Client-Side Monitoring of HTTP Clusters Using Machine Learning Techniques

Ricardo Filipe and Filipe Araujo

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

***Abstract*—Large online web sites are supported in the back-end by a cluster of servers behind a load balancer. Ensuring proper operation of the cluster with minimal monitoring efforts from the load balancer is necessary to ensure performance. Previous monitoring efforts require extensive data from the system and fail to include the client perspective. We monitor the cluster using machine learning techniques that process data collected and uploaded by web clients, an approach that might complement system-side information. To experiment our solution, we trained the machine learning algorithms in a cluster of 10 machines with a load balancer and evaluated the results of these algorithms when one of the machines is overloaded. While a fine-grained view of the state of the machines, may require much effort to accomplish, given the compensation effect of the remaining healthy machines, the results show that we can achieve a coarse grained view of the entire system, to produce relevant insight about the cluster.**

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

I. Introduction

Web applications running over the Hypertext Transfer Protocol [1] (HTTP) are the common standard used worldwide, on a daily basis, to navigate the Internet. To ensure success of services and businesses, system administrators must ensure that their users enjoy a good quality-of-experience (QoE). They have to detect, mitigate and react to problems as quickly as possible, using monitoring tools and applications, throughout the system.

System monitoring tools, Application Performance Monitoring (APM) or Real User monitoring (RUM) can help with this goal. Basic system tools together with `Nagios` or `Zabbix` may help system administrators to analyze a large array of raw metrics, such as memory, bandwidth or CPU occupation. Unlike these, APM suites, such as New Relic [2] or AppDynamics [3] enable analysis of specific applications, at the cost of being more intrusive. Normally, these tools launch agents on systems, which can, under certain conditions, decompose application running times by local or event components, say database time, or service invocation. The advantages of one approach are the disadvantages of the other: system monitoring tools may lack application context, but are less intrusive and need fewer resources.

White-box suites, such as the aforementioned approaches, have two important disadvantages: they are closely coupled to the infrastructure, and they ignore the fact that the client, not the server, is the main entity of the business application. The client has some specific conditions, such as network and third-party resources that cannot be controlled (and monitored) by the system itself.

Therefore, to gain a system-wide perspective, client-side data should be integrated with white-box monitoring tools. In [4], an approach focused on using JavaScript snippets to gather client-side information with an approach similar to Google Analytics [5] was used, to analyze the feasibility of a black-box approach. This work inspired us, to create a more complex and realistic scenario, considering a more realistic setting with a cluster of server machines. Our goal is to determine the conditions of operation of the cluster of servers and the network operation conditions, using a limited amount of information. We aim to evaluate experimentally whether a client can tell when a specific machine of the cluster is underperforming. This is simple, and brings the benefit of a monitoring suite that is minimally tied to the architecture and software used in the system.

There are applications that seem similar in behavior, namely Real User Monitoring (RUM) applications, like Pingdom [7] or the open source project Bucky [8]. However, they are more focused on the presentation layer, with dashboards and notification rules to warn system administrators. Our work is different and more complex, because we try to infer the internal state of the system *from* the metrics, we are not interested in the metrics *per se*.

To get information from the server, we use a metric called *request time*, the time between the beginning of the request and the first byte of the response, from the client's perspective. To collect this time, we resorted to the Navigation Timing API framework [9], available in the most common browsers.

Based on client data, we analyze the possible causes associated with that pattern, regardless of being an internal or external bottleneck. Our experiment uses an open source video service, named Mediadrop [10]. We ran several clients that accessed the Mediadrop application, under a combinations of different CPU and network operation conditions. For each

combination, we collected the request times, for the batch of clients accessing the web page. The data was used to train two distinct machine learning algorithms, a linear and a non-linear one.

Our results demonstrate that it is possible to have CPU and network bottleneck detection using client-side data for a cluster of servers. Since data is gathered using a JavaScript snippet, it would be easy to upload this information to a central point, with the convenience of this approach being operation system and platform independent. Other machines of the cluster can compensate for problems in one of their peers, however, we can identify in coarse terms the operating conditions of each machine. Given the simplicity and the complete scope of this form of monitoring, we argue that this approach should be a complement to standard monitoring solutions, with the goal to improve quality-of-experience to clients.

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

II. Problem Description

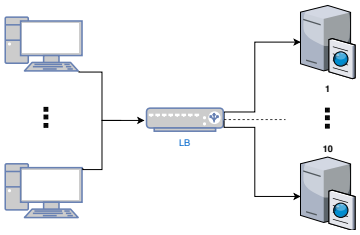

Fig. 1: Representation of the considered infrastructure

In our experimental evaluation, as illustrated in Figure 1, we consider an HTTP infrastructure comprised of 10 server machines running a social media application. In the front of the server machines we have a load balancer and a set of clients requesting HTTP objects.

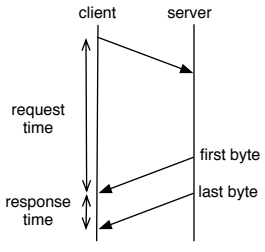

Fig. 2: Request and response times [4]

We use a metric that is available in all modern web browsers, via the framework Navigation Timing API: the request time. Figure 2 depicts this metric, including the server processing time and the network round-trip-time from the beginning of the request, until the first byte of the response has arrived. There is a complementary metric that we do not need, the response time, which includes the time needed to get all the response. Since we were always able to infer network occupation, based on request time alone, we simply did not use the response time.

Besides the request times observed by each client, we added an indication of which back-end server sent the reply in the response. Clients should then upload their data to a central point, from where we aim to automatically infer two very concrete metrics: 1) the level of occupation of the server's CPU and 2) the client-server network occupation. This information will enable system administrators to know the real operating conditions that clients experiment, and to react to bottlenecks in parts they control.

We could directly use raw data collected by the clients without any kind of processing. However, the order of arrival of the clients' responses had an impact in the method, i.e., having client A's response before client B's response was not the same as having the responses in reverse order. The reason for this is that we fed the request times to the machine learning algorithms in some specific order. Therefore, we want to avoid this dependency. Hence, we used the averages and median times of all the clients' raw data, in a rolling window approach, as a pre-processing step in our machine learning pipeline. These values were then mapped with the vacancy of the resources in the interval $[0, 1]$ (0 is entirely occupied, while 1 is entirely vacant).

As we will analyze on Section V to achieve better results in the machine learning algorithms, we also added information of where the request was processed, having this way a more fine grain information concerning the system. In a production environment, to collect this information, an approach similar to Google Analytics [5] could be made to collect and determine the operation status of the server.

The server was run in different operation sets, knowing *a priori* the vacancy of each resource. The clients for each set, run several requests, and all the data was collected. Using this data, we trained the machine learning algorithms, to understand the feasibility of a black-box client-side monitoring solution.

III. Experimental Setup

In our experimental setup, we used an open source platform for video contents, named Mediadrop [10], with features such as video statistics, popularity, and integration with major social networks, including Youtube. We opted for this software, because it has been used before in other benchmarks, such as BenchLab [11].

The Mediadrop software was installed in ten machines running Ubuntu 16.04, running on a virtualized platform. Each virtual machine has 2 single-core Intel Xeon CPU E5-2650 0 @ 2.00GHz virtual processors, and 1 GiB of RAM. We used the Mediadrop default settings.

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
NGINX	load balancing	1.10.3

To simulate the CPU and network loads, we used two distinct tools. In one specific machine of the cluster, we used the cpulimit tool [12]. This tool limits the CPU of the process running Mediadrop. To limit the network, we used the traffic control tool [13]. CPU vacancy for the Mediadrop processes was limited from 10% (almost entirely occupied) to 100% (entirely vacant) in steps of 10%. For the network, we used 5 levels, as we did not notice much performance differences in having additional levels. The network levels used were $50, 100, 250, 500$ and 1000 kbps. This gives a total of $10 \times 5 = 50$ different server conditions.

To control the actual effect produced on the servers, we used standard tools, such as the information obtained from `/proc/stat` or `top` , for the CPU utilization, and `bmw-ng` [14] for the network usage.

The load balancer used to distribute requests among the 10 back-end servers was an `NGINX` proxy server. The load balancing algorithm used was round-robin. It is relevant to mention that the load balancer includes a health-check mechanism that cuts the load on heavily stressed machines, thus having some interference on the experiments we are doing.

Client applications use a similar hardware configuration, with two virtual CPUs. The operating system is CentOS 6.7 x64. Since we wanted to invoke the Mediadrop webpage, we used a tool named Selenium [15], normally used to create automated front-end tests, which emulates accesses to web sites. In this scenario, we used this tool to access the Mediadrop webpage in an autonomous way, and collected the request times.

The Selenium framework allows us to use any kind of browser, as long as the corresponding *WebDriver* exists. In our setup we used the Firefox browser. To emulate real screens, we used Xvfb [16]. To control the several clients that access the infrastructure, with minimal effort, we integrate the Selenium clients with Apache JMeter [17], which is a standard performance tool. Hence, our experimental setup concerning the clients, consisted of JMeter, invoking our clients emulated by Selenium and Firefox, which collected the information from the Navigation Timing API [9]. Table I summarizes the most important components of our experimental setup.

We used a total of 4 browsers, each one of them associated with a JMeter thread. We configured each thread (i.e. each client) to invoke the entry page of Mediadrop application 50 times - for a total of 200 requests. To provide more data to the machine learning algorithms, we made 20 iterations for each combination of CPU and network occupations. This means that for the initial 200 requests, we generated $4,000$ raw results, for each of the 50 configurations — a grand total of $200,000$ requests of total raw data for all the experiment.

The 4000 requests of each setup were aggregated to produce results independent of the order of invocations. For each of the $4,000$ requests concerning one configuration, we aggregated request times by computing the following data: the number of requests processed by the back-end server machine (as we said, the server leaks the machine ID to the client), the minimal request time, the maximum request time, the median of the request time and the mean request time observed. With this information, we added the CPU or network level, depending on the regression case we are trying and fed the data to the algorithms of Section IV.

IV. MACHINE LEARNING APPROACH

Fig. 3. Example of Machine Learning regression models for CPU prediction of availabilities

With the information collected in our experiment, we used two machine learning algorithms to predict the CPU and network occupation, based on the clients' data. Since the output takes a value instead of a class label, we used a regression model for the network and CPU bottlenecks (in alternative to a classification). We opted for regression, since we want to predict a real value, either CPU or network occupation [18]. We aggregated the raw data for the several iterations, where each line had the target value of the configuration of CPU or network. Each line has the features represented in Figure 3 and the output to be predict, which is the vacancy of the resource (i.e., the actual output value, e.g., CPU in the figure). This latter value is available in the test scenario, but unavailable in the validation cases. The vacancy of resources is normalized in the interval $[0, 1]$, being 1 an entirely free, and 0 a totally occupied resource.

There is a wide body of knowledge regarding regression models, e.g. [19]. In this context, we opted for two distinct algorithms: Simple Linear Regression (SLR) and Support Vector Regression (SVR). SLR is simple, but linear. We also wanted to analyze more complex nonlinear models, so we used SVR, a particular case of Support Vector Machines, using a non-linear kernel. We also used this algorithm, since SVR has outperformed other regression models in distinct scenarios and applications, e.g. [20]

Both algorithms were run using the Weka framework [21]. For the SVR, we used the normalized polynomial kernel, with normalized data in the interval $[0, 1]$, to achieve better behavior in the training set. For SVR and SLR we used default values for the remaining parameters. To analyze the data, we used 10-fold cross validations, with 20 repetitions.

V. RESULTS

The results obtained for CPU and network occupation for the two distinct models are presented, respectively, in Table II

TABLE II. Regression results for CPU and network availabilities

Method	CPU		Network	
	MAE	CC	MAE	CC
SLR	0.18 ± 0.03	0.43 ± 0.11	0.19 ± 0.01	0.56 ± 0.11
SVR	0.16 ± 0.06	0.45 ± 0.17	0.21 ± 0.13	0.51 ± 0.19

TABLE III. Regression results for CPU and network availabilities

Method	CPU 3 levels		Network 3 levels	
	MAE	CC	MAE	CC
SLR	0.12 ± 0.03	0.70 ± 0.11	0.18 ± 0.08	0.71 ± 0.16
SVR	0.15 ± 0.10	0.85 ± 0.12	0.15 ± 0.13	0.76 ± 0.17

and in Table III. Table II presents the values for the 10 levels of CPU and 5 levels of network, and in Table III we present the values with a subset of the collected values. This subset was chosen taking into consideration the small, medium and larger value of the interval of both CPU and network mentioned in Section III. The problem of having the complete range of availabilities is that similar levels of CPU and network are very difficult to separate due to the influence of the remaining machines in the cluster. We thus wanted to check if the methods can actually do a good job for the simpler goal of telling the big picture regarding the resource state (resource available, unavailable or halfway).

We show the mean absolute error (MAE) and the Pearson correlation coefficient (CC). The MAE results show the distance between the estimated and the real value. Since we make a 10-fold cross-validation, with 20 repetitions, we present the values associated with the average and standard deviation of the all the iterations.

While we get acceptable results in Table II, results in Table III are much better. We can see that the former results are only average, but if we only consider 3 ranges of vacancy (small, medium, large), as in the latter table, the results are very good. We get a particularly high correlation, as well as low MAE, this meaning that both machine learning methods, specially SVR can do an excellent regression.

Results are better for the CPU bottlenecks, because the CPU bottleneck tends to dominate the overall request time. Specially in lower CPU vacancy (10%) the overall request time grows considerably (in the order of $10,000$ ms), whereas a very occupied network bottleneck only produces a delay of 400 ms. Therefore in the extreme case of low CPU and network resources, the time is largely dominated by the CPU starvation.

Since the infrastructure is complex, with a load balancer and 10 servers, patterns in the request time become elaborate. It is easy to see that the other 9 would compensate the affected machine for any trouble, creating complex patterns. Nevertheless, this was mitigated leaking to the client the machine processing the response.

VI. Related Work

We divided the related work in internal system-base monitoring frameworks, and tools that resort to client-side information.

Regarding internal monitoring solutions, we have the work of Malkowski *et al.* [22] that aims to pinpoint the resources responsible for the low performance. Although this works, they collect more than two hundred metrics from the server system. Additionally in [23], Malkowski *et al.* make an even deeper study about bottlenecks and the phenomenon of multi-bottlenecks, concluding that the chain-reaction may occur even in low saturated systems. In [24], the goal is to understand how transient bottlenecks work. To achieve this, authors used a fine-grained analysis of the system components; but this is tightly coupled to the system architecture.

In the industry, we can also find a plethora of white-box approaches that collect several server metrics, with the overall increase in the maintenance and operation of the system [25]. Additionally, there are some open-source projects, such as IOVisor that can pinpoint the bottleneck in thousands of VMs [26]. Compared to our work, these tools have a white-box methodology, needing some sort of instrumentation, agents at the virtual machine, or are very coupled to the system to be monitored. Additionally, the purpose is normally the creation of dashboards, resulting in more tools to the administrators to look at when some issue occurs in the system.

Looking only to client-side applications in industry, we have some applications that resort to techniques similar to ours, such as in [27]. However, they aim to create rules, alerts or dashboard to help administrators, thus being unable to enrich monitoring with some "intelligence" to understand the area where the system has low performance. On the other hand, if we look for academic efforts, we have some articles that create applets or plugins that collect information from distinct clients, to detect network connectivity issues. As in our case, information is processed in a central point [28–33]. It is relevant to mention that all the aforementioned approaches assume a volunteer perspective, where the clients allow the plugin and the collection of raw data. [6], uses a similar methodology to detect internal and external bottlenecks in the system using only client-side data, but with a less complex architecture.

Comparing to the previous work, we are not coupled to any kind of system infrastructure or technology, and secondly, we have little instrumentation on the server-side. To achieve this, we resort to standard tools, such as the browser standard Navigation Timing API, and other already proven methods, such as Google Analytics, to collect the information. Additionally,

the fact that we have client metrics allows us to understand the influence of the client-to-server network, and also to validate the interaction of users with third party resources, thus getting useful information to improve the quality-of-experience.

VII. Discussion and Future Work

Monitoring of web pages is a major challenge, due to the complexity, third-party resources and the need to achieve excellent quality-of-experience. The standard approach is to rely on white-box tools that collect performance metrics from the server. However, this tends to be intrinsically associated with the architecture and software used, thus excluding the clients' point-of-view. Since the goal is to understand if the quality-of-experience is good, it is important to collect metrics from the client, especially due to the client-to-server network and external resources that are out of the administrators' control.

In this paper, we aim to perform black-box monitoring, by using a realistic and powerful server infrastructure. As a result, we have a good insight of the client-side point-of-view and a solution that is almost entirely independent of the technology and software used in the system.

The evidences presented in this paper show that it is possible to separate some internal from external bottlenecks, using raw data collected in the client. This approach might work best to improve the results of standard white-box tools, instead of being a simple alternative.

As future work, we want to mitigate some of the disadvantages of using a supervised machine learning technique. In fact, using a trained data set may be unfeasible to do in large production systems. We are now doing an attempt to build a model of complex systems from the client's observations, and use that model to infer the internal state of the system.

Acknowledgments

This work was carried out under the project PTDC/EEI-ESS/1189/2014 — Data Science for Non- Programmers, supported by COMPETE 2020, Portugal 2020- POCI, UE-FEDER and FCT.

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 Feb, 2019.

[3] Appdynamics. https://www.appdynamics.com/product/application-performance-management/. Retrieved Feb, 2019.

[4] 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.

[5] Google Analytics. https://analytics.google.com/analytics/web/. Retrieved: Feb, 2019.

[6] R. Filipe, R. P. Paiva, and F. Araujo. Client-side black-box monitoring for web sites. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–5, Oct 2017.

[7] Website performance monitoring - pingdom. https://www.pingdom.com/. Retrieved: Feb, 2019.

[8] Bucky — performance measurement of your app's actual users. http://github.hubspot.com/bucky/. Retrieved: Feb, 2019.

[9] Papers — Navigation Timing. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html. Retrieved: May, 2018.

[10] Mediadrop - mediadrop open source project. http://mediadrop.video/. Retrieved: Feb, 2019.

[11] 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.

[12] Cpulimit - cpu usage limiter for linux. https://github.com/opsengine/cpulimit. Retrieved Feb, 2019.

[13] Traffic control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html. Retrieved Feb, 2019.

[14] Bandwidth Monitor. https://github.com/vgropp/bwm-ng. Retrieved Feb, 2019.

[15] Selenium browser automation. http://www.seleniumhq.org/. Retrieved: Feb, 2019.

[16] Xvfb. http://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml. Retrieved: Feb, 2019.

[17] Papers — Apache JMeter^{TM}. http://jmeter.apache.org/. Retrieved: Feb, 2019.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] External site monitoring services - web test tools. http://softwareqatest.com/qatweb1.html#MONITORING. Retrieved Feb, 2019.

[26] Iovisor. https://www.iovisor.org/. Retrieved Feb, 2019.

[27] Check my website. https://checkmy.ws/en/features/. Retrieved Feb, 2019.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.