Fine grain performance evaluation of e-commerce sites

Mauro Andreolini, Michele Colajanni, Riccardo Lancellotti, Francesca Mazzoni
Department of Information Engineering
University of Modena and Reggio Emilia, Italy
{*andreolini.mauro, colajanni, lancellotti.riccardo, mazzoni.francesca*}*@unimo.it*

Abstract

E-commerce sites are still a reference for the Web technology in terms of complexity and performance requirements, including availability and scalability. In this paper we show that a coarse grain analysis, that is used in most performance studies, may lead to incomplete or false deductions about the behavior of the hardware and software components supporting e-commerce sites. Through a fine grain performance evaluation of a medium size e-commerce site, we find some interesting results that demonstrate the importance of an analysis approach that is carried out at the software function level with the combination of distribution oriented metrics instead of average values.

1 Introduction

E-commerce sites have become the de-facto standard for offering business-oriented services through the Web. They are characterized by services and flows of information that in the large majority of cases are created on the fly and often are personalized for each customer. The need for such complex (and often critical) services has led to the deployment of specific Web-based technologies, which have evolved into todays' *e-commerce* systems. There is a large variety of open source and proprietary components to build these systems, but all of them are nowadays based on a multi-tier software architecture that tends to separate the presentation, the business and the information layers.

The *front end* layer is the interface of the e-commerce system. It accepts HTTP connection requests from the clients, serves static content from the file system, and offers an interface towards the business logic running on the second layer. The most popular software for the front end layer is the Apache Web server, although others exists (e.g., MS Internet Information Services, Sun Java System Web Server, Zeus). The *application server* layer is at the heart of an e-commerce system: it handles all the business logic and computes the information which will be used to construct HTTP documents. There is a huge number of technologies for deploying a business logic on the middle layer, for example CGI, ASP, JSP, PHP, EJB, Java Servlets. The *back end* layer typically consists of a database server and storage of critical information that is the basis for generating dynamic content. Database servers have a long history and there are many possible choices. The most common for e-commerce systems are Oracle, IBM DB2, MS SQL Server on the proprietary side, and MySQL and PostgreSQL on the open source side.

The complexity of hardware and software components and of their interactions, the heavy-tail characteristics of the Web workload, the burst arrivals that cause sudden peaks make the performance evaluation of an e-commerce system a quite difficult task. Most performance studies use coarse grain load monitors and average performance metrics that are useful just to give a first idea about the system behavior at the level of hardware resources, such as disk, CPU, network interface. They are also helpful for a preliminary bottleneck analysis. The limits of this coarse grain approach arise when it is necessary to understand the real motivations that are behind poor performance or a bottleneck in an e-commerce system, where there are literally hundreds of processes/threads cooperating and conflicting for the same few resources.

In these cases, a coarse grain view of the system resources may lead to incomplete or false deductions. Hence, we found necessary to pass to a fine grain performance analysis that has a twofold consequence: to look the system at the function level instead of the node level; to consider distribution-oriented metrics instead of the often misused average-oriented metrics. Understanding which fine grain functions are having the biggest impact on the performance of a system resource and considering more accurate metrics, such as cumulative distributions and percentiles, are also the necessary basis to settle SLA-based e-services.

We apply this approach to the analysis of a medium size e-commerce site, that is built through open source software and runs on PC-like hardware. Although the performance results of this study cannot be immediately extended to consider the complexity of an e-commerce site built through sophisticated proprietary suites, such as Oracle and IBM Web Sphere, the spirit of the approach and main conclusions are quite representative for

the large majority of medium-size dynamic Web sites receiving something in the order of hundreds of requests per minute.

It is interesting to observe that the proposed fine grain level approach is suitable not only to indicate which software component(s) is(are) likely to be behind a system bottleneck, but it also helps to evidence other interesting results. For example, this study allow us to understand the impact that the continuous improvement of hardware components even at the entry level may have on the present and future performance of medium-size e-commerce sites. It evidences the consequences of WAN effects even on the performance of the middle and back end layers; it shows the impact on system performance of token-based resources (e.g., file and socket descriptors) that do not degrade gracefully as the CPU does.

This paper has many contributions that in part confirm other results in literature and in part are original with respect to multi-tier e-commerce systems. There are a lot of interesting fine grain performance studies concerning the HTTP servers, especially Apache Web server, such as [18, 10]. However, we are not aware of other analyses based on low-level kernel profiling oriented to the performance evaluation of e-commerce systems. The WAN effects on Web server performance has been considered in [16], however this paper is limited to a platform consisting of one Web server hosting static contents. None of the previous studies seems to take into account the impact of the Internet to the performance of the internal layers of an e-commerce systems. The identification of the indexes and metrics which result critical for the performance evaluation of e-commerce systems have been considered in many contexts, but not with the goal of analyze the consequences of the different granularities. For example, Dahlin [4] addresses the issue of using stale server load information in the context of a distributed system. The proposed algorithms may improve the performance of distributed e-commerce systems.

Other studies compare different technologies for implementing the same e-commerce system [13, 8], evaluate the performance characteristics of different e-commerce sites [3], or focus on the TPC-W model [5]. In [11] the authors present an overview of key factors affecting performance of Web sites which may be using Web services protocols.

The remainder of this paper is organized as following. Section 2 outlines the main functions of the components of a representative e-commerce site of medium size and popularity. Section 3 focuses on the different levels of granularity and resource metrics for the evaluation of the performance of the considered e-commerce system. Section 4 describes the experimental testbed and the workload model that we used for the experiments. Section 5 discusses the experimental results, confirms some previous conclusions, and outlines some novel results. Finally, Section 6 concludes the paper with some final remark.

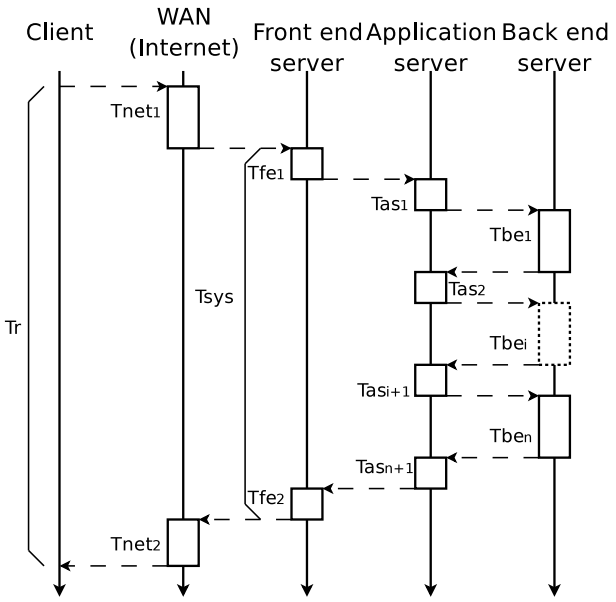

Figure 1: Temporal diagram of a request for a Web object to an e-commerce site

2 Coarse grain times in an e-commerce system

A user request for a Web resource is typically processed by the browser in terms of multiple requests to the front end server to get the HTML page container and its embedded objects. Most of the embedded objects are static files that can be served directly by the front end HTTP server. Other objects are generated on the fly through one or multiple interactions with the application layer and the back end layer. Let us focus on these dynamic requests that imply the most complex interactions.

The *response time* T_r for a Web object consists of two main components: the network contribution T_{net} and the e-commerce system contribution T_{sys}. This latter, in its turn, may be composed by one or multiple sojourn times in the three main system components: the *front end time* T_{fe}, the *application server time* T_{as}, the *back end time* T_{be}. (For the sake of completeness we should also consider the time spent by each node to interact with the others, but we neglect it because it has always been demonstrated irrelevant with respect to the other components.)

Figure 1 shows the temporal diagram corresponding to a request for a dynamically generated resource. It evidence the main sojourn times caused by the three main *software components* of the e-commerce system. The network time is typically independent of the type of request (unless it requires a secure channel), while the important

parameter is the dimension of the transmitted information: when the client sends a request to the e-commerce site, the request reaches the front end after the time T_{net_1} that is determined by the network conditions between the client and the server hosts. After the e-commerce system time T_{sys}, it is necessary to consider an additional network time T_{net_2}.

When the request is for a dynamic object, the front end activates the application server after a processing time T_{fe_1}. We can assume that the server at the application layer handles the logic associated with the request with a T_{as_1} processing time. If the application server needs some data from the back end layer, then it must issue one or more queries to the database server. This server receives the first query, processes it in a T_{be_1} time, and returns the result to the application server that processes the received data (T_{as_2} time) and possibly issues other $n-1$ queries to the database server. When the application server has gathered the results from the database server ($T_{as_{n+1}}$ time), it passes all information to the Web server that builds the HTTP response (T_{fe_2} processing time) and sends it back to the client.

Hence, the response time for a dynamic resource may be written as in Eq. 1, where n denotes the number of queries to the database server.

$$T_r = \sum_{i=1,2} T_{net_i} + \sum_{i=1,2} T_{fe_i} + \sum_{i=0,n+1} T_{as_i} + \sum_{i=0,n} T_{be_i}. \quad (1)$$

The total response time does not give any clue about possible system bottlenecks. But even the T_{net}, T_{fe}, T_{as}, T_{be} times considered as separate terms give a false impression of mutual independence. On the other hand, the components of the e-commerce system are strictly correlated. For example, the application server relies on the back end to provide the necessary information for building the application logic data. If the back end layer fails or is slow, the performance of the application server may be severely degraded and, as a domino effect, the overall performance of the dynamic requests drops. The above considerations give a first hint about the necessity of considering performance indexes at a finer grain level.

It is important to observe that there is no one-to-one mapping between the multi-tier logical layers and the physical architectures. The software components (front end, application and back end servers) may run on a single physical node or on a cluster of nodes [2], or even may be distributed over a geographical area. The best choice depends on many factors, such as the adopted technology, the size and popularity of the e-commerce system, the security goals. If we refer to medium-size e-commerce sites, the real choice is between two or three physical nodes, because the common tendency is to map the database server on a separate node. With present technologies, we can say that a software such as J2EE [12] would lead to a physical separation of the three logical layers on three nodes. On the other hand, other software such as ASP, JSP, PHP tends to concentrate the front-end server and the application server on the same machine.

3 Resource metrics at different levels of granularity

We discuss some problems related to the choice of the appropriate resource metrics for evaluating the performance of an e-commerce system. This is a quite difficult task for several reasons. First, there is a multitude of possible resource metrics (utilizations, throughputs, response times) at each layer of the system. With the advent of new, more sophisticated OS and software technologies, this number is doomed to increase further. Thus, it is important to determine which of the available metrics should be used as *explanatory variables* that have a major impact on system performance.

The task is further complicated by the modern software components, that are designed to support hundreds or even thousands of users at the same time. In this context, common monitoring tools, such as the system activity report [7], which should help in the identification of potential explanatory variables, are beginning to show the limits of sampling measures at a coarse granularity. As an example, many database management systems (including MySQL) support fast, asynchronous I/O operations and table buffering; preserving buffer consistence under these conditions are transforming some I/O operations in computationally intensive tasks instead of disk-bound operations.

Hence, depending on the performance study, we have to find the most appropriate *granularity level* for each resource metric. There exists many levels of granularity: *system*, *node*, *hardware resource*, *software component*, *process*, *function*. As a testbed example, we consider the system described in Figure 2, that has been implemented on two nodes by means of the PHP technology.

System-level metrics represent the overall behavior of the system. An example is the response time T_r in Figure 2. More generally, the performance samples collected by the clients are all at the system level. They are the easiest metrics to collect, but from these indexes we can only verify whether the system is providing or not services at an acceptable level of performance. It is not possible to draw any other conclusion by simply looking at these coarse grain indexes.

Node-level metrics describe the behavior of a single, physical machine. In the architecture of Figure 2, T_{be} and $(T_{fe} + T_{as})$ are two examples of node-level metrics.

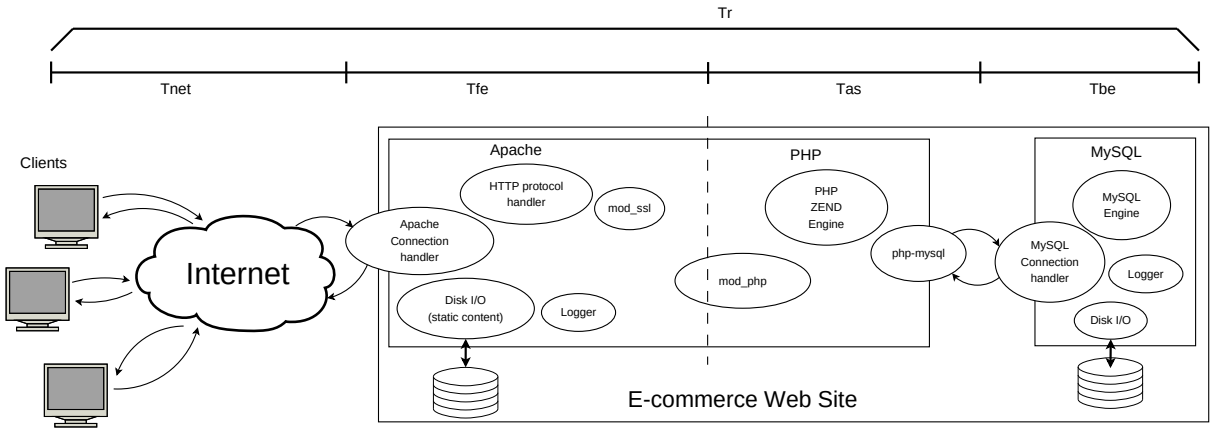

Figure 2: An example of PHP-based architecture of an e-commerce site

At this granularity level, unexpected behavior at the single nodes can be spotted, but no clue is given about what is slowing down that machine.

To delve a deeper analysis, we have to consider at least the *resource-level* metrics, that are associated to the hardware and operating system resources of a node. Typical examples include: the utilization of the CPU, disk and network interface that are gracefully degrading resources; the number of available file and socket descriptors, the amount of free memory that are token-based resources that may cause a sudden performance degradation. The vast majority of monitoring tools provide performance samples at this level of granularity. Indeed, they help find to the lacking node resource, but it is difficult or impossible to derive the motivation for sure by simply looking at resource-level metrics. For example, in a node hosting a database server, if the disk results as the node bottleneck, we can easily assume that this is due to some database operations, but we cannot know which component is causing the heavy I/O operations that are degrading the node performance. The problem become even worse when more software components run on the same node (which is usually the case with Apache and PHP). In these case to separate the utilizations of the different components is a difficult or impossible task, and we have to pass to a finer granularity.

The *component-level* metrics reflect the behavior of a software component, such as the HTTP server, the application server or the database server. The problem is that they are quite difficult to sample. Some metrics, such the response time, may even require modifications to the source code. On the other hand, component-level metrics usually give a detailed view of the system performance and permit to identify the critical components.

Even after the identification of the most critical component, when dealing with multiprocess or multithreaded applications, it is not easy to understand the motivations why this components is not performing according to the expected specifications. Often, it is necessary to go down to consider the *process-level* granularity. A typical examples include the response times of the single processes composing the application. The process-level granularity is of great help in spotting the critical areas, but the related metrics are almost impossible to collect with ordinary monitoring tools that are typically limited to the node-level granularity. The process-level metrics can be gathered by modifying the appropriate source code or by running more sophisticated performance analyzers, such as node profilers.

Some classes of software components, such as the database server, are so complex that even after the individuation of the critical area, it is difficult to understand *exactly* the hot spots in the code. To this purpose, it is necessary to resort to the finest granularity level, the so called *function level*, which relates to the main functions of each process (including the operating system). This level of granularity requires special instrumentation (node and kernel profilers [14]). They cause some overhead on the system and yield data (call profile graphs) that are quite expensive to analyze. For this reason, the analysis of profiled data is usually performed off-line. On the positive hand, the function-level metrics identify precisely the critical areas of the software component.

The analysis is not completed because, once a bottleneck is identified, we have to remove it. These actions depend on the particular type of bottleneck. We may have a misconfigured system component that is under-performing, where a simple reconfiguration may help boost its performance. Typical examples of *configuration bottlenecks* include the number of worker processes or the maximum number of TCP connections. We may find that one physical resource at a node (CPU, disk, network interface) is completely utilized. In this case, instead of reconfiguring the system, its capacity must be upgraded. There are two ways for improving the capacity of a component: *hardware upgrades* which simply augment the system capacity with the same number of node(s), and *hardware*

replications which allow the system to distribute the computation among additional nodes. A *software bottleneck* may be removed through a better design. The latter case is rather infrequent, because several application servers and the major back end servers are not shipped with the source code.

It is clear that the broadest picture is taken when sampling all resource metrics at the function level. While the collection in itself may be executed with relatively low overhead through system kernel profilers, the real problem is the off-line analysis of call-graphs for thousands of different application-level and OS-level functions. There is usually a tradeoff between the granularity level of the resource metric and the completeness of the obtained samples. There is no need for sampling at the function-level when bottlenecks result obvious from the component-level metrics. However, in most instances it is rather easy to locate a bottleneck even at a higher level, but when we need to understand the motivations of that bottleneck we have to carry out the performance analysis at the finest granularity of the previous scale.

After the choice of the resource metrics, it is necessary to determine the most representative statistics for the considered samples. We do not want to enter into many details, but we should consider that average values are still common choices for many performance studies, even if the characteristics of the e-commerce systems (e.g., workload models characterized by quantities at different orders of magnitude, heavy-tailed distributions, burst arrivals, complex correlations between the software components) would require higher moments. The choice of cumulative distributions or percentiles instead of average values becomes even more important when we consider that e-commerce systems may be interested to provide services based on some *Service Level Agreement* (SLA). And it is quite obvious that the service levels of a complex multi-tier system with large variances require statistics that are more representative than simple average values.

4 General experimental setting

4.1 Experimental testbed

Figure 3 shows the architecture of the prototype system. The experiments are carried out in a clustered environment of nodes running the Linux operating system (kernel version 2.6.8). Each node is equipped with a 2.4GHz hyperthreaded Xeon, 1GB of main memory, 80GB ATA disks (7200 rpm, transfer rate 55MB/s) and a Fast Ethernet adapter. All nodes are connected through a Fast Ethernet switch.

One node runs both the Apache [1] Web server (version 2.0.52) and the PHP4 [17] engine, which is used to implement the scripts at the application layer. The

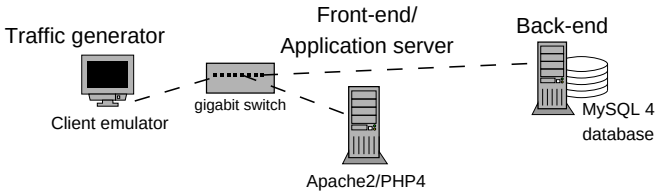

Figure 3: Architecture of the testbed for the experiments

Table 1: Composition of the workload scenarios.

Scenario	Static requests	Dynamic requests
Browsing	60%	40%
Buying	5%	95%

database server MySQL [15] (version 4.0.20) runs on a different node. To reflect a realistic workload scenario, we enabled the support for table locking and two phase commits. Data collection is performed through monitoring tools at the node level (the *system activity report* [7]) and at the function level (oprofile [14]).

To take into account the effects of wide area networks, we instrumented the *netem* packet scheduler [9] that creates a virtual link between th clients and the e-commerce system with the following characteristics: maximum link bandwidth 8Mbit/sec, packet delay normally distributed with $\mu = 200ms$ and $\sigma = 10ms$, packet drop probability of 1%.

4.2 Workload model

The choice of the workload model to test an e-commerce system is a problem by itself. Unlike the workload models oriented to browsing where the interaction is basically with the Web server and the mix is mainly oriented to define the number and size of embedded objects together with the user think time, it is impossible to define *THE* model for the e-commerce because of the dozen of possible alternatives at any level of the multi-tier architecture, often dependent also on the adopted software technology. The research community is being oriented to use the TPC-W benchmarking model that is the only complete specification of an e-commerce site (online book store). Even in this paper we use a TCP-W like model. We implement two different scenarios, *browsing* and *buying*, which capture client activities towards the e-commerce system. The percentage of requests for dynamic and static Web resources for both scenarios is shown in Table 1. Space reasons lead us to present the experimental results only for the buying scenario.

Web traffic is generated by means of a TPC-W like *client emulator*, which is executed on a separate node. The client emulator creates a fixed number of client processes which instantiate sessions made up of multiple requests to the e-commerce system. For each customer session, the client emulator opens a persistent HTTP connection

to the Web server which lasts until the end of the session. Session length is exponentially distributed with a mean of 15 minutes. Before initiating the next request, each emulated client waits for a specified think time, which is distributed exponentially with an average of 7 seconds. The sequence of requests is determined through a state transition matrix that specifies the probability to pass from one Web page to another one.

5 Experimental results

In this section we present the main results of a fine grain performance evaluation of the e-commerce system when it is subject to the *buying* workload scenario. We evaluate multiple performance metrics at different granularity levels (system, node, hardware resource and software component levels) that allow us to identify the exact causes of system congestion, and motivate some initially unexpected behavior. To this purpose, we use node, resource and component level measurement to get a deeper understanding of the hot spots in the system. Finer grained metrics, at the function level, permit to narrow the root of the problem and fully explain the causes of the unexpected performance level.

In the first part, we show how the complete picture of the performance metrics can give useful information about the corrections that should be applied to the e-commerce system to avoid bottlenecks or to improve its performance. In the second part, we evaluate the impact of wide area network dynamics on the system performance. This study allow us to show the necessity of considering in the analysis even the token-based resources that are often neglected, but that cause the worst performance problems when the pool is empty.

5.1 Fine grain performance evaluation

The first set of experiments is carried out in a local environment without taking into account WAN effects, which will be discussed in greater detail in Section 5.2. First, we focus on system level measures (system response time T_{sys} and the system throughput Thr_r) to find the capacity of the system. Figure 4 shows the system throughput (in served objects per second) as a function of the client population. Saturation occurs around 250 clients when the system is able to serve approximately 400 objects per second.

We consider also cumulative distributions and percentiles that are more representative of the system behavior in the context of Web-based information systems. For example, in Figure 5 we show the average and the 90-percentile of the system response time for a single Web object. The growth of the 90-percentile around the knee is much more evident (from 0.075 at 240 clients to 0.68 seconds at 360 clients) than that of the average response

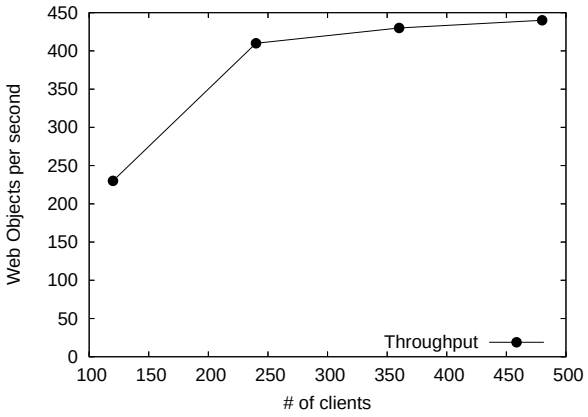

Figure 4: Throughput of the e-commerce system

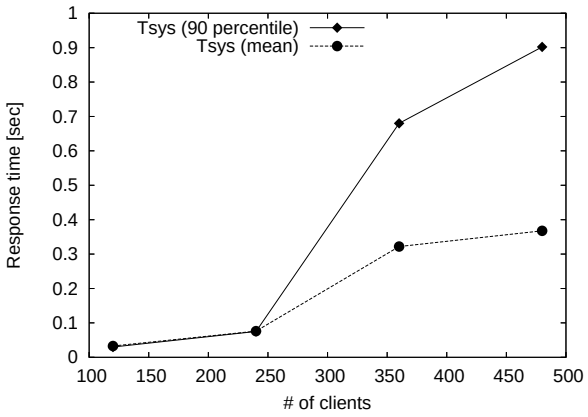

Figure 5: Response time of the e-commerce system

time (from 0.0759 at 240 clients to 0.322 at 360 clients). Indeed, mean value may tend to underestimate the explosion of the response time for increasing client populations.

Granularity at the system level is fairly easy to collect, but it only allows for congestion detection. For example, it does not provide any information about the causes that lead to poor performance. Thus, it is necessary to investigate the system at a finer grain level to find out where and why congestion occurs. Let us pass to consider the contribution to the system response time of the two nodes that compose the considered e-commerce system: one hosting the Apache Web server and the PHP component, the others running the MySQL server. Node-level measurement allows us to identify where the major part of the response time is spent. Furthermore, we can easily identify the node that gets overloaded by observing which node-level response time explodes first. Figure 6 shows the 90-percentile of the system response time T_{sys} and the contributions of the two nodes T_{fe-as} and T_{be} (logarithmic scale). There is no doubt that T_{be} represents the predominant factor of T_r. Hence, with the considered workload oriented to dynamic requests, the

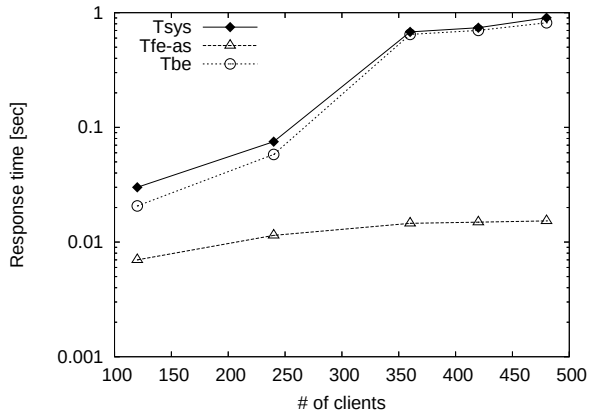

Figure 6: 90-percentile of the contributions to the system response time at the node level

bottleneck that limits the performance of the system is on the back-end node.

At this level of granularity we are still missing most of the information on the possible interventions on the system to avoid the congestion. A common approach to better identify the potential bottlenecks is to move to a finer granularity level such as the resource level. We use metrics such as CPU, disk and network utilization collected on each node that allow to evidence the hardware resource of the back-end node that is over-utilized with respect to its capacity.

Figures 7 and 8 show the CPU utilization of the back-end node during the experiments carried out with a population of 240 and 360 clients, respectively. The two horizontal lines represent the mean values. CPU utilization is burst at 240 clients (Figure 7), while it bumps to 100% for the majority of time when there are 360 clients (Figure 8). Table 2 shows the average resource utilization of the database node for different client populations. In particular, user-space and kernel-space measures are separated to better identify the source of the problem (application computations or system calls?). To complete the picture, we also report the utilization of the disk and the network interface. This table confirm that the CPU utilization is extremely high, while disk and network seem underutilized. The low disk activity is an initially unexpected result that we can explain due to the database size which in large part fits in the main memory of 1Gbyte. This aspect is interesting because with the hardware improvements even at the entry level, it is becoming common for medium-size e-commerce databases to fit for a large part in the main memory. As a trend result, we can conclude that the disk activities may not be the most significant component for understanding the behavior of the back-end node hosting the database server. A similar result has been obtained in [6].

If we consider the CPU utilization for 360 clients (Table 2), we recognize a 80-20% ratio between the time spent in the user and kernel space. This initially unexpected result suggests that the application level computations are much more intensive than the cost necessary for the system calls. As there is only one major process running on the back-end, we can easily assume that the *mysqld* server process is the source of the bottleneck. However, if we limit the analysis at this granularity level, we cannot exactly motivate the high CPU utilization of the database server application.

# of clients	CPU utilization		disk utilization	Network utilization
	user	kernel		
120	17%	4%	0.10%	0.018%
240	51%	9%	0.12%	0.019%
360	76%	14%	0.15%	0.020%

Table 2: Hardware resource utilizations

Hence, to identify the hot spots in the database server process we pass to get measurements at the function level that represent our finest grain. The profiler output shows more than 800 mysqld functions, hence a detailed analysis is quite difficult and even useless. The idea is to focus on the functions the use more CPU time, while we aggregate the others that are not significant for a performance study. Figure 9 shows the percentages of CPU utilizations that are used by the main functions of the mysqld process that is, tuple management, I/O management, *buf_page_is_corrupted()*, others. The result of the pie is clear and unexpected: the function *buf_page_is_corrupted()* that checksums asynchronous I/O buffers uses almost 70% of the CPU time. This result is not at all obvious from the measurements carried out at the hardware resource level, hence some technical details may be useful. This function is part of the asynchronous I/O buffer management of mysqld. Asynchronous I/O is used to improve I/O performance by caching frequently accessed portions of the database thus bypassing the operating system disk buffer cache. To provide data consistency a checksum is calculated on every mysqld buffer through the buf_page_is_corrupted() function. We can conclude that the asynchronous I/O subsystem is the real bottleneck of the mysqld process.

To improve the database performance, we need to reduce this checksumming activity by decreasing the number of buffer accesses. This can be easily done by augmenting the size of the query cache. Figure 10 reports the cumulative probabilities of the response time of the back-end node before and after the intervention. This figures proves the validity of our deduction because the 90 percentile of T_{be} drops from the original 0.646 seconds to 0.273 seconds after the database tuning. A similar improvement is reflected on the system response time T_{sys}.

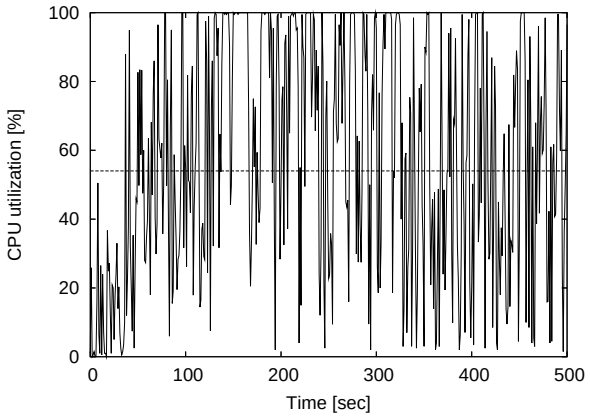

Figure 7: CPU utilization of the back-end node (240 clients)

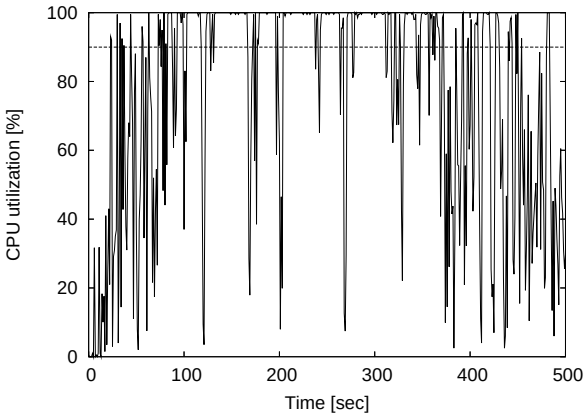

Figure 8: CPU utilization of the back-end node (360 clients)

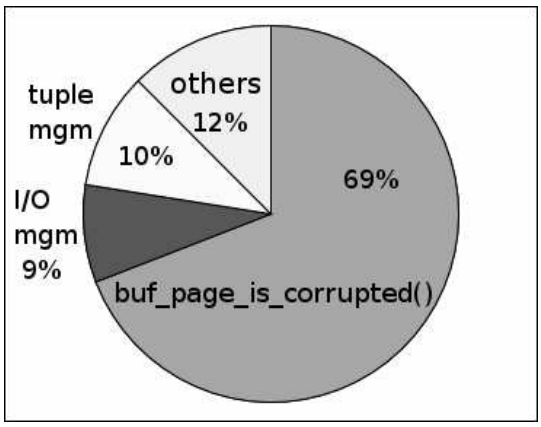

Figure 9: CPU utilization percentages at the function level (database process)

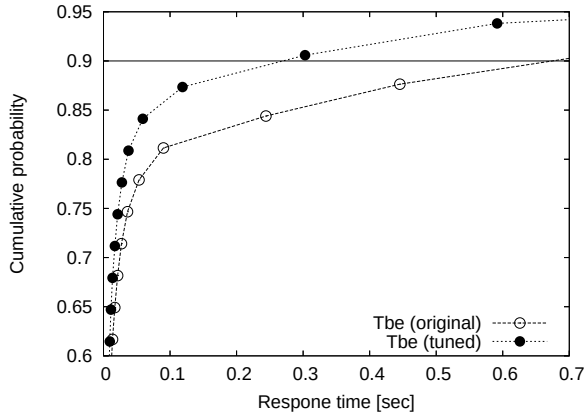

Figure 10: Cumulative distribution functions of the back-end response time before and after the tuning operation

5.2 Impact of wide area network effects

The performance study on the e-commerce system involved a large number of experiments. In this section we report another interesting result that was initially unexpected, at least for the motivations. The goal is to evaluate the impact of WAN effects on the performance of the e-commerce system. The initial motivation for this study comes from the results of Nahum *et al.* [16] that were among the first authors to describe the details of the wide area network effects on the behavior of the Web server system. The results of this section extends their work from a single layer Web site to a multi-tier e-commerce systems.

The testbed and workload model is the same of the previous section, with the addition of wide area network emulation between the clients and the front-end node. Due to space limitations, we focus on a client population of 240 clients and outline the impact of wide area network characteristics on the results obtained previously. We denote the scenario without and with wide area effects through the terms *no-WAN* and *WAN*, respectively.

We first compare the response time T_r for the two network scenarios. Figure 11 shows the cumulative distribution of the response time of one dynamic request to the e-commerce system T_r and the network delay T_{net} for the WAN scenario. The 90-percentile of $T_r(WAN)$ is equal to 2.5 seconds. From the previous experiments we have also that the 90-percentile of the system response time for the no-WAN scenario is equal to 0.075. The difference of two orders of magnitude is too large to be motivated only by the introduction of the network delays. Indeed, the network delay has a 90-percentile below 1.5 seconds, which cannot explain the 90-percentile of 2.5 seconds for $T_r(WAN)$, when the $T_r(no-WAN)$ counterpart is below 0.1 seconds.

To look for other motivations we pass to a finer grain analysis of the system. To this purpose, we show in Fig-

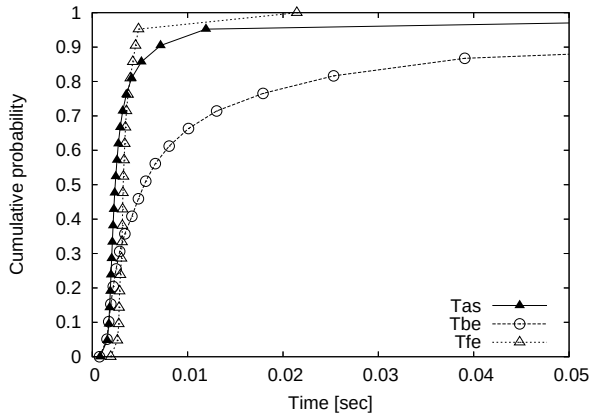

Figure 12: Cumulative distributions of the components of T_{sys} (no WAN scenario)

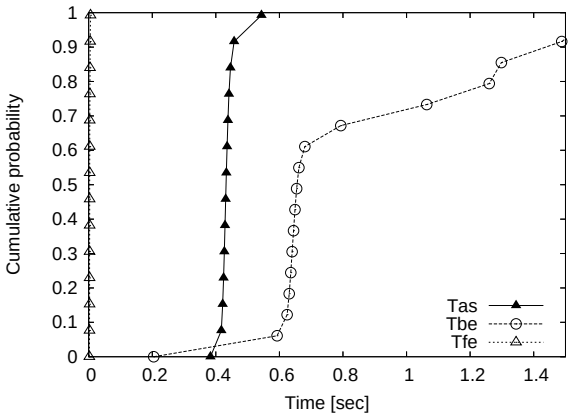

Figure 13: Cumulative distribution function of the components of T_{sys} (WAN scenario)

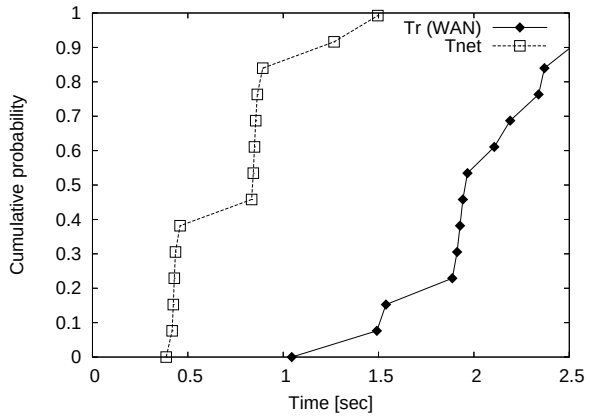

Figure 11: Cumulative distributions of the response time T_r and the network time T_{net} (WAN scenario)

ures 12 and 13 the contributions to the response times at the level of the three main software components. In the comparison, it is necessary to consider the different scale on the x-axis.

While the contributions of the front-end servers T_{fe} remain low for both scenarios (the 90-percentile is equal to 0.0045 and 0.005 for the no-WAN and WAN scenario, respectively), the contributions of the application and back-end server grow substantially with the introduction of the WAN effects. In particular, the 90-percentile of T_{as} passes from 0.008 to 0.46 seconds, and the 90-percentile of T_{be} passes from 0.055 to 1.5 seconds. The poor performance of the database and the application server in a WAN scenario are explained through a finer granularity analysis.

Figure 14 shows the number of open sockets during the experiment in the WAN and no-WAN scenarios. From this figure, it is immediate to get two results: the number of sockets simultaneously used by the database server is three times higher in the WAN scenario (on average, 45 vs. 128); the sockets are a limited set that in this experiment is fully utilized (128 is the default number). The growth in socket needs is explained by considering that connections between the application server and the database server last much longer in the WAN scenario due to the network slowdown on client requests. Sockets are token based resources that are not gracefully degradable. This means that once the number of available sockets is exhausted, further requests to the database server are queued. The contention for access to the limited pool of available sockets connecting to the database further increases the concurrency level leading to an amplification of the phenomenon that is similar to a thrashing event. The macroscopic effect of socket shortage is the poor performance of the application server and back-end server components.

The previous finer granularity analysis show two important results. In an e-commerce system there are many token-based resources that are often neglected in the performance studies: ignoring these resources may have serious consequences. There is a high interdependence among the multiple components of an e-commerce system, hence a bottleneck on one component can easily be reflected on other parts of the system with an amplification effect that sometimes makes difficult to understand the initial cause.

6 Conclusions

This paper investigate different granularity levels for the performance evaluation of medium-size e-commerce systems. We discuss the trade-off between the complexity of the analysis and the importance of having detailed results. We also demonstrate that, if a coarse grain view of the system performance may be useful for the bottleneck localization, it is necessary to pass to a fine grain analysis to understand the motivations and remove the

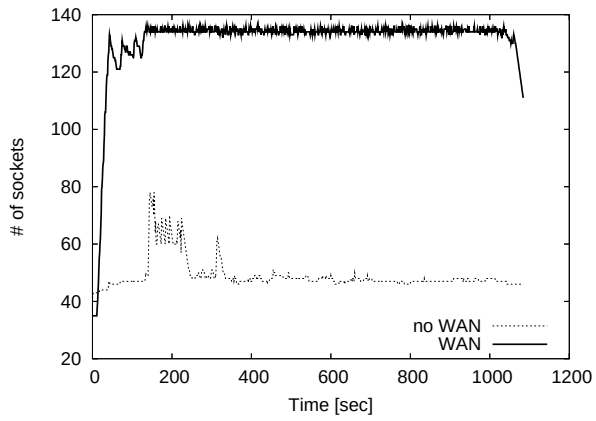

Figure 14: Number of utilized sockets by the back-end server in the WAN and no-WAN scenarios

problems.

We also notice the importance of using percentiles and cumulative distribution functions instead of average values in e-commerce systems that are characterized by complex sub-system interactions, burst arrivals, heavy-tailed workload models.

These premises are allowed us also to evaluate the impact of WAN effects even on the internal layers of the e-commerce system, and the consequences on performance of having a certain number of token-based resources that degrade suddenly.

References

[1] Apache web server, 2004. – http://httpd.apache.org/.

[2] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.

[3] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proc. of 4th Middleware Conference*, Jun 2003.

[4] M. Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10), 2000.

[5] R. C. Dodge, D. A. Menascé, and D. Barbará. Testing e-commerce site scalability with tpc-w. In *Proc. of 2001 Computer Measurement Group Conference*, Dec 2001.

[6] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proc. of the 13th Int'l Conference on World Wide Web (WWW2004)*, May 2004.

[7] S. Godard. Syssyat: System performance tools for linux os, 2004. – http://perso.wanadoo.fr/sebastien.godard/.

[8] X. He and Q. Yang. Performance evaluation of distributed web server architectures under e-commerce workloads. In *Proc. of the 1st Int'l Conference on Internet Computing (IC'2000)*, Jun 2000.

[9] S. Hemminger. netem: Network emulator, 2004. – http://developer.osdl.org/shemminger/netem/.

[10] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the apache web server. *International Journal of Computers and Their Applications*, 8(4), Dec. 2001.

[11] A. Iyengar, R. King, H. Ludwig, and I. Rouvellou. Performance and service level considerations for distributed web applications. In *In Proc. of the 7th World Multiconference on Systems, Cybernetics, and Informatics (SCI 2003)*, Jul 2003.

[12] Java Technology. Java 2 platform, enterprise edition (j2ee), 2004. – http://java.sun.com/j2ee/index.jsp.

[13] K. S. Juse, S. Kounev, and A. Buchmann. Petstore-ws: Measuring the performance implications of web services. In *Proc. of the 29th Int'l Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems - CMG2003*, Dec. 2003.

[14] J. Levon. Oprofile - a system profiler for linux, 2004. – http://oprofile.sourceforge.net/.

[15] Mysql database server, 2004. – http://www.mysql.com/.

[16] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The effects of wide-area conditions on www server performance. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 257–267, 2001.

[17] Php scripting language, 2004. – http://www.php.net/.

[18] H. Xie, L. Bhuyan, and Y.-K. Chang. Benchmarking web server architectures: A simulation study on micro performance. In *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-02), with HPCA-8*, Feb 2002.