Client-Side Black-Box Monitoring for Web Sites

Ricardo Filipe, Rui Pedro Paiva and Filipe Araujo

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

***Abstract*—In spite of their growing maturity, current web monitoring tools are unable to observe all operating conditions. For example, clients in different geographical locations might get very diverse latencies to the server; the network between client and server might be slow; or third-party servers with external page resources might underperform. Ultimately, only the clients can determine whether a site is up and running in good conditions.**

In this paper, we use the response times experienced by clients, to infer about server and network performance. The goal is to detect internal and external bottlenecks doing black-box monitoring, in particular CPU (internal) and network (external). We aim to determine to what extent are the clients able to tell one type of bottleneck from the other, i.e., what kind of information do the server and network leak, regarding their operating conditions.

To answer this question, we resort to an empirical approach. We submit an HTTP server and network to a large number of operating conditions and train two machine learning algorithms, a linear and a non-linear one, to identify the cause of the congestion affecting the system. Results show that the server and network leak information to a level of detail that allows sorting out CPU from network bottlenecks, or even a combination of the two, in a large spectrum of cases. This suggests that a black-box monitoring approach is not only possible, but promising, as it may complement traditional white-box approaches.

***Index Terms*—Black-box monitoring; Client-side monitoring; Web-site monitoring; Analytics**

I. Introduction

The Hypertext Transfer Protocol [1] (HTTP) is used by millions around the globe on a daily basis and makes up for a large fraction of the Internet traffic. In the HTTP ecosystem, networks and servers are crucial, as the quality of interactions depends on their quick responses to requests. Developers and administrators must keep a watchful eye on applications, to detect and mitigate server and infrastructure bottlenecks, as accurately and swiftly as possible.

Unix monitoring tools, Application Performance Monitoring (APM) or Real User monitoring (RUM) suites can help with this goal. Unix boxes usually ship with a plethora of monitoring tools, such as `top`, `lsof`, or `netstat`. With these, system administrators can observe a large array of metrics out-of-the-box, such as memory or CPU occupation by process, open files, network connections, and so on. On the other extreme, Application Performance Monitoring suites, like New Relic [2], AppDynamics [3] or Dynatrace [4], tend to be a lot more intrusive, but provide impressive results in return. They launch agents to reverse engineer distributed systems, provide information on their architecture, and access logs to correlate failures with performance metrics. While the former Unix tools cannot grasp the users' quality of experience, the latter commercial APM suites can amass abundant information on the application, for some programming languages, usually at the expense of significant resources and configuration.

Unfortunately, even the most intrusive monitoring suites tend to ignore the actual clients' waiting times, often a result of their real locations and network conditions. Furthermore, some resources on a web page are served from third-party providers, thus slipping under the radar of server-side instrumentation. Hence, to gather a complete picture of the system, white-box monitoring solutions should include client-side data. Furthermore, the simplicity of installing and changing as little as possible and *automatically* observing beyond administration frontiers is extremely appealing. In this paper, we follow this path and evaluate to what extent can black-box monitoring identify different bottlenecks, using only clients' data. Real user monitoring suites, like Pingdom [5], Monitis [6] or open source project Bucky [7] do this to some extent, by relying on clients' data, but they mostly serve to create dashboards and trigger notifications according to a set of rules. In simple terms, our goal in this work differs from these previous approaches, because we want to automatically infer more information about the internal state of the server, using only readings from clients.

In [8], [9], we used JavaScript snippets to collect and send client-side information to the server using an approach similar to Google Analytics [10], while in [8] we surveyed a number of sites looking for bottlenecks. We now extend our previous work, by performing automatic detection of bottlenecks. In the process, we realized that there are limits to what we can infer from the client perspective, or more precisely, and in converse terms, we identified some of the information that the server and networks leak about its internal condition, during its operation (in a process that bears some similarities to black-box attacks that aim at gaining information from the physical implementation of crypto systems).

To extract information from the server, we resort to two types of timings, known as *request* and *response* times. Shortly, the request time is the time that elapses from the request to the first byte of the response, on the client, whereas the response time is the time to transfer the entire information following this byte. Based on these times, we aim at pinpointing the precise cause for internal and external bottlenecks. We created a laboratory experiment with the Mediadrop [11] open source video service, to identify two possible sources of bottlenecks: CPU and network. We ran a set of clients under a combination of 100 different types of CPU and network performance restrictions. For each of these combinations, we collected the request and response times, for the batch of client requests. Using these data, we trained two algorithms, a linear and a non-linear one, to identify the state of the server.

Our results demonstrate that client-side data enable CPU and network bottleneck identification, thus showing not only the utility, but the necessity of considering client metrics for performance monitoring. The increase in complexity to collect the additional data is also fairly small, because JavaScript snippets can do the trick of uploading limited amounts of performance data to the server. The trade-off involved thus seems to be quite favorable and independent of the operating system, programming language, or platform. On the other hand, we were also confronted with the difficulty to exactly pinpoint all sorts of problems. Black-box monitoring seems unable to distinguish some bottlenecks from others, as timing information from different sources gets mashed on its way to the client. It is an interesting open problem to determine if doing such distinction is indeed impossible or just a matter of improving the identification algorithms.

The rest of the paper is organized as follows. Section II describes the problem we tackle in this paper and the method we used to solve it. Section III describes the experimental settings. Section IV discusses distinct machine learning alternatives to solve our problem. In Section V we show the results of our experiment and evaluate the meaning of these results, the strengths of this approach and the limitations. Section VI presents the related work. Section VII concludes the paper and describes future directions.

II. Problem Description

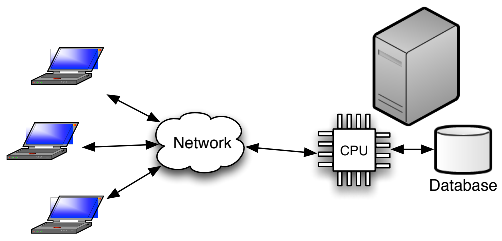

Fig. 1: Representation of the considered server-side bottlenecks

As we show in Figure 1, we consider a stand-alone HTTP server, including a relational database, and a set of clients requesting objects available on the server. We collect the request and response times seen by each client, and, based on these times, we aim at determining the level of utilization of network and CPU on the HTTP infrastructure, including the client-server network. We assign a real number in the interval $[0, 1]$ to the availability of each of these resources, where 1 stands for a completely available resource and 0 for a completely occupied one.

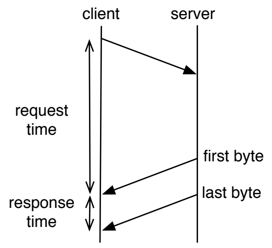

Fig. 2: Request and response times

We use two metrics that are available on web browsers, via JavaScript: the request time and the response time. In Figure 2, we can see that the request time is the time it takes from the client's request to the first byte of the response. This time includes the network round-trip-time, plus all the server processing delays. The response time is the time from the first byte to the last byte of the response. To some extent, the former time mostly involves latencies, whereas the latter mostly concerns network transfer throughput, although in some cases the server might also need to perform extensive computation and disk operations to produce the first byte, or even the remaining bytes of the response.

To automatically determine the kind of bottleneck affecting the server, clients perform object requests to the server and measure the request and response times. As we discuss in Section V, we eventually discarded the response time, and used the request time alone. Based on a sequence of request times, the goal is to determine the level of availability of CPU and network in the interval $[0, 1]$. Note that in real conditions, system administrators can follow an approach similar to Google Analytics [10], to upload client data to their facilities and analyze such data, to determine server operating bottlenecks.

In our experiments, we run the server under a wide range of controlled conditions, knowing beforehand the exact level of availability of each resource. Each of these conditions produces different request times on the clients. We then get the times directly from the clients and add the availability numbers for the two resources, in the $[0, 1]$ interval. We then train machine learning algorithms using these data, to perform an offline analysis of the overall performance, to understand the possibility to pinpoint the server status using only client data.

III. Experimental Setup

To experiment our method, we used the Mediadrop [11] open source video platform. Mediadrop is one of the components of BenchLab [12]. BenchLab addresses the limitations of

TABLE I. SOFTWARE USED AND DISTRIBUTION.

Component	Observations	Version
Mediadrop	open source video platform	0.10.3
Selenium	selenium-server-standalone jar	2.53.1
Firefox	browser	45.4.0
JMeter	performance application	3.0
Xvfb	xorg-server	1.13.3
cpulimit	binary	0.2
traffic control	change network bandwidth	1.0.2

older benchmark tools like TCP-W [13], or RUBIS [14], which are outdated and do not include Web 2.0 features. Cecchet *et al.* [12] list some more possibilities, but their adoption is somewhat more complicated, because these benchmarks lack open source implementations.

We installed Mediadrop on a Ubuntu 14.04.1 LTS Server x64, running on a Citrix XenServer virtualization platform. The virtual machine has 2 single-core Intel Xeon CPU E5-2650 0 @ 2.00GHz virtual processors, and 2 GiB of RAM. We used a server thread pool with size 10, along with other Mediadrop default settings and components. The storage is accessed through a storage area network, via 10 Gbps fiber channel. Mediadrop supports HTML5, Flash, and includes features such as video statistics or popularity, social network integration with Facebook and Twitter, content management and the ability to import videos from Youtube. Mediadrop has an off-the-shell front-end that can be accessed through a browser to let users see, import or comment videos. It was written in Python and can be extended via plugins.

To simulate CPU bottlenecks, we used the cpulimit [15] tool, which limits the CPU usage of a process. To restrain the network available to the Mediadrop server, we used the traffic control tool [16]. We varied the available CPU from 10% to 100% in steps of 10% (for 10 different values overall) plus the network from 100 kbps to 1000 kbps in steps of 100 (another set of 10 values). In total, with these two tools, we operated the server under 100 different conditions (10×10), by doing requests from the client. The goal is simple: understand if, from the client's point-of-view, we can recognize these distinct patterns of network and CPU usage.

To observe the effect of the aforementioned tools, and ensure the correct outcome of the experiments, we resorted to the `/proc` virtual file system. For the CPU, we got data directly from `/proc/stat`; for the network utilization, we used `bmw-ng` [17].

Client processes ran on the same virtualization infrastructure as the server, using a similar hardware configuration, with two identical single-core virtual CPUs, in the same local area network. The client's operating system is CentOS 6.7 x64. To perform client-side operations, we used a test tool called Selenium [18]. Selenium is a framework that emulates clients accessing the Internet using browsers. Normally, this tool is used for front-end tests, but it can be used to automate tasks, such as accessing a Uniform Resource Locator (URL) and collect the responses, as we do in this paper. To avoid being tied to a specific browser, Selenium uses a *WebDriver*,

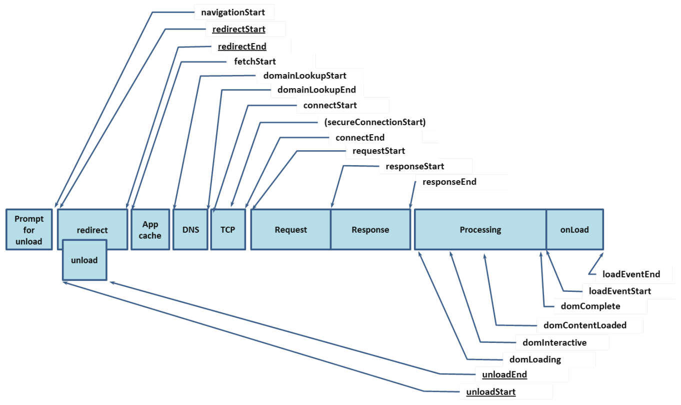

Fig. 3. Navigation Timing metrics (figure from [21])

which allows the framework to use pretty much any option. In our experiments we used Firefox. To run multiple browsers without real screens, we used Xvfb [19], to emulate a display and perform the graphical operations. To control the clients, we injected the browser requests through Apache JMeter [20], which is a standard performance evaluation tool. JMeter triggered the clients, each one of them using the Selenium framework, coupled with Firefox. We resorted to the Navigation Timing Application Programming Interface (API) [21], to collect performance times. This API, is a JavaScript-based mechanism that runs on the browser and enables collection of several performance times, including interaction times with the server, as well as rendering and processing times of the browser itself. Figure 3 depicts the different metrics that are available to this library, as defined by the World Wide Web (W3) Consortium.

This tool collects information like DNS or TCP times, as well as the request and response times we need for our algorithms. Table I summarizes the software and respective versions of the most important components of the setup. Our utilization of standard tools reproduces production site conditions, something that would not be possible with customized clients.

In our experiments, we used 4 browsers, each one of them triggered by a distinct JMeter thread. Each browser requested the entry page of Mediadrop 25 times to the server. With the combination of the two metrics, plus three iterations, our infrastructure generated 300 distinct results (10 steps for CPU $\times$ 10 steps for network $\times$ 3 iterations). From this 300 we collected 100 distinct results using the median of the three iterations that each client did for the 25 requests. These 100 values, together with the operating server conditions, will be the input of the machine learning algorithms of Section IV.

The program we used in the experiment is summarized in Algorithm 1.

IV. MACHINE LEARNING APPROACH

We followed a machine learning approach to predict CPU and network availabilities from the clients' input data. We

Algorithm 1 WebPage report

```
Input: Range of metrics to measure
Output: web pages metrics
    Initialization :
 1: cpu_range = list(range(10,100,10))
 2: network_range = list(range(100kbit, 1000kbit, 100))
 3: iteration_range = list(range(1, 3, 1))
 4: Open Mediadrop Application
    LOOP Process
 5: for all cpu_rank in cpu_range do
 6:   Limit Mediadrop Application to cpu_rank
 7:   for all nw_rank in network_range do
 8:     Limit NW Bandwith to nw_rank
 9:     for all it_rank in iteration_range do
10:       Run it_rank iteration
11:       Invoke JMeter from the client machine;
12:       Create 4 threads with respective Firefox Browsers;
13:       Invoke web page link;
14:       Save metrics to File;
15:     end for
16:   end for
17: end for
18: Parse Data and create final output with the median from
    all experiments.
```

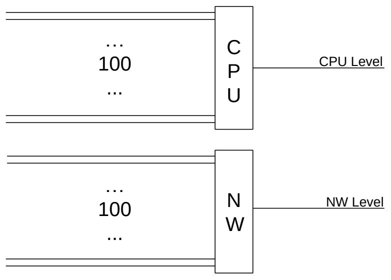

Fig. 4. Machine learning regression models for CPU and network prediction of availabilities

created a regression model (rather than a classification) for each of the two problems, as illustrated in Figure 4, since the output variables take continuous values (rather than class labels). The idea behind regression is to predict a real value, based on a previous set of training examples [22]. We provide 100 different lines to each regression model, where each of the lines has 100 inputs: 25 request times seen by the first client, another 25 by the second, etc.. There is a 101^{st} value in the line, which is the availability of the resource (i.e., the actual output value, e.g., CPU). This reference is necessary for training, but it is not available in the test cases, as we show in Figure 4.

A wide range of regression methods are available in the literature [23]. In the context of the present study, two of them assume particular relevance: simple linear regression (SLR) and Support Vector Regression (SVR). SLR is a simple algorithm where input-output linearity is assumed. This method was selected for its simplicity, speed and adequacy as a baseline approach, following the principle of Occam's razor or law of parsimony ("when you have two competing theories that make exactly the same predictions, the simpler one is the better") [24]. However, since the linearity assumption may not be valid, we also evaluate a non-linear solution, in this case, SVR. SVR is a particular case of Support Vector Machines, where the outputs are real-valued and input-output non-linearity is typically assumed, by using a non-linear kernel, e.g., polynomial or radial basis function kernels. SVR have proved to outperform other regression approaches in a wide range of problems, e.g. [25].

These algorithms were run using the Weka framework [26]. For SVR, the normalized polynomial kernel was selected based on experimental results. Both input and output data were normalized to the $[0, 1]$ interval, to attain better numerical behavior in SVR training. As for algorithmic parameterization, both algorithms were employed with default parameters. Finally, all experiments were performed using 10-fold cross validation with 20 repetitions.

V. Results

As we referred in Section III, we collected a set of 100 request and response times for the Selenium and JMeter client invocations. As we went through the data, we observed that the request time completely dominated the overall time necessary to retrieve the webpage, whereas the response time only contributed with an insignificant offset, thus providing little or no information at all. For this reason, we used the request times alone as the input to the machine learning algorithms.

The results obtained for the CPU and network availabilities regression models are summarized in Table II. We report average results for two evaluation metrics: the mean absolute error (MAE) and the Pearson correlation coefficient (CC), between the predicted and actual values. Since we performed 20 repetitions of 10-fold cross-validation, we present average and standard deviation results.

As one can observe, good results were attained for CPU and network availability prediction. There, average 0.12 MAE (with 0.02 standard deviation) and 0.14 (with 0.02 standard deviation) were attained using SVR, respectively. On a $[0, 1]$ range this is a good result. Regarding correlation, high correlation coefficient values (0.82 and 0.79, respectively) also denote that the proposed SVR approaches adequately model CPU and network availabilities.

Both SLR and SVR work better with the CPU than with the network, because CPU bottlenecks tend to dominate the request time. When the availability of the CPU is only 0.1 the average request time of the 100 requests (4 clients times 25 requests) can grow as large as $10,000$ ms, whereas a very occupied network can only raise request times to around 400 ms. Hence, very large request times immediately point to low CPU availability, but can make it quite difficult to identify the state of the network, because this component

TABLE II. Regression results for CPU and network availabilities

Method	CPU		Network	
	MAE	CC	MAE	CC
SLR	0.21 ± 0.03	0.59 ± 0.12	0.28 ± 0.05	0.29 ± 0.17
SVR	0.12 ± 0.02	0.82 ± 0.05	0.14 ± 0.02	0.79 ± 0.05

contributes relatively less to the overall delay. Indeed, when CPU availability is only 10%, the correlation coefficient (CC) of the network decreases to 0.4, for SVR, as we try all network availability levels. Finally, we observed that when the CPU and the network contribute with delays of the same magnitude, in the order of $200 - 300$ ms, the correlation coefficient of the network is 0.75, i.e., quite reasonable. Put in other words, SVR could easily identify the availability of the resource that is responsible for the longer delays (CPU), and also did a good job for a second resource (network), as long as the relative magnitudes of both delays are more or less the same. In the extreme case where one of the resources (CPU) dominates the delay, the other (network) becomes almost invisible.

Comparing SVR and SLR, the former clearly outperforms the latter in all cases and in both metrics (MAE and CC). To evaluate the significance of these results, statistical significance tests were performed using the MAE results obtained for SLR and SVR. As both MAE distributions were found to be Gaussian using the Kolmogorov–Smirnov test, the paired T-test was carried out. These results proved statistically significant (p-value < 0.001). Hence, the improved performance obtained from the non-linear SVR model shows that the input-output relationship cannot be adequately captured using a simple linear regression model.

VI. Related Work

We divide previous related work into two main areas: methods concerning internal server data and methods that resort to client-side data. Concerning server internal methods, we can enumerate some analytic models that gather several metrics to detect and predict server bottlenecks. Malkowski *et al.* [27] collect more than two hundred metrics from the server system, including the application. With this information, they show the metrics responsible for the low system performance. In [28], Malkowski *et al.* made a deeper study concerning bottlenecks. They focused in the phenomenon of multi-bottlenecks, and concluded that these may result from a chain reaction, even in slightly saturated resources. Wang *et al.*, in [29], aimed to detect transient bottlenecks recurring to a fine-grained analysis of each component of the infrastructure. Although functional, this approach suffers from the same problem of [27]: it is closely coupled to the system architecture.

Looking only to non-academic research, we have three types of tools. First, we have internal tools that gather server metrics, but require a huge amount of maintenance and operation [30]. Other tools [30], [31], which are more similar to our own experiment, allow the client to configure an URL to monitor. If something goes wrong, an alert (SMS or email) will be sent to the site owners. Unfortunately, these tools can only detect some network bottlenecks or a “slow” system, because they lack the fine-grained evaluation we propose in this paper. Finally, we have tools that use a hybrid solution, with internal and external metrics [30].

Concerning client-side data, in [32] a Java applet, called Netalyzer for browsers, tries to understand network errors. To make the test in the presence of a slow URL, it invokes the same HTTP from distinct locations and analyzes the results obtained. Others articles, such as [33–36] use a similar approach, with a browser plugin that collects information from distinct clients, to detect connectivity issues (discarding application errors). In all the aforementioned papers, the gathered information is then processed at a central point. Again, all these articles are based on a volunteer perspective, where the clients allow the plugins to run on their browsers.

Dasu [37] is a platform to measure network from distinct locations. It is also volunteer, because clients must allow the program to run on their machines. [38] also uses information from multiple crawlers in different locations to collect network errors, HTTP, DNS and TCP connection data.

WebProphet [39] and Polaris [40] are more focused on determining the dependencies between objects referred in the HTML, to decrease the page load time on the client. For this, they compute the critical path of dependencies and change the load sequence from the webpage. Our research is somehow orthogonal and complementary to the aforementioned methods.

Unlike most previous approaches, we are not tied to any specific architecture. Our goal is to evaluate the server from the client, while, aiming to identify the type of bottleneck or performance pattern that is occurring on the server. To stay away from any platform ties, we use the browser standard Navigation Timing API, thus avoiding plugins or customized crawlers. Additionally, with the client metrics, we can understand how the network influences the response in terms of quality-of-experience, something that could not be made using only the data from the server. Furthermore, we showed that it is possible to use machine learning techniques on the information gathered by the clients and, in this way, get vital information about the system reliability.

VII. Discussion and Future Work

Proper monitoring of web pages is a challenge to system developers and administrators, due to the size and complexity of the overall system and to the presence of third-party physical resources. The most popular monitoring tools are implemented on the server alone, thus not including the clients' point-of-view. This is quite limiting, because the client-to-server network is a crucial factor for performance, not to mention that some resources come from other domains.

The evidence we collected in this paper supports the idea that a black-box monitoring system, using client metrics alone, can be achieved with limited effort. In particular, results demonstrate that it is possible to accurately separate an internal server bottleneck, such as CPU, from external network bottlenecks. Furthermore, since this method is not tied to any operating system, or programming language, it is compatible with current monitoring systems, thus being able to complement standard white-box monitoring tools.

As future work, we intend to explore several directions. First, by testing a wider range of client-side setups using the same server configuration. This involves two main points: running clients on different sites, over a wide-area network, to check the impact of diversity on the machine learning approach; and requesting additional and varying resources, with different sizes that require observation of the response times (besides request times). We also plan to consider a more complex server, with more machines and a load balancer. In this case, as long as we can recover the parts of the infrastructure that were responsible for the response, we believe that we might still be able to pinpoint bottlenecks.

Finally, determining the limits of information that client-side (black-box) monitoring can extract from server operation, using only request (or other) timings, would be a particularly interesting challenge.

REFERENCES

[1] Rfc 2616 - Hypertext Transfer Protocol – HTTP/1.1. Internet Engineering Task Force (IETF), June 1999.

[2] New Relic. https://newrelic.com/application-monitoring/features. Retrieved May, 2017.

[3] Appdynamics. https://www.appdynamics.com/product/application-performance-management/. Retrieved May, 2017.

[4] Dynatrace. https://www.dynatrace.com/platform/. Retrieved May, 2017.

[5] Website performance monitoring - pingdom. https://www.pingdom.com/. Retrieved: Jun, 2017.

[6] Real user monitoring (rum) - monitis. http://www.monitis.com/real-user-monitoring/. Retrieved: Jun, 2017.

[7] Bucky performance measurement of your app's actual users. http://github.hubspot.com/bucky/. Retrieved: Jun, 2017.

[8] Ricardo Filipe, Serhiy Boychenko, and Filipe Araujo. Online client-side bottleneck identification on HTTP server infrastructures. In *The Tenth International Conference on Internet and Web Applications and Services (ICIW 2015)*, pages 22–27, Brussels, Belgium, June 2015.

[9] R. Filipe and F. Araujo. Client-side monitoring techniques for web sites. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 363–366, Oct 2016.

[10] Google Analytics. https://analytics.google.com/analytics/web/. Retrieved: Jun, 2017.

[11] Mediadrop - mediadrop open source project. http://mediadrop.video/. Retrieved: Jun, 2017.

[12] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 4–4. USENIX Association, 2011.

[13] Tpc-w benchmark, objectweb implementation. http://jmob.ow2.org/tpcw.html. Retrieved May, 2017.

[14] Rubis home page. http://rubis.ow2.org/. Retrieved Jun, 2017.

[15] Cpulimit - cpu usage limiter for linux. https://github.com/opsengine/cpulimit. Retrieved May, 2017.

[16] Traffic control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html. Retrieved May, 2017.

[17] Bandwidth Monitor. https://github.com/vgropp/bwm-ng. Retrieved May, 2017.

[18] Selenium browser automation. http://www.seleniumhq.org/. Retrieved: May, 2017.

[19] Xvfb. http://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml. Retrieved: Feb, 2017.

[20] Papers — Apache JMeter^{TM}. http://jmeter.apache.org/. Retrieved: May, 2017.

[21] Papers — Navigation Timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html. Retrieved: May, 2017.

[22] Ashish Sen and Muni Srivastava. *Regression analysis: theory, methods, and applications*. Springer Science & Business Media, 2012.

[23] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Burlington, MA, 4 edition, 2016.

[24] William M Thorburn. Occam's razor. *Mind*, 24(2):287–288, 1915.

[25] Renato Panda, Bruno Rocha, and Rui Pedro Paiva. Dimensional music emotion recognition: Combining standard and melodic audio features. In *Proceedings of the 10th International Symposium on Computer Music Multidisciplinary Research (CMMR)*, pages 583–593, 2013.

[26] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[27] Simon Malkowski, Markus Hedwig, Jason Parekh, Calton Pu, and Akhil Sahai. Bottleneck detection using statistical intervention analysis. In *Managing Virtualization of Networks and Services*, pages 122–134. Springer, 2007.

[28] Simon Malkowski, Markus Hedwig, and Calton Pu. Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 118–127. IEEE, 2009.

[29] Qingyang Wang *et al.* Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *ICDCS*, pages 31–40. IEEE Computer Society, 2013.

[30] External site monitoring services - web test tools. http://softwareqatest.com/qatweb1.html#MONITORING. Retrieved May, 2017.

[31] Check my website. https://checkmy.ws/en/features/. Retrieved May, 2017.

[32] Christian Kreibich, Nicholas Weaver, Boris Nechaev, and Vern Paxson. Netalyzr: Illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 246–259, New York, NY, USA, 2010. ACM.

[33] Tobias Flach, Ethan Katz-Bassett, and Ramesh Govindan. Diagnosing slow web page access at the client side. In *Proceedings of the 2013 Workshop on Student Workhop*, CoNEXT Student Workhop '13, pages 59–62, New York, NY, USA, 2013. ACM.

[34] Mohan Dhawan, Justin Samuel, Renata Teixeira, Christian Kreibich, Mark Allman, Nicholas Weaver, and Vern Paxson. Fathom: A browser-based network measurement platform. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, pages 73–86, New York, NY, USA, 2012. ACM.

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

[36] Heng Cui and E. Biersack. Troubleshooting slow webpage downloads. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 405–410, April 2013.

[37] Mario A. Sánchez, John S. Otto, Zachary S. Bischof, David R. Choffnes, Fabián E. Bustamante, Balachander Krishnamurthy, and Walter Willinger. Dasu: Pushing experiments to the internet's edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 487–499, Lombard, IL, 2013. USENIX.

[38] C.M. Vaz, L.M. Silva, and A. Dourado. Detecting user-visible failures in web-sites by using end-to-end fine-grained monitoring: An experimental study. In *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pages 338–341, Aug 2011.

[39] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert G Greenberg, and Yi-Min Wang. Webprophet: Automating performance prediction for web services. In *NSDI*, volume 10, pages 143–158, 2010.

[40] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, pages 123–136, 2016.