. Because of their new features, in the case of AJAX applications, there are different aspects that impact the web server's performance. However, AJAX features impact not only the performance profile but also arise some issues when reprovisioning the application. Chapter 6 discusses these problems and different tunings having relevance in the context of resource provisioning. Finally, Chapter 7 concludes by discussing the impact of our research and giving possible paths for future work and improvements.

Chapter 2

Related Work

Playing an important role in nowadays life, web applications have become more and more complex. They are designed as groups of interconnected but independent services. Each service exposes through standard invocation interfaces an elementary functionality (e.g., database maintaining customer information, web application serving search requests, etc). Responses delivered by the web application to the users are generated by composing the results of multiple services based on pre-defined workflows [22]. For instance, a page request to amazon.com, the e-commerce site, requires the rendering engine to construct the response by sending requests to over 150 services. Moreover, most of the services have multiple dependencies (and in turn invoke other services) such that the call graph of an application is complex, having multiple levels [5].

Not only do modern web applications consist of a big number of services and that the relations between these services are very complex, but also the interactions with the users are more complicated. Because of these aspects, major web applications have started posing more and more challenges in order to maintain their end-to-end performance within a predefined SLA when facing unpredictable workload or flash crowds (e.g., events such as the September 2001 terrorist attack in the US, when news sites such as www.cnn.com noticed a dramatic increase in the number of requests and many sites became unavailable).

Precise performance models and resource provisioning algorithms have to be designed for these web applications in order to be able to keep up with the demands and to achieve the imposed performance targets. Whenever the SLA targets are violated because of a flash crowd or a workload modification, applications should be able to reprovision their resources on-the-fly with minimum costs to offer good user experience. Because of the complexity of nowadays web applications, interesting challenges are arising when maintaining the performance, such that the problem of resource provisioning of web applications has captured the attention of many research groups.

The next chapter shortly presents the studies already conducted on resource provisioning and reviews the aspects that differentiate modern web applications enriched with AJAX techniques from the classical ones. Because of the new features added by AJAX, different aspects impact the server's performance and have to be considered when accomplishing the reprovisioning task.

2.1 Resource provisioning

The very first studies on resource provisioning considered simple web applications presenting a single-tier model [2, 8] or a multi-tier one [14, 20, 23, 28, 29]. These studies model either the most constrained tier of the web application [20] or they simplify the tiers' operation model [14]. [28] handle session-based workload and concurrency limits at different tiers. Moreover, their studies also capture the performance impacts of techniques such as caching and database replication. These models represented the starting point of our research as the application case study we have considered presents such a multi-tier architecture. However, our study extends their work as we do not address classical applications but applications enriched with AJAX that present more complex features and interactions (e.g., persistent connections and asynchronous requests).

However, some major web applications such as maps.google.com, amazon.com or ebay.com were not designed in a one-tier or multi-tier fashion but as complex groups of independent services querying each other [22]. As a consequence, further studies addressed the problem of resource provisioning of such multi-service web applications. Studies presented in [31] model the workflow patterns within multi-service applications in order to predict future workloads of each service component. The required number of servers per service can be derived. This model considers that each server has a fixed maximum capacity, approach basically equivalent to assigning an SLA to each service. As proved in [6], this can lead to resources wasting and over-provisioning.

Studies presented in [6] focus on multi-service web applications designed as directed acyclic graphs. They consider that only the front-end service should be assigned an SLA as only its end-to-end performance is observed by users. More, each service is autonomously responsible for its own provisioning by collaboratively negotiating its performance objectives with other services. Negotiation between services is based on "what-if analysis": each service estimates its performance in case it was assigned more or less resources, or if it received more or less traffic. The front-end service has the perspective of the whole application and is responsible for selecting the optimal service to be provisioned.

Our study on multi-tier AJAX applications can be extended for more complex multi-service applications as observations we make do not rely on the applica-

tion's model but on the new features added by AJAX at the front-end service's side. More, we also consider that only the front-end service should have a SLA associated, as its performance is the sole one observed by users.

All studies previously discussed and ours make the assumption that the underlying provisioning machines are homogeneous. This assumption states in our case as the experiments were conducted in medium scale environments (cluster computers). In the last years, however, cloud computing has become a very attractive platform to host web applications and in this situation, resources are heterogeneous and, thus, the homogeneous assumption does not state. Recent researches focus on the problem of provisioning web applications in heterogeneous resource environments. Research presented in [7] addresses the heterogeneity issue by efficiently benchmarking the performance profile of each individual virtual machine obtained from the cloud. For better performance, the request load is balanced according to these performance profiles.

Future work should focus on integration of our study with these studies addressing resource provisioning in heterogeneous environments as major web applications (that are enriched with AJAX techniques) are often hosted in cloud environments.

Our research extends previous studies on resource provisioning by focusing on modern web applications enriched with the AJAX technique. To the best of our knowledge, our study is the very first attempt to address these applications. The AJAX technique adds new features and more complex interactions between users and the web application. These characteristics have great impact on the web server's performance profile and, as a consequence, different aspects have to be considered when reprovisioning the server's resources.

The next section reviews the differences between classical web applications and modern ones and makes a brief introduction of the AJAX's impact on server's performance.

2.2 AJAX Web applications

The classic web application model uses a request/response/display workflow and supposes that most of the user actions in the interface trigger an HTTP request back to a web server. The server does the complete processing and then returns an entire HTML page to the client (Figure 2.1). This is a model adapted from the original use of the web as a hypertext medium and makes a lot of technical sense, but it doesn't make sense for a great user experience because of to the full-page transmission and synchronous communication's nature. The user has to wait while the request is sent, the server is doing its processing, and then the response is received.

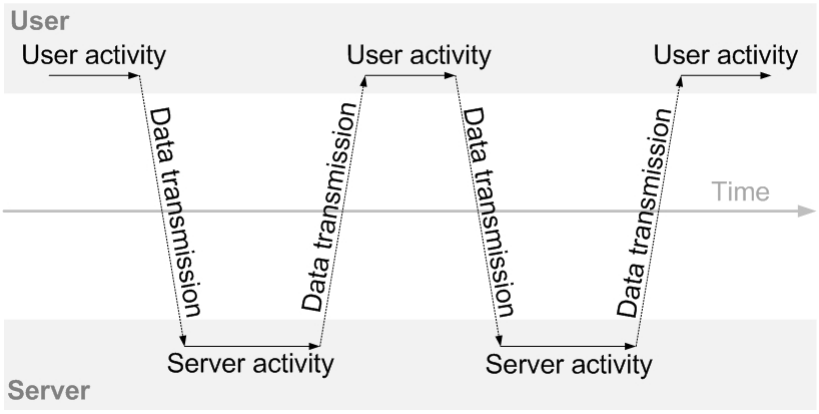

Figure 2.1: Classical web application model (synchronous interactions

Once the web page's interface is loaded, the user activity should not be halted every time the application needs something from the server. In fact, for a good experience, the user should not feel at all that the application is waiting for information from the web server.

2.2.1 Introduction

AJAX (an acronym for Asynchronous JavaScript and XML) is a group of interrelated web development methods used on the client-side to create interactive web applications. AJAX can be used to implement a web application that communicates with a server in the background, without interfering with the current state of the page. In [11], Garrett explained that AJAX incorporates the following technologies:

- HTML or XHTML and CSS for presentation;
- the Document Object Model (DOM) for dynamic display of and interaction with data;
- XML for the interchange of data and XSLT for its manipulation or JavaScript Object Notation (JSON), preformatted HTML or plain text as alternative formats;
- the XMLHttpRequest object for asynchronous communication;
- JavaScript (or another client-side scripting language) to bring these technologies together.

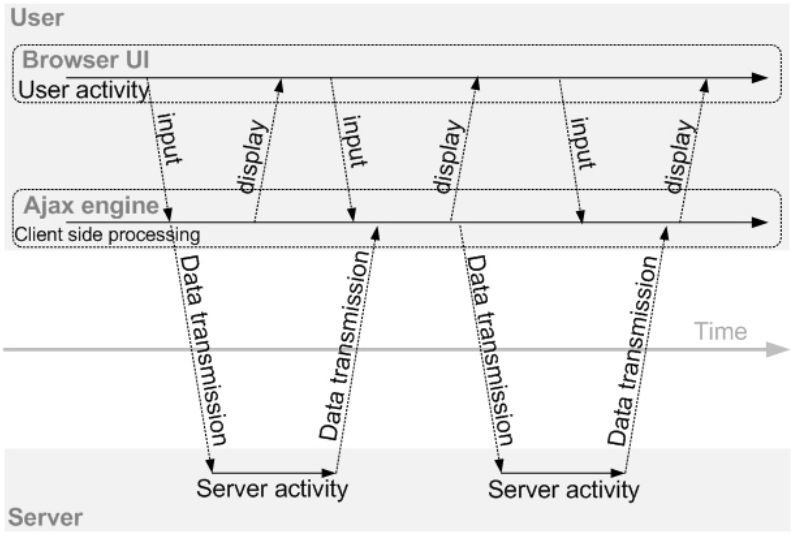

Figure 2.2: AJAX web application model (asynchronous interactions). [11]

An AJAX application moves some of the processing at the client side and eliminates the start-stop nature of interaction with the web server supposed by classical web applications by introducing an intermediary (namely, the AJAX engine) between the user and the server. Instead of loading a webpage, at the start of the session, the browser loads an AJAX engine that is responsible for both rendering the interface the user sees and communicating with the web server on the user's behalf. The AJAX engine allows the user's interaction with the application to happen asynchronously and most of the times independent of the communication with the server (Figure 2.2).

Every user action that would generate an HTTP request in the case of classical web applications, now takes the form of a JavaScript call to the AJAX engine instead. Because part of the application's processing is moved at the client side, any response to a user action that doesn't require a trip back to the web server (e.g. simple data validation, editing data in memory, and even some navigation) is handled by the AJAX engine itself. On the one hand, this highly improves the application's response time and on the other hand reduces the server's load. If the engine needs something from the server in order to respond (e.g. submitting data for processing, loading additional interface code, or retrieving new data) the engine makes those requests asynchronously, usually using XML, without stalling a user's interaction with the application.

The original HTTP protocol did not allow keep-alives, such that a new TCP connection was established between the users and the web server for each pair of request and response. This method led to inefficiency because of the high over-

head implied by frequent connections establishments and terminations and, as a consequence, the new version 1.1 of HTTP corrected this weakness and incorporated the concepts of keep-alives: a connection between the client and the web server is kept open indefinitely, or at least as long as the server permitted. Even if keep-alive is somehow against HTTP's original design goal of being "stateless", it allowed for it to overcome its speed and overhead problems and it is a feature intensively used by modern web applications. Anyway, because of the fact that interactions with the web server is handled by the AJAX engine, this technology intensively uses long-lived connections.

2.2.2 Impact of AJAX on server performance

A series of studies focus on analyzing the server performance of AJAX web applications. In [12], Smullen studies an web application following a three-tier model that supplies real-time class information extracted from a university student information system. Authors realize a comparative study on the performance of the HTML application and the AJAX application implementing the same functionality. Their results show that AJAX significantly reduces the bandwidth required for the client to receive the response than the traditional HTML application. A 56% reduction in the bytes making up a response was observed. Moreover, they indicate a reduction in time that the server spends generating a query (16%) in the case of the AJAX application. However, their study takes into consideration only synchronous calls, neglecting one of the great AJAX features.

[21] presents the study conducted on the performance gain offered by AJAX in the case of Hyves (http://www.hyves.nl), The Netherlands' largest social networking website having over 2 million unique users and 10 million page views per day. When using AJAX, their studies registered a reduction with 41.3% from the perspective of the network traffic. Moreover, the processing time on the server side in the AJAX scenario represents about one third of the time required by classical processing.

Studies analyzing the impact of maintaining persistent connections between users and the web server reveal first of all that this technique reduces the overhead implied by frequently opening of TCP connections as at least 3 packets have to be initiated (for SYN, SYN-ACK and ACK). In fact, the server CPU requirements are reduced if an average TCP connection carries at least one successful HTTP transaction. The time spent actually processing the request won't change, but the time spent opening and closing TCP connections and launching new threads or processes to handle them would be reduced [4].

One server can maintain a restricted number of active TCP connections and associated HTTP server threads or processes. Also, both open connections (including ESTABLISHED, CLOSING, etc. states) and TIME_WAIT connections

require some protocol control block (PCB) table space [16]. A busy server could end up with its tables full of connections in this TIME_WAIT state, or having no room for new connections or imposing excessive connection table management costs. When a server runs out of TCP connections descriptors or out of processes or threads for managing individual connections, it has to close idle TCP connections.

However, even if the impact on server's performance implied by asynchronous AJAX and persistent connections were intensively studied, currently, to the best of our knowledge, there is no study addressing the design of predictive models of the server performance that could be used by a resource provisioning system to determine on-the-fly the optimum requirement of resources.

Chapter 3

Methodology

This chapter presents the methodology followed by our research on dynamic resource provisioning for AJAX web applications. We have chosen as proof of concept a live chatting web application exposing a rich AJAX functionality to users, that is representative for its class of asynchronous AJAX web applications both from the perspective of types of interactions and from the perspective of AJAX engine's behavior type. On the one hand, persistent connections are maintained over long periods of time between users and the web server. On the other hand, asynchronous request are made by the AJAX engine on the user's behalf during the interaction with the web server, generating an interesting interaction, similar to the one of major applications successful nowadays. Moreover, following the trend of nowadays applications, our study is fundamentally concerned with web applications serving predominantly dynamic content.

The considered AJAX web application mimics one of the functionalities of eBay.com [22], a very popular online auction and shopping site, currently ranked 22th among the most visited web sites on the Internet and presenting a three hours average connections' duration [9]. The behavior of AJAX engines in the cases of both applications are very similar, the sole difference consisting on the significance of the information exchanged with the server: information regarding bids when referring to eBay.com or short text messages when referring to the application considered by our research.

Studies presented in [3] emphasize the importance of realistic benchmarking of modern web applications enriched with AJAX techniques in order to be capable to fully capture their interactions' complexity. Our experiments precisely mimic the behavior of real users and the behavior of the AJAX engine interacting with the web application. However, experiments are conducted on a LAN so they do not capture real latencies perceived by users that are normally geographically distributed on the Internet. Moreover, the resources provisioning the application are homogeneous as they belong to a grid environment. Future work could extend

our research by addressing this aspect.

First, we analyze the differences and impact on the server's performance implied by the AJAX's specific characteristics. To accomplish this task, we conducted a comparative study on the performance of two applications, both of them exposing the same live chatting functionality from users' perspective. One of them uses the AJAX technology and persistent TCP connections, while the second one lacks the AJAX features.

The comparative study revealed the fact that, under similar configurations and workload, the performance profiles of the servers serving the two applications are completely different. This result leads us to the conclusion that, because of the new features added by the AJAX technology and because of the use of long-lived persistent connections, different aspects heavily impact the server's performance profile and have to be considered when reprovisioning AJAX applications. As a consequence, AJAX applications heavily differentiate from classical web applications and thus, require separate provisioning techniques.

As AJAX applications expose a new performance profile, the next step of our research consisted of designing of the web application's performance model. The performance model is a mathematical model capable to predict what the web server's performance would be when more or less resources would be assigned or if the workload would increase or decrease. The performance model is heavily influenced by the features added by the AJAX technology and, as a consequence, the web server has a completely different performance profile than the one in the case of a classical web application.

The performance model is used to accomplish the resource provisioning task. Each service can to continuously estimate its performance based on the "what-if" analysis and to collaboratively negotiate its performance objectives with other services [6]. The front end service has the perspective of the whole application and whenever the SLA is no more achieved selects for re-provisioning the services that bring the best global benefit. Similar action is taken when the application is over-provisioned compared to the current workload - the resource that incurs the lowest global performance is removed.

This chapter continues with the description of the functionality, characteristics and features of the AJAX web application considered as proof of concept and then discusses the significance of the performance metric chosen in our research. The chapter ends by presenting the experimental environment and setup used in all evaluations.

3.1 Application Benchmark

We have chosen to analyze the characteristics of AJAX applications by considering as proof of concept a web application exposing a rich AJAX functionality to users (asynchronous requests and long-lived persistent connections). More, it serves to users highly dynamic content. Research was conducted on the Wordpress web application [18], the largest self-hosted blogging tool in the world, used on millions of sites [25]. Another reason for this choice is that Wordpress is an open-source project and thus, we have access to all its internals, a mandatory requirement of our research.

As AJAX features are used only in some administrative screens of the Wordpress web site, the basic functionality was enriched with Pierre's Wordspew plugin [30] that creates a live shoutbox, using AJAX as a backend. The shoutbox adds basic live chatting functionality to the web application and permits a real time interaction between users through short text messages (posts). Currently, the number of users who have enriched their Wordpress web pages with this plugin is greater than 33,000, placing the plugin among the most popular Wordpress plugins.

3.1.1 Functionality

As any live chatting application, the plugin's functionalities imply on the one hand the reading of posts created by other users and on the other hand the creation of new posts. Both types of interactions are intermediated by the AJAX engine placed at the client side.

The reading of new posts is implemented as a HTTP GET request made by the AJAX engine with fixed frequency (Figure 3.1, unbolded lines) controlled by T. T is a configurable parameter that influences on the one hand the application's degree of interactivity and on the other hand, the server's load. A small value of T determines very frequent requests, and thus, a high rate of updates received by clients. However, it would heavily increase the workload on the server side. On the other hand, a big value of T reduced the workload registered at the server side but also reduces the application's interactivity as updates are more rarely received. In all experiments, to provide a highly interactive application, we considered $T = 0.5$ *seconds*.

A GET request specifies the identifier of the last message "known" by the current user to have been added to the live chat. The response sent by the web server contains the lists of all timestamp-identifier-message tuples newer than the identifier specified by the request. GET requests are made regardless of the user's actions and thus, this type of interaction generates only asynchronous communication with the web server.

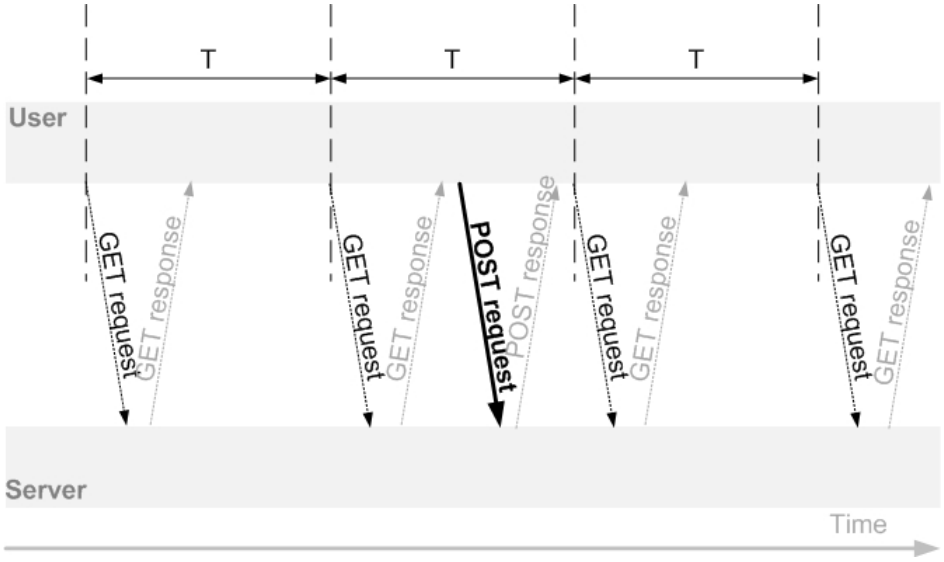

Figure 3.1: The asynchronous interactions model of the web application. Exemplification of the two interaction types.

To complete the chat functionality, users have to be able to add new posts. The adding of new posts is implemented as a HTTP POST request that describes the content of the text message that will be posted to the live chat. It implies a synchronous interaction because it requires the user's will to take action (Figure 3.1, bolded line).

This live chatting application's behavior is very similar to the one of eBay.com in the context of an online auction. In the last minutes of the auction, the AJAX engine keeps the user up-to-date with the new bids placed by others and also permits the placing of a better bid. In the case of both applications, the AJAX engines make requests with fixed frequency to the web server for new updates. Moreover, at the user's will, the AJAX engines add new information (i.e. text messages in the context of the Wordpress chatting application or new bids on eBay.com). These similarities prove that the functionality exposed by the application considered in our research as proof of concept is reach and representative for the class of modern web applications that are popular nowadays.

As previously discussed, it is important to ensure a realistic benchmarking of web applications generally speaking and thus, of AJAX applications in the context of our research. Normally, the client of any web application would be a web browser that retrieves and renders the content. However, in our research we re-implemented in Java the client side as a standalone component. This component mimics the behavior of the AJAX engine that would be normally placed on the client side to intermediate the users' communication with the web

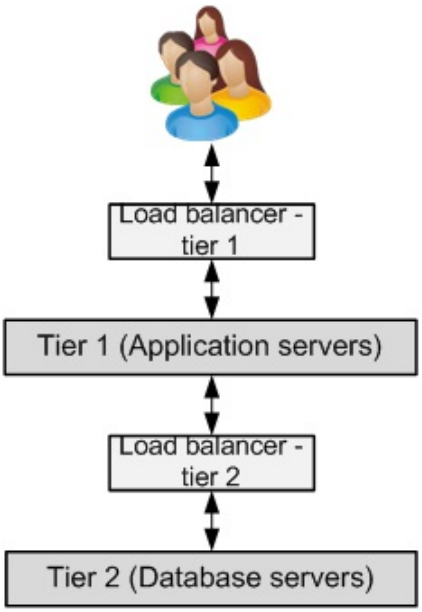

Figure 3.2: The application's two tier model

server. In both simulated and real environment, GET requests are issued with fixed frequency and no user action is required. On the other hand, POST requests require user action and the simulations we considered generate these requests with uniform distribution. However, there is no need to simulate advanced features such as user's typing speed as they have no big impact on the workload registered at the web application server side.

In the case of AJAX applications, the first interaction between the user and the web server normally implies the transmission of all scripts that are mandatory for the AJAX engine to work at the client side. Moreover, this interaction has a synchronous character and happens at most once as these scripts could be already cached from a previous session or could be served by a separate entity such as CDNs. However, the first interaction has no big relevance for our research such that we neglect it and consider strictly the AJAX specific interactions supposed by the web application.

3.1.2 Web Server's Architecture

The Wordpress live chatting application is served by an Apache web server and was designed in a classical multi-tier fashion (Figure 3.2), being composed by:

- the application tier (where the application logic is implemented);
- the database tier (where the persistent state of the chat is stored).

The application tier receives HTTP (GET and POST) requests issued by the AJAX engine placed on the client side. A GET request issues one query to the database required to search for posts having an identifier newer than the one specified by the request. On the other hand, a POST request issues two queries to the database; one query is required to actually save the new post message into the database and another one is required for archiving it.

As a consequence of the fact that the experimental environment (described in Section 3.2) is homogeneous and thus, resources have the same processing capabilities, the load balancers equally distribute the workload between the resources provisioning the application tier, and respectively resources provisioning the database tier.

3.1.3 Performance metric

To illustrate the way multiple users interact through this web application, we will analyze the separate interactions between the web server and two users that are connected during overlapping periods of time (Figure 3.3). On behalf of each one of them, the AJAX engines placed on their machines, will make GET requests with the same fixed frequency influenced by the T parameter.

Considering that, for instance, on the time axis, at the moment of time $T1$, $User$ 1 decides to post a new text message to the chat, this new message will be visible by $User$ 2 at the moment of time $T4$. In fact, $T4$ marks the receiving of the response corresponding to the very first GET request made by the AJAX engine on the $User$ 2's behalf after the effects of the POST request performed by $User$ 1 are committed as persistent state by the database layer.

The delay between times $T1$ and $T4$ is denoted by T_{int} that expresses the degree of interactivity between users offered by the web application. A reduced value for T_{int} denotes a high degree of interactivity and implies small update delays. This is the desired behavior for any interactive application. T_{int} will be considered the performance metric of the web application considered as proof of concept.

As expected, T_{int}, the degree of interactivity, depends on the user perceived response times of both GET and POST requests, denoted by R_{GET}, and respectively by R_{POST}. Moreover, it depends on the "phase shift" (PS) between the two events - the time elapsed between the moment when $User$ 1 posts his message and the moment of time the AJAX engine triggers the very first request for new updates (Figure 3.3):

$$T_{int} = f(R_{POST}, R_{GET}, PS)$$

$$PS = T3 - T1$$

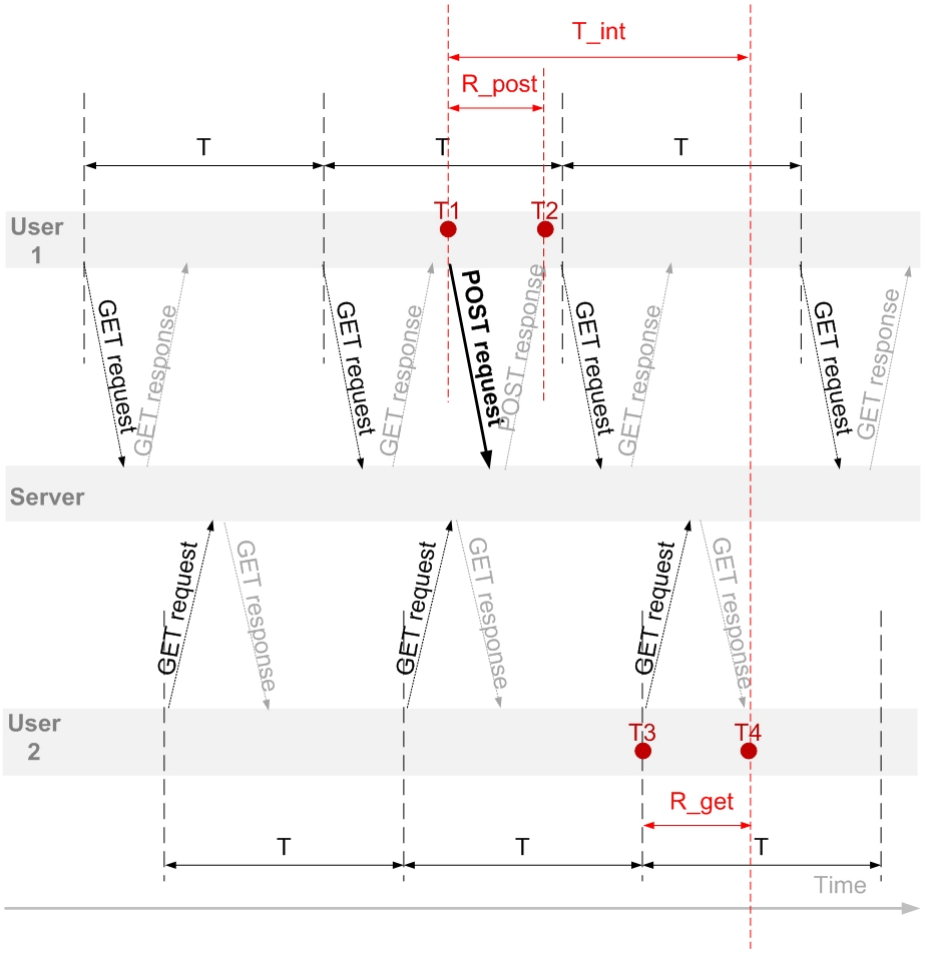

Figure 3.3: Inter-users interactions through the web application. Performance metric exemplification: the update made by $User1$ at $T1$ moment of time is received by $User2$ at $T4$.

However, the value of PS varies between any two users and is directly influenced by T. In the worst case $PS \rightarrow 0.5$ *seconds* as two GET requests are generated each second and a GET request on the behalf of $User$ 2 is received just before the POST request of $User$ 1, such that only the very next GET requests will catch the update. Anyway, the value of PS is not influenced by the current performance of the web server and as a consequence, the application's performance metric neglects the PS term.

Furthermore, a good user experience in the case of such web applications assuming live interactions supposes that updates have to be quickly seen by the other participants. The amount of time required by an update to be announced to the other users characterizes the performance of our web application and it is

Parameter	Value	Meaning
KeepAlive	On	Enables HTTP persistent connections.
KeepAliveTimeout	2	The number of seconds Apache will wait for a subsequent request before closing the connection.
MaxKeepAliveRequests	100	Number of requests allowed on a persistent connection.
MaxClients	150	Maximum number of connections that will be processed simultaneously.
MaxAliveTimeout	2	Amount of time the server will wait for subsequent requests on a persistent connection.
MaxSpareServers	150	Maximum number of idle child server processes.

Table 3.1: Web server's configuration of the keep-alive parameters

considered to have the following form:

$$T_{int} = R_{POST} + R_{GET} \tag{3.1}$$

To offer good user experience and real-time interactivity between users, we consider in all evaluations that the performance target to be $T_{int} \leq 0.3$ *seconds*.

The manner we set the application's performance metric can be extended also for the case of other AJAX web applications that generate real time interactions between users. Considering for instance the example of eBay.com [22], our performance metric expresses in fact the time necessary for a new bid placing for a specific product to be visible by another users who are interested in the same item. The lower this duration is, the better the interactivity and the improved the user's experience are as users can react to others' actions.

3.2 Experimental Setup

The web application considered as case study is Wordpress version 3.0.4, enriched with Pierre's Wordspew plugin version 6.1. The application layer was implemented using PHP version 5.3.5 and Apache HTTP server 2.2.17 (configured as specified in Table 3.1) while the database layer is implemented using MySQL version 5.0.92. According to W3 Techs [27], Wordpress is used by over 14% of the 1,000,000 biggest websites. Apache was chosen instead of a event-based web server because it has been, since 1996, the most popular HTTP server on the

World Wide Web, serving in 2011 over 63% of all websites and over 66% of the million busiest, according to Netcraft [26].

Experimental testing was conducted in a homogeneous environment: DAS-4 (The Distributed ASCI Supercomputer 4) [1], a six-cluster wide-area distributed system designed by the Advanced School for Computing and Imaging. The VU cluster of DAS-4 includes 74 dual-quad-core compute nodes having the following configuration: 2.4 GHz and 24 GB memory, interconnected by an InfiniBand network. The operating system the DAS-4 runs is CentOS Linux.

Chapter 4

AJAX impact on server performance

To illustrate the impact that AJAX features have on server's performance and to better emphasize the difference between these applications' behavior and the one of classical web applications, we conducted experiments on the two applications exposing the same application-specific functionality to the users (as described in the previous chapter) but one of them using AJAX features and persistent connections, in contrast to the second one.

4.1 Comparative study of AJAX and non-AJAX scenarios

Basically, the main difference between the two scenarios is represented by the way TCP connections between the users and the web server are handled. These connections are maintained persistent in the case of the AJAX web application in contrast to the other scenario. As a consequence, as Figure 4.1(a) describes, in the case of non-AJAX behavior, each application-specific request made to the server implies that a new HTTP connection is established (and terminated afterwards). As a consequence, a TCP connection is used only for one pair of request/response and never reused. In contrast, the AJAX behavior (Figure 4.1(b)) implies that TCP connections between users and the web server are kept persistent over long periods of time and are reused by multiple request/response pairs.

Figure 4.1 presents a simplified representation of the states a TCP connection changes during its lifetime. As only two of these states have big relevance for our study, only they were represented:

- **Established state** indicates that the connection is ready to send and re-

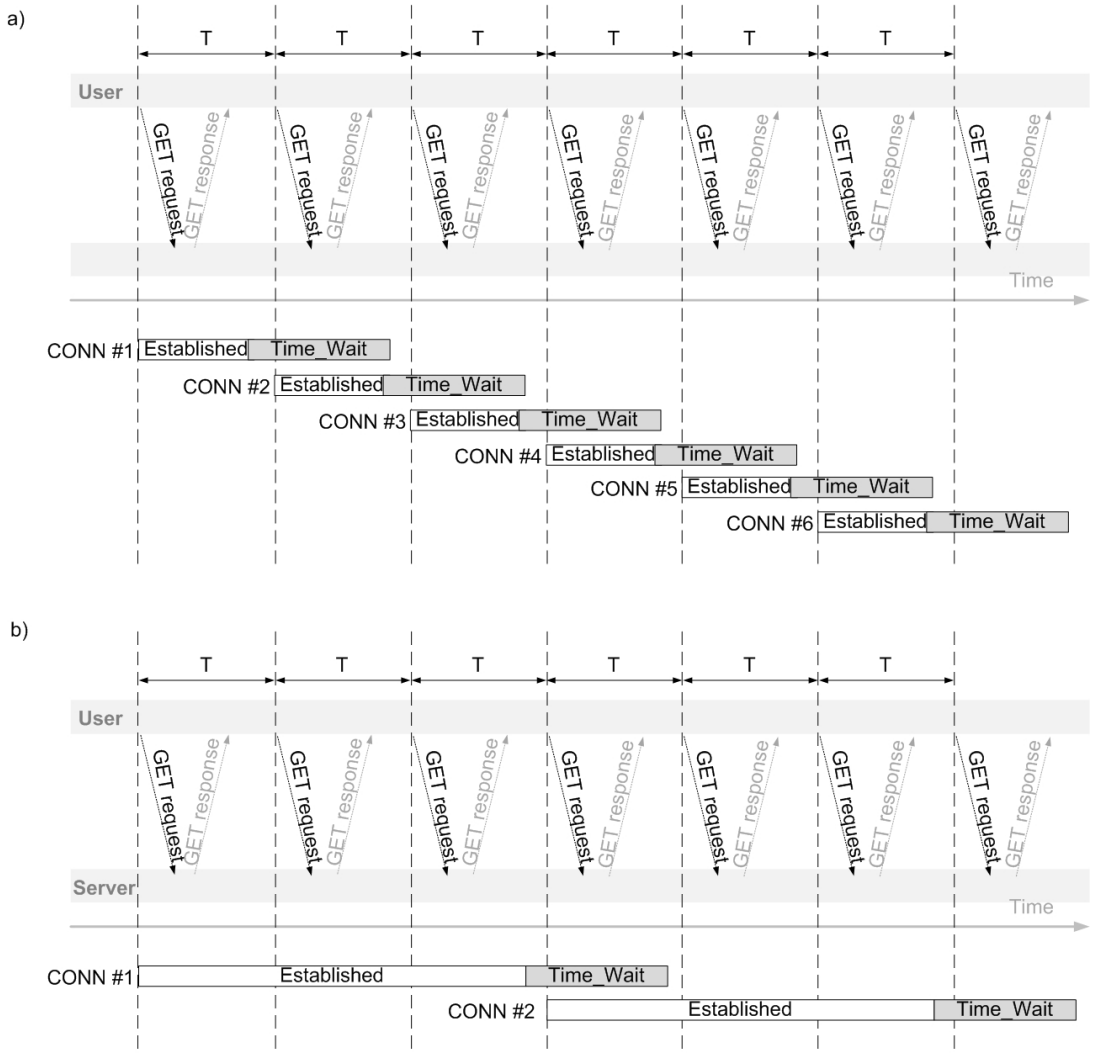

Figure 4.1: TCP connections handling. a) The non-AJAX scenario implies non-persistent connections. b) The AJAX scenario supposes persistent connections.

Table 4.1: TCP connection lifetimes

	AJAX scenario	non-AJAX scenario
ESTABLISHED state	50 s	100 ms
TIME_WAIT state	60 s	60 s

ceive data between the user and the web server. The time a connection is kept in this state depends on the connection's character (persistent or non-persistent). In the AJAX scenario, up to 100 requests (and corresponding responses) are sent reusing the same TPC connection. As a consequence, the lifetime of the active connection is long (up to 50 seconds, Table 4.1), in contrast to the lifetime of an active connection in the non-AJAX scenario that lasts for few hundreds of milliseconds.

- **Time Wait state** represents waiting for enough time to pass to be sure the remote peer received the acknowledgment of the connection termination request. According to RFC 793 [24] a connection can stay in this state for a maximum of four minutes known as a MSL (maximum segment lifetime). However, in both scenarios, the lifetimes of the idle connection are comparable (up to 60 seconds).

4.1.1 Experimental Setup

To perform this comparative study, we considered for both applications the same setup and the same experimental environment, as described in Section 3.2, except for the $MaxClients$ parameter that was set to 200. This modification was done with the purpose to better emphasize important performance aspects that differ from one scenario to the other. Moreover, in the case of both scenarios, each of the two tiers of the web application has assigned only one resource.

The workload is generated by 200 users. On the behalf of each one of them, the AJAX engine generates 2 GET requests/second, resulting thus, a request rate of 400 requests/second. POST requests are generated with an uniform distribution.

Both applications performed well and had no problems to complete all requests without violating the imposed SLA. (As specified in Section 3.1.3, the performance target is imposes on the interactivity time that should satisfy the condition $T_{int} \leq 0.3\ seconds$).

4.1.2 Differences regarding the number of TCP connections

The distinction from the perspective of the TCP connection handling between the AJAX and non-AJAX scenarios is reflected first of all by big differences in terms of the number of concurrent TCP connections in the ESTABLISHED and TIME_WAIT state registered at server side. Figure 4.2 presents the cumulative distribution function of the number of concurrent established TCP connections encountered in both scenarios.

As expected, because TCP connections are kept persistent over longer periods of time and they are established and terminated less frequently in the AJAX scenario, the number of concurrent open connections is much higher than in the non-AJAX scenario. As a further consequence of the long persistency, the number of open connections does not vary too much over time: more than 80% of the time more than 80% of the users are connected (the number of connections is around 164, with a standard deviation equal to 13).

In contrast, as the non-AJAX scenario supposes short-lived TCP connections, the total number has large variations and uniformly covers a wider range. In this case, the average number of concurent active connections is around 47.7 (more than 3 times less compared to the AJAX scenario) while the standard deviation of values is 32.4 (almost 3 times more compared to the AJAX scenario).

Figure 4.3 presents the cumulative distribution function of the number of idle TCP connections for the same configuration and setup. In the non-AJAX case, the web server handles about 400 requests/second and each of these requests is establishing a new TCP connection. As a consequence, the web server handles in fact the establishment of 400 connections/second. As they are short-lived connections, the number of concurrent idle connections has a uniform distribution over a wide range.

On the other hand, as AJAX scenario maintains long-lived connections and one connection carries up to 100 requests, in this case the number of idle TCP connections is drastically reduced (by about 100 times) (Figure 4.3). Basically, the lifetime of a persistent connection is about 50 seconds and the approximate duration of the TIME_WAIT state is 60 seconds. Due to the ratio between these two values, in the AJAX case, the number of idle connections is just slightly greater than the number of active connections and its cumulative distribution has a similar profile.

In conclusion, because of the long-lived connections, from the perspective of concurrent active and idle TCP connections, the web server's state is more stable and easier to predict in the AJAX scenario.

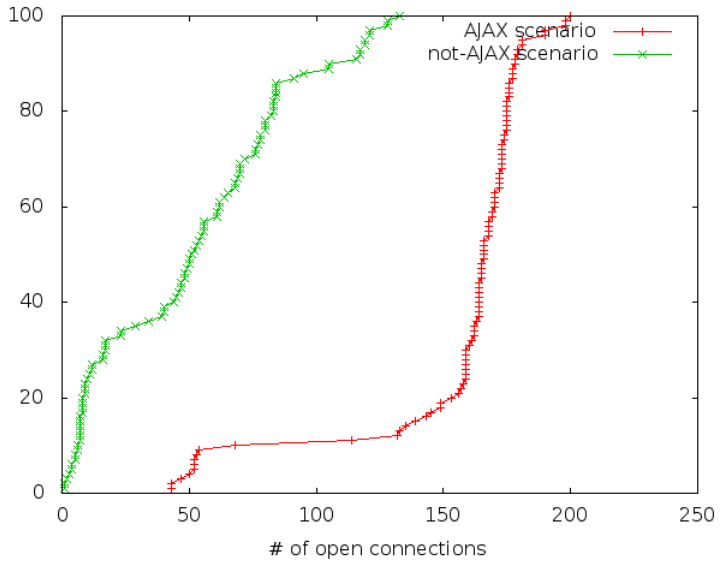

Figure 4.2: Cumulative distribution function of the number of ESTABLISHED connections for the AJAX and non-AJAX scenarios.

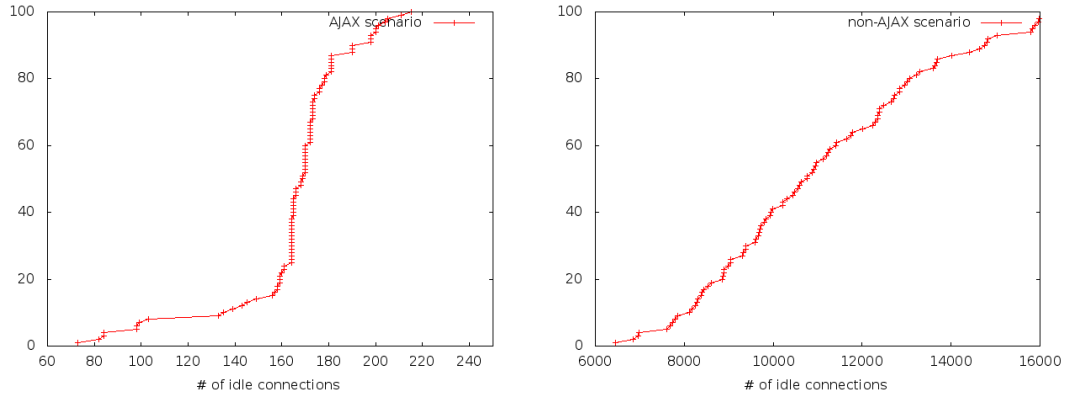

Figure 4.3: Cumulative distribution function of the number of connections in TIME_WAIT state for the AJAX and non-AJAX scenarios.

4.1.3 Differences regarding the network traffic

Maintaining short-lived TCP connections in the non-AJAX scenario comes with an overhead to be paid at the server side, as any application specific request made by an user will determine a new connection to be established (and terminated afterwards). Each TCP connection establishment requires an overhead of at least 3 packets to be initiated (i.e. SYN, SYN-ACK, and ACK packets). Moreover, each connection termination requires two pairs of FIN and FIN ACK packets.

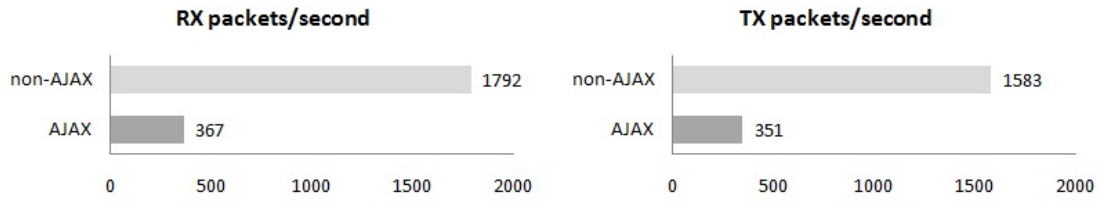

Figure 4.4: Average number of received and transmitted packets/second in the AJAX and non-AJAX scenarios

As both scenarios expose the same functionality to users, the number of packets/second required to transmit the application-specific data is similar. The sole difference consists of the number of packets required for TCP connections establishment and termination.

In the non-AJAX scenario, the web server receives on each connection an average of five packets: one SYN, one ACK, one application-specific request, one FIN and one FIN ACK. On the other hand, the web server sends on each connection an average of four packets: one SYN ACK, one application-specific response, one FIN and one FIN ACK. In contrast to this, in the AJAX scenario, the web server receives and sends on each connection not one, but up to 100 application-specific requests, respectively responses.

Figure 4.4 compares the number of received and transmitted packets/second registered at the server side for the two scenarios, considering the discussed setup and workload. Maintaining persistent TCP connections reduces the number of received and transmitted packets/second with up to 80%. This result is generated by the fact that, in the AJAX scenario, users send around 2.08 packets/second while in the non-AJAX scenario, they send around 10 packets/second to accomplish the same application-specific functionality.

As both GET response and POST request packages carry the messages added to the chat and the size of these messages present big variations from one experiment to another, we avoid to compare the network traffic in terms of number of received and transmitted bytes. The statistic of the number of received and transmitted packets is, however, very suggestive and emphasizes the big behavioral difference between the two scenarios.

4.1.4 Differences regarding the memory usage

As discussed earlier, the sole difference between the two scenarios consists of the way the TCP connections with users are handled by the web server. As seen in Section 4.1.2, the number of active and idle connections in the AJAX scenario is several hundred times less than in the non-AJAX scenario. This big difference is

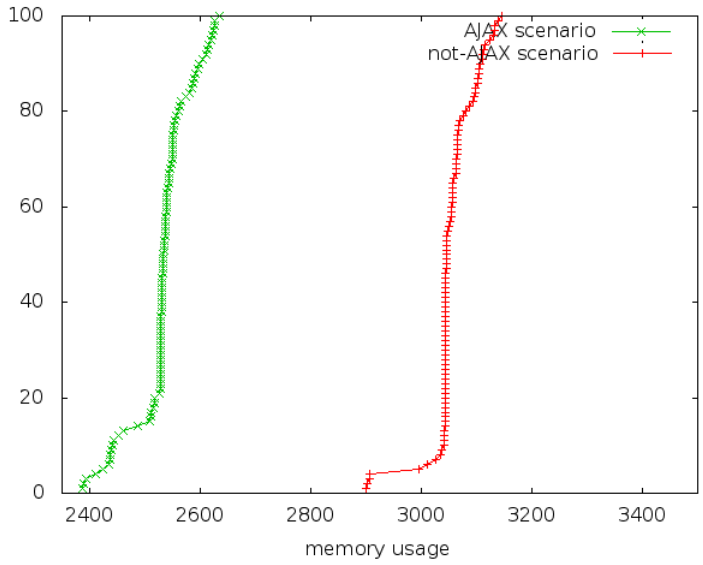

Figure 4.5: Cumulative distribution functions of the memory usage for the AJAX and non-AJAX scenarios.

largely caused by the huge number of idle connections. However, the operating system permits, if needed, to close an idle connection at any moment of time because a certain duration of the TIME_WAIT state is an advice of the protocol designers for safety but it is not mandatory. Anyway, a TCP connection, either being active or idle implies some maintenance costs because of the following resources[4]:

- the socket file descriptor and the kernel data structures corresponding to it;
- the TCP connection tuple (consisting of protocol, source:port, destination:port) and its data structures;
- the socket send and receive buffers.

As a consequence of the big difference between the number of managed TCP connections, the amount of memory used in the non-AJAX scenario is bigger. Reducing the lifetime of an idle connection would give advantage, however, to the non-AJAX scenario.

4.1.5 Differences regarding the CPU usage

As the AJAX scenario implies less varying number of open and idle TCP connections and reduced network traffic, the CPU usage presents small variations:

less than 30% during the entire experiment. On the other hand, the non-AJAX behavior implies additional overhead because of extra establishments and terminations of TCP connections. As a consequence, in this case the CPU usage increased significantly to values just below 100%. However, this big difference is determined not only by frequent TCP connection establishments and terminations, but also by more demanding searches in TCP connections table. This happens because TCP packets do not include a session identifier, but both endpoints identify the session using the client's address and port. Whenever a packet is received, the TCP implementation must perform a lookup on this table to find the destination process. As discussed in Section 4.1.2, in the non-AJAX scenario, the table contains much more entries, thus requires more CPU usage.

4.1.6 Differences regarding the user perceived performance

The non-AJAX scenario supposes that each application-specific request issued by users requires one TCP connection establishment to the web server. As a consequence, the response time of the each GET and POST request incorporates also the duration of the connection establishment. In contrast, in the AJAX scenario, only about 1% of the requests present a response time that incorporates this connection establishment delay. However, the application-specific processing time is similar for both scenarios because the current request rate is below the maximum throughput of the server. As an expected consequence of these facts, the average response time in the non-AJAX scenario is up to 11.2% greater compared to the other scenario. The difference is caused by the fact that the application specific processing time in both cases is comparable to the TCP establishment time (around 20 milliseconds, respectively 2-3 milliseconds). This stands, however, because no network delays are measured as experiments are conducted in a LAN.

4.1.7 Conclusions

This section illustrated the impact of AJAX features on the web server's performance in the context of our proof of concept application. However, these observations can be extended for other applications and with small differences (caused by application specific characteristics) these observations stand for them too.

As seen, persistent connections improve the CPU usage, the network traffic and the user perceived performance but in the meantime negatively impact the physical memory usage which should be the very first concern of the administrators of such web applications. This is reflected by the $MaxClients$ parameter that limits the number of (persistent) connections that can be concurrently served and

thus, in our case, the number of users than can be handled concurrently without violating the SLA.

However, all observations we made in this section consider a fixed configuration and setup. Varying the Apache parameters (e.g. $MaxKeepAliveRequests$, $MaxRequestsPerChild$, etc) would lead obviously to different results.

In the scenario of modern AJAX application handling very frequent requests (such as ours or eBay.com), modifying the $MaxRequestsPerChild$ parameter would have the greatest impact on the server's performance. For instance, reducing the value of the parameter would reduce the lifetime of the active connection but would increase the network traffic and the CPU usage as more packets would be sent to establish and terminate TCP connections. The performance would be somewhere between the ones discussed in this section. On the other hand, increasing the value of the $MaxRequestsPerChild$ parameter would increase even more the connection lifetime, but could lead at some point to memory leaks. More, the task of de-provisioning would be harden as it should firstly "wait" for all (persistent) connections handled by an instance to consume their lifetime and to be closed and only after that to release the resource.

However, there is no perfect solution and major Apache-based web sites have different keep-alive configurations. Some of them, such as Yahoo! do not permit persistent connections at all while BBC has a very short keep-alive timeout (i.e. less than 5 seconds). On the other hand, there are other sites such as Apple and CNET that use a large keep-alive timeout [19].

4.2 The choice of web server software

As previously discussed in this chapter, the maintaining of keep-alive connections has a great impact on the web server's performance. However, the keep-alive concept was been implemented differently by different HTTP servers, and from this regard, there are two main directions in which they have evolved: process-based servers (e.g. Apache) and event-based servers (e.g. nginx, lighttp). The difference between them is made by the way they map the received requests on the working processes.

4.2.1 Process-based Web servers

The best known example of process-based web server is represented by Apache [10], the web server used also in our experiments. When Apache is started with pre-fork MPM (which is required by mod_php), a main "coordinator" process is created, that is responsible for accepting incoming connections and passing them to "worker" processes previously created. These workers read users' requests and

send back responses. When a worker is done servicing a user's requests, it reports to the main process and waits for a new connection to be handed to it.

In a perfect world, we would set to infinity the lifetime of a connection such that clients would maintain a connection to the web server for as long as possible to remove the overhead of TCP connection establishment and termination. Unfortunately, each client connection requires Apache to use a worker process to serve the requests associated with. A worker process can only handle one connection at a time, and each connection is persistent over a long period of time. Thus, Apache will create a new worker process for each new connection until it reaches its limit of $MaxClients$.

Let's consider as example what happens when 1000 web clients concurrently access a web site, considering that the web site was $MaxClients = 150$, the default value for Apache. The first 150 clients will successfully connect, because Apache will create workers to service their requests. However, as these clients maintain persistent connections over long periods of time, they do not immediately leave so the next 850 clients will be unable to access the web server, as all Apache worker processes are already assigned. Those 850 clients will queue and wait for an Apache process to become available and to serve their request, but when the maximum wait time is reached, they will give up by dropping the connection and leaving the web application after a bad user experience.

Just increasing the $MaxClients$ setting to something high enough to handle the flash crowd won't work because each Apache process consumes memory and thus, $MaxClients$ is limited by the memory dedicated to the web server:

$$MaxClients = \frac{TotalRAMdedicated}{MaximumWorkerProcessSize} \tag{4.1}$$

Setting up a higher number of worker processes than this limit will determine the web server to begin exaggeratedly thrashing and swapping between RAM and the hard drive in a futile attempt to make it work. Unfortunately, in the end, this leads to a totally unresponsive server and a bad user experience. There are two solutions to fix the problem: by adding many gigabytes of RAM to the server or by reducing the duration of persistent connections. The latter solution is out of the question in this research.

4.2.2 Event-based Web servers

Event-based web servers are trying to avoid the large memory footprint that process-based servers such as Apache have. The best known examples are Nginx (http://nginx.org/) and Lighttpd (http://www.lighttpd.net/). Chris Lea said: "Apache is like Microsoft Word, it has a million options but you only need

six. Nginx does those six things, and it does five of them 50 times faster than Apache." [17].

Studies reveal that Nginx can handle much more traffic as compared to Apache on same hosting plan. Hosting Wordpress [17], the nginx setup presented the same CPU and memory usage but hosted four times more sites and servers five times more traffic without any downtime or without getting slow.

The performance improvement is offered by the way nginx manages the requests received from users. Similarly to process-based web servers, in nginx, the worker process that is selected to serve a keep-alive HTTP connection will continue to serve all subsequent HTTP requests over that same connection. However, in contrast to process-based web servers, nginx worker processes handle requests "on-demand" in an asynchronous manner.

Nginx workers do not fork or create new threads upon receipt of a new connection, they just add the new connection to their connection set, and go back to their main select/poll loop. When HTTP requests are received, the nginx worker process it and then go back to their select() loops. Since each worker handles multiple connections to backend, the only issue is to find the right number of worker processes to use for the nginx instance.

Web application administrators and future studies on resource provisioning should take into consideration such event-based servers as the web server performance profile could be different than the one when setting up a process-based web server.

Chapter 5

Performance model

To optimally accomplish the resource provisioning of an AJAX application, a good performance model is required. It represents a mathematical model capable to predict the future behavior of each individual service composing the web application if more or less resources would be allocated and if the workload would increase or decrease. This chapter designs the performance model of the AJAX application considered as a proof of concept in our research.

The web application is hosted and configured according to the setup specified in Section 3.2. As previously discussed, the application was designed in a simple, multi-tier fashion. The application tier implements the logic of the live chatting system, while the database tier stores the persistent state of the system in a MySQL database. In all experiments we considered Apache as being the web server hosting the web application.

5.1 Application tier performance model

Starting with an initial configuration in which each tier of the AJAX live chatting web application is provisioned with only one resource, we analyzed the impact on the user perceived performance when increasing the number of users and, respectively, the workload registered at the server side. On the behalf of each user, the AJAX engine generates two GET requests/second. Each one of them further generates one read query to the database tier, such that the response time depends on the response time of both the application and the database tier:

$$R_{GET} = R_{AS} + R_{DB} \tag{5.1}$$

On the other hand, POST requests are generated with an uniform distribution and each one of them issues two insert queries to the database tier, such that:

$$R_{POST} = R_{AS} + 2 \cdot R_{DB} \tag{5.2}$$

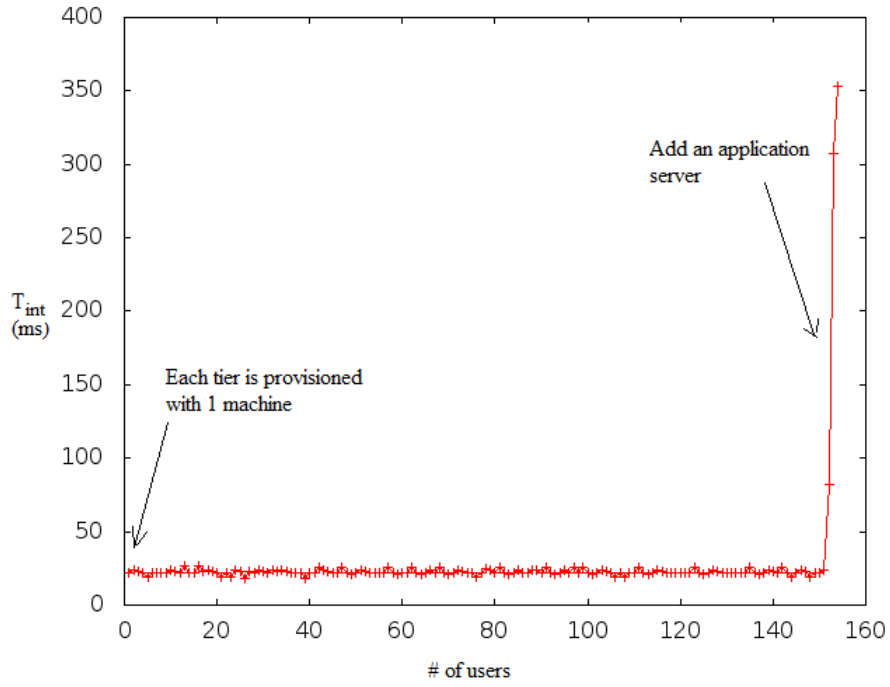

Figure 5.1: Response time of the AJAX application when increasing the number of users. Each tier is provisioned with one resource (1 AP, 1 DB).

Replacing formulas 5.1 and 5.2 into 3.1, we obtain:

$$T_{int} = ct_1 \cdot R_{AS} + ct_2 \cdot R_{DB} \tag{5.3}$$

where ct_1 and ct_2 are two constant values.

Moreover, the number of requests (L) handled by the web server can be approximated as linearly depending on the number of users (U): $L = ct \cdot U$.

Figure 5.1 presents the evolution of the user perceived response times provided by this setup of the AJAX web application when increasing the workload. The performance target set by the SLA specifies the restriction: $T_{int} \leq 0.3$ *seconds*. As observed, the SLA starts to be violated when the number of concurrent users exceeds 150.

To deeply understand the cause of this performance loss, we analyze the state of the most important system parameters that are describing the state of the machine that is provisioning the application tier of the chatting system. Table 5.1 specifies the values of these system parameters close to the crash moment of time, namely when the number of concurrent users that are accessing the application exceeds 150. It is easy to observe that, even if, under this workload, the performance target is no more achieved, the states of the system parameters reveal,

CPU usage	25%
memory usage	1.8 GB
RX packets/second	272
TX packets/second	259

Table 5.1: State of the system parameters when SLA is violated (1 AP, 1 DB)

however, that none of them reaches its maximum hardware limit and thus, none of them is the bottleneck generating the current performance loss.

The moment of time the system starts losing its performance has a special meaning as the web application is hosted by Apache. As discussed in Section 4.2, because Apache is a process-based web server, the number of threads (denoted by the $MaxClients$ parameter) that can be created by the web server is limited and depends on the amount of memory allocated to the server (according to formula 4.1). A thread is responsible to respond to *all* requests sent over the same TCP connection but can serve only one connection a time. Because of this one-to-one relationship between one TPC connection and one Apache thread, the $MaxClients$ parameter sets a limit on the number of active connections that can be concurrently served successfully by the machine provisioning the application tier. In this experiment, the web server's setup specified in Section 3.2 states the limit for the $MaxClients$ parameters to 150.

When the number of users concurrently accessing the web server is approaching this value, a sudden performance degradation is encountered. This big performance loss is caused by the fact that the number of users trying to connect to the web server exceeds the values of the $MaxClients$ parameter such that, for the new users that are trying to access the application, there is no available thread to handle their requests. As a consequence, these users are placed in a waiting queue until some Apache threads finish their current tasks and become available to handle them.

AJAX web applications offering support for real-time interaction between users, such as the Wordpress live chatting application, normally take maximum advantage of the keep-alive feature of the HTTP version 1.1 protocol and maintain very long-lived TCP connections. For instance, in the case of the chatting application, each TCP connection is used to carry up to 100 pairs of application-specific requests and their corresponding responses, such that the lifetime of these connections is long (i.e. up to 50 seconds). This aspect generates long waiting queues when the number of concurrent users exceeds the value specified by the $MaxClients$ parameter and leads to the situation when these users have to wait unacceptable amounts of time, and most of the times, they even start quitting and have a poor experience. In the worst case, one user is waiting in the queue

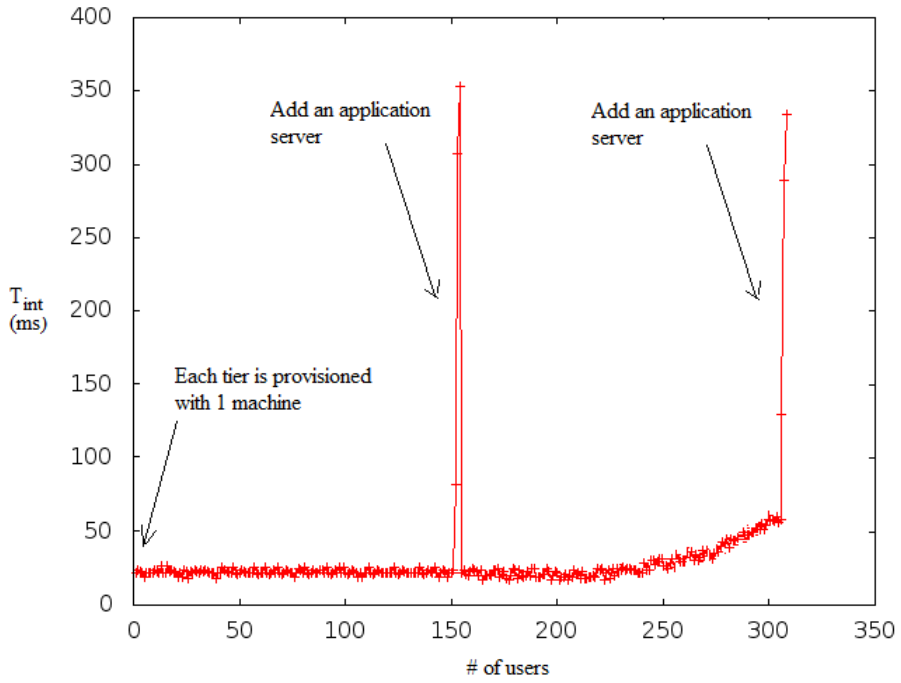

Figure 5.2: Response time of the AJAX application when increasing the number of users. The application tier is reprovisioned with one more resource (2 AP, 1 DB).

for an amount of time equal to the maximum lifetime of a persistent TCP connection. This is unacceptable as studies revealed that usually users leave the page if its response time is greater than few seconds.

Taking all these into consideration and also the observation that the machine that is provisioning the database tier is even less loaded, it is obvious that the application tier is the one that need to be reprovisioned with one more machine because of its impossibility to handle the current workload as there are no enough Apache threads to serve all incoming requests.

Figure 5.2 extends Figure 5.1 by presenting the user perceived performance after the application tier is reprovisioned with one more machine. We continue to increase the number of users, respectively the workload seen by the web servers. As all resources provisioning the web server are homogeneous (belonging to a grid environment) and they have the same configuration and setup, the load balancer will uniformly distribute the workload between the two machines provisioning the application tier. Each one of these machines can handle up to 150 concurrent active TCP connections.

Moreover, because of the nature of the analyzed application, there is no com-

munication or relationship between the two machines provisioning the application tier. Furthermore, none of them is aware of the other's existence, facts that heavily influence the application's performance profile as a whole.

Because the workload is uniformly distributed between the two machines, the workload generated by 150 users that initially overwhelmed the web server is successfully handled after the reprovisioning and thus, the user perceived performance returns to good values. This good performance is maintained until the number of users accessing the application exceeds 300. At this moment, each one of the two machines provisioning the application tier handles almost half of the connections and requests (i.e. around 150). In this situation, the performance loss is caused, as previously, by the same hardware limitation of the number of active connections that can be concurrently handled by each one of the instances. The current states of the system parameters for each one of the machines provisioning the applications' tiers are similar to the one presented in Table 5.1, proving that the system is overwhelmed again because of the impossibility of the application tier to handle more active TCP connections, such that a new machine should be assigned to it.

As a consequence, we reprovisioned the application tier with one more machine. After this reprovisioning, each one of the three instances will serve about one third of the concurrent active TCP connections (i.e. around 100) and their corresponding requests. Furthermore, the user perceived performance will return to good values.

In conclusion, because of the AJAX's particular characteristics and because all machines are homogeneous, the number of users that can be successfully served by the application tier linearly depends on the number of machines provisioning this tier.

$$Th_{AS}(n) = ct \cdot n \tag{5.4}$$

Bassically, whenever the number of users concurrently accessing the application is below the number of Apache threads the web server can create, the application tier offers good response times. Otherwise, because of the long-lived persistent connections, its response time violates the SLA. From this perspective, considering that the database tier has infinite capacity, the application tier works similar to a switch:

$$R_{AS}(n) = \begin{cases} \sim ct, & if\ U \leq Th_{AS}(n) \\ \infty, & otherwise \end{cases} \tag{5.5}$$

We continued to analyze the web application's performance profile when increasing even more the number of concurrent users accessing it. Currently, the application tier is provisioned with three resources that uniformly split the workload among them while the database tier is provisioned with one resource. Figure 5.3

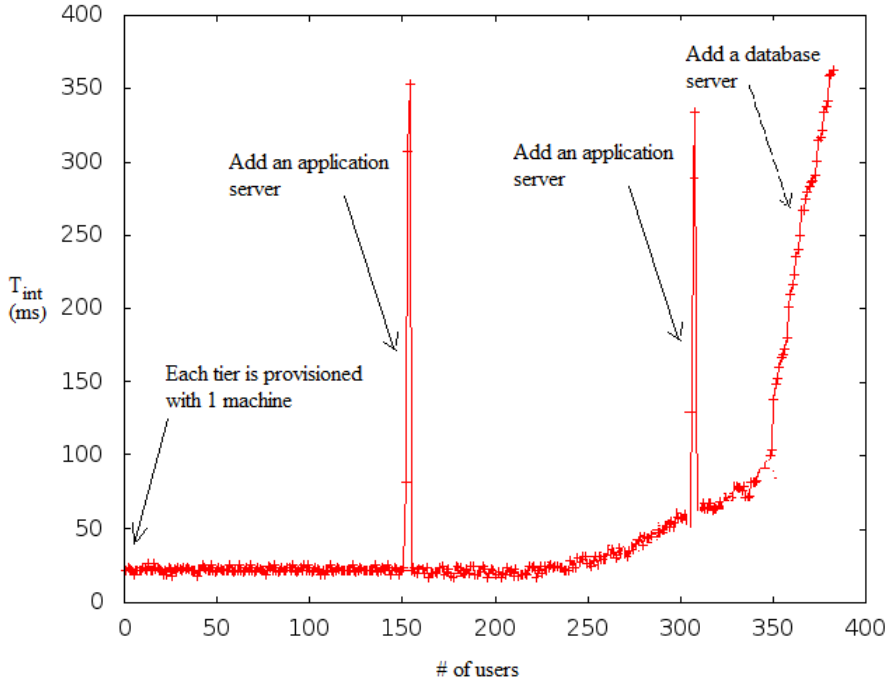

Figure 5.3: Response time of the AJAX application when increasing the workload. The application tier is reprovisioned with one more resource (3 AP, 1 DB).

extends the previous two figures and presents the further user perceived response time of the application when continuing to increase the number of concurrent users over 300.

As observed, the workload increasing generates a progressive performance loss that starts violating the imposed SLA when the number of users is approaching to 390. This workload is successfully served by the machines provisioning the application tier (that are uniformly splitting it among them). In this moment, the performance loss is generated by the database tier, currently provisioned with one machine. This machine must respond to (read and insert) requests issued by all users and cannot serve them into an acceptable amount of time such that to avoid the SLA violation. In this situation, the database tier should be reprovisioned with one more machine.

5.2 Database tier performance model

As a consequence of the last reprovisioning decision, the permanent state of the Wordpress live chatting application is replicated across multiple database servers.

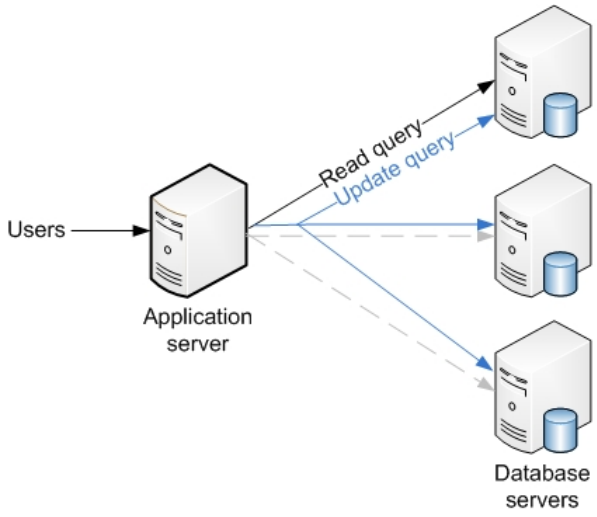

Figure 5.4: Provisioning the database tier.

Each GET request issued by an AJAX engine on the behalf of an user generates one *Read query* to the database tier. As MySQL ensures strong consistency among all database replicas, this read query can be addressed to any replica. Moreover, because all resources provisioning the database tier are homogeneous, read queries are uniformly distributed across all database replicas. On the other hand, POST requests generate two *Insert queries* but, for consistency reasons, they must be issued at every replica.

Considering that the application's database tier is provisioned with N homogeneous resources, each one of them must process a fraction $\frac{1}{N}$ *Read queries*. In the meantime, however, to ensure strong consistency, each resource processes all *Insert queries* (Figure 5.4). Because of this, anyway, at some point, increasing N, the number of resources provisioning the database tier, won't bring more benefits when the number of *Insert queries* alone saturate the database servers' capabilities. More scalable solutions such as NoSQL datastores were designed to overcome the scalability problem of relational databases such as MySQL but this discussion is out of the scope of this thesis.

The performance model of MySQL databases is a well-studied subject in the literature. It presents the characteristics of a queuing system. However, the main AJAX features we are interested in our research do not impact the performance profile of the database tier such that detailed study on the performance profile of relational databases is not required. This relational database might be replaced by a NoSQL datastore, as well.

In terms of thread management, MySQL is different from Apache because it does not use a thread pool model but a thread cache instead. Thread cache

is different from thread pool in the sense that thread cache does not pre-create threads at system startup, but the threads are managed in a dynamic fashion. When workload is heavy such that the number of required concurrent threads exceeds the cache size, it creates new threads to serve extra requests. After that only the cache size number of threads are reused and maintained alive. [15] proposes a model using load-independent multi-station queuing center to model MySQL where the number of stations is the averaged number of all worker threads of MySQL during a run.

The Wordpress web application we considered as proof of concept maintains long-lived persistent TCP connections only between users and the application tier. Any request issued by the application tier to the database tier is sent using a fresh TCP connection. The PHP extension for MySQL supports persistent connections but by default they are disabled in the new mysqli extension, maybe because of the additional caveats that have to be treated. One is that, when using table locking on a persistent connection, if the script cannot release the lock, then subsequent scripts using the same connection will block indefinitely. Moreover, when using transactions, a transaction block will also carry over to the next script which uses that connection if the script execution ends before the transaction block does. Also, another problem with MySQL persistent connections is that the connection will stay open even if the user leaves the page and the connection goes out of scope.

5.3 Evaluation

This section evaluates the correctness of the performance model previously designed. To accomplish this task, we modified some of the parameters considered, as it will be described in Section 5.3.1. In the initial configuration of the web server, each one of its tiers is provisioned with only one machine.

5.3.1 Experimental setup

We evaluated our performance model considering a different experimental setup of the Wordpress live chatting application, as described in Table 5.2. The table presents only the values of parameters that differ from the ones presented in Table 3.1. The AJAX engine placed on the client side issues also two GET requests/second and POST requests are generated with an uniform distribution. As a result of the current configuration, on each persistent TCP connection are sent up to 150 pairs of requests and their corresponding responses such that the lifetime of the active connection is increased to 90 seconds. Moreover, each application server can successfully serve 200 active connections.

MaxKeepAliveRequests	150
MaxClients	200

Table 5.2: Web server's configuration

ESTABLISHED state	90 s
TIME_WAIT state	60 s

Table 5.3: TCP connection lifetimes

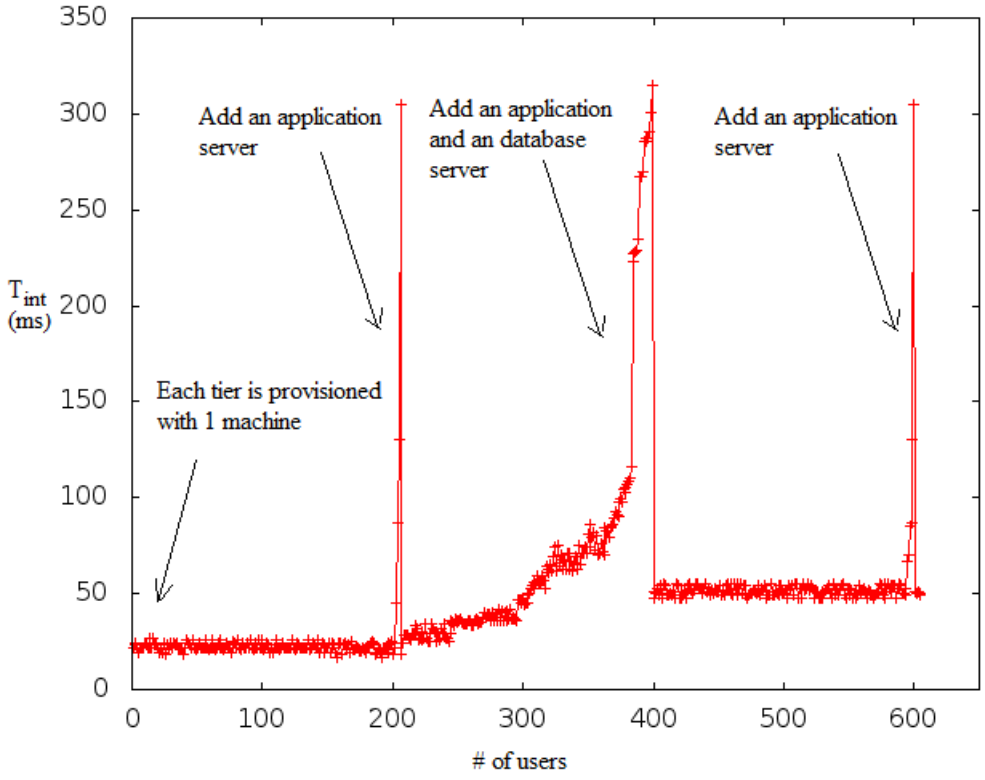

Figure 5.5: Model validation

5.3.2 Model validation

Figure 5.5 presents the web application's performance when increasing the workload. The user perceived performance offered by this initial configuration of the web server starts to violate the SLA ($T_{int} \leq 0.3$ *seconds*) when the number of users U exceeds 200. This situation, however, is confirmed by the formula 5.5

that describes the performance model of the application tier: currently, the condition of the first branch of the formula is no more satisfied (i.e. $U > 200, n = 1$) and thus, the application tier is no more capable to successfully handle all active TCP connections that would be created by the users accessing the application. As a consequence, the application should be reprovisioned with one more machine in order to double the number of Apache threads.

The workload will be uniformly distributed among the two application servers and all queries will be solved by the machine provisioning the database tier. We continue to increase the number of users after this reprovisioning. The application presents gradual performance degradation and the imposed SLA starts to be violated when the number of users is close to 400. On the one hand, the application tier must be reprovisioned as $U \simeq 400, n = 2$ and the condition of the first branch of the formula 5.5 is not satisfied. On the other hand, the current request rate ($\simeq$ 800 requests/second) exceeds the maximum throughput of the database server such that the database tier should be reprovisioned. After taking these actions, the application's services are provisioned with two and respectively three machines and the user perceived performance returns to good values.

Increasing even more the number of users, the workload will be successfully handled by the database tier but, when approaching 600, will generate too many active TCP connections and will cause the server to respond with unpredictable delays.

As seen, changing the setup of the AJAX application considered as proof of concept creates similar behavior and performance profile to the one discussed at the beginning of this chapter. The AJAX's features have a similar impact on the server's performance and determine the same linear relation between the number of concurrent active users that can be successfully served and the number of machines provisioning the application tier.

Chapter 6

Provisioning AJAX Applications

To be capable to keep up with the demand and to successfully handle workload variations or flash crowds, major web applications should be able to request on-the-fly new resources. These requests are based on resource provisioning algorithms that use the mathematical performance profile of each service composing the web application to determine the optimal configuration of the server according to the workload. The reprovisioning task can be based on the "what-if" analysis such that each service can be responsible for its own provisioning as proposed in [6]. When the imposed performance target is no more achieved, the service that would bring the most benefit when reprovisioning will be reprovisioned. This chapter discusses particular aspects and issues raised by the use of the AJAX technology in the context of resource provisioning. In contrast to classical applications, AJAX creates even more interesting challenges because of some of its characteristics and because of the long-lived persistent TCP connections maintained with users.

Among the use of the "what-if" analysis to reprovision the application, there are other tunings and approaches relevant in the context of resource provisioning the application.

6.1 Resource provisioning and web server's parameters

As previously seen, the server's performance when serving an AJAX application heavily depends on the amount of memory allocated to the server because it controls the number of active connections that can be concurrently served. From the resource provisioning perspective however, it would be beneficial that each machine provisioning the application tier to be capable to serve as many concurrent users as possible. Considering that machines are requested to a grid or a cloud

and their amount of memory is fixed, the performance can still be improved by trying to maintain as small as possible the size of the worker process. According to formula 4.1, this would lead to an increase of the $MaxClients$ parameter. Depending on the application's particular aspects, this parameter can be increased until the bottleneck moves to the CPU usage or the network traffic.

As seen previously, tuning the server's parameters configuring persistent connections (that were discussed in Table 3.1) can greatly impact the performance. Among these parameters, $KeepAliveTimeout$ and $MaxKeepAliveRequests$ have the biggest impact on server's performance. According to the current workload, web servers could dynamically tune these parameters to be able to serve it with minimal costs.

The $KeepAliveTimeout$ parameter enables the closing of connections when the user becomes idle and as a result, reduces the number of concurrent active connections. Depending on the application's particularities, tuning this parameter could help improving the performance and minimizing the number of machines provisioning the application. Anyway, in the context of AJAX applications supposing real-time interactions between users, normally the AJAX engine requests very frequently updates such that no connection will be recognized as been idle. In this situation, a greater impact has the $MaxKeepAliveRequests$ parameter that controls the number of requests that can be sent over the same connection. Increasing the value of this parameter leads to an increase of the active connection lifetime but in the mean time it is important to recycle each process because Apache threads gradually increase their memory allocation as they run. Because of this, there is a tradeoff between the lifetime of an active connection and the number of concurrent active connections that can be handled. However, there is no configuration of these parameters that fits to each and every application but the application's characteristics are the one that determine them. Because of their complexity, there is no perfect solution applying to all AJAX applications.

6.2 Reprovisioning and deprovisioning issues

Moreover, AJAX applications create even more interesting challenges when reprovisioning them. As previously discussed in Section 4.2, in the context of process-based web servers, such as Apache, that was considered as proof of concept in our research, a thread of the web server is responsible to handle *all* requests sent over a TCP connection during its entire active lifetime. As a consequence of this fact, even if the application tier has been reprovisioned in the meanwhile with new machines, the already established TCP connections cannot be relocated to these new machines. They will continue to be served by the same machines as before the reprovisioning. Only the new incoming connections would benefit from

the new provisioned machines.

The very first implication of this aspect is the fact that the uniform balancing between all machines provisioning the application tier will take much more time than in the case of classical web applications and depends on the moment of time the reprovisioning decision is taken. Moreover, just after the reprovisioning, if the workload does not follow the predicted increase, no user will benefit from the reprovisioning.

A similar situation is encountered when one decides to deprovision some of the resources previously allocated to the application tier. To maintain a good user experience, a machine previously assigned to the application tier can be deprovisioned only after *all* connections handled by all its Apache threads were terminated.

The maximum delay between the moment of time the decision of deprovisioning the application tier is taken and the moment of time the resource can be actually deprovisioned in completely safe conditions depends on the lifetime of both active and idle connection. Anyway, the maintenance of idle connections is a recommendation of operating systems' designers but not a mandatory behavior such that they can be reduced to short periods and their impact can be minimized. However, in contrast to the case of classical web applications, the resource deprovisioning of AJAX applications is delayed because of the long-lived persistent TCP connections.

Chapter 7

Conclusions

During the last decades, web applications have evolved from simple applications sharing static content to very complex ones serving highly dynamic content to millions of users spread around the world. Nowadays' web applications' design has become more and more complicated because of the new features added in order to ensure a good user experience and to keep up with the demand even in the presence of workload variations or flash crowds.

We have seen that the AJAX technique plays an important role among the new features added by designers of modern web applications in order to improve the user experience. This technique started from the idea to offer a desktop application like experience to users by placing at the client side an intermediary that incorporates part of the application processing and mediates the users' interactions with the application asynchronously to the interactions with the web server. Furthermore, AJAX makes intensive use of the keep-alive feature added by the version 1.1 of the HTTP protocol by maintaining long-lived TCP connections.

Our study revealed that these new features added by the AJAX technique greatly impact the performance of the web server such that its performance profile is completely different from the one of a classical web application, proving that different aspects influence the user perceived performance of the application or cause the web server to crash. We concluded that AJAX applications should be studied separately in the context of resource provisioning and different performance models should be designed for them. This thesis explored the challenges posed by AJAX applications by analyzing the AJAX's impact on server's performance and by identifying the key elements relevant in the context of resource provisioning. To the best of our knowledge, it is the very first study addressing these aspects.

As a consequence of the AJAX applications' characteristics, we observed that the web server's performance profile is much different than the one of classical applications in that it saves CPU usage by reducing the number of connections

establishments and terminations required to provide to users the same application specific functionality. This also reduces the size of the tables storing state information correlated to all active and idle TCP connections and implicitly the complexity and the processing required by the operations issued on these tables. Moreover, because of the same reasons, a significant reduction of the network traffic is encountered. AJAX improves the user perceived performance as the response time of the applications most of the times does not incorporates the time required by a new TCP connection to be established.

We identified the amount of memory allocated to web serving purposes as a key aspect since it has direct influence on the number of active connections that can be successfully served concurrently when hosting the application using a process-based web server, such as Apache, the most popular web server nowadays. This limitation exists because there is a one-to-one relationship between a thread of the web server and a TCP connection with a user: a thread serves *all* requests sent over a connection during its entire lifetime but the thread can handle only *one* connection a time.

In the case of such web applications enabling long-lived TCP connections, we registered a much higher number of concurrent active connections than in the case of classical applications such that a greater number of threads will be busy serving them. Whenever the number of users concurrently accessing the application exceeds the maximum number of threads that can be created by the web server, waiting queues become long, cause the users to quit and generate poor user experience.

AJAX applications serving highly dynamic content and supposes no computational intensive processing at the server side determine a performance profile of the web server similar to the one of the web application considered as proof of concept in our research. When facing workload variations, the performance degradation is caused by the server's impossibility to handle the respective number of concurrent active TCP connections.

After understanding the performance profile imposed by AJAX, we designed the mathematical performance model of the web server serving the applications. This model is capable to predict the future performance of each service making up the AJAX application when adding or removing machines and when increasing or decreasing the workload. Whenever the performance target is no more achieved or, in contrast, the web application makes waste of resources, the performance model is further used by resource provisioning algorithms to determine what is the service that offers the most benefit when reprovisioning and then to reprovision it.

Our research focuses of process-based servers, as they are nowadays the most popular web servers. Our study revealed the fact that the number of concurrent users that can be successfully served increases linearly with the number of

resources provisioning the tier influenced by AJAX. This behavior is directly caused first of all by the internal characteristics of the web server and secondly, by the fact that only the application service is directly influenced by the AJAX features.

This performance model does not stand, however, for other services composing the application where the direct impact of AJAX features is not felt and which behave exactly the same regardless of the using or not using of AJAX. For instance, in the case of the application considered as proof of concept in our research, the database service presents a classical queuing model.

7.1 Future work

There are few directions in which the current research on AJAX web applications can be improved and extended by future work.

First of all, as discussed in Section 4.2, web servers have evolved in two main directions: process-based web servers or event-based web servers. The most important difference between the two categories consists of the way active TCP connections are allocated to the web server's threads to serve them: following an one-to-one relationship in the case of process-based servers or a multi-to-one relationship in the case of event-based servers. This design decision has great relevance in the context of AJAX web applications because of the long-lived TCP connections maintained with the users. As a consequence, future work should analyze the impact on the performance profile of the design decisions of process-based web servers.

Moreover, depending on the application's particular character and also on the types of interactions exposed to users, some AJAX web applications could take great advantage of the use of caching. Future work should analyze the benefits of caching at different tiers of the AJAX web applications and should capture it into performance models.

As previously discussed in Chapter 3, because of the complexity of the interactions supposed by AJAX web applications, it is important to ensure their realistic benchmarking in order to be capable to fully capture different aspects that greatly impact both the server's performance profile and the user perceived experience. Among these aspects, one can name the key typing speed, the network latency, the distribution of clients and even the web browsers types used by users to access the application. Recent research proposes to use real browser when benchmarking web applications, not to emulate them, such that, future work on resource provisioning of AJAX web applications should take these aspects into consideration.

The testing environment considered in our research is a grid such that all

machines provisioning the web application are homogeneous. However, recently, thanks of its great advantages, cloud computing has become an attractive platform to host web applications but in this context machines are no more homogeneous, such that future research on AJAX applications should take this into consideration.

Last but not least, our research focused on AJAX web applications following a pull-based model similar to most of the modern applications that are successful nowadays. However, the category of applications designed in a push fashion also raises interesting challenges that should be explored by future work.

Bibliography

[1] The Distributed ASCI Supercomputer 4. http://www.cs.vu.nl/das4/.

[2] Tarek Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems Journal*, 2001.

[3] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[4] Edith Cohen, Haim Kaplan, and Jeffrey Oldham. Managing tcp connection under persistent http. In *Proceedings of the eighth international conference on World Wide Web*, pages 1709–1723, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[6] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proceedings of the International World-Wide Web Conference*, apr 2010.

[7] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource provisioning of Web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, jun 2011.

[8] Ronald P. Doyle. Model-based resource provisioning in a web service utility. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems*, 2003.

[9] Ebay. Sales statistics sources for ebay uk and ebay.com, 2009.

[10] The Apache Software Foundation. http://www.apache.org/.

[11] Jesse James Garrett. Ajax: A new approach to web applications. In *http://www.adaptivepath.com/ideas/e000385*. Adaptive Path, 2008.

[12] Clinton W. Smullen III and Stephanie A. Smullen. Ajax application server performance. In *Proceedings of SoutheastCon, IEEE*, pages 154 – 158, 2007.

[13] Andrea James. Amazon's 2-hour crash thwarts shoppers, 2008.

[14] Abhinav Kamra. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *In International Workshop on Quality of Service (IWQoS)*, pages 47–56, 2004.

[15] Xue Liu, Jin Heo, and Lui Sha. Modeling the 3-tiered web sites. *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 888–895, 2005.

[16] Jeffrey C. Mogul. The case for persistent-connection http. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 299–313, New York, NY, USA. ACM.

[17] Why NGinx? http://wpnginx.com/why-nginx/.

[18] Wordpress official website. http://wordpress.org/.

[19] Apache optimization. A brief history of keep-alives, 2011.

[20] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers, 2002.

[21] Youri Roodt. The effect of ajax on performance and usability in web environments. Research supervised by Hyves, Universiteit van Amsterdam, Hogeschool van Amsterdam, Vrije Universiteit.

[22] Randy Shoup. ebay's architectural principles. In *http://qconlondon.com/dl/qcon-london-2008/slides/RandyShoup_eBaysArchitecturalPrinciples.pdf*.

[23] Swaminathan Sivasubramanian. Scalable hosting of web applications, 2007.

[24] TCP Specification. http://tools.ietf.org/html/rfc793. September 1981.

[25] WordPress Usage Statistics. http://en.wordpress.com/stats/.

[26] Netcraft WebServer Surveys.

[27] W3 Techs. Web technology surveys, usage of content management systems for web sites, 2011.

[28] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the ACM SIGMETRICS'2005*, pages 291–302, 2005.

[29] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Ch, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 2008.

[30] Pierres Wordspew WordPress Plugins. http://wordpress.org/plugins

[31] BangYu Wu, Chi-Hung Chi, and Zhe Chen. Resource allocation based on workflow for enhancing the performance of composite service. *Services Computing, IEEE International Conference on*, 0:552–559, 2007.