

Custom Load Shedding for Non-Cooperative Network Monitoring Applications

Pere Barlet-Ros*, Gianluca Iannaccone[†], Josep Sanjuà-Cuxart* and Josep Solé-Pareta*

*Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

[†]Intel Research, Berkeley, USA

Abstract—Network monitoring applications are prone to continuous overload situations due to the extreme and highly variable data rates of network traffic, traffic anomalies and network attacks. These overload situations can have a severe and unpredictable impact on the accuracy of monitoring applications, right when they are most valuable to network operators.

Load shedding via traffic sampling has recently been proposed as an effective alternative to overprovisioning for efficiently handling overload situations in network monitoring. However, not all monitoring applications are robust against sampling and often other techniques can be devised to shed load more effectively.

In order to provide a generic solution for arbitrary applications, we propose a load shedding scheme that, besides supporting standard sampling techniques, allows network monitoring applications to define custom load shedding methods. The main novelty of our approach is that the monitoring system can delegate the task of shedding excess load to applications in a safe manner and still achieve robustness in the presence of non-cooperative and selfish users.

We use real-world packet traces and deploy a real implementation of our load shedding scheme in a large university network in order to show the robustness of our monitoring system against deliberate traffic anomalies and queries that fail to shed load correctly. Our results also show that the monitoring system is able to preserve a high degree of accuracy and fairness during extreme overload conditions.

I. INTRODUCTION

Developing and deploying network monitoring applications for large enterprise network or service providers present several familiar challenges. Applications have to deal with continuous traffic streams from a large number of high speed data sources. They have to operate across a wide range of network devices, transport technologies, hardware architectures and system configurations. Furthermore, applications have no control over the input rate and need to be provisioned correctly to handle traffic burst or sudden increase in the computation requirements. To add to the complexity of building monitoring applications, there is a growing demand from network operators to obtain an ever more detailed representation of the traffic traversing the network to improve the end-user experience and the overall “health” of the infrastructure [1]–[3].

As a result, the network measurement research community has put forward several proposals aiming at reducing the burden on the developers of monitoring applications. A common approach among the various proposal is to abstract away the inner workings of the measurement infrastructure [4], [5] and allow arbitrary monitoring applications, developed

independently by third parties, to run effectively on a shared measurement infrastructure [5]–[8].

One of the key differences from previous designs is that these systems are not tailor-made for a single specific application, but instead can handle multiple, concurrent and competing monitoring applications. In this context, the monitoring system has to guarantee fair access to the system resources (e.g., processor cycles, memory space and bandwidth) across all monitoring applications. This is even more true during overload situations due to extreme traffic conditions or other anomalies that could be malicious (e.g., denial of service attacks or worm infections) or unexpected (e.g., network misconfigurations, flash crowds). During these events, the resource requirements of the applications could easily overwhelm the system leading to unpredictable results, or even interrupted service, right when the measurements are the most valuable to the network operators. Unfortunately, proposed system designs do not directly address this problem while the alternative of over-provisioning to handle peak data rates or worst case traffic mix is in general not economically feasible.

In order to address this problem, in a previous work [9] we proposed a load shedding scheme that operates without any explicit knowledge of the internal implementation and cost of the monitoring applications, and without relying on any specific model for the incoming traffic. This way, the monitoring system can preserve a high degree of flexibility, allowing for fast implementation and deployment of new monitoring applications. In contrast, previous solutions require perfect knowledge of either the system implementation [10] or the cost and selectivity of each query operator [11]–[13].

Our load shedding scheme (briefly reviewed in Section III) builds an online prediction model of the resource requirements of each monitoring application by continuously observing the correlation between its actual resource usage and a large set of traffic features (that summarize the main characteristics of the input traffic). This prediction is used to anticipate overload situations and discard part of the incoming traffic using standard sampling techniques, such as packet or flow sampling, to avoid uncontrolled packet losses even in front of highly variable workloads.

The main limitation of this approach is the lack of support for those monitoring applications that are not robust to packet or flow sampling. Those applications however could return accurate results in presence of more advanced load shedding mechanisms. In this paper we extend our load shedding

scheme to support custom load shedding methods defined by the end users. The main novelty of our scheme is that it allows the monitoring system to *safely* delegate the task of shedding excess load to untrusted applications and still guarantee fairness of service among non-cooperative users. Similar custom load shedding solutions proposed for other systems [14] require applications to behave in a collaborative fashion, which is not possible in an open environment. Our method instead is able to automatically police applications that do not implement custom load shedding methods properly. This is an important feature given that applications may fail to shed the correct amount of load (due to inherent limitations) or refuse to do so (maliciously or due to an incorrect implementation).

The rest of the paper is organized as follows. Section II presents in greater detail the related work. Section III introduces the basic architecture of our monitoring system and reviews the design of its load shedding scheme. We describe how users can safely define custom load shedding mechanisms in Section IV and validate the method in Section V. Section VI presents a performance evaluation of an actual implementation of our system using real-world packet traces, while Section VII shows the results we obtained when deploying the system in a large university network. Finally, Section VIII concludes the paper and discusses future work.

II. RELATED WORK

Data processing systems that deal with live input streams are becoming increasingly common. These systems present several particularities, such as their push-based nature and support for continuous queries, that make the problem of handling overload situations challenging and different from that addressed in traditional systems. Examples of such systems include, apart from network monitoring systems, those commonly known as data stream management systems (DSMS) [15].

In the context of DSMS, the standard solution to the overload problem is known as load shedding. Load shedding is the process of dropping excess load in such a way that the system remains stable and no overflow occurs in the system buffers. Load shedding in Aurora [11] and STREAM [12] consists of dropping unprocessed tuples by automatically inserting drop operators into query plans, such that the impact on the utility of the system is minimized. TelegraphCQ [13] implements instead an architecture called Data Triage, which uses summarization techniques to provide approximate and delay-bounded answers in the presence of overload.

These solutions require queries to be built out of small set of operators, whose cost and selectivity are assumed to be known. This significantly limits the flexibility of these solutions to be used for network monitoring, where there is a clear need for supporting arbitrary traffic queries and complex monitoring applications that cannot be easily expressed using standard declarative languages [5]. In contrast, our load shedding scheme can handle arbitrary monitoring applications and operate without any explicit knowledge of their actual implementation.

Ref. [14] proposes an interesting collaborative load shedding approach for media-based applications that resembles our idea of custom load shedding. However, [14] requires applications to behave in a cooperative fashion and does not define any explicit enforcement policy, but relies on a social welfare assumption that is not met in our scenario.

In network monitoring, the load shedding problem has not been extensively studied yet. Most solutions in this context try to avoid overload situations using specialized hardware [16] and as such are only viable as short term solutions. More sophisticated monitoring systems implement packet filtering, traffic sampling and/or data aggregation techniques [10], [17]–[19]. However, these techniques are either designed to handle overload situations only in the collection devices [17]–[19] or limited to a set of pre-defined traffic reports [10], while in this work we focus on the problem of handling multiple, arbitrary monitoring applications with unknown and variable cost.

Similar open network monitoring infrastructures have opted instead for more strict and inflexible resource management policies. For example, FLAME [6] bounds the execution time of measurement modules using a cycle counter, while Scriptroute [7] runs each measurement task on an independent resource-limited sandbox, with no local storage access and limited execution time, memory and bandwidth. The inflexible solution of simply limiting the amount of resources in advance can result in poor accuracy due to excessive and arbitrary packet drops, and does not allow the system to degrade gracefully in the presence of overload.

In the Internet services space, SEDA [20] proposes an architecture to develop highly concurrent server applications. SEDA implements a reactive load shedding approach by dropping incoming requests when an overload situation is detected. In [9] we show that, in a network monitoring system, a predictive approach can significantly reduce the impact of overload compared to a reactive one, given the extremely high data rates typically involved in network monitoring.

III. BACKGROUND

In this work we use the CoMo platform [5] as a case study to evaluate our load shedding methods. CoMo is a modular open source network monitoring system that allows users to easily develop network monitoring applications as plug-in modules written in the C language. The plug-in modules can contain the code needed to answer a particular network traffic query or even complex monitoring applications,¹ such as systems for intrusion and anomaly detection, traffic accounting and classification, network performance evaluation, billing and pricing, etc. More details about the CoMo platform can be found in [5].

In order to provide developers with maximum flexibility, CoMo does not restrict the type of computations a plug-in module can perform nor the data structures it can use. As a consequence, the load shedding scheme in CoMo must operate

¹In the rest of the paper, the terms *monitoring application*, *plug-in module* and *query* are used interchangeably.

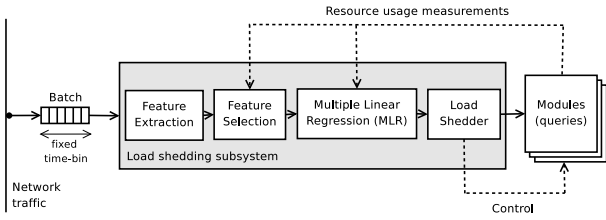


Fig. 1. Load shedding subsystem

only with external observations of the resource requirements of the modules, because the platform considers them as black boxes. Moreover, the CoMo system is open in the sense that any user can submit a plug-in module to the network infrastructure and cause arbitrary resource consumption at any time. Therefore, such an open infrastructure must manage its resources carefully in order to assure fairness of service and offer a graceful performance degradation in the presence of overload. The resource management problem is even harder considering that the input network traffic is also arbitrary and bursty in nature, with sustained peaks that can be orders of magnitude greater than the average traffic. Thus, it is also important that the load shedding scheme in CoMo does not rely on a specific model for the input traffic.

A. Prediction Methodology

Without any a-priori knowledge of the plug-in modules, the load shedding scheme in CoMo infers the cost of each module from the relation between its actual CPU usage and a pre-defined set of features of the incoming traffic. A *traffic feature* is simply a counter that describes a specific property of the input traffic stream (e.g., number of packets, bytes, flows, unique IP destination addresses, etc.). The current version of CoMo extracts about fifty traffic features online using CPU- and memory-efficient counting algorithms, such as multi-resolution bitmaps [21]. A complete list of the traffic features supported in our current prototype is available in [9].

In CoMo, the input traffic is grouped in fixed-time batches as shown in Figure 1. For each batch, the prediction system extracts its traffic features and measures the CPU cycles used by each module to process it. With these previous observations, the system builds an online prediction model for each plug-in module based on a multiple linear regression (MLR), where the response variable is the CPU usage and the predictor variables are the values of the extracted traffic features.

In order to reduce the overhead of running the MLR online and increase the accuracy of the prediction process, we use a variant of the Fast Correlation-Based Filter (FCBF) [22] that allows the system to remove both redundant and irrelevant features, before running the MLR.

Figure 1 shows the components and summarizes the operation of the prediction process. A complete description and evaluation of the prediction method is available in [9].

B. Load Shedding

The load shedder in CoMo acts when the sum of the predictions of all modules for a given batch exceeds the system

capacity. The advantage of a predictive load shedder is that the monitoring system can shed excess load in advance before being actually overloaded.

The load shedder applies packet or flow sampling to the incoming batches in order to reduce the load of the monitoring system. In CoMo, apart from the sampling method, each query $q \in Q$ can provide a minimum sampling rate (m_q) that indicates its minimum accuracy requirements. The strategy used by the CoMo load shedder [23] to select the sampling rates (p_q) of each query $q \in Q$ consists of satisfying the minimum sampling rate constraints of all queries and distributing the remaining cycles in such a way that the minimum sampling rate among all queries is maximized. In [23] we show that, in an open network monitoring system, this strategy results in better performance than the traditional approach of providing fair access to the CPU used by typical OS task schedulers.

Depending on the query requirements and the system capacity, an allocation that satisfies the minimum rate constraints of all queries may or may not exist. When no feasible solution exists, some queries have to be disabled. The strategy used by CoMo to enforce that non-cooperative users provide correct information about m_q is to stop first those queries with greater resource demands. That is, those with greater values of $m_q \times \hat{d}_q$, where \hat{d}_q is the prediction of the CPU cycles that would be needed to process the current batch by the query q .

On the one hand, this (intentionally) simple strategy can be applied online given that an algorithm exists to compute the optimal solution in polynomial time [23]. On the other hand, we demonstrated [23] that when using this strategy there is a single *Nash Equilibrium* when all queries demand $\frac{C}{|Q|}$ cycles, where C is the system capacity in CPU cycles. In other words, this solution intrinsically assures that no user has an incentive of demanding more cycles than $\frac{C}{|Q|}$ in a system with capacity C and Q queries, which is exactly the fair share of C . Moreover, given that $|Q|$ and C are unknown for the users, this strategy discourages them to provide minimum sampling rates (m_q) greater than their actual requirements, because it would increase the probability of demanding more than $\frac{C}{|Q|}$ and, as a consequence, the probability of being disabled in the presence of overload. This feature is very important because it allows the monitoring system to maintain the accuracy of the queries within bounds defined by non-cooperative or selfish users, while still assuring a fair allocation of computing resources.

IV. CUSTOM LOAD SHEDDING

The load shedding strategy described in Section III assures a fair allocation of resources to queries as long as all queries are equally robust against the load shedding mechanisms provided by the core monitoring platform (i.e., packet and flow sampling). However, this strategy penalizes those queries that do not support sampling (e.g., signature-based IDS queries), forcing them to set their minimum tolerable sampling rate (m_q) to 1. As a consequence, they have a high probability of being disabled in the presence of overload since, according to the strategy presented in Section III-B, the system stops first

those queries with greater resource demands when it cannot satisfy the minimum sampling rates.

On the other hand, there are queries that, although being robust against sampling, can compute more accurate results using different sampling methods than those provided by the core platform. For example, previous works have shown that Sample and Hold [24] achieves better accuracy than uniform sampling for detecting heavy-hitters. In that case, using packet or flow sampling would force these queries to use greater values of m_q than those actually needed when using other, more appropriate sampling methods. This would result not only in a waste of resources, but also in worse accuracy, given that the query would have a higher probability of being disabled during overload situations.

A possible solution would consist of including as many load shedding mechanisms as possible in the core system (e.g., lightweight summaries [13] or different sampling algorithms [24]) to increase the probability of finding a suitable one for any possible traffic query the system receives. However, this solution is not viable in practice for a system that supports arbitrary network traffic queries, such as CoMo, and it does not allow for testing or deploying new load shedding methods.

We propose instead a simple yet effective alternative: to allow the queries to provide their own, customized load shedding mechanisms. This way, when a suitable load shedding mechanism is not found for a given query, the system can delegate the task of shedding excess load to the query itself. Applications can potentially shed load in a more effective and graceful manner than the monitoring system. For example, they can take into account the particular measurement technique employed and the data structures involved in order to implement a load shedding mechanism that has a lower impact on their accuracy.

A. Enforcement Policy

In this solution queries are part of the load shedding procedure, which raises additional fairness concerns. Similar custom load shedding designs have been proposed for other environments (e.g., [14]) where applications behave in a collaborative fashion, a requirement that is not met in the presence of non-cooperative users. For example, in such an environment, there is no guarantee that queries will implement their custom load shedding correctly, for malicious reasons or otherwise.

Our solution instead consists of ensuring that queries shed the requested amount of load by penalizing those that do not shed it correctly. Although several policies would be possible for this purpose [6], we empirically verified that our prediction method inherently penalizes selfish queries by increasing exponentially their predicted cycles (\hat{d}_q) and thus their probability of being disabled.

Figure 2(a) illustrates this property with a real example. The line labeled as ‘selfish prediction’ shows the predicted cycles for a selfish signature-based P2P flow detector query²

²A description of this query and the packet trace used for this experiment is available in Section V.

that does not shed any load, irrespective of the amount of load shedding requested by the core. The figure confirms that \hat{d}_q increases exponentially with the load shedding rate, instead of remaining constant (note logarithmic axes). As a result, the running probability of this query decreases exponentially, because it depends directly on the value of \hat{d}_q .

This interesting behavior is consequence of the way we update the history maintained by the MLR to perform the prediction. In particular, the value used to update the MLR history is computed as $\frac{d_q}{1-r_q}$, where d_q stands for the actual CPU cycles used by a query $q \in Q$ and r_q is the load shedding rate requested by the core system. This correction is necessary because the actual resource consumption of a query can only be measured after shedding excess load.

It is then clear that the value of $\frac{d_q}{1-r_q}$ will increase exponentially if r_q increases but d_q is not reduced in the same proportion (i.e., when q sheds less load than requested). For example, the line labeled as ‘selfish actual’ in Figure 2(a) shows that the value of d_q is almost constant (i.e., q never sheds excess load), resulting in the mentioned exponential ramp on the predicted cycles (‘selfish prediction’ line). Therefore, users have no option other than implementing their custom load shedding mechanisms correctly, otherwise their queries will not have any chance to run under overload conditions.

Note that the alternative of recomputing the traffic features used in [9], rather than scaling the CPU cycles, would be only valid for those queries that use sampling as their load shedding option, besides being computationally more expensive.

B. Implementation

Our current implementation offers two equivalent ways of notifying the magnitude of load shedding to those queries that implement a custom mechanism. First, it provides the query with the load shedding rate (r_q) to be applied, which is relative to its current CPU usage. This rate is computed as $1 - p_q$, where p_q is obtained as described in Section III-B. Second, it informs the queries about the absolute amount of CPU cycles to be shed, which is simply computed as $\hat{d}_q \times r_q$.

As an example, we implemented a custom load shedding method for our previous example of a signature-based P2P flow detector query. Typically, the signatures employed to detect P2P applications appear within the first bytes of a flow [25], [26]. Therefore, an effective way to shed load is to always scan the first packet of a flow and inspect subsequent packets with probability $1 - r_q$. In order to efficiently detect new flows, we use a Bloom filter [27].

While this query could use packet or flow sampling instead, Figure 2(b) shows the notable improvement achieved after implementing our custom-defined load shedding mechanism. Note that the error of this query is defined as 1 minus the ratio of the number of flows correctly classified (according to the results obtained with the same query when all packets are inspected) and the total number of flows. The error of 1 when the load shedding rate is greater than 0.8 is due to the fact that the query is stopped at that point, as we will explain shortly.

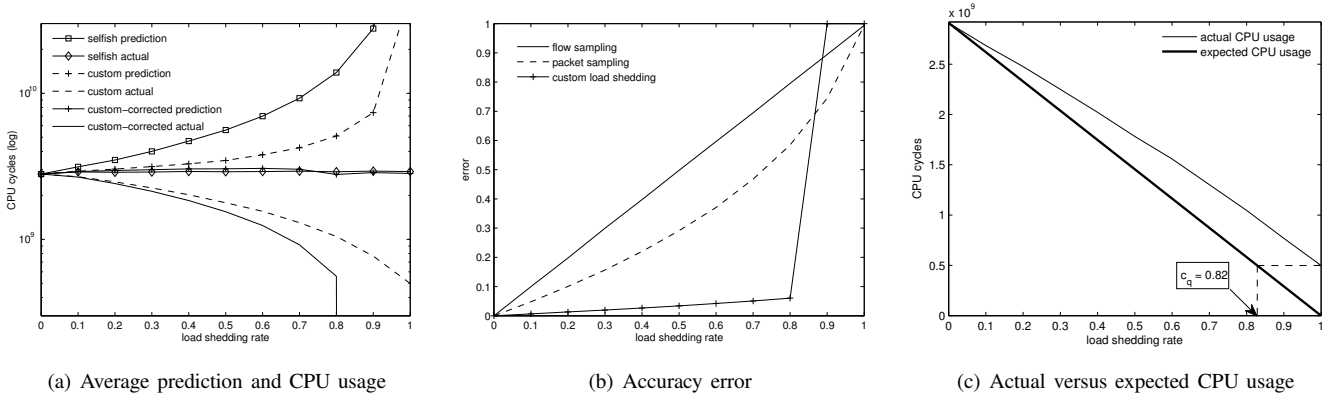


Fig. 2. Performance of a signature-based P2P flow detector query when using different load shedding methods

Nevertheless, there are cases where reducing the amount of computations by the load shedding rate does not guarantee an equivalent decrease in the query’s CPU usage. For example, in the case of the P2P detector query, there is a fixed cost that the query cannot reduce, which corresponds to checking/updating the Bloom filter and scanning the first packet of each flow.

Figure 2(a) plots the prediction and actual resource usage of this query when using the mentioned load shedding method. The line labeled as ‘custom prediction’ shows that the predicted cycles still increase exponentially, although the resource consumption decreases linearly (note logarithmic axes) with the load shedding rate (‘custom actual’ line). This is a result of the query shedding less load than requested by the core system due to this fixed cost, and therefore being penalized by our prediction algorithm. Figure 2(c) shows the actual resource usage of the query compared to that expected by the core system as a function of the load shedding rate. The figure verifies that the query is actually shedding less load than requested by the core.

In order to solve this practical problem, we allow the query to inform the core system about this extra cost. This way, the system can correct beforehand the amount of load shedding requested to the query in order to compensate for this cost and avoid exponential penalization. Assuming a linear relationship between the load shedding rate and the resource consumption, the core system computes the actual load shedding rate (r_q) to be applied to those queries that implement a custom method as $r_q = \min(1, r'_q/c_q)$, where r'_q is the original load shedding rate computed as $r'_q = 1 - p_q$, and c_q is a value provided by the query, which indicates the minimum load shedding rate from which the query q is not able to further reduce its resource consumption. When r'_q is greater than or equal to c_q , the query is disabled given that it cannot shed the amount of load requested by the core system. In the case of the P2P detector query, c_q was obtained empirically as illustrated in Figure 2(c). In particular, c_q is 1 minus the ratio of cycles consumed with a load shedding rate of 0 and those consumed with a load shedding rate of 1, resulting in a value of $c_q = 0.82$.³

³Although c_q could be computed automatically by the core system from previous observations, this option is not yet available in our implementation.

TABLE I
DESCRIPTION OF THE QUERIES USED IN THE VALIDATION

Query	Description	$m_q(c_q)$
<i>application</i>	Port-based application classification	0.03
<i>autofocus</i>	High volume clusters per subnet [28]	0.51
<i>counter</i>	Traffic load in packets and bytes	0.02
<i>high-watermark</i>	High watermark of link usage over time	0.10
<i>p2p-detector</i>	Signature-based P2P detector [25], [26]	0.91 (0.82)
<i>super-sources</i>	Sources with largest fan-out [29]	0.91
<i>top-k</i>	Ranking top- k dest. IP addresses [30]	0.50
<i>trace</i>	Full-payload packet collection	0.95 (0.49)
<i>tuple</i>	Number of active flows	0.03

Note that another option would consist of performing this correction internally within the query. In fact, by implementing it in the core system we are actually allowing both options, since a query can always provide a value of $c_q = 1$ and perform the correction by itself.

Figure 2(a) shows the impact of applying the mentioned correction to the load shedding rate. The line labeled as ‘custom-corrected prediction’ shows that, in this case, the predicted cycles are constant and independent of the load shedding rate being applied, which indicates that now this query is shedding the correct amount of load, and therefore is not penalized by our prediction algorithm. The ‘custom-corrected actual’ line shows the actual resource consumption of the query after applying the correction.

V. VALIDATION

In order to validate the load shedding scheme presented in Section IV, we implemented it in the CoMo monitoring platform and performed several executions using real-world packet traces and a diverse set of traffic queries.

A. Testbed Scenario

Table I briefly describes the traffic queries used in the validation. The table contains the queries included in the standard distribution of CoMo plus three additional (and more complex) queries that we have expressly developed to validate our proposal, namely *uni-dimensional autofocus* [28], *super-sources* [29] and *p2p-detector* [25], [26].

While most queries use packet or flow sampling, which are provided by the core platform, two queries (*p2p-detector* and

trace) implement a custom load shedding method. The *p2p-detector* query uses the technique already described in Section IV, which consists of dynamically reducing the number of packets inspected per flow. Instead, the *trace* query implements a custom method based on dynamically reducing the number of payload bytes collected per packet. The values of c_q for these queries are shown in Table I and they were computed as described in Section IV.

Table I also presents the minimum constraints (m_q) of those queries that use traffic sampling, which are set to the minimum sampling rate that guarantees an average error below or equal to 5% for each query.⁴ The error and m_q constraints of the queries are computed using the same methodology as in [23].

For our testing purposes, we collected two 30-minute packet-level traces in November 2007 and April 2008 at the access link of the Technical University of Catalonia (UPC), which connects 10 campuses, 25 faculties and 40 departments to the Internet through the Spanish Research and Education network (RedIRIS). Real-time statistics of the traffic traversing this Gigabit Ethernet link are publicly available at [31].

Both traces contain the entire packet payloads, which are needed to study those queries that require the packet contents to operate (e.g., *p2p-detector*). The first trace accounts for 95.2M packets, with an average data rate of 253.5 Mbps and a peak rate of 399.0 Mbps, while the second trace contains 61.3M packets, with average and peak rates of 222.2 and 281.2 Mbps. The second trace will not be used until Section VII, where we validate the online performance of the system.

B. System Accuracy

In order to show the benefits of our custom load shedding strategy (*custom*), we compare its performance to three different load shedding alternatives. The first alternative (*no_lshed*) consists of the original version of CoMo [5], which does not implement any explicit load shedding scheme. Instead, it simply discards packets without control as the buffers fill in the presence of overload. The second alternative (*eq_rates*) implements the simple load shedding strategy proposed in [9], which assigns an equal sampling rate to all queries. That is, in this system the amount of cycles allocated to each query is proportional to its relative cost. Finally, the third alternative (*maxmin*) implements the load shedding strategy proposed in [23] and reviewed in Section III-B.

Throughout the validation we use the accuracy of the queries as the performance metric to compare the different load shedding alternatives. As in [23], we define the accuracy of a query $q \in Q$ as $1 - \epsilon_q$ when $p_q \geq m_q$ and 0 otherwise, where ϵ_q is the actual error of the query and p_q is the sampling rate. In order to make all systems comparable, the accuracy of the *no_lshed* system is assumed to be 0 when the error is greater than 5%, given that the minimum constraints are not considered in this system.

⁴Note that the value of 5% is arbitrary and is used just as an example to validate our proposal. Similar conclusions would be also drawn with different values for the maximum error. See [23] for additional discussion.

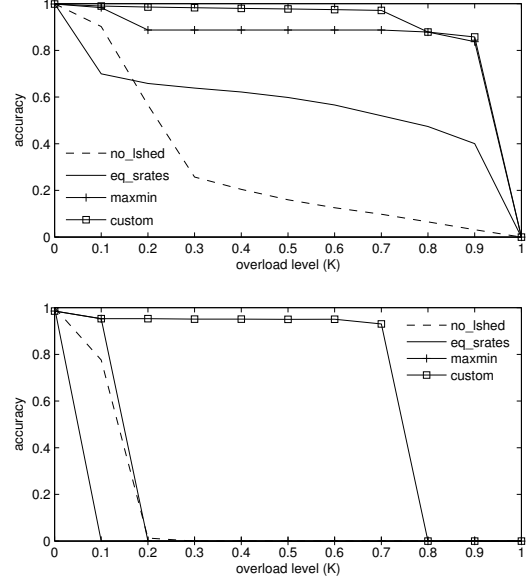


Fig. 3. Average (top) and minimum (bottom) accuracy of the monitoring system at increasing overload levels

The top plot in Figure 3 shows the average accuracy of the various load shedding strategies as a function of the overload level in the monitoring system. The overload level K is defined as 1 minus the ratio between the system capacity and the sum of the query demands. In order to simulate the different levels of overload in our testbed, we perform 10 executions ranging the value of K from 0 to 1 (in steps of 0.1) by manually setting the capacity of the monitoring system to $C \times (1 - K)$. C is experimentally determined according to the minimum number of cycles that assures that no load shedding is applied in our testbed. Therefore, $K = 0$ denotes no overload (the system capacity is equal to the sum of all demands), whereas $K = 1$ expresses infinite overload (the system capacity is 0).

The figure shows a consistent improvement of around 10% in the average accuracy of the *custom* system compared to the best alternative (*maxmin*). This improvement is achieved thanks to *p2p-detector* and *trace* that now implement a custom load shedding method, and can therefore significantly increase their accuracy. This improvement is more evident when K reaches 0.2, which is the point from which *p2p-detector* is disabled in the *maxmin* system. Note that in the *eq_rates* and *maxmin* systems, *p2p-detector* and *trace* use packet sampling and their m_q constraints are set to the values shown in Table I.

The bottom plot in Figure 3 shows the minimum accuracy among all queries. The improvement in the minimum accuracy is much more significant than in the average case. In particular, the minimum accuracy is sustained above 0.95 until $K = 0.8$, when the *p2p-detector* query is stopped. This result confirms that the *custom* system is significantly fairer in terms of accuracy than the other alternatives, given that the *trace* and *p2p-detector* queries can now compete under fair conditions for the shared resources with the other queries. Note that in the best of the other alternatives, the accuracy of at least one query is already zero when K reaches 0.2.

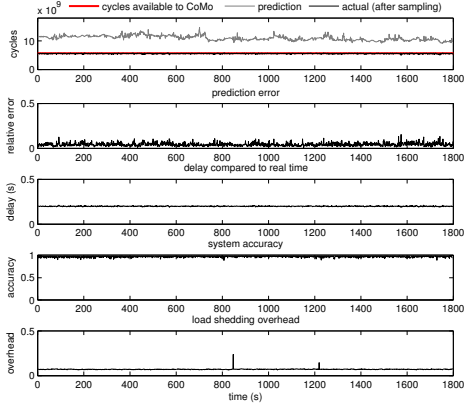


Fig. 4. Performance of a network monitoring system that supports *custom* load shedding and implements the *maxmin* strategy

On the other hand, the good performance of the original version of CoMo when $K = 0.1$ is explained by the fact that the capacity of this system is slightly larger than the rest, since it does not incur the additional load shedding overhead. The poor performance of the *eq_srates* system is also expected, given that it is not designed to consider the minimum sampling rates, resulting in a large number of violations of the minimum constraints, even when $K = 0.1$.

VI. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of our custom load shedding strategy in a fully operative network monitoring system. With these experiments we verify that, even when delegating the task of shedding excess load to non-cooperative users, the system is able to achieve robustness and a high degree of accuracy.

A. Performance under Normal Traffic

We first evaluate the performance of our custom load shedding scheme under normal traffic conditions. Figure 4 plots five different system performance parameters over time (i.e., predicted and actual CPU usage, prediction error, system delay, system accuracy and load shedding overhead, respectively) when running the nine queries on our first trace.

The first two plots show the time series of the predicted cycles per second compared to the actual CPU usage and the prediction error. The bold horizontal line depicts the total CPU cycles allocated to CoMo, which in this experiment are set in such a way that the overload factor is $K = 0.5$ in average during the entire execution. The figure shows that the prediction error is very low, resulting in an overall CPU usage (‘actual’ line) very close to the limit of available cycles (‘cycles available to CoMo’ line).

The third plot (‘delay’) shows the delay of the system compared to the real time. In principle, the delay should be at least $0.1s$, given that batches contain $0.1s$ of traffic [9]. This figure gives us a measure of the buffer size required to avoid packet losses in the case of running the system on a real link. The figure shows that the delay is almost constant and centered in $0.2s$, indicating that the system is stable and only requires a buffer able to hold two batches, the one that is

being processed and the next one. On the contrary, the delay in an unstable system would increase without bound.

The fourth plot (‘system accuracy’) shows the overall accuracy of the system over time, which is computed as the sum of the accuracy for all queries divided by the total number of queries. The figure shows that the overall accuracy of the monitoring system is very close to the maximum value of 1, even when the system is highly overloaded.

Finally, the bottom plot (‘overhead’) shows the overhead over time of the load shedding subsystem. The overhead is constant (about 7%) although the predicted cycles are quite variable (as shown in the top plot), thanks to the space-efficient algorithms used in the feature extraction phase, which have a deterministic worst case computational cost. The few spikes in the overhead are caused by context switches during the execution of the load shedding procedure.

B. Robustness against Traffic Anomalies

The performance of a network monitoring system can be highly affected by anomalies in the network traffic. For example, a system can be underutilized for a long period of time and suddenly become highly overloaded due to the presence of a Distributed Denial-of-Service attack (DDoS) or a worm spread. During these situations, the results of the system, even if approximate, are extremely valuable to network operators.

In this experiment, we evaluate the impact of network anomalies on the robustness of the monitoring system. In this particular example, we inject a massive DDoS attack every 3 minutes into our traces. The attack consists of injecting 1 new flow, with spoofed IP source addresses and ports, for every 3 flows already existing in the original trace. Although each attack lasts 1 minute, the prediction history is set to 6 seconds. Therefore, when a new attack arrives after 3 minutes the system has forgotten all previously observed attacks.

Figure 5 plots the predicted and actual CPU usage together with the system accuracy and delay during the attacks. The figure shows that the predicted cycles increase significantly during the anomaly (note logarithmic scale in the top plot) since the queries that depend on the number of flows in the traffic (e.g., *p2p-detector*) are highly affected by this type of attacks. However, the system is stable during the anomaly and its impact on the overall system accuracy and delay is negligible. Note also that although the prediction accuracy is somewhat affected at the end of each attack, all prediction errors are overestimations, given that the delay of the system decreases when the prediction error increases.

Another interesting behavior is the decrease in the actual CPU usage and delay during the first two attacks. The cause of this behavior is that the fixed cost (c_q) of the *p2p-detector* query is highly affected by these attacks, given that the first packet of each flow is always inspected. As a consequence, the core system detects that the query is not shedding the correct amount of load during the first anomalies and proceeds to penalize the query increasing its prediction as described in Section IV. Nevertheless, this situation has no impact on the

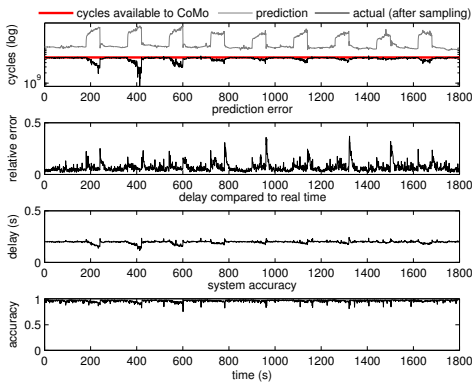


Fig. 5. Performance of the network monitoring system in the presence of massive DDoS attacks

accuracy of the other queries running on the system, as can be observed in the bottom plot of Figure 5.

C. Robustness against Selfish Queries

One of the critical aspects of our custom load shedding strategy is that the monitoring system has to operate in a non-cooperative environment with selfish users. Therefore, the enforcement policy presented in Section IV-A is crucial to achieve robustness and assure a fair allocation of computing resources to queries.

In this experiment, we evaluate the robustness of our enforcement policy in the presence of a selfish query. In particular, the selfish behavior is simulated by employing a custom load shedding method that never sheds excess load, irrespective of the amount of load shedding (r_q) requested by the core platform. We modified the *p2p-detector* query to implement this selfish custom load shedding method. The resulting query is then submitted to the monitoring system every 3 minutes and withdrawn after 1 minute. Initially, the system is running the remaining eight queries listed in Table I and it is not experiencing overload. Note that the *p2p-detector* query is the most expensive in Table I, with a cost more than 10 times greater than the rest of the queries.

Figure 6 shows that this selfish query is quickly penalized and does not have any chance to run after very few observations of its selfish behavior. Note also that the system would have enough cycles to run a version of the same query that implements a correct load shedding method instead.

The bottom plot shows the overall system accuracy without including the selfish query and confirms that the impact of this query on the accuracy of the rest of the queries is negligible. In subsequent arrivals, the query is never executed again, given that the system still maintains its MLR history for some time. This additional check to identify the query is simply done by computing a hash of the query binary code.

VII. OPERATIONAL EXPERIENCES

The objective of this section is to validate the results obtained through packet traces in an operational network with live network traffic. We deploy our monitoring system in the access link of the university presented in Section V-A and

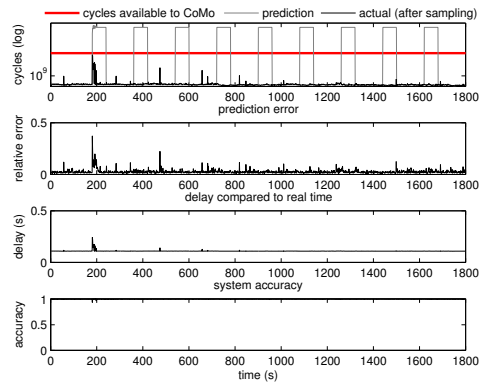


Fig. 6. Performance of the system when receiving a selfish version of the *p2p-detector* query every 3 minutes

run online the nine queries described in Table I. We present the results of a 30-minute execution of our monitoring system in an Intel Xeon running at 3 GHz. In order to measure the accuracy of the queries, we also deploy a second computer that receives an exact copy of the traffic traversing the monitored Gigabit Ethernet link using a pair of optical splitters. This additional computer only collects a packet-level trace without loss. Details of this trace are available in Section V-A. Both computers are equipped with an Endace DAG 4.3GE card [16] and their buffers are set to 256 MB.

Figure 7(a) plots the time series of the CPU cycles consumed by the monitoring system to process one second of traffic (i.e., 10 batches) together with the predicted cycles and the overhead of our load shedding scheme over time. It is clear that the monitoring system is highly overloaded since the predicted load is more than twice the total system capacity. The prediction increases over time due to the increase in the network traffic, as can be observed in Figure 7(b).

The figure confirms that our load shedding system is able to keep the CPU usage of the monitoring system consistently below the 3 GHz threshold, which marks the limit from which the system would not be stable. It succeeds in predicting the increase in the CPU demands and in adapting the CPU usage of the queries accordingly. Figure 7(c) shows how the average load shedding rate increases with the traffic load. Moreover, during the entire execution, the CPU usage is very close to the limit of available cycles, which indicates that the predictions are accurate and the system is shedding the bare minimum amount of load. The CPU usage only decreases a little bit at the end, when the predicted load is so high that the minimum requirements of all queries cannot be satisfied for some batches, and the *p2p-detector* (the most expensive query) is sometimes stopped to avoid uncontrolled packet drops.

As a result, Figure 7(b) shows that the occupation of the incoming buffer of the DAG card is controlled around 5 MB (2% of the total buffer size) and only reaches values of up to 50 MB at the beginning of the execution when the prediction system is still not trained. Since the buffer limit is never reached, no packets are dropped by the DAG card during the entire execution as depicted in Figure 7(b).

Figure 7(c) plots the overall accuracy of the queries over

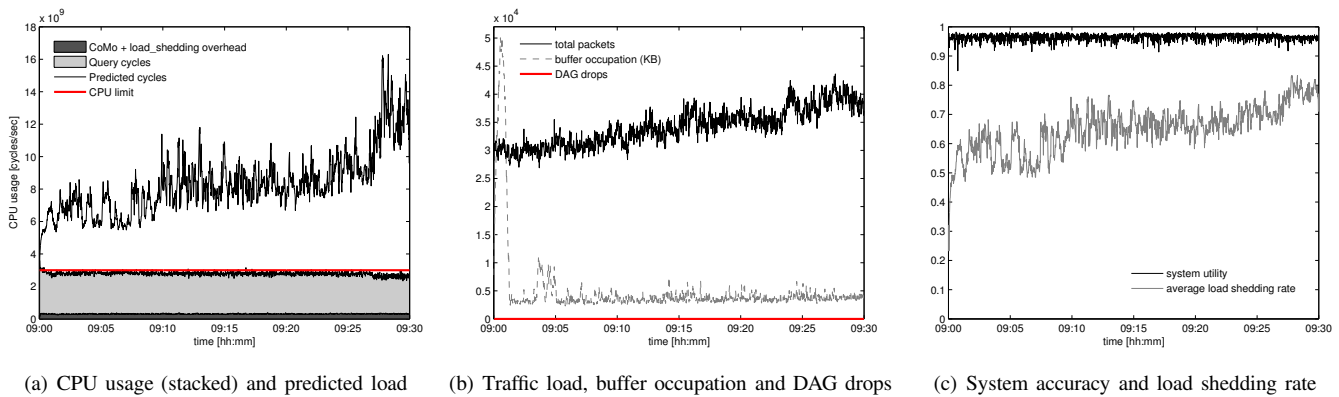


Fig. 7. Online performance of the load shedding scheme over time when running the monitoring system at the UPC access link

time. As expected, the accuracy is very high, even when the system is more overloaded, given that the minimum constraints of all queries (except *p2p-detector*) are preserved and not a single packet is dropped without control.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a load shedding framework that allows monitoring applications to use custom-defined load shedding methods. This feature is very important to those applications that are not robust against the sampling methods provided by the core platform and to those that can achieve better accuracy by using other, more appropriate, techniques.

The main novelty of our approach is that the monitoring system can still achieve robustness and fairness of service in the presence of overload situations, even when delegating the task of shedding excess load to untrusted applications. The proposed method is able to police applications that do not implement custom load shedding methods correctly using a lightweight and easy to implement technique, given that the enforcement policy is an intrinsic feature of the prediction algorithm used by the load shedding scheme.

We validated our solution to support custom load shedding methods using real-world packet traces and evaluated its online performance in a large university network. We also showed the robustness of our load shedding scheme in front of extreme overload conditions, anomalous traffic and selfish applications.

Our ongoing work is focused on extending our methods to address the resource management problem in a distributed network monitoring system. We are also interested in applying similar techniques to other system resources, such as memory, disk bandwidth and storage space.

REFERENCES

- [1] S. Guha, J. Chandrashekar, N. Taft, and K. Papagiannaki, "How healthy are today's enterprise networks?" in *Proc. of ACM IMC*, Oct. 2008.
- [2] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney, "A first look at modern enterprise traffic," in *Proc. of ACM IMC*, Oct. 2005.
- [3] V. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye, "A study of end-to-end web access failures," in *Proc. of ACM CoNEXT*, Dec. 2006.
- [4] C. Cranor *et al.*, "Gigascop: A stream database for network applications," in *Proc. of ACM SIGMOD*, Jun. 2003.
- [5] G. Iannaccone, "Fast prototyping of network data mining applications," in *Proc. of PAM*, Mar. 2006.
- [6] K. G. Anagnostakis *et al.*, "Open packet monitoring on FLAME: Safety, performance, and applications," in *Proc. of IFIP-TC6 IWAN*, Dec. 2002.
- [7] N. Spring, D. Wetherall, and T. Anderson, "Scriptroute: a public internet measurement facility," in *Proc. of USENIX USITS*, Mar. 2003.
- [8] K. Claffy *et al.*, "Community-oriented network measurement infrastructure (CONMI) workshop report," *SIGCOMM CCR*, vol. 36, no. 2, 2006.
- [9] P. Barlet-Ros *et al.*, "Load shedding in network monitoring applications," in *Proc. of USENIX Annual Technical Conf.*, Jun. 2007.
- [10] K. Keys, D. Moore, and C. Estan, "A robust system for accurate real-time summaries of internet traffic," in *Proc. of SIGMETRICS*, Jun. 2005.
- [11] N. Tatbul *et al.*, "Load shedding in a data stream manager," in *Proc. of VLDB*, Sep. 2003.
- [12] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *Proc. of IEEE ICDE*, Mar. 2004.
- [13] F. Reiss and J. M. Hellerstein, "Declarative network monitoring with an underprovisioned query processor," in *Proc. of IEEE ICDE*, Apr. 2006.
- [14] C. L. Compton and D. L. Tennenhouse, "Collaborative load shedding for media-based applications," in *Proc. of ICMS*, May 1994.
- [15] B. Babcock *et al.*, "Models and issues in data stream systems," in *Proc. of ACM PODS*, Jun. 2002.
- [16] Endace, "DAG network monitoring cards," <http://www.endace.com>.
- [17] Cisco Systems, "Sampled NetFlow," http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/12s_sanf.html.
- [18] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better NetFlow," in *Proc. of ACM SIGCOMM*, Aug. 2004.
- [19] R. R. Kompella and C. Estan, "The power of slicing in internet flow measurement," in *Proc. of ACM IMC*, Oct. 2005.
- [20] M. Welsh, D. Culler, and E. A. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *Proc. of SOSP*, Oct. 2001.
- [21] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proc. of ACM IMC*, Oct. 2003.
- [22] L. Yu and H. Liu, "Feature selection for high-dimensional data: A fast correlation-based filter solution," in *Proc. of ICML*, Aug. 2003.
- [23] P. Barlet-Ros, G. Iannaccone, J. Sanjuà-Cuxart, and J. Solé-Pareta, "Robust network monitoring in the presence of non-cooperative traffic queries," Technical University of Catalonia, Tech. Rep. DAC-2008-49, 2008, <http://people.ac.upc.edu/pbarlet/noncooperative.techrep2008.pdf>.
- [24] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, 2003.
- [25] S. Sen *et al.*, "Accurate, scalable in-network identification of P2P traffic using application signatures," in *Proc. of WWW*, May 2004.
- [26] T. Karagiannis *et al.*, "Is P2P dying or just hiding?" in *Proc. IEEE GLOBECOM*, Nov. 2004.
- [27] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, 1970.
- [28] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in *Proc. of ACM SIGCOMM*, Aug. 2003.
- [29] Q. Zhao, J. Xu, and A. Kumar, "Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation," *IEEE J. Select. Areas Commun.*, vol. 24, no. 10, Oct. 2006.
- [30] C. Barakat, G. Iannaccone, and C. Diot, "Ranking flows from sampled traffic," in *Proc. of ACM CoNEXT*, Oct. 2005.
- [31] IST-Lobster sensor at UPC, <http://loadshedding.ccaba.upc.edu/appmon>.