An Experimental Evaluation of Performance Problems in HTTP Server Infrastructures Using Online Clients

Ricardo Filipe, Serhiy Boychenko, and Filipe Araujo

CISUC, Dept. of Informatics Engineering
University of Coimbra
Coimbra, Portugal
rafilipe@dei.uc.pt, serhiy@dei.uc.pt, filipius@uc.pt

***Abstract*—Ensuring short response times is a major concern for all web site administrators. To keep these times under control, they usually resort to monitoring tools that collect a large spectrum of system metrics, such as CPU and memory occupation, network traffic, number of processes, etc. Despite providing a reasonably accurate picture of the server, the times that really matter are those experienced by the user. However, not surprisingly, system administrators will usually not have access to these end-to-end figures, due to their lack of control over web browsers. To overcome this problem, we follow the opposite approach of monitoring a site based on times collected from browsers. We use two browser-side metrics for this: i) the time it takes for the first byte of the response to reach the user (request time) and ii) the time it takes for the entire response to arrive (response time). We conjecture that an appropriate choice of the resources to control, more precisely, one or two web pages, suffices to detect CPU, network and disk input/output bottlenecks. In support of this conjecture, we run periodical evaluations of request and response times on some very popular web sites to detect bottlenecks. In this paper, we present a new experiment using pairs of synchronized clients, to extend the results we achieved with single-client requests in our previous work. Results suggest that collecting timing data from the browsers can indeed contribute to detect server performance problems and raise interesting questions regarding unfair delays that seem to exist in some specific requests.**

Keywords–Web monitoring; Client-side monitoring; Bottleneck.

I. INTRODUCTION

In the operation of a Hypertext Transfer Protocol (HTTP) server, bottlenecks may emerge at different points of the system often with negative consequences for the quality of the interaction with users. To control this problem, system administrators must keep a watchful eye on a large range of system parameters, like CPU, disk and memory occupation, network interface utilization, among an endless number of other metrics, some of them specifically related to HTTP, such as response times or queue sizes. Despite being very powerful, these mechanisms cannot provide a completely accurate picture of the HTTP protocol performance. Indeed, the network latency and transfer times can only be seen from the client, not to mention that some server metrics might not translate easily to the quality of the interaction with users. Moreover, increasing the number of metrics involved in monitoring adds complexity to the system and makes monitoring more intrusive. In Section II, we overview different techniques to monitor servers and to detect different sorts of bottlenecks.

We hypothesize that a simpler mechanism, based on client-side monitoring, can fulfill the task of detecting and identifying an HTTP server bottleneck from a list of three: CPU, network, or disk input/output (simply I/O hereafter). The arguments in favor of this idea are quite powerful: client-side monitoring provides the most relevant performance numbers, while, at the same time, requiring no modifications to the server, which, additionally, can run on any technology. This approach can provide a very effective option to complement available monitoring tools.

To achieve this goal, we require two metrics taken from the web browser: i) the time it takes from requesting an object to receiving the first byte (request time), and ii) the time it takes from the first byte of the response, to the last byte of data (response time). We need to collect time series of these metrics for, at least, one or two carefully chosen URLs. These URLs should be selected according to the resources they use, either I/O or CPU. As we describe in Section III, the main idea is that each kind of bottleneck exposes itself with a different signature in the request and response time series.

To try our conjecture, and create such time series, in Section IV, we resorted to experiments on real web sites, by automatically requesting one or two URLs with a browser every minute, and collecting the correspondent request and response times. With these experiments, we managed to discover a case of network bottleneck and another one of I/O bottleneck. We believe that this simple mechanism can improve the web browsing experience, by providing web site developers with qualitative results that add to the purely quantitative metrics they already own.

We now extend these results, which we presented before in [1], with an additional experiment. In Section V, we fetch pages from the same server using two synchronized clients. This enables separation between client-side network and server-side problems. However, the main goal of this experiment was to verify whether observations from one of the clients takes us into a set of conclusions that fits the observations of the second one. Furthermore, to avoid any bias, in Section VI, we introduce a simple algorithm that evaluates request and response times from both clients, before outputting the cause of the problem.

Surprisingly, we noted that, occasionally, the two clients disagree about the quality of the interaction with the server. One of them suffers from an isolated server-side problem, which does not occur again, while the other client does not

suffer from any problem at all. This suggests that some requests get a very unfair treatment along their way. Even a network and server that seem to be lightly loaded can exhibit this sort of delay at times. Determining exactly where and how frequently does this happen is, we believe, an interesting practical concern.

To summarize, this paper makes the following major contributions:

- it proposes a mechanism to detect bottlenecks on HTTP server infrastructures, based on taking periodic client-side metrics;
- it shows evidence of particularly long delays in specific isolated requests.

The rest of the paper is organized as follows. Section II presents the related work in this field and provides a comparison of different methods. Section III describes our conjecture of client-side detection and identification of HTTP server bottlenecks. In Section IV, we show monitoring results from popular web sites, thus exposing different types of bottlenecks. In Section V, we extend our previous work, now using two clients in different networks. In Section VI, we present an automated mechanism to detect bottlenecks. Finally, in Section VII we discuss the results and conclude the paper.

II. Related Work

In the literature, we can find a large body of work focused on timely scaling resources up or down, usually in N-tier HTTP server systems, [2–8]. We divide these efforts into three main categories: (i) analytic models that collect multiple metrics to ensure detection or prediction of bottlenecks; (ii) rule-based approaches, which change resources depending on utilization thresholds, like network or CPU; (iii) system configuration analysis, to ensure correct functionality against bottlenecks and peak period operations.

First, regarding analytic models, authors usually resort to queues and respective theories to represent N-tier systems [9][10]. Malkowski *et al.* [11] try to satisfy service level objectives (SLOs), by keeping low service response times. They collect a large number of system metrics, like CPU and memory utilization, cache, pool sizes and so on, to correlate these metrics with system performance. This should expose the metrics responsible for bottlenecks. However, the analytic model uses more than two hundred application and system level metrics. In [12], Malkowski *et al.* studied bottlenecks in N-tier systems even further, to expose the phenomenon of multi-bottlenecks, which are not due to a single resource that reaches saturation. Furthermore, they managed to show that lightly loaded resources may be responsible for such multi-bottlenecks. As in their previous work, the framework resorts to system metrics that require full access to the infrastructure. The number of system metrics to collect is not clear. Wang *et al.* continued this line of reasoning in [8], to detect transient bottlenecks with durations as low as 50 milliseconds. The transient anomalies are detected recurring to depth analysis of metrics in each component of the system. Although functional, this approach is so fine-grained that it is closely tied to a specific hardware and software architecture.

In [3], authors try to discover bottlenecks in data flow programs running in the cloud. In [7], Bodík *et al.* try to predict bottlenecks to provide automatic elasticity. The work in [6] presents a dynamic allocation of Virtual Machines (VMs) based on Service Level Agreement (SLA) restrictions. The framework consists of a continuous "loop" that monitors the cloud system, to detect and predict component saturation. The paper does not address questions related to resource consumption of the monitoring approach or scalability to large cloud providers. Unlike other approaches that try to detect bottlenecks, [13] uses heuristic models to achieve optimal resource management. Authors use a database rule set that, for a given workload, returns the optimal configuration of the system. The work in [14] presents a technique to analyze workloads using k-means clustering. This approach also uses a queuing model to predict the server capacity for a given workload for each tier of the system.

TABLE I. Bottleneck detection in related work.

Article	CPU/Threads/VM	I/O	Network
[3]	X	X	
[4]	X		
[11]	X	X	
[5]	X		
[8]	X	X	Internal
[12]	X	X	Internal
[17]	X		X
[15]			External

In [15], authors propose a collaborative approach. They use a web browser plug-in on each client, to monitor all Internet activity, gather several network metrics, and send the information to a central point, for processing. The focus of the plug-in is the main web (HTML) page. The impact of this approach on network bandwidth and client data security is unclear, as authors only handle external network connectivity issues.

Other researchers have focused on rule-based schemes to control resource utilization. Iqbal *et al.* [4][16] propose an algorithm that processes proxy logs and, at a second layer, all CPU metrics of web servers. The goal is to increase or decrease the number of instances of the saturated component. Reference [17] also scales up or down servers based on CPU and network metrics of the server components. If a component resource saturation is observed, then, the user will be migrated to a new virtual machine through IP dynamic configuration. This approach uses simpler criteria to scale up or down compared to bottleneck-based approaches, because it uses static performance-based rules.

Table I illustrates the kind of resource problem detected by the mentioned papers. The second column concerns the need to increase CPU resources or VM instances. The third column is associated to I/O, normally an access to a database. The network column represents delays inside the server network or in the connection to the client. It is relevant to mention that several articles [3][12][18] only consider CPU (or instantiated VM) and I/O bottlenecks, thus not considering internal (between the different components) or external (client-server) bandwidth.

Some techniques scan the system looking for misconfigurations that may cause inconsistencies or performance issues. Attariyan *et al.* [20] elaborated a tool that scans the system in real time, to discover root cause errors in the configuration. In [21], authors use previous correct configurations to eliminate unwanted or mistaken operator configuration.

It is also worth mentioning client-side tools like HTTPerf [22] or JMeter [23], which serve to test HTTP servers, frequently under stress, by running a large number of simultaneous invocations of a service. However, these tools work better for benchmarking a site before it goes online. Nevertheless, in [19], we demonstrated that it is possible to detect bottlenecks with limited access to the server using JMeter. However, the bursts of requests of JMeter could hardly work on the real internet, and could potentially be considered as a denial-of-service attack.

Our current work is different from the previously mentioned literature in at least two aspects: we are not tied to any specific architecture and we try to evaluate the bottlenecks from the client's perspective. This point of view provides a better insight on the quality of the response, offering a much more accurate picture regarding the quality of the service. While our method could replace some server-side mechanisms, we believe that it serves better as a complementary mechanism.

III. A Conjecture on Client-Side Monitoring of HTTP Servers

This section presents our conjecture concerning network, CPU and I/O bottleneck identification. The first subsection shows how to identify network bottlenecks and the second subsection shows how to distinguish CPU from I/O bottlenecks.

A. Identification of Network Bottlenecks

We now evaluate the possibility of detecting bottlenecks, based on the download times of web pages, as seen by a client. We conjecture that we can, not only, detect the presence of a bottleneck, something that would be relatively simple to do, but actually determine the kind of resource causing the bottleneck, CPU, I/O or network. CPU limitations may be due to thread pool constraints of the HTTP Server (specially the front-end machines), or CPU machine exhaustion, e.g., due to bad code design that causes unnecessary processing. I/O bottlenecks will probably be related to the database (DB) operation, which clearly depend on query complexity, DB configuration and DB access patterns. Network bottlenecks are related to network congestion.

To illustrate this possibility, we propose to systematically collect timing information of one or two web pages from a given server, using the browser side JavaScript Navigation Timing API [24]. Figure 1 depicts the different metrics that are available to this JavaScript library, as defined by the World Wide Web (W3) Consortium. Of these, we will use the most relevant ones for network and server performance: the request time (computed as the time that goes from the request start to the response start) and the response time (which is the time that goes from the response start to the response end). We chose these, because the request and response times are directly related to the request *and* involve server actions, which is not the case of browser processing times, occurring afterwards, or TCP connection times, happening before.

Consider now the following decomposition of the times of interest for us:

- Request Time: client-to-server network transfer time + server processing time + server-to-client network latency.
- Response Time: server-to-client network transfer time.

To make use of these times, we must assume that the server actions, once the server has the first byte of the response ready, do not delay the network transfer of the response. In practice, our analysis depends on the server not causing any delays due to CPU or (disk) I/O, once it starts responding. Note that this is compatible with chunked transfer encoding: the server might compress or sign the next chunk, while delivering the previous one.

We argue that identifying network bottlenecks, and their cause, with time series of these two metrics is actually possible, whenever congestion occurs in both directions of traffic. In this case, the request and response times will correlate strongly. If no network congestion exists, but the response is still slow, the correlation of request and response times will be small, as processing time on the server dominates. Small correlation points to a bottleneck in the server, whereas high correlation points toward the network. Hence, repeated requests to a single resource of the system, such as the entry page can help to identify network congestion, although we cannot tell exactly where in the network does this congestion occur. Henceforth, we will call "single-page request" analysis to this correlation-based evaluation of the request and response time series from a single URL. In this paper, we improve from our previous work [1], by providing evidence in Section V supporting that, when the service is slow, a high (low) correlation between the request and response times results from network (server) congestion.

B. Identification of CPU bottlenecks

Separating CPU from I/O bottlenecks is a much more difficult problem. We resort to a further assumption here: the CPU tasks share a single pool of resources, possibly with several (virtual) machines, while I/O is often partitioned. This, we believe, reflects the conditions of many large systems, as load balancers forward requests to a single pool of machines, whereas data requests may end up in separate DB tables, served by different machines, depending on the items requested. Since scarce CPU resources affect *all* requests, this type of bottleneck synchronizes all the delays (i.e., different parallel requests tend to be simultaneously slow or fast). Thus, logically, unsynchronized delays must point to I/O bottlenecks. On the other hand, one cannot immediately conclude anything, with respect to the type of bottleneck, if the delays are synchronized (requests might be suffering either from CPU or similar I/O limitations).

The challenge is, therefore, to identify pairs of URLs showing unsynchronized delays, to pinpoint I/O bottlenecks. Ensuring that a request for an URL has I/O is usually simple, as most have. In a news site, fetching a specific news item will most likely access I/O. To have a request using only CPU or, at least, using some different I/O resource, one might fetch non-existing resources, preferably using a path outside the logic of the site. We call "independent requests" to this mechanism of using two URLs requesting different types of resources.

One should notice that responses must occupy more than a single TCP [25] segment. Otherwise, one cannot compute any meaningful correlation between request and response times, as this would always be very small.

In our experiments, we will start by evaluating the correlation between request and response times. Then, we will experimentally try the "single-page request" and the "independent

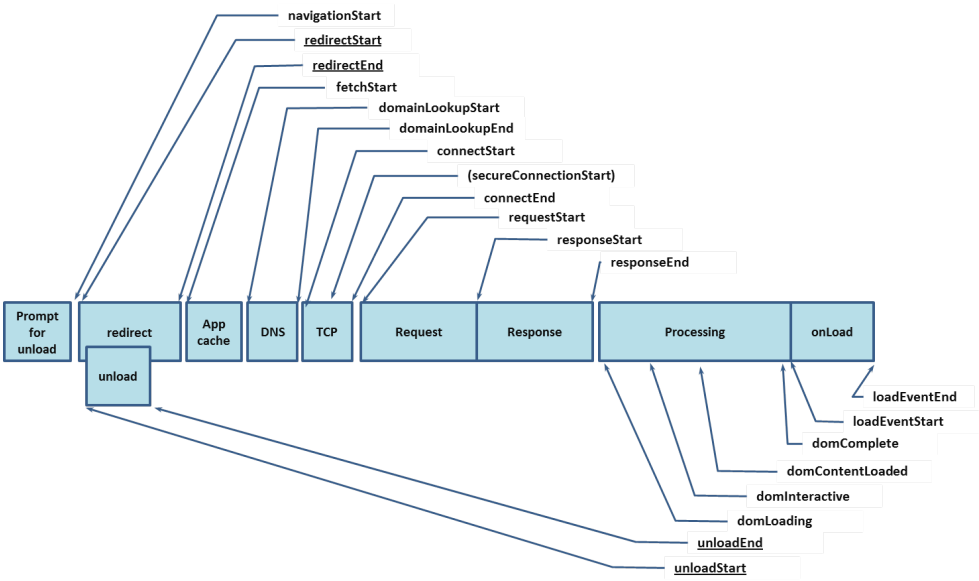

Figure 1. Navigation Timing metrics (figure from [24])

TABLE II. SOFTWARE USED AND DISTRIBUTION.

Component	Observations	Version
Selenium	selenium-server-standalone jar	2.43.0
Firefox	browser	23.0
Xvfb	xorg-server	1.13.3

requests" mechanisms, to observe whether they can actually spot bottlenecks in real web sites.

IV. EXPERIMENTAL EVALUATION

In this section, we present the results of our experimental evaluation. First, we present the setup and afterwards the most important results obtained with the experiments.

A. Experimental Setup

For the sake of doing an online analysis, we used a software testing framework for web applications, called Selenium [26]. The Selenium framework emulates clients accessing web pages using the Firefox browser, thus retaining access to the Javascript Navigation Timing API [24]. We use this API to read the request and response times necessary for the "single-page request" and "independent requests" mechanisms. We used a UNIX client machine, with a crontab process, to request a page each minute [27], using Selenium and the Firefox browser. We emulated a virtual display for the client machine using Xvfb [28]. Table II lists the software and versions used.

One of the criteria we used to choose the pages to monitor was their popularity. However, to conserve space, we only show results of pages that provided interesting results, thus omitting sites that displayed excellent performance during the entire course of the days we tested (e.g., CNN [29] or Amazon [30]) — these latter experiments would have little to show regarding bottlenecks. On the other hand, we could find some bottlenecks in a number of other real web sites:

- **Photo repository** — We kept downloading the same 46 KiloBytes (KiB) Facebook photo, which was actually delivered by a third-party provider Content Delivery Network (CDN). During the time of this test, the CDN was retrieving the photo from Ireland. This experiment displays network performance problems.
- **Portuguese News Site** — this web page is the 5th most used portal in Portugal (only behind Google – domain .pt and .com, Facebook and Youtube) and the 1st page of Portuguese language in Portugal [31]. This web page shows considerable performance perturbations on the server side, especially during the wake up hours.
- **Portuguese Sports News** — This is an online sports newspaper. We downloaded an old 129 KiB news item and an inexistent one for several days. The old news item certainly involves I/O, to retrieve the item from a DB, whereas the inexistent may or may not use I/O, we cannot tell for sure. We ensured a separation of 10 seconds between both requests. One should notice that having a resource URL involving only CPU would be a better choice to separate bottlenecks. However, since we could not find such resource, a non-existing one actually helped us to identify an I/O bottleneck.
- **Social Network Site** — We used the 1st popular social network and the largest social network worldwide. The technology demands are enormous to ensure quality-of-experience to their users and, therefore, preventing bottleneck occurrences. However, recent blackouts in the system have shown the potential of our tool to detect system anomalies and predict web page disruptions.

B. Results

We start by analyzing the results from the Content Delivery Network and from Portuguese News site, in Figures 2, 3, and 4. These figures show the normal behavior of the systems and enable us to identify periods where the response times fall out of the ordinary.

Figure 2 shows the response of the CDN site for a lapse of several days. We can clearly observe a pattern in the

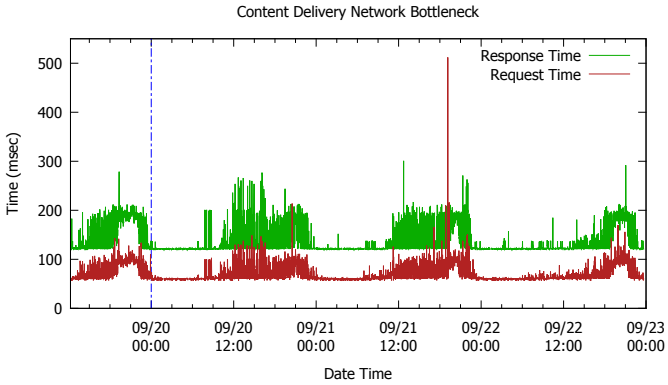

Figure 2. CDN bottleneck.

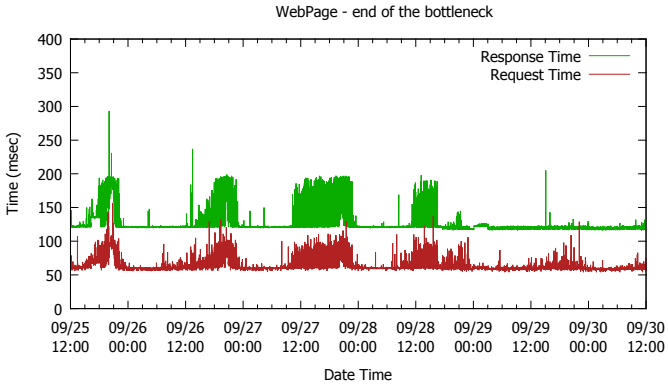

Figure 3. CDN - end of the bottleneck.

response that is directly associated to the hour of the day. During working hours and evening in Europe, we observed a degradation in the request and response times (see, for example, the left area of the blue line on September 19, 2014, a Friday). The green and the red lines (respectively, the response and the request times), clearly follow similar patterns, a sign that they are strongly correlated. Computing the *correlation coefficient* of these variables, $r(Req, Res)$, for the left side of the blue line we have $r(Req, Res) = 0.89881$, this showing that the correlation exists indeed. However, for the period where the platform is more "stable" (between the first peak periods) we have $r(Req, Res) = -0.06728$. In normal conditions the correlation between these two parameters is low. This allows us to conclude that in the former (peak) period we found a network bottleneck that does not exist in the latter. However, our method cannot determine where in the network is the bottleneck. Interestingly, in Figure 3, we can observe that the bottleneck disappeared after a few days. On September 29^{th}, we can no longer see any sign of it.

Regarding Figure 4, which shows request and response times of the main page of a news site, we can make the same analysis for two distinct periods: before and after 9 AM (consider the blue vertical line) of December 13, 2013 (also a Friday). Visually, we can easily see the different profiles of the two areas. Their correlations are:

- $r(Req, Res)_{before9AM} = 0.36621$
- $r(Req, Res)_{after9AM} = 0.08887$

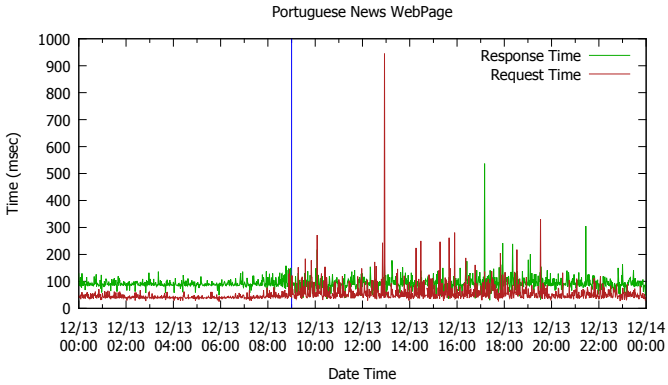

Figure 4. Portuguese News Site bottleneck.

The correlation is low, especially during the peak period, where the response time is more irregular. This case is therefore quite different from the previous one, and suggests that no network bottleneck exists in the system, during periods of intense usage. With the "single-page request" method only, and without having any further data of the site, it is difficult to precisely determine the source of the bottleneck (CPU or I/O).

To separate the CPU from the I/O bottleneck, we need to resort to the "independent requests" approach, which we followed in the Portuguese Sports News case. Figures 5, 6, 7, and 8 show time series starting on February 18^{th}, up to February 21^{st} 2015. We do not show the response times of the inexistent page as these are always 0 or 1, thus having very little information of interest for us. In all these figures, we add a plot of the moving average with a period of 100, as the moving average is extremely helpful to identify tendencies.

Figures 5 and 6 show the request time of the old 129 KiB page request. The former figure shows the actual times we got, whereas in the latter we deleted the highest peaks (those above average), to get a clearer picture of the request times. A daily pattern emerges in these figures, as daytime hours have longer delays in the response than night hours. To exclude the network as a bottleneck, we can visually see that the response times of Figure 7 do not exhibit this pattern, which suggests a low correlation between request and response times (which is indeed low). Next, we observe that the request times of the existent and inexistent pages (refer to Figure 8) are out of sync. The latter seems to have much smaller cycles along the day, although (different) daily patterns seem to exist as well. For the reasons we mentioned before, in Section III, under the assumption that processing bottlenecks would *simultaneously* affect both plots, we conclude that the main source of bottlenecks in the existent page is I/O. This also suggests the impossibility of having the request time dominated by access to a cache on the server, as this would impact processing, thus causing synchronized delays. A final word for the peaks that affect the request time: they weakly correlate with response times. Hence, their source is also likely to be I/O.

Figures 9 and 10 show a period when the social network web page was down in the entire world, due to a system misconfiguration. Figure 9 shows how the page behaved, regarding request and response times — before, during and

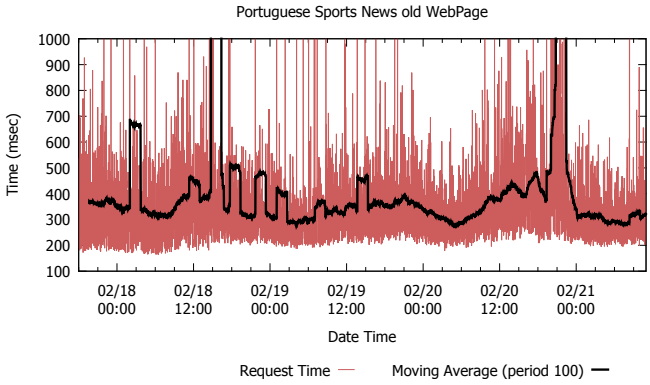

Figure 5. Portuguese Sports News old page — request times.

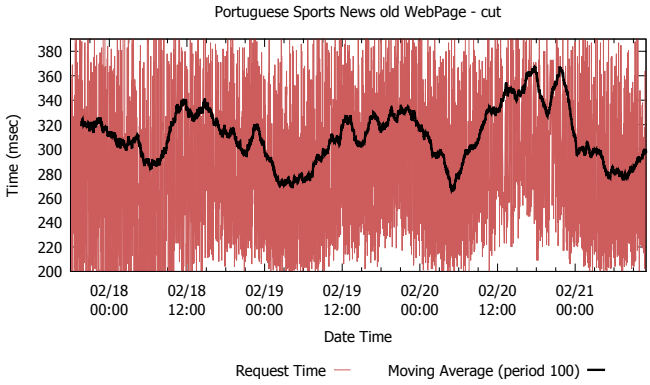

Figure 6. Portuguese Sports News old page — request times with peaks cut.

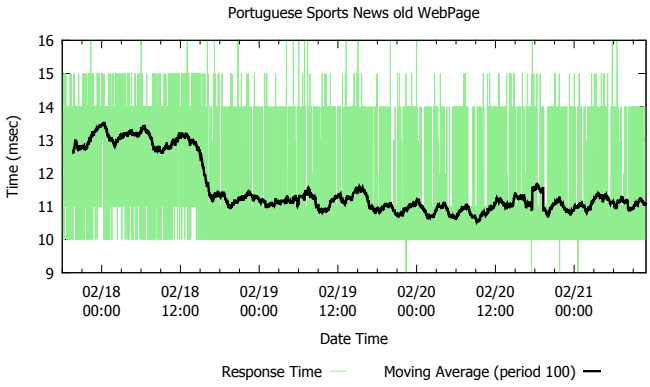

Figure 7. Portuguese Sports News old page — response times.

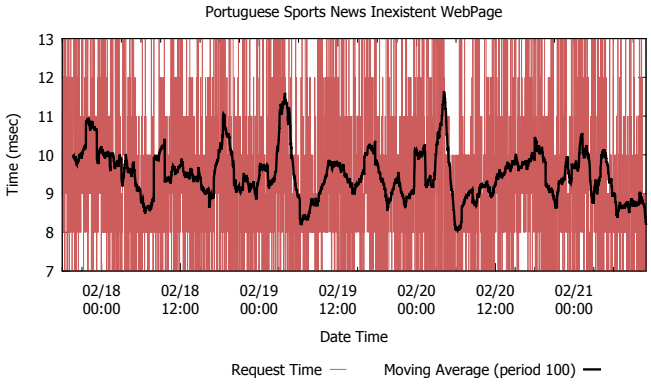

Figure 8. Portuguese Sports News inexistent page — request times.

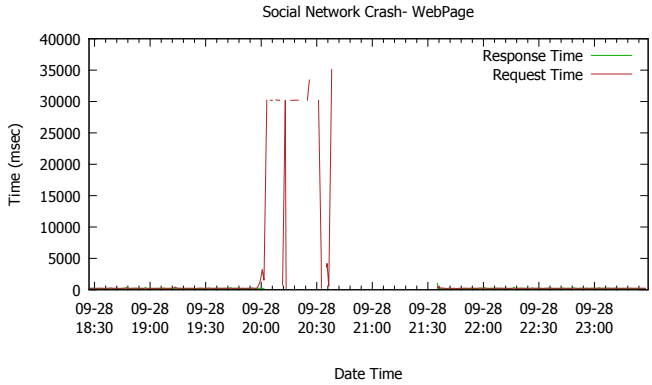

Figure 9. Social Network Web Page crash.

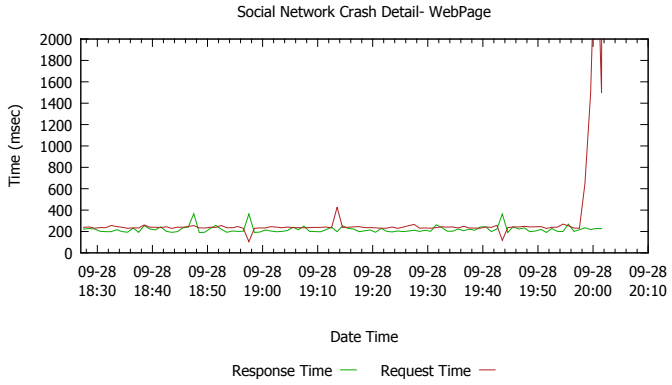

Figure 10. Social Network Web Page crash detail.

after the system resumed responding correctly. Figure 10 gives a closer look of the period before the web page failure. Time periods without request or response times, occurred when the client reached the configured timeout and aborted the web page request. Currently, the timeout is configured to be 60 seconds. Analyzing Figure 9, we can identify the period of time when the web page was down or responding incorrectly. This might be important, if the web page is hosted in a third-party provider that might be held responsible for the failure and the user wants to complain for a refund [32–34]. Figure 10 gives a closer look of the minutes before the failure. The request time increased significantly, while the response time remained unchanged — a weak correlation that points to a server-side bottleneck. This agrees with what was mentioned by the media [35]. Additionally, prior to the complete failure, we can observe a 5-minute window, were the problem was starting to become apparent, and that could be used to fix the problem or alert system administrators.

V. Client-Side Monitoring Using Two Distinct Networks

In Section III, we focused on the problem of detecting bottlenecks using only client metrics. We used a time series approach based on the assumption that we can distinguish server from network congestions, by looking at the correlation between the request and response times. Unfortunately, since we have no access to the internal server-side data of the web servers we monitored, we cannot directly confirm this

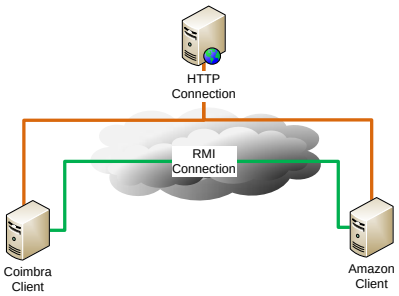

Figure 11. Experimental Setup - 2 clients

correlation hypothesis. To confirm this hypothesis and thus verify the feasibility of this method, we now resort to two distinct clients. The idea is straightforward: the observation from two different clients should be coherent; clients should agree on whether a bottleneck exists, and where does it come from. As we shall see, the results we got slightly deviated from what we expected, thus raising a very interesting question.

A. Experimental Settings

We used the same technologies mentioned in Table II for each client. This means that each client was running a Selenium instance to invoke a specific web page, through Firefox. However, having two unsynchronized clients would invalidate the results, because we would not know if the requests were made at the same time. To eliminate this limitation we created a communication protocol between the clients in Java RMI [36]. Before fetching a page, the two clients communicate with each other, to determine which page to get and to ensure some degree of synchronicity. The process consists in 3 steps — first, client A notifies client B to invoke a determined URL; second, both clients invoke the URL and save the request and response time; and finally, client A receives the data from the web page invocation from client B. One of the clients (client A in this description) has the role of "master", triggering the web page invocation and the collection data from both requests. Clients should be in different locations to have different network connectives. We picked our own department facilities in Coimbra, and a virtual machine in the Amazon Web Service cloud in the Northern Virginia Region [37]. Figure 11 illustrates this interaction — the RMI connection is identified in green, whereas the HTTP connections to the web page are depicted in orange.

One of the criteria we used to choose the pages to monitor was their popularity. Additionally, the web page should have the same location regardless of the origin of the client (Amazon in America or Coimbra in Europe). To ensure this, we compared the IP given by the DNS to each client, to ensure that they were, indeed, monitoring the same server. A second criterion was to monitor web pages from different geographic locations. However, to conserve space, we only show results of pages that provided interesting results. Among these, we could find some bottlenecks in the following real web sites:

- **American electronic commerce and cloud computing company** — We kept downloading the main page from this popular web page hosted in the United States from our two clients. This experiment displays a significant performance improvement, at some point in time. This improvement was observed in both clients.
- **Chinese Search Engine** — this web page is the front-end for one of the most popular search engines in the world [31]. It is hosted in China. This web page shows some network perturbations during a specific time in both clients.
- **Portuguese Sports News** — This is an online sports newspaper already used in Section III. The web page is located in Portugal (Europe). We downloaded the main page during several days. We verified several pattern changes associated with system bottlenecks in both clients.

B. Results

Since we now have two clients invoking the same URL, at the same time, we expect similar server response patterns at both clients. One would assume that whenever the response time pattern in only one of the clients changes, the difference should result from some bottleneck in the client-server network path that is specific to the client observing the change. However, if both clients observe a modification in the response patterns (in terms of request and response time series), we can conclude that this is the result of a component that is common to both clients. Hence, this is the result of a system bottleneck or a common network path.

We will now experimentally try the two clients "synchronous request" mechanism, to observe whether they can actually spot bottlenecks in real web sites and achieve consistent observation of response patterns.

We start by analyzing the results of the American electronic commerce web page in Figure 12, which shows the response of the main page for a lapse of several days. The pair of figures mostly shows normal behavior seen in Coimbra and in the AWS, thus allowing us to identify periods when the response times fell out of the ordinary for one or both clients. We can clearly observe a pattern in the response that is directly associated to the hour of the day. Additionally, the pattern exists for both clients, this meaning that both were experiencing the same constraints (system or network) from the web page. To have a better understanding of the trends, we also show a moving average of the last 100 samples of response time, in black. Computing the *correlation coefficient* for the response and request times for both clients, $r(Req, Res)$, for the interval between September 12^{th} 13:00 and September 13^{th} 13:00 2015, we get a correlation of $r(Req, Res)_{Coimbra} = 0.13896$ and $r(Req, Res)_{AWS} = -0.07370$. Since none of the clients observed significant congestion conditions, these low correlations provide us very little information and suggest a normal behavior of the system. Near the end of the experiment, still in Figure 12, we can see an improvement in the response times of both clients. Calculating the *correlation coefficient* for this period, we have $r(Req, Res)_{Coimbra} = 0.07974$ and $r(Req, Res)_{AWS} = 0.14808$. Hence, having in consideration the low correlations and what was mentioned in Section III, we can infer that the improvement experienced by both clients seems to be a consequence of a change in the American electronic commerce web page. An improvement in the system network is less likely, because the request time rested unchanged for this period.

Figure 13 shows the request and response times of the Chinese Search Engine web page. The web page presents a relatively stable pattern during most of the days. During

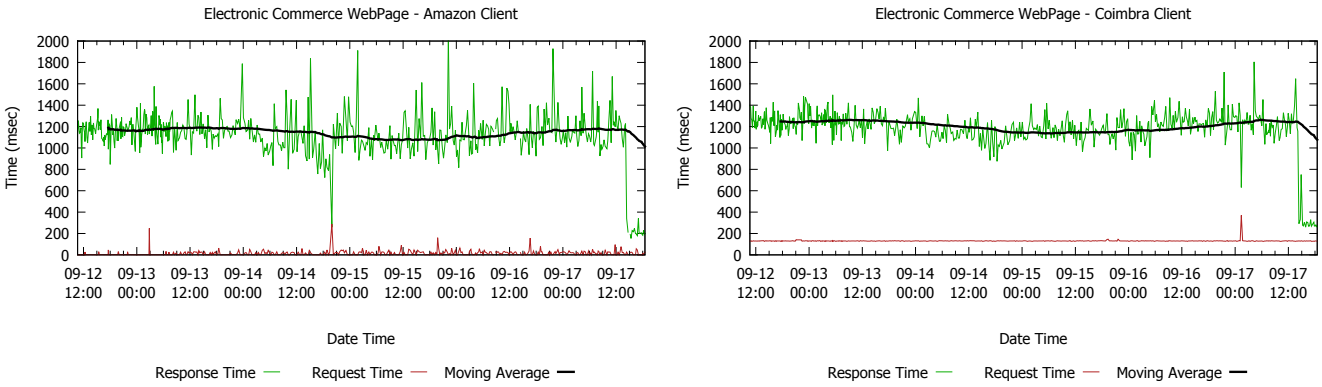

Figure 12. American electronic commerce web page

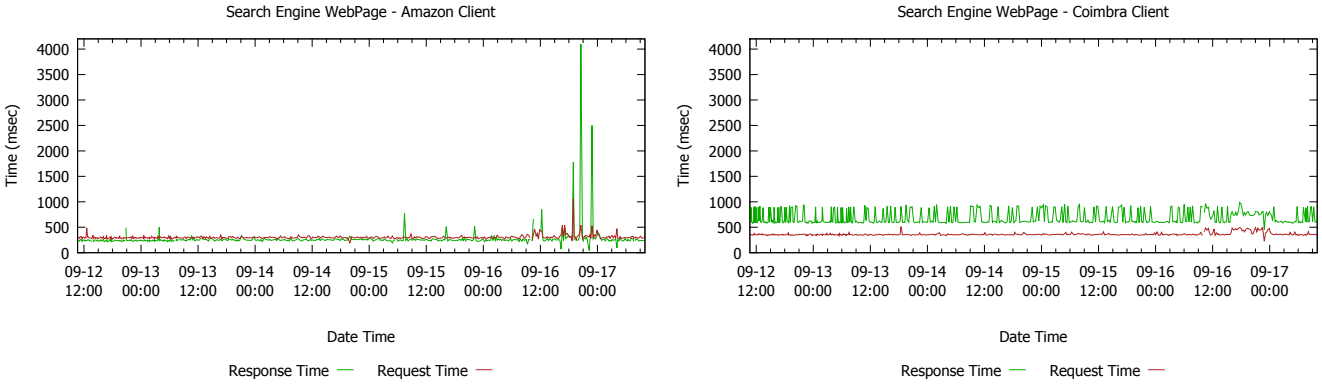

Figure 13. Chinese search engine web page

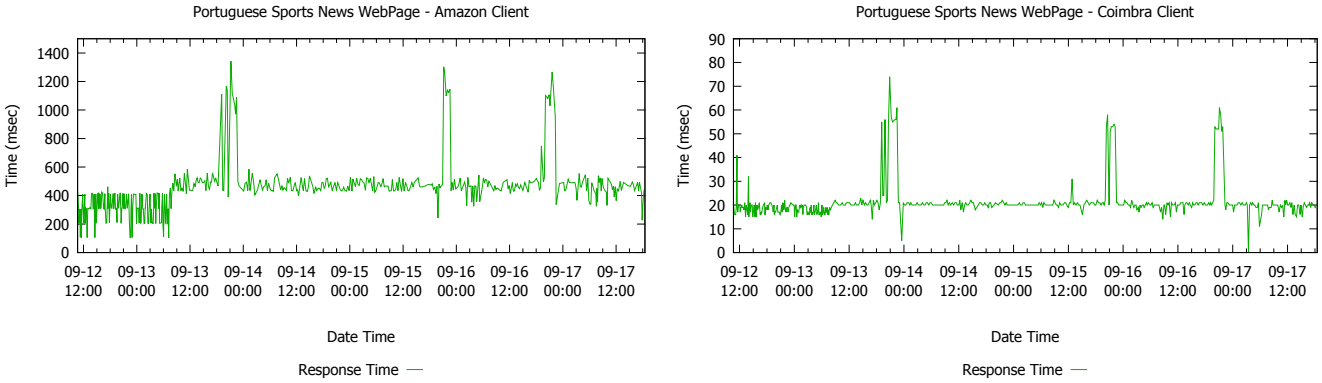

Figure 14. Portuguese sports news web page

this period (before September 16^{th}), the *correlation coefficient* was $r(Req, Res)_{Coimbra} = 0.04532$ and $r(Req, Res)_{AWS} = 0.16566$, this meaning that in normal conditions there was no correlation between request and response times. However, for the period after September 16^{th}, there was a significant change observed by the AWS client, and although less significant, also by the Coimbra client. This is even clearer when we calculate the *correlation coefficient* of both clients for this period. The correlation in Coimbra was $r(Req, Res)_{Coimbra} = 0.69707$ and in AWS $r(Req, Res)_{Aws} = 0.57794$. This means that the correlation for request and response times in both clients increased significantly, when compared to the normal pattern. Hence, taking in consideration what was mentioned in Section III, we are, most likely, observing a network bottleneck in a common path between the server and the clients.

Figure 14 shows a degradation of the response time in 3 distinct moments. This degradation was observed in both clients at the same time. When we calculate the correlation

TABLE III. CORRELATION FOR THE 3 PEAKS OF PORTUGUESE SPORTS NEWS SITE

Correlation	Coimbra	AWS
$r(Req, Res)_{1^{st}peak}$	-0.32036	0.35263
$r(Req, Res)_{2^{nd}peak}$	0.33789	0.35815
$r(Req, Res)_{3^{rd}peak}$	0.28955	0.15749

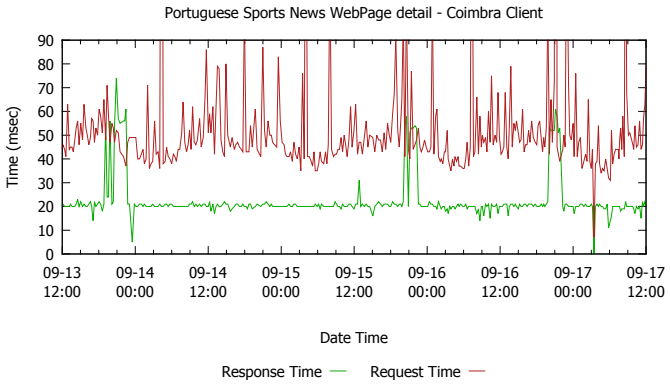

Figure 15. Portuguese Sports News Web Page - Detail

for a stable period of good performance (e.g., between the first and second peak), we get $r(Req, Res)_{Coimbra} = -0.02381$ in Coimbra and $r(Req, Res)_{AWS} = 0.17699$ in the AWS. Then, for the three observed peaks, we have the correlation values of Table III.

This *correlation coefficient* for both clients in the three peaks differs considerably, especially in Coimbra. We show with finer detail the request and response times of this client in Figure 15. The correlation is never very high, thus pointing to a server problem, especially in the first and third peaks. However, it is also never low, thus suggesting that the network, more specifically a common path of the network might have been affected as well, at least in the second peak.

VI. AUTOMATIC DETECTION OF BOTTLENECKS USING TWO DISTINCT NETWORKS

Our next step was to do a simple automated mechanism to detect bottlenecks, using information coming from the pair of clients. This is beneficial, not only because automation may eventually lead to a quicker detection of performance problems, but mainly because a visual inspection as we did in the previous sections is error-prone and is subject to all sort of biases. Indeed, with this new scheme we were able to achieve new conclusions and get a deeper insight of performance bottleneck problems.

A. Overview

The pair of clients provides four different variables that we can feed into an algorithm: a boolean value per client telling whether or not the client sees a congestion; and the correlation between the request and the response times, for both clients. Determining if the client is observing a congestion in the service is not trivial, in the sense that different algorithms may respond differently. In our experiments, we used an algorithm based on a moving average. Unlike congestion, the correlation is easier to determine. As before, we use the last 100 metrics

TABLE IV. All possible combinations of congestion and correlation

Client 1		Client 2		Cause
Congestion?	Correlation	Congestion?	Correlation	
No	Irrelevant	No	Irrelevant	No Bottleneck
No	Irrelevant	Yes	Low	Impossible
No	Irrelevant	Yes	High	Client 2's Network
Yes	Low	Yes	Low	Server
Yes	Low	Yes	High	Server & C. 2 Net.
Yes	High	Yes	High	Common Network

of request and response times. Note that for the correlation to tell us something, we must be careful enough to request pages that are relatively large, but still go in a single non-chunked HTTP message. Considering high and low correlations, we get 16 possible combinations of the four variables, of which we arrange the cases of interest in Table IV. We omit the redundant cases, where client 1 and client 2 would simply swap their variables. For example, since we already have "Yes, Low, Yes, High", in the line before last, it would be pointless to include a "Yes, High, Yes, Low".

Line 1 of the table is pretty much trivial: none of the clients observes a bottleneck, therefore, looking for correlation is not relevant. In line 2, one of the nodes observes a congestion that does not come from the network (low correlation). Hence, the other node should also observe a congestion. Hence, this line is seemingly impossible. However, as we shall discuss, it may, in fact, happen. In line 3, the client observing the congestion can tell from the correlation that the network is congested. Since client 1 observes no congestion, the network of client 2 is the culprit. The following case is equally straightforward: when both clients observe a congestion and a low correlation, the server is the culprit. In the following case, more than one bottleneck exists: client 1 can tell that the server is the most likely cause for the low correlation between request and response times, because responses are taking quite a long time. On the other hand, client 2 is also observing congestion, but it can see a high correlation in the request and response times, thus concluding that the problem lies in the network. Despite seeming contradictory, this is possible, if both the server and client 2 network are sources for delays. In the final line, the problem lies in the network that is common to clients 1 and 2. We will now try to confirm to what extent do real observations actually fit into this model.

B. Algorithm to detect bottlenecks

In this section, we present an algorithm that evaluates the variables of Table IV and outputs the cause of the problem. The algorithm combines the request and response time series retrieved at the same time, by two distinct clients for the same web page, collected in our "synchronous request" experiments. We wrote the algorithm in Python and we describe its high-level details in pseudo-code in Algorithm 1. The expression CongestionClient (1 or 2) $\geq$ CongestionThreshold becomes true when the moving average of the request plus receive time of the last 100 values grows above 15% of the average of all samples, for that client. We consider the threshold that splits low from high correlation to be 0.2. The correlation also takes into account the last 100 measurements.

C. Results

In this section, we present some of the results that we obtained with Algorithm 1 and compare them to the data previously analyzed in Section V. Although we ran our algorithm

Algorithm 1 Identify Bottleneck

```
if CongestionClient1 ≥CongestionThreshold then
  if CongestionClient2 ≥CongestionThreshold then
    if CorrelClient1 ≥HighCorrelationThreshold then
      if CorrelClient2 ≥HighCorrelationThreshold then
        Common Network
      else
        Client 1's Network and Server
      end if
    else if CorrelClient2 ≥ HighCorrelationThreshold
    then
      Client 2's Network and Server
    else
      Server
    end if
  else
    if CorrelClient1 ≥ HighCorrelationThreshold then
      Client 1's Network
    else
      Impossible
    end if
  end if
else if CongestionClient2 ≥ CongestionThreshold then
  if CorrelClient2 ≥HighCorrelationThreshold then
    Client 2's Network
  else
    Impossible
  end if
else
  No Bottleneck
end if
```

under varied conditions, we will focus on its responses for the inputs depicted in Figures 12, 13, and 14. We can say that all the bottlenecks we identified by visual inspection in these figures were also identified by the algorithm, which pointed out the same sources for problems. Although this might result from the specific thresholds we selected and from the size of the sliding window in the moving average and network correlation, the effort of tuning the algorithm and making it run under real data allowed us to reach two results:

- The algorithm cannot cope with request or response times that are too low. For instance, if the request or response times fall to values near the millisecond range, as in one case where the client and server were very close to each other, any small increase in the response time, no matter how small it is, will look as a congestion.
- The algorithm tags some congestion cases as being impossible (line 2 in Table IV). This happens because some requests take so long to get an answer that they are able to push the moving average above the congestion threshold. The interesting thing is that this sort of delay, which happens rarely, is usually not seen by the peer client, which is fetching pages from the same server; and it is not seen neither before, nor after, by the same client. This suggests that some requests get an unfair amount of wait. If this is really the case, what is the source of the problem (network, CPU, or I/O) and how often does this happen remains as an open question.

VII. Discussion and Conclusion

We proposed to detect bottlenecks on HTTP servers using client-side observations of request and response times. A comparison of these signals, either over the same, or a small number of resources, enables the identification of CPU, network and I/O bottlenecks. We did this work having no access to internal server data and mostly resorting to visual inspection of the request and response times. If run by the owners of the site, we see a number of additional options:

- Simply follow our approach of periodically, invoking URLs in one or more clients, as a means to complement current server-side monitoring tools. This may help to reply to questions such as "what is the impact of a CPU occupation of 80% for interactivity?".
- A hybrid approach, with client-side and server-side data is also possible. I.e., the server may add some internal data to *each* request, like the time the request takes on the CPU or waiting for the database. Although much more elaborate and dependent on the architecture, instrumenting the client and the server sides is, indeed, the only way to achieve a full decomposition of request timings.
- To improve the quality of the analysis we did in Section IV, site owners could also add a number of very specific resources, like a page that has known access time to the DB, or known computation time.
- It is also possible to automatically collect timing information from real user browsers, as in Google Analytics [38], to do subsequent analysis of the system performance. In other words, instead of setting up clients for monitoring, site owners might use their real clients, with the help of some JavaScript.

In summary, we collected evidence in support of the idea of identifying bottlenecks from the client side. In our previous work [1], we recognized that to unambiguously demonstrate these results we needed further evidence from a larger number of sites, and from supplementary server data. We now managed to run several experiments with a second client. Although mostly concurring with our initial observations, the second client opened an entirely new perspective: sometimes one of the clients observes a delay in a single very concrete request, which is neither observed by the other client, nor by the client itself, either before or after. I.e., even when the server seems to be delivering a normal service, clients may occasionally fail to receive a response in reasonable time. We are certain that the problem does not come from the network, because we classify network problems in a different category. This result agrees with [1]. We thus know that the server itself is the culprit for such delays.

Let us think for a moment on the unique route taken by each request: the TCP connection takes the request up to the server, where some thread reads it, processes it, and (most likely) forwards it to another layer of the system, where some thread will eventually fetch several items from the database, before enqueuing or sending the response back to the client. Each request might follow slightly different routes, depending on the threads that get it. This suggests a simple, but significant conclusion: a few unlucky requests get blocked at some point

inside the server. To be fair, there was never any guarantee that *all* requests would get their fair chance, or that they would all get a quick response. But observing such cases in a moderate number of samples is, we think, a rather interesting result. This observation raises the question of determining the exact mechanism behind starvation of some specific requests, and how likely is such mechanism to come into play.

REFERENCES

[1] R. Filipe, S. Boychenko, and F. Araujo, "Online client-side bottleneck identification on HTTP server infrastructures," in *The Tenth International Conference on Internet and Web Applications and Services (ICIW 2015)*, Brussels, Belgium, June 2015, pp. 22–27.

[2] *RFC 2616 - Hypertext Transfer Protocol – HTTP/1.1*, Internet Engineering Task Force (IETF), Internet Engineering Task Force (IETF) Std., June 1999. [Online]. Available: http://www.faqs.org/rfcs/rfc2616.html

[3] D. Battre, M. Hovestadt, B. Lohrmann, A. Stanik, and D. Warneke, "Detecting bottlenecks in parallel dag-based data flow programs," in *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, 2010, pp. 1–10.

[4] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[5] Y. Shoaib and O. Das, "Using layered bottlenecks for virtual machine provisioning in the clouds," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, 2012, pp. 109–116.

[6] N. Huber, F. Brosig, and S. Kounev, "Model-based self-adaptive resource allocation in virtualized environments," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 90–99. [Online]. Available: http://doi.acm.org/10.1145/1988008.1988021

[7] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for Internet datacenters," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855533.1855545

[8] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, July 2013, pp. 31–40.

[9] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Autonomic Computing, 2007. ICAC '07. Fourth International Conference on*, June 2007, pp. 27–27.

[10] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, "Layered bottlenecks and their mitigation," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, Sept 2006, pp. 103–114.

[11] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, "Bottleneck detection using statistical intervention analysis," in *Managing Virtualization of Networks and Services*. Springer, 2007, pp. 122–134.

[12] S. Malkowski, M. Hedwig, and C. Pu, "Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 118–127.

[13] R. Chi, Z. Qian, and S. Lu, "A heuristic approach for scalability of multi-tiers web application in clouds," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, 2011, pp. 28–35.

[14] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809053

[15] S. Agarwal, N. Liogkas, P. Mohan, and V. Padmanabhan, "Webprofiler: Cooperative diagnosis of web failures," in *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, Jan 2010, pp. 1–11.

[16] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Sla-driven automatic bottleneck detection and resolution for read intensive multi-tier applications hosted on a cloud," in *Advances in Grid and Pervasive Computing*. Springer, 2010, pp. 37–46.

[17] H. Liu and S. Wee, "Web server farm in the cloud: Performance evaluation and dynamic architecture," in *Proceedings of the 1st International Conference on Cloud Computing*, ser. CloudCom '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 369–380. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10665-1_34

[18] B. Singh and P. Nain, "Article: Bottleneck occurrence in cloud computing," *IJCA Proceedings on National Conference on Advances in Computer Science and Applications (NCACSA 2012)*, vol. NCACSA, no. 5, pp. 1–4, May 2012, published by Foundation of Computer Science, New York, USA.

[19] R. Filipe, S. Boychenko, and F. Araujo, "On client-side bottleneck identification in HTTP servers," in *Proceedings of the 5th INForum — Simpósio de Informática*, Covilh, Portugal, September 2015.

[20] M. Attariyan, M. Chow, and J. Flinn, "X-ray: automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 307–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387910

[21] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Barricade: defending systems against operator mistakes," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 83–96. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755924

[22] "Papers — HP Web server performance tool," http://www.hpl.hp.com/research/linux/httperf/, retrieved: May, 2015.

[23] "Performance tools — Apache JMeter^{TM}," http://jmeter.apache.org/, retrieved: May, 2015.

[24] "Papers — Navigation Timing," https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html, retrieved: May, 2015.

[25] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[26] "Papers — Selenium Browser automation," http://www.seleniumhq.org/, retrieved: May, 2015.

[27] "Crontab - quick reference — admin's choice - choice of unix and linux administrators," http://www.adminschoice.com/crontab-quick-reference, retrieved: May, 2015.

[28] "Xvfb," http://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml, retrieved: May, 2015.

[29] "Breaking news, u.s., world, weather, entertainment & video news - cnn.com," http://edition.cnn.com, retrieved: May, 2015.

[30] "Amazon.com: Online shopping for electronics, apparel, computers, books, dvds & more," http://www.amazon.com, retrieved: May, 2015.

[31] "Alexa — Top Sites in Portugal," http://www.alexa.com/topsites/countries/PT, retrieved: May, 2015.

[32] "Papers — Windows Azure Service Level Agreement," http://www.windowsazure.com/en-us/support/legal/sla/, retrieved: May, 2015.

[33] "Papers — HP Service Level Agreement," https://www.hpcloud.com/SLA, retrieved: May, 2015.

[34] "Papers — Amazon EC2 Service Level Agreement," http://aws.amazon.com/ec2-sla/, retrieved: May, 2015.

[35] "Facebook crash," http://www.dailymail.co.uk/sciencetech/article-3252603/Facebook-goes-Social-network-crashes-time-month-leaving-users-panic.html, retrieved: Nov, 2015.

[36] "Papers — RMI overview," https://docs.oracle.com/javase/tutorial/rmi/overview.html, retrieved: Nov, 2015.

[37] "Papers — Amazon Web Services," http://aws.amazon.com/, retrieved: Nov, 2015.

[38] B. Clifton, *Advanced Web Metrics with Google Analytics*. Alameda, CA, USA: SYBEX Inc., 2008.