Performance-aware load shedding for monitoring events in container based environments

Rolando Brondolin
Politecnico di Milano
DEIB department
rolando.brondolin@polimi.it

Matteo Ferroni
Politecnico di Milano
DEIB department
matteo.ferroni@polimi.it

Marco Santambrogio
Politecnico di Milano
DEIB department
marco.santambrogio@polimi.it

ABSTRACT

Runtime monitoring tools have become fundamental to assess the correct operation of complex systems and applications. Unfortunately, the more precise is the monitoring (sampling rate, information granularity, and so on), the higher is the overhead introduced in the system itself. In this paper, we propose a new load shedding framework that enables runtime adaptation of monitoring agents under heavy system load, exploiting an heuristic Load Manager to control the agent status and a runtime support for domain-specific policies. We implemented the proposed methodology on Sysdig, with an average control error improvement of 3.51x (12.25x at most), w.r.t. previous solutions.

CCS Concepts

•Information systems → Stream management;•General and reference → *Performance;***•Software and its engineering** → *Monitors;*

Keywords

Monitoring; Data-stream processing; Load Shedding

1. INTRODUCTION

In the last few decades, embedded systems has moved from System-on-Chip (SoC) based on micro-controllers to fully-fledged platforms powered by multi-core processors. This allowed the concurrent execution of multiple tasks and services on the same hardware, enabling the development of complex applications in multiple scenarios (e.g., automotive, Internet TV, mobile and other embedded use cases like low-power microservers).

Such a technological shift is bringing into the embedded world a set of technologies that had been an exclusive for workstations and servers for years. On the one hand, this is the case of *virtualization*, whose objectives are resources partitioning and applications isolation in a multi-tenant system, even if this is embedded [21]. On the other hand, *software containers* can further reduce the resource footprint of applications and decrease the overhead of isolation, an important aspect especially for embedded platforms [6]. In both cases, a thorough and precise *monitoring* of hardware platform, operating system and of tenant applications is fundamental to audit the correct operation of a complex system, embedded or not: unfortunately, the more precise is the monitoring (sampling rate, information granularity, and so on), the higher is the overhead introduced in the system itself.

EWiLi'18, 4 October 2018, Torino, Italy.

An interesting use case is Sysdig [24], a monitoring tool based on system calls tracing [15], meant to monitor Docker containers [10] at runtime: it provides a monitoring agent that runs in the system and traces every system call, collecting data for threads and File Descriptors (FDs) and grouping the resulting metrics for each application, container and host. When the monitored applications are serving a small amount of requests, Sysdig has to collect few system calls, with a negligible overhead on the system. Then, when the activity of the applications rises, the number of system calls parsed becomes significant, increasing the agent load and rapidly filling its memory buffers: the consequence are an excessive computational overhead and an undiscriminated loss of events. *Self adaptation* then becomes an important requirement to prevent this critical situation at runtime.

In this context, we propose a new Load Shedding (LS) framework that enables runtime adaptation of a monitoring system, downscaling the quality of the output metrics depending on the system load. The contribution of this work is twofold:

1. we propose a LS methodology for monitoring agents based on pluggable and domain-specific policies, that relies on input characteristics and allows to drop events given their meta-data (and not their whole content);
2. we designed and implemented the LS framework, which exploits performance degradation techniques to adapt the throughput of Sysdig to the input stream, allowing it to have a limited impact in case of system overload.

The rest of this paper is organized as follows: Section 2 details the previous works in the field; Section 3 presents a preliminary analysis of the LS problem, discussing also the system calls considered in the monitoring activity; Section 4 details the methodology, the design and its implementation in the LS framework; Section 5 presents the results of the experiments conducted to validate the proposed methodology; finally, Section 6 draws the most remarkable conclusions, pointing out future directions.

2. RELATED WORKS

Monitoring is fundamental to audit applications and infrastructures performance and effectiveness, both in embedded environments and cloud-based ones [2]. As for cloud environments, *System-level* monitoring tools are available from the same providers (e.g. Amazon Cloudwatch [8]), as well as from third party producers, like Azurewatch [4], Nagios [17, 12], OpenNebula [19], Ganglia [16], and others.

However, the intrinsic dynamism of container-based infrastructures moved the focus from system-level to *application-level* monitoring [11], where application containers are the fundamental building block of complex ecosystems. Monitoring tools can

rely on instrumentation [22], plug-ins, Management Interfaces (MIs) (e.g. Java Management eXtentions (JMX) [13]) or custom builds of applications.

Another possibility is to retrieve information directly at the *OS-level*, thus tracing system calls and network activity [20] to collect advanced metrics. Tools like NewRelic [18], Datadog [9] and Sysdig [24] then rose to cope with this new challenge: Sysdig has been our choice for the validation of the proposed methodology, although any of the others can be chosen. Sysdig is a Software as a Service (SaaS) solution which relies on system calls tracing and MIs to collect aggregated metrics about CPU and Memory usage, Network and File activity, segmenting and grouping them by application, container, host and infrastructure using an agent deployed on each machine. Sysdig is open-source [24] and provides detailed information about system status using Linux tracepoints [15]: it is composed by (1) a kernel module, which traces system calls and exposes some buffers to (2) the user-space application, which analyzes the events stream.

In this context, the problem of optimizing the agent becomes a key aspect as the amount of system calls increases. LS techniques have been introduced mainly in centralized and distributed Data Stream Management Systems (DSMSs) like STREAM [3] and Aurora [1] to reduce the overhead manipulating the input stream and dropping events at runtime. Aurora defines a LS system [25] based on drop operators added inside the streaming graph, introducing the concept of Load Shedding Road Map (LSRM). The algorithm computes several LSRMs off-line and applies one of them to the query network depending on the system load. CTRL [26] is a LS system based on Aurora that efficiently controls the DSMS overloading with identification techniques.

These approaches work on the single input data and do not tackle the typical data aggregation performed by monitoring agents. The LS system of STREAM [5] instead, works on aggregation queries over data streams, minimizing the quality loss in the output data and guaranteeing the desired Quality of Service (QoS). Finally, a recent works like FFWD [7] starts from that point and tackles the LS problem with a framework composed by an heuristic controller and pluggable policies in the context of sentiment analysis; however, only response time is addressed, while no consideration on the overhead is made.

3. MOTIVATING EXAMPLE AND GOALS

Sysdig is the monitoring tool we chose to validate the methodology proposed in this paper: its agent is able to trace system calls [23] in a large variety of Linux kernel versions, and its source code is available open-source. Monitoring these events is extremely useful for both troubleshooting and runtime audit: the difference is that the CPU overhead introduced is not a concern in the former use case, while is critical in the latter.

If we consider the graph of Figure 1, we can see a run of Sysdig when observing the system calls of the pts/nginx benchmark from the Phoronix test suite [14], which is producing $\simeq$ 1.4 million events per second. The graph shows CPU usage of Sysdig obtained on a Dell Power Edge T630, equipped with two Intel Xeon E5-2650 v3 @ 2.3GHz and 128GB RAM DDR4 in three different scenarios. The first one is the CPU load of the Sysdig troubleshooting tool (in red): it consumes almost one core of the whole system, as the agent traces every system call, even non strictly useful ones. If we select only the system calls related to process creation, context switch and FDs activity (e.g. open, close, read, write, dup, socket, connect, accept, bind, etc.), the CPU load of the agent for the same workload is traced in blue: even if it decreased, it is still far from being negligible.

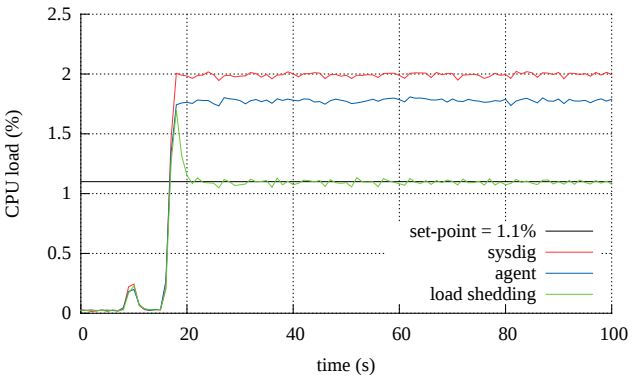


Figure 1: CPU load of sysdig, agent and agent with LS with nginx from Phoronix test suite.

This work aims at (1) controlling the overhead of the monitoring agent, adapting the volume of events processed, while (2) maximizing metrics precision and monitoring accuracy. This is showed by the green plot, produced by the agent equipped with the proposed LS framework: it considers the same events tackled by the blue plot, however, it drops part of them to meet both the aforementioned goals, as detailed in the rest of the paper.

4. SYSTEM DESIGN

Figure 2 shows the main components of the Sysdig agent, highlighting the integrations made by the LS framework.

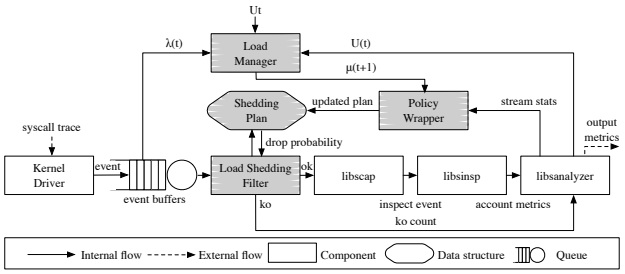


Figure 2: Sysdig and LS framework integrations with filter, shedding plan, policy wrapper and load manager.

The agent is composed by two main actors: the kernel driver and the user-space agent. The former traces system calls and maps them on the events header and payload, sending them to the interface buffers, one per core. The latter consumes events from the buffers, parsing them to account the system call parameters to the proper thread and FD.

The user-space application is then built upon three components: *a*) *libscap*, *b*) *libsinsp*, *c*) *libsanalyzer*. *Libscap* is in charge of selecting the oldest event from the buffers, sending it to *libsinsp*, which extracts the data and updates the monitoring state. Then, the *libsanalyzer* component groups data per application, container and host. Finally, the metrics are sent to a back-end infrastructure at a fixed time interval: no more details about this are provided here, as this is outside the scope of the work.

The LS framework adds the *LS Filter* between the interface buffers and *libscap*, thus defining *where to shed* the excessive load.

The filter is in charge of discarding events, depending on the drop probabilities stated for each process in the *Shedding Plan*. This component is fed by the LS policies hosted in the *Policy Wrapper*, which computes *how much load to shed* using the output of the *Load Manager*. Finally, the *Load Manager* decides *when to shed* the load computing the throughput of the agent using the buffers state, the CPU usage set-point and the current CPU utilization. The framework is also in charge of rescaling the output metrics depending on the amount of dropped events.

4.1 Load management

The user-space application can be modeled as a single-server node fed by a set of queues, that can be abstracted to a single queue given the event consumption strategy adopted. The application measures the arrival rate $\lambda(t)$ as the number of incoming events and $\mu(t)$ as the number of served events in a given time interval. The event consumption behavior of the agent can then be modeled by means of the *Little's Law*, i.e., the CPU utilization of the application is proportional to the arrival rate $\lambda(t)$ and the service time $S(t)$.

If we consider a one second time interval, we can approximate the service time as $\frac{1}{\mu_{max}}$, where μ_{max} is the estimated maximum throughput of the agent. The resulting equation gives an estimation of the utilization of the user-space application, but still does not consider the events waiting in the queue, which are addressed by equation (1). The queue length at time t is a function of the queue length at time $t-1$ and the number of incoming and served events, as shown in equation (2).

$$U(t) = \frac{\lambda(t)}{\mu_{max}} + \frac{Q(t)}{\mu_{max}} \tag{1}$$

$$Q(t) = Q(t-1) + \lambda(t) - \mu(t) \tag{2}$$

We can then combine the previous formulas to obtain the formulation of equation (3):

$$\mu(t) = 2 \cdot \lambda(t) - \lambda(t-1) - \mu_{max} \cdot \big(U(t) - U(t-1)\big) \tag{3}$$

If we assume that the input arrival rate does not change abruptly over time, we can approximate $\lambda(t)$ as the arrival rate at last time interval $\lambda(t-1)$: this assumption holds as far the system evolution is slower than the controller.

The last step through the definition of the heuristic *Load Manager* is to define the feedback error $e(t)$, as shown in equation (4) This leads to the final formulation of the *Load Manager*, as in equation (5):

$$e(t) = U(t) - \bar{U} \tag{4}$$

$$\mu(t+1) = \lambda(t) + \mu_{max} \cdot e(t) \tag{5}$$

The *Load Manager* formulation just obtained is composed by two contributions: on the one hand, when the contribution of the feedback error $e(t)$ tends to 0, the stability condition: $\mu(t) \leq \lambda(t)$ is met; on the other hand, the second contribution: $\mu_{max} \cdot e(t)$ ensures a fast response of the *Load Manager* in case of deviations from the CPU load set-point, allowing the application to consume all the events in the queue in the given time interval. Equation (5) then represents the throughput that allows the application to keep up with the events in the queue.

4.2 Event selection

Once the *Load Manager* has computed the new throughput, it needs to plan *how much* load to shed and *which events* to drop. To this aim, we split the agent throughput into $\mu_c(t)$, i.e., the system capacity (measured as the number of computed events per second), and $\mu_d(t)$, i.e., the *dropping rate* of the LS system, thus obtaining (6):

$$\mu(t) = \mu_c(t-1) + \mu_d(t) \tag{6}$$

As we did for $\lambda(t)$, if the input stream does not change abruptly over time, we can approximate $\mu_c(t)$ to $\mu_c(t-1)$. Starting from equation (6), the proposed LS system selects which events to drop using a probabilistic approach; where, in any case, the approximated metrics computed with the LS framework should be as close as possible to the exact ones. Keeping this in mind, we can define policies able to express which events must be dropped and with which probability, still guaranteeing the throughput $\mu(t+1)$ defined by the *Load Manager*. The rest of this section proposes two policies related to the case study at hand, which use domain-specific knowledge on system and application monitoring.

4.2.1 Fair policy

Each process in the system produces different amount of events: the fair policy computes a shedding probability for each one to maintain good visibility on small processes and achieving good accuracy in estimating metrics of bigger ones. We can define the number of computed events $\mu_{c_i}(t+1)$ for each process as in equation (7), where $|H|$ is the number of processes. We can then define for each process a probability $P(X_i)$ (where X_i means "*the event of process i is not taken*") as 1 minus $\mu_{c_i}(t+1)$ over the service rate of that process $\mu_i(t+1)$, as in equation (8). The value $\mu_i(t+1)$ is approximated rescaling $\mu_i(t)$ on $\mu(t+1)$. Moreover, if we substitute (7) in (8), we obtain the final formulation of the probability $P(X_i)$.

$$\mu_{c_i}(t+1) = \frac{\mu_c(t+1)}{|H|} \tag{7}$$

$$P(X_i) = 1 - \frac{\mu_{c_i}(t+1)}{\mu_i(t+1)} = 1 - \frac{\mu_c(t+1)}{|H| \cdot \mu_i(t+1)} \tag{8}$$

In case of processes with unbalanced $\mu_i(t)$, the smaller ones will have $P(X_i) \simeq 0$, while the bigger ones will have $P(X_i) \simeq 1$. To achieve optimality in event attribution, the policy orders the processes by their $\mu_i(t)$ in ascending order, assigning the events to be computed as in equation (9) and rescaling the remaining events accordingly. Each probability computed by the fair policy is then stored in the *Shedding Plan*, that in our case is represented by the thread table of the agent.

$$\mu_{c_i}(t+1) = min\left(\frac{\mu_c(t+1)}{|H|}, \mu_i(t+1)\right) \tag{9}$$

4.2.2 Priority-based policy

The priority-based policy is meant to monitor critical processes with higher accuracy with respect to others. It defines priorities for each relevant process, while all the others are considered with minimum priority (i.e., best-effort monitoring). Given a priority $p_i \in \mathbb{N}$ and the set of processes H, we can define the *normalized priority* $w_i \in \mathbb{R}$ as in equation (10), where w_i represents the weight of the process in the input flow. We can then apply the normalized priority to obtain the number of computed events for each process $\mu_{c_i}(t+1)$ as in equation (11).

$$w_i = \frac{p_i}{\sum_{i=1}^{|H|} p_i} \tag{10}$$

$$\mu_{c_i}(t+1) = w_i \cdot \mu_c(t+1) \tag{11}$$

As we can see from equation (11) and equation (7), w_i is the generalization of the "fair" weight used for the policy of section

Table 1: test configurations for homogeneous tests.

test id	name	# instances	priority	# evts/s
A	nginx 1.1.0	3	3	884846
B	postmark 1.1.0	3	4	1274787
C	fio 1.4.0	1	4	1284232
D	simplefile	1	2	1507153
E	apache 1.6.1	2	2	1975027

4.2.1, where the processes have priority equal to 1. We can now define the probability $P(X_i)$ as one minus the weighted priority w_i times the ratio between the total number of computed events $\mu_c(t+1)$ and the service rate for that process $\mu_i(t+1)$ as in equation (12):

$$P(X_i) = 1 - \frac{\mu_{c_i}(t+1)}{\mu_i(t+1)} = 1 - w_i \cdot \frac{\mu_c(t+1)}{\mu_i(t+1)} \tag{12}$$

Again, when the processes have unbalanced $\mu_i(t)$ we assign the number of computed events for each process as in equation (13). In this way, $\mu_{c_i}(t+1)$ is selected only if it is smaller than the process service rate $\mu_i(t+1)$. The remaining events, if any, are reassigned to the bigger processes in the list rescaling $\mu_{c_i}(t+1)$ accordingly.

$$\mu_{c_i}(t+1) = min\Big(\big(w_i \cdot \mu_c(t+1)\big), \mu_i(t+1)\Big) \tag{13}$$

4.3 Implementation details

The agent already implements a filtering mechanism in the kernel module, which incrementally halves the input stream on a 1 second basis in case of overload until the user-space application reaches the target CPU load. On the one hand, this solution relieves the user-space application from processing all the events. On the other hand, it makes impossible to accurately reconstruct the aggregated metrics.

To cope with this issue, the *LS filter* is instantiated at the beginning of the user-space pipeline, discarding events as soon as they arrive in the buffers and precisely accounting them for metric reconstruction. Moreover, the *LS filter* works only on the events header to avoid duplicated analysis that should be done by other agent components.

Each system call has two events associated, thus the *LS filter* should treat them together. For each event, the *LS filter* finds the related thread in the thread table and the FD when the system call uses it. If the thread or the FD is missing, the event is discarded, otherwise the *LS filter* selects whether to drop the event or not depending on the probabilities computed by the policies of section 4.2, and accounts the event for the metric correction phase.

5. EXPERIMENTAL RESULTS

The experimental evaluation focuses on two key performance indicators: the *a*) *stability* of the load manager (i.e., it is able to maintain the monitoring overhead bounded in all the test cases) and the *b*) *quality* of the output (i.e., we are able to provide good approximations for the output metrics). On the one hand, the monitoring agent should maintain the CPU overhead as close as possible to the set-point, adapting its throughput in case of sudden spikes of the input stream. On the other hand, the load adaptation of the agent should minimize the events drop and the quality degradation of the output metrics.

With these goals in mind, we tested the *Load Manager* and the policies described in Section 4.2 w.r.t. the kernel drop technique described in Section 4.3, when executing the following five benchmarks: *apache*, *nginx*, *fio*, *postmark* from Phoronix test suite [14] and a custom one we called *simplefile*. The benchmarks are executed in two configurations: *homogeneous*, where we simulate a dedicated server (as described in Table 1), and *heterogeneous*, where we simulate a scenario in which resources are shared among different workloads (as shown in Table 2). For each test, we analyze the *stability* property, providing the Mean Absolute Percentage Error (MAPE) between the effective CPU usage and the set-point, and the *quality* property, providing the MAPE between the approximated and the exact metrics computation. We conducted the tests on a Dell Power Edge T630, equipped with two Intel Xeon E5-2650 v3 @ 2.3GHz and 128GB RAM DDR4.

Table 2: test configurations for heterogeneous tests.

test id	instances	# evts/s
F	3x nginx, 1x fio	1381261
G	1x nginx, 1x simplefile	1394268
H	1x apache, 2x postmark, 1x fio	1782978

5.1 Stability of the Load Manager

Table 3 shows the results obtained with the agent equipped with the *kernel-drop* filtering system and the agents with the two policies described in Section 4.2. We tested the agents with three different set-points (1.0%, 1.1% and 1.2% of the whole system capacity) to evaluate the *Load Manager* response with different constraints, repeating the experiment more than 20 times for each case. Results shows that the *Load Manager* performs better than the kernel-drop solution in most cases, with an average improvement of the MAPE of 3.51x (12.35x at most). Table 3 also shows that, in both the fair and priority cases, the MAPE decreases as we increase the target set-point, as expected. Moreover, in some cases the kernel-drop algorithm leads to an oscillatory behavior with a significant amplitude. On the contrary, the proposed solution allows a fine grain control and action that allow to converge the CPU usage near the set-point.

The proposed *Load Manager* performs worse in only one case (i.e. test H) and reaches saturation in only another case, i.e., with test E and test H with set point 1.0%. This happens as the system is under heavy load and the agent is discarding $\simeq$ 96% of the events. This situation is mainly due to the cost of reading events inside the input buffers, which can be mitigated moving part of the filtering process in the kernel driver. The kernel-drop implementation does not suffer this problem, given that the event filter is deployed directly in the kernel driver.

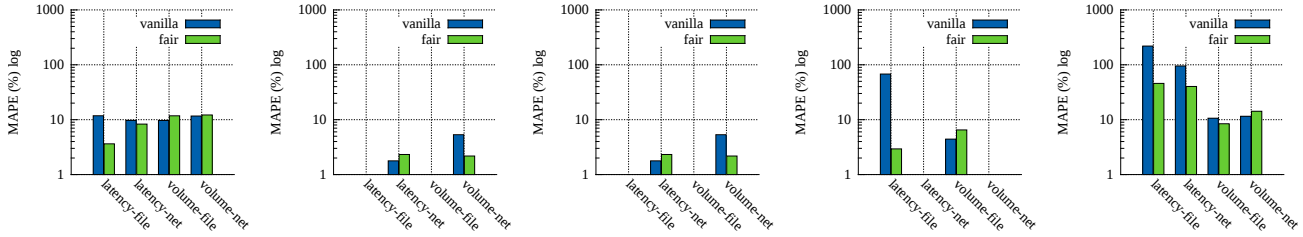
5.2 Quality of the output metrics

For what concerns the quality of the output under heavy load conditions, we collected the output metrics of the agent equipped with the kernel-drop system and with the two policies of section 4.2, for each test listed in Tables 1 and 2. Results for set-point 1.1% are shown in Figure 3 for the tests of Table 1 and in Figures 4, 5 and 6 for the tests of Table 2. For each test, we measured the MAPE of the file and network latency metrics and of the file and network volume metrics, obtained comparing the estimated and exact metrics of the agents. All the graphs have the y-axis in logarithmic scale, to allow a better visualization of the results.

As for the homogeneous tests, the kernel-drop baseline and the proposed LS framework have similar performances with nginx, postmark, fio and apache, as shown in Figures 3(a), 3(b), 3(c), 3(e) respectively: this happens as these tests have a regular behavior in terms of system calls, thus making the results inde-

Table 3: **MAPE (lower is better) between U and $\bar{U}$ ($\bar{U}$ set to 1.0%, 1.1%, 1.2% of system capacity) with tests of table 1 (A, B, C, D, E) and table 2 (F, G, H). The proposed solution is highlighted in bold.**

test	$U=1.0\%$			$U=1.1\%$			$U=1.2\%$		
	kernel-drop	**fair**	**priority**	kernel-drop	**fair**	**priority**	kernel-drop	**fair**	**priority**
A	4.11%	1.79%	2.86%	7.12%	1.78%	3.78%	7.44%	1.63%	2.41%
B	27.93%	5.51%	5.89%	34.06%	4.37%	4.46%	6.07%	3.04%	4.58%
C	24.12%	2.19%	2.15%	28.03%	2.27%	2.24%	19.48%	2.04%	2.05%
D	14.14%	1.55%	1.77%	11.52%	1.41%	1.54%	15.52%	1.52%	1.65%
E	21.46%	15.94%	17.45%	26.02%	8.51%	8.99%	18.71%	3.58%	7.97%
F	16.95%	6.33%	6.81%	22.67%	8.11%	3.74%	27.00%	4.39%	2.76%
G	18.88%	3.81%	3.04%	16.42%	3.37%	2.73%	19.74%	3.35%	3.29%
H	5.26%	13.06%	25.67%	13.49%	8.41%	8.01%	20.36%	10.05%	5.8%
avg	16.61%	6.65%	8.2%	19,92%	4.78%	4.44%	16.54%	3.70%	3.81%



(a) kernel drop 50.67%, fair drop 75.8%, pts/nginx (A)

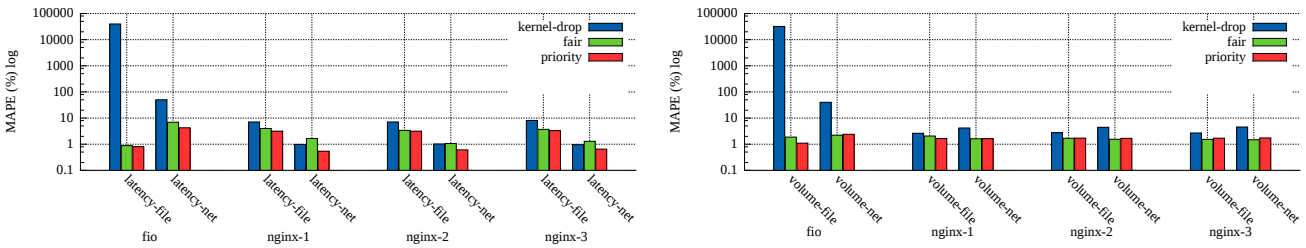
(b) kernel drop 40.49%, fair drop 43.2%, pts/postmark (B)

(c) kernel drop 27.02%, fair drop 57.13%, pts/fio (C)

(d) kernel drop 4.48%, fair drop 81.43%, simplefile (D)

(e) kernel drop 69.02%, fair drop 94.73%, pts/apache (E)

Figure 3: MAPE between exact and approximated metrics for tests A,B,C,D,E with file and network latency, file and network volume (lower is better). The graphs are in log scale.



(a) MAPE latency for 1x fio, 3x nginx

(b) MAPE volume for 1x fio, 3x nginx

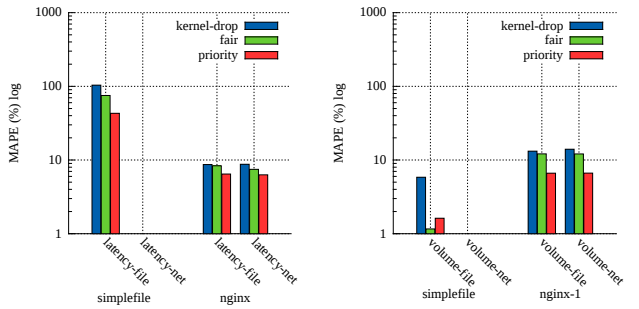
Figure 4: MAPE between exact and approximated metrics for test F (lower is better), with kernel drop 69.77%, fair drop 90.39%, priority drop 88.89%. The graphs are in log scale.

pendent from the drop strategy. On the contrary, this is not the case of the simplefile test, which performs a set of read and write operations on several files in a regular manner, with writes ops buffered by the Operating System (OS): with this optimization, the volume of bytes read and written remains the same, but the latency can vary unpredictably, leading to the result in Figure 3(d), where our solution better estimates the file latency metric.

We then mixed some tests to simulate co-located workloads competing for the same resources. Figure 4 shows the results of test F, where 3 instances of nginx and one instance of postmark are running at the same time. In this case, the proposed solution behaves better in estimating all the metrics of all the running instances. Moreover, the priorities listed in Table 1 and applied in the heterogeneous tests enhanced the accuracy on all the instances with respect to the other processes running in the system: our solution outperforms by $\simeq$3 orders of magnitude the kernel-drop solution with the fio instance in the file metrics. We experienced the same behavior in test H, with one fio instance, one apache instance and two postmark instances: Figure 6 shows that the three implementations performs similarly for apache and postmark with a limited MAPE, while our solution performs better with the fio instance. Finally, Figure 5 shows the results of test G, where one nginx instance and one simplefile instance run concurrently: results show how we are able to obtain better estimations w.r.t. kernel-drop for both nginx and simplefile benchmarks. As a final remark, it is interesting to note how the proposed solution is able to improve the precision of the output metrics w.r.t. the kernel-drop solution, even if we often enforce a higher drop percentage because of the event extraction cost.

6. CONCLUSION

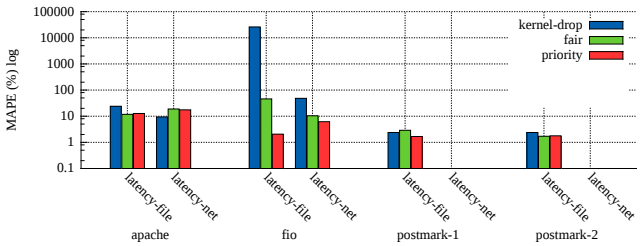
This paper presented a LS framework that allows a monitoring agent to adapt its behavior to both limit CPU overhead and maximize the accuracy of its output metrics under heavy load conditions. The framework is composed of an heuristic *Load*



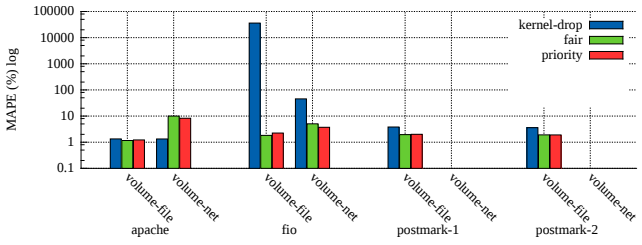
(a) MAPE latency for 1x simplefile, 1x nginx

(b) MAPE volume for 1x simplefile, 1x nginx

Figure 5: MAPE between exact and approximated metrics for test G (lower is better), with kernel drop 41.95%, fair drop 78.37%, priority drop 73.95%. The graphs are in log scale.



(a) MAPE latency for 1x apache, 1x fio, 2x postmark



(b) MAPE volume for 1x apache, 1x fio, 2x postmark

Figure 6: MAPE between exact and approximated metrics for test H (lower is better), with kernel drop 48.43%, fair drop 85.31%, priority drop 85.04%. The graphs are in log scale.

Manager, which autonomously corrects the monitoring agent throughput, a *LS Filter*, that accurately discards events, and a runtime management for domain-specific policies.

Results show how the *Load Manager* outperforms the previous filtering implementation by 3.51x on average, 12.25x at most. At the same time, the domain-specific policies and the *LS Filter* enable an accurate metric estimation, outperforming the previous solution in most of the cases.

We are actively working to bring our LS framework to the next level. As the experimental evaluation showed, the framework pays a high filtering cost related to the event extraction process. This happens because the event processing of the Sysdig case study is memory bound, as the agent performs few operations on each event. This issue does not affect the kernel-drop solution, which is the baseline with which we compare in this work.

Future work will exploit a kernel-level filtering, combining the efficiency of this solution with the control precision guaranteed by the *Load Manager* and the flexibility provided by the domain-specific policies.

7. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal-The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[2] G. Aceto, A. Botta, W. De Donato, and A. Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Book chapter*, 2004.

[4] azurewatch. http://www.paraleap.com/azurewatch.

[5] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[6] P. Bellavista and A. Zanni. Feasibility of fog computing deployment based on docker containerization over raspberrypi. In *Proceedings of the 18th international conference on distributed computing and networking*, page 16. ACM, 2017.

[7] R. Brondolin, M. Ferroni, and M. Santambrogio. Ffwd: Latency-aware event stream processing via domain-specific load-shedding policies. In *Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), 2016 IEEE Intl Conference on*, pages 130–137. IEEE, 2016.

[8] amazon cloudwatch. https://aws.amazon.com/cloudwatch/.

[9] Datadog. https://www.datadoghq.com.

[10] Docker. https://www.docker.com.

[11] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 74(10):2918–2933, 2014.

[12] E. Imamagic and D. Dobrenic. Grid infrastructure monitoring system based on nagios. In *Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28. ACM, 2007.

[13] Java management extentions. http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html. [Online; accessed 02-Oct-2016].

[14] M. Larabel and M. Tippett. Phoronix test suite, 2011.

[15] Linux tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt. [Online; accessed 03-Oct-2016].

[16] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[17] Nagios. https://www.nagios.com.

[18] Newrelic. https://newrelic.com.

[19] Opennebula monitoring. http://docs.opennebula.org/5.2/deployment/open_cloud_host_setup/monitoring.html. [Online; accessed 08-Nov-2016].

[20] F. Risso and L. Degioanni. An architecture for high performance network analysis. In *Computers and Communications, 2001. Proceedings. Sixth IEEE Symposium on*, pages 686–693. IEEE, 2001.

[21] D. Rossier. Embeddedxen: A revisited architecture of the xen hypervisor to support arm-based embedded virtualization. *White paper, Switzerland*, 2012.

[22] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[23] Linux system calls. http://man7.org/linux/man-pages/man2/syscalls.2.html. [Online; accessed 03-Oct-2016].

[24] Sysdig monitoring tool. https://sysdig.com.

[25] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320. VLDB Endowment, 2003.

[26] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, pages 787–798. VLDB Endowment, 2006.