Robust Network Monitoring in the presence of Non-Cooperative Traffic Queries

Pere Barlet-Ros [a,*], Gianluca Iannaccone [b],
Josep Sanjuàs-Cuxart [a], Josep Solé-Pareta [a]

[a] *Dept. Arquitectura de Computadors, Universitat Politècnica de Catalunya (UPC), Campus Nord, Edif. D6, C. Jordi Girona, 1-3, 08034 Barcelona, Spain*

[b] *Intel Research, 2150 Shattuck Avenue, Berkeley, CA 94704, USA*

Abstract

We present the design of a predictive load shedding scheme for a network monitoring platform that supports multiple and competing traffic queries. The proposed scheme can anticipate overload situations and minimize their impact on the accuracy of the traffic queries. The main novelty of our approach is that it considers queries as black boxes, with arbitrary (and highly variable) input traffic and processing cost. Our system only requires a high-level specification of the accuracy requirements of each query to guide the load shedding procedure and assures a fair allocation of computing resources to queries in a non-cooperative environment. We present an implementation of our load shedding scheme in an existing network monitoring system and evaluate it with a diverse set of traffic queries. Our results show that, with the load shedding mechanism in place, the monitoring system can preserve the accuracy of the queries within predefined error bounds even during extreme overload conditions.

Key words: Network monitoring, Load shedding, Traffic sampling

1 Introduction

The ability to extract detailed information from live traffic streams is critical to network management applications such as traffic engineering, performance

* Corresponding author.
Email addresses: pbarlet@ac.upc.edu (Pere Barlet-Ros), gianluca.iannaccone@intel.com (Gianluca Iannaccone), jsanjuas@ac.upc.edu (Josep Sanjuàs-Cuxart), pareta@ac.upc.edu (Josep Solé-Pareta).

analysis and network security. The challenges in this context include the unpredictable nature of network traffic as well as the types of computations, usually unknown in advance, to be performed on the packet streams. At any given point in time, a variable number of applications may need access to the packet stream traversing the same monitoring system.

Recently, several research efforts have proposed network monitoring frameworks that abstract away the low-level details of the network traffic and allow developers to quickly design and implement new methods to process and extract information from packet streams [7,14]. These systems differ from previous designs in that they are not tailor made for a single specific application, but instead can handle multiple, concurrent monitoring applications.

In an environment where multiple monitoring applications compete for the same shared resources, ensuring fairness of service in the presence of overload is a basic requirement. Load shedding has been recently proposed as an effective alternative to over-provisioning for handling overload situations in several real-time systems [23,20,25,3]. 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. Traditionally, load shedding has been implemented by dynamically discarding part of the incoming data in the presence of overload. In this work, we address the problem of how to efficiently and fairly shed excess load from an arbitrary set of monitoring applications while keeping the measurement error within bounds defined by the end users.

There are three main requirements that make this problem particularly challenging. First, the system operates in real-time with live packet streams. Therefore, the load shedding scheme must be lightweight and quickly adapt to sudden overload situations to prevent undesired packet losses. Second, the monitoring applications are unaware of other applications running on the same system and cannot be assumed to behave in a cooperative fashion. Instead, they will always try to obtain the maximum share of the system resources. The system however must ensure fairness of service and avoid starvation of any application, while trying to satisfy their accuracy requirements. Third, to provide developers with maximum flexibility, the system has to support arbitrary monitoring applications for which the resource demands are unknown *a priori*. In addition, the input data (i.e., the network traffic) is continuous, highly variable and unpredictable in nature. As a consequence, the system cannot make any assumptions about the input traffic nor use any explicit knowledge of the cost of the monitoring applications to decide, for example, when it is the right time to shed load.

To address this third challenge, in a previous work [3] we designed a load shedding scheme that can efficiently handle extreme overload situations, without requiring explicit knowledge of the monitoring applications. The core of our

load shedding scheme is based on an on-line prediction model that allows the monitoring system to anticipate future overload situations. It infers the cost of each application from the relationship between its actual resource usage and a large set of simple (and lightweight) traffic features that summarize the incoming traffic (e.g., the number of packets, flows, unique IP destination addresses, etc.). This prediction is then used for gracefully degrading the accuracy of the monitoring applications by deciding, for example, *when* or *how much* load to shed using well-known traffic sampling techniques.

In this paper, we extend our previous load shedding scheme to address the problem of *where* to shed excess load (i.e., the amount of load to be shed in each application), which ensures robustness and fairness of service when dealing with *non-cooperative* monitoring applications.

In our previous prototype [3], an equal sampling rate is applied to each monitoring application in the presence of overload. Although this solution is fair in the number of packets processed by each application, this paper shows that the system can shed load more effectively by applying different sampling rates to different applications according to external information about their accuracy requirements (e.g., maximum loss the application can tolerate to guarantee a maximum error in the results).

Other strategies used by similar systems to decide *where* to shed load fall into two broad categories. The first includes solutions that maximize an aggregate performance metric, such as the overall system utility [23] or throughput [1,22]. We argue that these approaches, when applied to non-cooperative environments, suffer from serious fairness issues and therefore are only suitable for scenarios where the system administrator has complete control over the utility functions or priorities of each application. In this paper, we propose instead a variant of the classical max-min fair share policy that ensures fairness of service even with non-cooperative users. We model our system using game theory and show that it has a single Nash equilibrium when all players provide correct information about their resource requirements. That is, our system has the appealing feature that a user obtains maximum benefit only when providing correct information to the system.

The second category includes solutions that achieve fairness of service by assuring that each application receives an equal share of the system computing resources [16]. In contrast, in this paper we show that, in the context of network monitoring, ensuring fair access to the packet stream can significantly improve the accuracy of monitoring applications. This result indicates that in a scenario where multiple monitoring applications have to run on the same system, a packet-based scheduler can obtain better performance than the Operating System task scheduler, which is basically designed to guarantee fair access to the CPU.

The remainder of this paper is organized as follows. The next two sections review the related work and our load shedding scheme, respectively. We present our method to handle non-cooperative monitoring applications in Section 4 and model it using game theory in Section 5. Section 6 describes our testbed scenario, while Section 7 presents a performance evaluation of an actual implementation. Finally, Section 8 concludes the paper and discusses future work.

2 Related Work

In network monitoring, the simplest form of load shedding consists of discarding packets without control in the presence of overload. This naive approach is still adopted by most monitoring applications, although it is known to have a severe (and unpredictable) impact on the accuracy and effectiveness of these applications.

In order to minimize this impact, critical monitoring systems often integrate specialized hardware (e.g., DAG cards [10]) or make use of ad-hoc configurations to avoid the inherent hardware limitations of the PC-based architecture for network monitoring [21]. Although these solutions have demonstrated their effectiveness in some scenarios, they present scalability issues that make them viable only as a short term solution.

Recently, several research works have proposed solutions that offer a more robust and predictable behavior in the presence of overload. Most proposals are based on data reduction techniques (some of them adaptive), such as packet filtering, traffic sampling or data aggregation [11,16,8,6,27].

Although most of these solutions are more effective and scalable, they incur one of the following two limitations: (*i*) they are designed to address overload situations only in traffic collection devices, without considering the cost of analyzing these data on-line [6,11], or (*ii*) they are limited to a pre-defined set of traffic reports or analyses [8,16,27].

Our load shedding scheme differs from these approaches in that it can handle arbitrary network monitoring applications and operate without any explicit knowledge of their actual implementation. This way, we significantly increase the potential applications and network scenarios where a monitoring system can be used.

This flexibility however raises different problems to those addressed in previous works, such as how to ensure fairness of service. For example, [16] assumes that the average cost per packet of each component of the monitoring system is a constant share of the total CPU usage. This assumption is a di-

rect consequence of their careful design, but does not hold for any arbitrary monitoring application. Conversely, our system has to deal with arbitrary, non-cooperative monitoring applications, with different accuracy requirements and variable cost per packet.

It is however in the context of Data Stream Management Systems (DSMS) where the load shedding problem has been explored more extensively. Most load shedding designs in this area (e.g., [23,20,22]) require queries to be built out of a small set of operators, whose cost and selectivity is assumed to be known and *constant*. Our work differs again from these approaches in that our system considers queries as black boxes and therefore cannot make any assumption on their cost or selectivity to guide the load shedding procedure.

A noticeable exception is the control-based approach proposed in [24]. Nevertheless, it addresses only the problem of *when* and *how much* load to shed, while in this paper we focus mainly on the question of *where* to shed it.

More recently, some of the techniques used in [23] have been extended to the Borealis distributed DSMS [22]. The feasibility of using similar techniques to the one proposed in this paper in a distributed network monitoring system constitutes an important part of our future work.

Finally, in the Internet services space, SEDA [25] 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 this paper, as well as in [3], we show that in a network monitoring system our predictive approach can significantly reduce the impact of overload compared to a reactive one.

3 Background

We implemented our proposal as an extension to the CoMo monitoring platform [14]. CoMo is an open-source passive network monitoring system that allows for fast implementation and deployment of network monitoring applications. Applications in CoMo (or "modules" [1]) are written in the C language, making use of a feature-rich API provided by the core platform.

In order to provide the user with maximum flexibility when writing queries, CoMo does not restrict the type of computations that a plug-in module can perform. As a consequence, the platform does not have any explicit knowledge of the data structures used by the plug-in modules nor the cost of maintaining

[1] In the rest of the paper the terms *monitoring application*, *module* and *query* are used interchangeably.

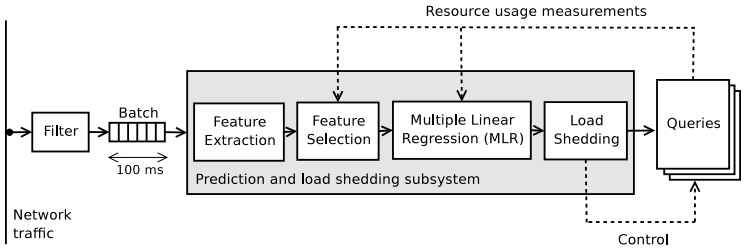


Fig. 1. System overview

them. This approach allows users to define traffic queries that otherwise could not be easily expressed using common declarative languages (e.g., SQL).

Without any knowledge of the plug-in modules, the load shedding scheme in CoMo [3] infers the cost of each query from the relation between its actual resource usage and a pre-defined set of traffic features. A *traffic feature* is simply a counter that describes a specific property of the incoming traffic (e.g., number of packets, bytes, flows, unique IP destination addresses, etc.).

The intuition behind this method comes from the empirical observation that each query incurs a different overhead when performing basic operations on the state it maintains while processing the input packet stream (e.g., creating new entries, updating existing ones or looking for a valid match). The time spent by a query is mostly dominated by the overhead of some of these operations, which can be modeled by considering the right set of simple traffic features.

Figure 1 shows the components and data flow in the CoMo system. The prediction and load shedding subsystem (in gray) intercepts the packets before they are sent to the plug-in modules implementing the traffic queries.

The system operates in four phases. First, it groups the incoming traffic in "batches" of packets. Each batch is then processed to extract a large set of pre-defined traffic features (Section 3.1). Next, the feature selection subsystem is in charge of selecting the most relevant features according to the recent history of the query's CPU usage (Section 3.2). This subset of relevant features is then given as input to the multiple linear regression (MLR) module to predict the CPU cycles required by the query to process the entire batch (Section 3.3). If the prediction exceeds the available cycles, the load shedding subsystem applies traffic sampling to reduce the load of the monitoring system (Section 4). Finally, the actual CPU usage is measured using the *time-stamp counter* [15] and fed back to the prediction subsystem to close the loop.

In case of overload, CoMo applies packet or flow sampling to the incoming traffic in order to shed excess load. Packet sampling consists of randomly selecting packets in a batch with probability p (i.e., the *sampling rate*), while flow sampling consists of randomly selecting entire flows, rather than single packets, with probability p. In order to efficiently implement flow sampling,

CoMo uses a hash-based technique called *Flowwise sampling* [9].

The remainder of this section is devoted to a quick review of the three components that perform the prediction. An extensive evaluation of the prediction accuracy and overhead is available in [3].

3.1 Feature Extraction

We are interested in finding a set of traffic features that are simple and lightweight to compute, while helpful to characterize the CPU usage of a wide range of queries. A feature that is too specific may be used to predict a given query with great accuracy, but could have a cost comparable to directly answering the query (e.g., counting the packets that contain a given pattern to predict the cost of signature-based IDS-like queries). Our goal is therefore to find features that may not explain in detail the entire cost of a query, but can provide enough information about the aspects that dominate the processing cost. For instance, in the previous example of a signature-based IDS query, the cost of matching a string will mainly depend on the number of collected bytes.

In addition to the number of packets and bytes, we maintain four counters per *traffic aggregate* that are updated every time a batch is received. A traffic aggregate considers one or more of the TCP/IP header fields: source and destination IP addresses, source and destination port numbers and protocol number. The four counters we monitor per aggregate are: (*i*) the number of unique items in the batch; (*ii*) the number of new items compared to all items seen in a measurement interval; (*iii*) the number of repeated items in the batch (i.e., items in the batch minus unique) and (*iv*) the number of repeated items compared to all items in a measurement interval (i.e., items in the batch minus new).

This large set of features (four counters per traffic aggregate plus the total packet and byte counts, i.e., 42 in our experiments) helps narrow down which basic operations performed by the queries dominate their processing costs (e.g., creating a new entry, updating an existing one or looking up entries).

In order to extract the features with minimum overhead, we implement the multi-resolution bitmap algorithms proposed in [13]. The advantage of the multi-resolution bitmaps is that they bound the number of memory accesses per packet as compared to classical hash tables and they can handle a large number of items with good accuracy and small memory footprint.

3.2 Feature Selection

Since modules consist of arbitrary code, the system cannot know in advance which features perform best as predictors for each query. Including all the extracted traffic features in the regression has several drawbacks: (*i*) the cost of the linear regression increases quadratically with the number of predictors, much faster than the gain in terms of accuracy (*irrelevant predictors*); (*ii*) even including all possible predictors, there would still be a certain amount of randomness that cannot be explained by any predictor; (*iii*) predictors that are linear functions of other predictors (*redundant predictors*) can negatively impact on the accuracy of the predictions (*multicollinearity*).

The CoMo platform uses a variant of the Fast Correlation-Based Filter (FCBF) [26], which can effectively remove both irrelevant and redundant features and is computationally very efficient.

3.3 Multiple Linear Regression

Regression analysis is a widely applied technique to study the relationship between a response variable and one or more predictor variables. The linear regression model assumes that the response variable is a linear function of the predictors. The fact that this relationship exists can be exploited for predicting the expected value of the response variable when the values of the predictors are known.

In CoMo, the linear regression is used to model the resource usage of each query from the relation between the features selected by the FCBF algorithm and the CPU usage of each query observed while processing previous batches. This model is used to predict the CPU cycles that will cost to process a given batch once the values of the individual features are known.

4 Fairness in a Non-Cooperative Environment

The load shedding strategy described in Section 3 has a major limitation: it does not differentiate among queries, since the load shedder always applies the same sampling rate to each of them. However, the system would make load shedding decisions in a more graceful and intelligent manner if it could consider some additional knowledge about the queries to guide the load shedding procedure, such as their level of tolerance to loss. For example, when using traffic sampling, some queries (e.g., ranking top flows [2]) require much higher

sampling rates than other simpler ones (e.g., packet/byte counts) to achieve the same degree of accuracy in the results.

Nevertheless, our system cannot directly measure the error of a query to infer its tolerance to loss, given that it considers them as black boxes. Thus, there is no option other than obtaining this information from the user. The main drawback of this approach is that users will tend to request the largest possible share of the resources. Therefore, the monitoring system must implement mechanisms to ensure fairness of service and make sure users provide accurate information about their queries.

4.1 Max-Min Fairness

Fairness can be defined in many different ways. A classical technique used to ensure fair access to a scarce shared resource is the *max-min fair share* allocation policy. Intuitively, the max-min fair policy maximizes the smallest allocation of the shared resource among all users: it assures that no user receives a resource share larger than its demand, whereas users with unsatisfied demands get an equal share of the resource.

Table 1 summarizes the notation used throughout this section. For each query $q \in Q$ at time t, $\widehat{d_q}$ and c_q denote the cycles predicted (using the method described in Section 3) and those actually allocated by the system, respectively. Let C be the system capacity in CPU cycles at time t.[2] A vector $c = \{c_q \mid q \in Q\}$ of allocated cycles is *feasible* if the following two constraints are satisfied:

$$\forall_{q \in Q} \;\; c_q \leq \widehat{d_q} \tag{1}$$

$$\sum_{q \in Q} c_q \leq C \tag{2}$$

The max-min fair share allocation policy is then defined as follows [4]:

Definition 1 *A vector of allocated cycles c is* max-min fair *if it is feasible, and for each $q \in Q$ and feasible $\bar{c}$ for which $c_q < \bar{c}_q$, there is some q' where $c_q \geq c_{q'}$ and $c_{q'} > \bar{c}_{q'}$.*

[2] In [3] we describe how the system capacity is measured. We also show that it varies over time due to the system overhead and the prediction error in previous time bins.

Table 1
Notation and definitions

Symbol	Definition
Q	Set of q continuous traffic queries
C	System capacity in CPU cycles
$\widehat{d_q}$	Cycles demanded by the query $q \in Q$ (prediction)
m_q	Minimum sampling rate constraint of the query $q \in Q$
c	Vector of allocated cycles
c_q	Cycles allocated to the query $q \in Q$
p	Vector of sampling rates
p_q	Sampling rate applied to the query $q \in Q$
a_q	Action of the query $q \in Q$
a_{-q}	Actions of all queries in Q except a_q
u_q	Payoff function of the query $q \in Q$

4.2 Fairness in terms of CPU Cycles

We aim at using external information to drive the load shedding decision. A possible way to express this information is by providing a utility function per query that describes how the utility varies with the sampling rate. To simplify the system and reduce the burden on the users we let the user specify only the minimum sampling rate (m_q) a query $q \in Q$ can tolerate. This allows to keep the load shedding algorithm very simple yet flexible enough to control resource usage.

Minimum constraints however are not considered in the classical definition of max-min fairness. For this reason, we modify the constraint (1) of the standard max-min fair policy by the following one in order to introduce the notion of a minimum sampling rate:

$$\forall_{q \in Q} \ (m_q \times \widehat{d_q}) \leq c_q \leq \widehat{d_q} \tag{3}$$

Depending on the query requirements and the system capacity, a max-min fair allocation that satisfies each query's minimum rate constraint may or may not exist. When no feasible solution exists, some queries have to be disabled. The strategy used by our system to encourage users to request the smallest amount of resources (i.e., low m_q) is to disable the smallest subset of $Q' \subseteq Q$ queries to satisfy (2) and (3), such that $\sum_{q' \in Q'} m_{q'} \times \widehat{d_{q'}}$ is *maximized*. That is, the system disables first the queries with the largest minimum demands.

As we show in Section 5, this (intentionally) simple strategy not only enforces users to specify m_q values as small as possible, since higher values increase the probability of being disabled in the presence of overload, but also encourages them to write queries in an efficient manner (i.e., small d_q), because given two

equivalent queries, the least demanding one will have more chances to run.

4.3 Fairness in terms of Packet Access

The strategy we described so far is max-min fair in terms of access to the CPU cycles. An alternative strategy is to be max-min fair in the access to the packet stream. The intuition behind this idea is that the number of processed packets has a stronger correlation with the accuracy of a query than just the number of allocated CPU cycles. Simpler queries, such as aggregate packet counters, tend to be more resilient to sampling and also require very few cycles to execute. On the other hand complex queries, such as top-k destinations, are more expensive and more sensitive to sampling. As a result, allocating CPU cycles may guarantee 100% sampling to simple (and cheap to execute) queries that do not need that high sampling rate while penalizing more complex queries.

A strategy that is max-min fair in terms of packet access consists of optimizing the minimum number of packets processed among all queries, rather than the allocated cycles, while satisfying the minimum sampling rate constraints.

Letting C be the system capacity in CPU cycles at time t, we say that a vector $p = \{p_q \mid q \in Q\}$ of sampling rates is *feasible* if the following two constraints are satisfied:

$$\forall_{q \in Q} \;\; m_q \leq p_q \leq 1 \tag{4}$$

$$\sum_{q \in Q} (p_q \times \widehat{d_q}) \leq C \tag{5}$$

We then define the max-min fair share policy in terms of access to the packet stream as follows:

Definition 2 *A vector of sampling rates p is* max-min fair in terms of access to the packet stream *if it is feasible, and for each $q \in Q$ and feasible $\bar{p}$ for which $p_q < \bar{p}_q$, there is some q' with $p_q \geq p_{q'}$ and $p_{q'} > \bar{p}_{q'}$.*

Like in the strategy described in Section 4.2, when no feasible solution exists, the system uses the minimum demands (i.e., $m_q \times \widehat{d_q}$) to decide which queries are allowed to run, but then allocate spare cycles according to each query per-packet processing cost.

4.4 On-line Algorithm

The main advantage of both strategies is that they are simple yet fair in a non-cooperative environment. Both strategies can run online given that an algorithm exists to compute the optimal solution in polynomial time [4].

Our algorithm is based on the classical max-min fair share algorithm [4], but it includes the minimum sampling rate constraint. It is divided into two main phases. The first phase is common to both strategies, since they both aim at satisfying the minimum requirements (m_q) and only differ on how the remaining cycles are distributed among the queries. First, it sorts the queries according to their $m_q \times \widehat{d_q}$ values and checks if the following condition can be satisfied without disabling any query:

$$\sum_{q \in Q} (m_q \times \widehat{d_q}) \leq C \tag{6}$$

If (6) is satisfied, the algorithm continues to the second phase. Otherwise, it sets c_q (or p_q when using the strategy that is fair in terms of packet access) of the query with the greatest value of $m_q \times \widehat{d_q}$ to 0 (i.e., the first query of the list is disabled), q is removed from the list, and the process is repeated again with the remaining queries.

The second phase differs depending on the strategy being implemented. In the strategy that is fair in terms of CPU access, the second phase consists of finding a vector $c' \subseteq c$ of allocated cycles that is max-min fair, while satisfying the minimum rate constraint of each query $q' \in Q'$, where Q' stands for the queries that are left in the list after the first phase. The algorithm starts allocating $m_{q'} \times \widehat{d_{q'}}$ cycles to each query. The queries are then divided in two lists. The first initially contains the query with the smallest $c_{q'}$, while the second list includes the rest of the queries sorted by ascending $c_{q'}$ values. Throughout the algorithm, the first list always contains queries with equal $c_{q'}$ that are also always smaller than any other in the second list. The $c_{q'}$ values of all queries in the first list are set to the minimum of: (*i*) the $c_{q'}$ value of the first query in the second list, (*ii*) the minimum $\widehat{d_{q'}}$ of the queries in the first list, and (*iii*) their current $c_{q'}$ plus the remaining cycles over the number of items in the first list. If (*i*) is used, the first query in the second list is moved to the first list, while if (*ii*) is used, the $c_{q'}$ of the query with minimum $\widehat{d_{q'}}$ is definitive and q' is removed from the first list. This process is repeated until there are no queries left on the lists or the system capacity is reached (i.e., when the value (*iii*) is used).

The second phase of the strategy that is fair in terms of packet access con-

sists of finding a vector $p' \subseteq p$ of sampling rates that is max-min fair, while satisfying the minimum sampling rate constraint of each query $q' \in Q'$. The algorithm starts computing a global sampling rate $r = C/\sum_{q'} d_{q'}$. Then, for all queries $q' \in Q'$ such that $m_{q'} > r$, the sampling rate $p_{q'}$ is set to $m_{q'}$. The sampling rate of these queries is definitive and they are removed from the list. Next, r is recomputed for the rest of the queries and the process is repeated again, but subtracting from the system capacity the cycles already allocated. The algorithm finishes when there is no query $q' \in Q'$ such that $m_{q'} > r$. In this case, $p_{q'}$ of the queries still remaining in the list is set to r.

5 System's Nash Equilibrium

To verify that no user has an incentive to provide incorrect m_q values, we evaluate our strategy in terms of game theory. In particular, our system can be modeled as a strategic game with Q players, where each player q corresponds to a query. Each player has a set of possible actions that consist of its minimum CPU demands, denoted by a_q (i.e., $m_q \times \widehat{d_q}$). [3] The objective of non-cooperative players is to obtain the maximum number of cycles from the system. Thus, the payoff function u_q, which specifies the player's preferences, is the number of cycles actually allocated by the system to the query q, which depends on a_q and the minimum demands of the rest of the queries a_{-q} (the $-q$ subscript stands for all queries except q).

In particular, according to the strategies described in Sections 4.2 and 4.3, our system tries to satisfy all minimum demands and eventually shares any spare cycles max-min fairly (in terms of CPU or packet access) among the queries. However, if the sum of all a_q values is greater than the system capacity, the system disables first the queries with largest a_q. We can express the payoff u_q of a query q as a function of the action profile $a = (a_q, a_{-q})$ as follows:

$$u_q(a_q, a_{-q}) = \begin{cases} a_q + mmfs_q(C - \sum_{i:u_i>0} a_i), & \text{if } \sum_{i:a_i \leq a_q} a_i \leq C \\ 0, & \text{if } \sum_{i:a_i \leq a_q} a_i > C \end{cases} \quad (7)$$

where i denotes the set of all queries ($i \in Q$) and $mmfs_q(x)$ is the max-min fair share of x cycles (in terms of CPU or packet access) that correspond to the query q given the action profile a. The first condition of Equation 7 gives us

[3] Note the difference between the full demand of a query ($\widehat{d_q}$) and its minimum demand (a_q), which denotes the number of cycles required by the query to achieve its minimum sampling rate (m_q).

the payoff u_q of a query q when the system can satisfy its minimum demand. This occurs when the sum of all minimum demands less than or equal to a_q (including a_q) is less than or equal to the system capacity C. In this case, the query will receive at least its minimum demand a_q and, if the sum of the minimum demands of the queries that remain active (i.e., those with $u_i > 0$) is less than C, the query will additionally receive its max-min fair share of the spare cycles. Note that although u_i is recursively defined, there is only one possible value for each u_i and no cycles occur. On the other hand, if a_q cannot be satisfied, no cycles are allocated to the query q and its payoff is 0, as captured in the second condition of Equation 7.

Definition 3 *A* Nash equilibrium (NE) *is an action profile a^* with the property that no player i can do better by choosing an action profile different from a_i^*, given that every player j adheres to a_j^** [19].

Theorem. Our resource allocation game has a single *Nash Equilibrium* when all players demand $\frac{C}{|Q|}$ cycles.

First, we prove that the action profile a^*, with $a_i^* = \frac{C}{|Q|}$ for all $i \in Q$, is a NE. Next, we show that in fact it is the only NE of our game.

Proof (a^* is a NE). According to Definition 3, an action profile a^* is a NE if $u_i(a^*) \geq u_i(a_i, a_{-i}^*)$ for every player i and action a_i. We differentiate two different cases that cover all possible actions with $a_i \neq a_i^*$ and show that, for none of them, a query i can obtain greater payoff than $\frac{C}{|Q|}$, which is the one it would obtain by playing a_i^*, if all other queries keep their actions fixed to $\frac{C}{|Q|}$.

(1) $a_i > \frac{C}{|Q|}$. In this case the sum of the minimum demands is greater than C. Therefore, according to Equation 7, the payoff $u_i(a_i, a_{-i}^*)$ is 0, since i is the query with the largest minimum demand.
(2) $a_i < \frac{C}{|Q|}$. In this case the sum of the minimum demands is less than C and $u_i(a_i, a_{-i}^*) = a_i + mmfs_i(\frac{C}{|Q|} - a_i)$, where $\frac{C}{|Q|} - a_i$ are the cycles left to reach the system capacity C. Independently of whether the system uses the strategy that is fair in terms of CPU or packet access, by definition $mmfs_i(x) \leq x$ and, therefore $u_i(a_i, a_{-i}^*) \leq \frac{C}{|Q|}$. □

Proof (a^* is the only NE). In order to prove that our game has a single NE, it is sufficient to show that for any action profile other than $a_i^* = \frac{C}{|Q|}$, for all $i \in Q$, there is at least one query that has an incentive to change its action. We differentiate three different cases that cover all possible situations:

(1) $\sum_i a_i > C$. In this case the system does not have enough resources to satisfy the minimum demands of all queries. Those with the largest minimum demands are disabled and obtain a payoff of 0. Therefore, at least these queries have an incentive to decrease their demands in order to

obtain a non-zero payoff.

(2) $\sum_i a_i < C$. In this case the system capacity is not reached and the spare cycles are distributed among the queries in a max-min fair fashion. Therefore, in this scenario any query would prefer to increase its minimum demand in order to capture the spare cycles rather than sharing them with other queries.

(3) $\sum_i a_i = C$ and $\exists_i : a_i \neq \frac{C}{Q}$. In this case at least one query has an incentive to increase its minimum demand in order to force the system to disable the query with the largest minimum demand and capture the cycles it would free. □

Therefore, we can conclude that our load shedding strategy intrinsically assures that no player 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 players, this strategy discourages them to specify a minimum sampling rate greater than their actual requirements, because it increases the probability of demanding more than $\frac{C}{|Q|}$ and, as a consequence, the probability of being disabled in the presence of overload.

Note also that a strategy that maximizes the sum of the query utilities, instead of the minimum allocation, such as the one used in Aurora [23], would be extremely unfair and not suitable for a non-cooperative setting. In Aurora, the Nash equilibrium is when all players ask for the maximum possible allocation, which in Aurora consists of providing an utility function that drops to zero if the sampling rate is less than 1.

6 Evaluation Scenario

In this section, we present the testbed scenario and the set of traffic queries we use to evaluate our load shedding scheme. We also study the tolerance to sampling of each query to select appropriate values for their minimum sampling rate constraints.

6.1 Dataset

We implemented both strategies in the CoMo platform and performed several experiments using a 30-minute trace collected at a Gigabit Ethernet link that connects the Catalan Research and Education Network (the *Scientific Ring*) to its Spanish counterpart (RedIRIS). The Scientific Ring is managed by the Supercomputing Center of Catalonia (CESCA) and connects more than fifty

Table 2
Description of the queries used in the experimental evaluation

Query	**Description**	**Sampling**	m_q
application	Port-based application classification	packet	0.03
autofocus	High volume traffic clusters per subnet [12]	packet	0.69
counter	Traffic load in packets and bytes	packet	0.03
high-watermark	High watermark of link utilization over time	packet	0.15
pattern search	Identifies sequences of bytes in the payload [5]	packet	0.10
super sources	Detection of sources with largest fan-out [28]	flow	0.93
top-k	Ranking of the top-k destination IP addresses [2]	packet	0.57
trace	Full-payload packet collection	packet	0.10
tuple	Per-flow classification and number of active flows	flow	0.05

universities and research centers using many different technologies that range from ADSL to Gigabit Ethernet [17].

The trace includes the entire packet contents (about 30 GB) and accounts for 49.43M packets, with an average rate of 133.04 Mbps and a peak rate of 212.22 Mbps. Although similar results were obtained using publicly available datasets, such as those in the NLANR repository [18], we present only the results when running the system on a single packet trace, given that it is the only dataset we have access to that allows us to evaluate those CoMo queries that require packet contents to operate (e.g., pattern search).

We use a packet trace for the sake of reproducibility, but all conclusions can be extended to an on-line system, given that CoMo does not make any distinctions between running on-line or off-line [3,14].

6.2 Queries

To evaluate the different load shedding strategies we use a representative set of queries that are part of the standard distribution of CoMo. We also implemented two additional queries (*uni-dimensional autofocus* [12] and *super sources* [28]) that make use of more complex data structures. Table 2 presents a summary of the queries that we believe that form a representative set of typical uses of a real-time network monitoring system.

In general, the minimum sampling rate constraint m_q of a query should be provided by the user. However, given that the queries in the standard distribution of CoMo do not provide this value yet, we perform 100 executions on our packet trace by ranging the sampling rate from 0.01 to 1 (in steps of 0.01) to determine reasonable values for m_q, which in a real scenario will depend on the user's requirements.

Table 2 presents the selected values for m_q. They are set to the minimum sampling rate that guarantees an average error below 5% in the output of each query. For all queries, we measure the relative error as $|1 - \frac{estimated\ value}{actual\ value}|$. The error of the *autofocus* query is defined as the number of clusters in the delta report (see [12]) over the total number of clusters reported by the query. The error of the *top-k* query is proportional to the number of misranked pairs of destination addresses, where the first element of a pair is in the top-k list returned by the query and the second one is outside the list [2]. The error of *super sources* is computed as the average relative error in the fan-out estimations [28]. For *pattern search* and *trace*, we define the accuracy as the overall ratio of packets processed by the query. To provide for realistic sampling requirements, we set m_q to 0.1 (i.e., 10% sampling) for these two queries.[4] Table 2 also shows that the level of tolerance of most queries to sampling is very different, resulting in very diverse values of m_q.

7 Experimental Evaluation

In this section, we evaluate our load shedding scheme in the CoMo platform. We study the performance of the two variants of our load shedding scheme, namely max-min fairness in terms of access to the CPU ($mmfs\_cpu$) and in terms of access to the incoming packet stream ($mmfs\_pkt$), with the traffic queries presented in Section 6.

7.1 Performance Results

To evaluate the performance of our solution, we compare the $mmfs\_cpu$ and $mmfs\_pkt$ strategies to three alternative systems. The first consists of a version of CoMo without any explicit load shedding scheme (no_lshed). It simply discards packets without control as buffers fill in the presence of overload. In order to estimate the error in the absence of load shedding when running on packet traces, we emulate a buffer of $200ms$ of traffic. The second alternative implements the load shedding strategy presented in [3], which applies the same sampling rate to all queries (eq_srates). In this system, when the sampling rate is below the minimum sampling rate of a query, the query is

[4] Note that usually the output of these two queries is not used directly by a user, but instead is given as input to other applications. In this case, the error should be measured in terms of the applications that use the output of these queries. Although the value of 0.1 is somewhat arbitrary, it can be considered fairly conservative given the lower sampling rates typically used by network operators for this class of resource-intensive queries.

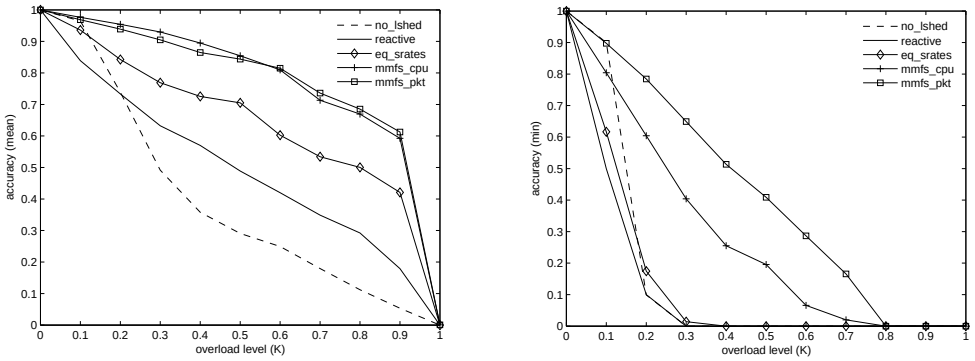


Fig. 2. Average (left) and minimum (right) accuracy of various load shedding strategies when running a representative set of queries with fixed minimum sampling rate constraints

disabled during one batch, and the sampling rate is computed again for the queries that remain active. The third alternative (*reactive*) also applies an equal sampling rate to all queries, but it implements a reactive approach instead. This system is equivalent to a predictive one, where the prediction for a batch is always equal to the cycles used to process the previous one. This strategy is similar to the one used by SEDA [25].

Figure 2 plots the average and minimum accuracy among all queries, when using the minimum sampling rate constraints defined in Table 2, at different overload levels (K) that range from 0 to 1 (in steps of 0.1). $K = 0$ denotes no overload (the system capacity C is equal to the sum of all demands), whereas $K = 1$ expresses infinite overload ($C = 0$). Therefore, the system capacity is computed as $C \times (1 - K)$. The accuracy of a query is defined as $1 - \varepsilon_q$, where ε_q is the actual error of the query as described in Section 6.2. 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% (or greater than 90% in the case of *trace* and *pattern search*), given that the minimum constraints are not considered in this system.

It is important to note that our system only requires the minimum sampling rates to operate and does not use any other external information, such as the complex utility functions needed by other systems (e.g., [23]). Throughout the evaluation, we use the accuracy of the queries as a performance metric to compare the different scheduling alternatives. However, in a real environment the users are responsible for selecting the minimum sampling rates according to their actual requirements, which may be very different for every user and may not necessarily depend on the accuracy of the queries. For example, in Section 6.2 we selected the m_q values of some queries in such a way that a maximum error in the results is guaranteed (e.g., *application*, *top-k*, etc.), while for other queries (e.g., *trace* and *pattern search*) m_q is selected according to a minimum performance requirement, without considering directly their

Table 3
Average accuracy when resource demands are twice the system capacity ($K = 0.5$)

Query	**Accuracy (mean ±stdev, $K = 0.5$)**				
	no_lshed	*reactive*	*eq_srates*	*mmfs_cpu*	*mmfs_pkt*
application	0.57 ±0.50	0.81 ±0.40	0.99 ±0.04	1.00 ±0.00	1.00 ±0.03
autofocus	0.00 ±0.00	0.00 ±0.00	0.05 ±0.12	0.97 ±0.06	0.98 ±0.04
counter	0.00 ±0.00	0.02 ±0.12	1.00 ±0.00	1.00 ±0.00	0.99 ±0.01
high-watermark	0.62 ±0.48	0.98 ±0.01	0.98 ±0.01	1.00 ±0.01	0.97 ±0.02
pattern search	0.66 ±0.08	0.63 ±0.18	0.69 ±0.07	0.20 ±0.08	0.41 ±0.08
super sources	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.95 ±0.04	0.95 ±0.04
top-k	0.42 ±0.50	0.67 ±0.47	0.96 ±0.09	0.99 ±0.03	0.96 ±0.07
trace	0.66 ±0.08	0.63 ±0.18	0.68 ±0.01	0.64 ±0.17	0.41 ±0.08
tuple	0.00 ±0.00	0.66 ±0.46	0.99 ±0.01	0.95 ±0.07	0.95 ±0.06

accuracy. Our system allows non-cooperative users to directly provide this different type of preferences without compromising the system integrity, given that a single Nash Equilibrium exists in $\frac{C}{|Q|}$, as shown in Section 5.

Figure 2 shows that the $mmfs\_cpu$ and $mmfs\_pkt$ strategies outperform the three alternative systems. 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 overhead of the load shedding scheme itself. The drop in the accuracy when $K = 1$ is also expected, given that $K = 1$ denotes zero cycles available to process queries. The figure also shows that the $mmfs\_pkt$ strategy significantly improves the minimum accuracy as compared to the $mmfs\_cpu$ strategy, while maintaining a similar average accuracy.

Table 3 presents the average accuracy broken down by query when $K = 0.5$ (i.e., when the resource demands are twice the system capacity). The table confirms that the accuracy of all queries is preserved within the predefined bounds, with a small standard deviation, when using the $mmfs\_cpu$ and $mmfs\_pkt$ strategies.

In this experiment, the minimum accuracy is driven by *pattern search* (i.e., the most expensive query in Table 3), which is commonly used for worm and intrusion detection purposes. In that case, a monitoring system implementing the $mmfs\_cpu$ strategy would miss much more intrusions than one using $mmfs\_pkt$, while obtaining similar accuracy for the rest of the queries. Although this query achieves greater accuracy in the alternative systems, note the large impact of this gain on the accuracy of the rest of the queries, resulting in a significant decrease in the fairness of service.

So far, we have only looked at the average accuracy over the entire duration of the experiment. In order to better understand the stability of the system, we plot in Figure 3 the accuracy of the *autofocus* query over time when $K = 0.2$.

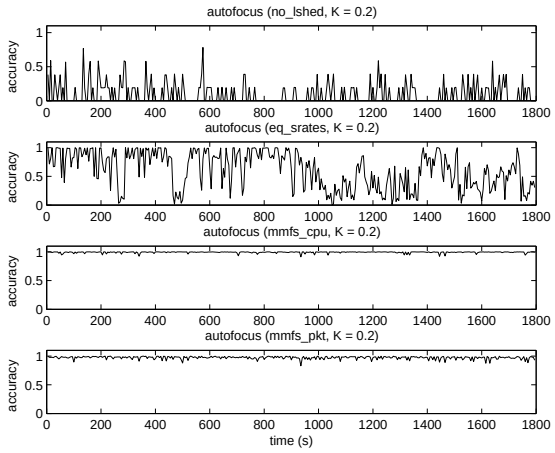


Fig. 3. *Autofocus* accuracy over time when $K = 0.2$

The figure shows the large impact of light overload situations in two alternative systems. In particular, the poor performance of the *eq_srates* system is due to the fact that the query is disabled quite frequently given the variability in the incoming traffic, although in average the sampling rate during the entire execution is above the minimum presented in Table 2. This has an important impact in the accuracy of this particular query. Instead, the $mmfs\_cpu$ and $mmfs\_pkt$ strategies are more stable and allow the system to keep the sampling rate above the minimum sampling rate, even if the incoming traffic is highly variable.

7.2 Max-Min Fairness in terms of CPU Cycles versus Packet Access

In the previous experiments, we showed the superiority of our load shedding strategies over alternative approaches. In order to study the differences between $mmfs\_cpu$ and $mmfs\_pkt$ in more detail, we set up a simple scenario that allows us to compare both load shedding strategies and easily discuss the effects of the level of overload and the minimum sampling rate constraints on the accuracy of the traffic queries.

For ease of exposition, in this experiment we only execute two queries with our packet trace in a scenario especially chosen to emphasize the differences between both load shedding strategies. The first query (*counter*) is a lightweight query that is tolerant to low sampling rates. The second query (*trace*) is more expensive, but less tolerant to packet loss. In particular, the computational cost of *trace* is approximately 10 times that of *counter*.

In this experiment, we perform 121 executions varying the minimum sampling rate (m_q) of all queries and the overload level (K) from 0 to 1. Recall that the system capacity is computed as $C \times (1 - K)$, where C is experimentally set

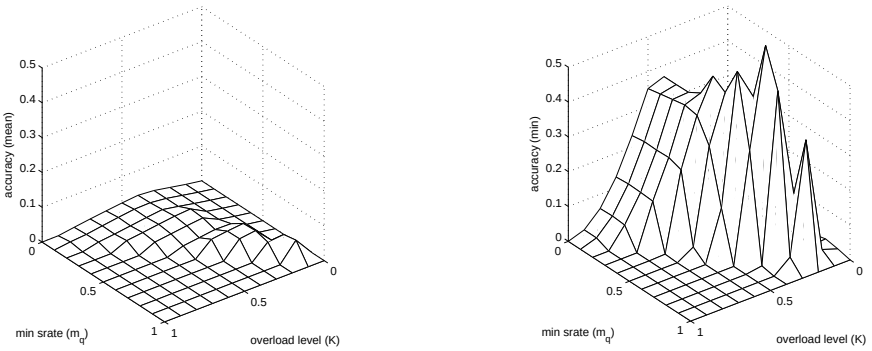


Fig. 4. Difference in the average (left) and minimum (right) accuracy between the $mmfs\_pkt$ and $mmfs\_cpu$ strategies when running 1 *trace* and 10 *counter* queries. Positive differences show the superiority of $mmfs\_pkt$ over $mmfs\_cpu$.

to the minimum number of cycles that assure that no sampling is applied in our testbed.

Figure 4 shows the difference in the accuracy of the system between the packet-based ($mmfs\_pkt$) and the CPU-based ($mmfs\_cpu$) strategies, when running 1 *trace* and 10 *counters* concurrently (i.e., the sum of the demands of the *counter* and *trace* queries is equal). While the difference in the average accuracy is negligible (it is almost a flat surface), the packet-based strategy significantly improves the minimum accuracy of the system, because $mmfs\_cpu$ highly penalizes the accuracy of the *trace* query. This result indicates that, in terms of accuracy, $mmfs\_pkt$ is significantly fairer than $mmfs\_cpu$.

7.3 Overhead

Applying different sampling rates to different queries has direct impact on the cost of running the prediction algorithm as compared to the system presented in [3], because the traffic features have to be recomputed for each query after applying traffic sampling in order to correctly update the MLR history. In contrast, in [3] the traffic features are recomputed just once, given that all queries always process equal batches.

An optimization that allows to reduce this overhead consists of scaling the actual CPU usage of each query with its sampling rate while using the original features to update the MLR history (avoiding a second full feature extraction on each batch). The overhead imposed by our load shedding system ($mmfs\_pkt$ strategy) on the entire CoMo platform is of 10.30%, with similar prediction accuracy as a system that recomputes the traffic features. The results presented in Section 7 were obtained on a system implementing this optimization.

Note also that the overhead of the different strategies to compute the max-min fair sampling rates is negligible compared to the cost of extracting the traffic features, as discussed in [3].

8 Conclusions and Future Work

Current network monitoring systems must inevitably deal with the effects of extreme overload situations, due to the large volumes, high data rates and bursty nature of network traffic. This often results in a severe and unpredictable impact on the accuracy and effectiveness of network monitoring applications.

To address this problem, we designed a load shedding scheme that can efficiently handle extreme overload situations by gracefully degrading the accuracy of the traffic queries based on an on-line prediction model of their resource requirements. The main novelty of our approach is that it considers the queries as black boxes with arbitrary (and highly variable) resource consumption. This way, we increase the potential applications and network scenarios where the monitoring system can be used.

In this paper, we presented a load shedding strategy that minimizes the impact of overload situations on the accuracy of the queries by using external information about their accuracy requirements to guide the load shedding procedure. Our method ensures that excess load is shed in a fair manner among the queries, and that any external information used by the system is correct and free of bias, even when dealing with non-cooperative users.

We implemented our load shedding scheme in an existing monitoring system and evaluated it with a diverse set of real traffic queries. Our results confirm that our strategy ensures fairness of service and maintains high levels of accuracy for all queries, even in the presence of severe overload situations. The results also show that a packet-based scheduler is preferable for handling multiple queries in a network monitoring system over the more common approach of providing fair access to the CPU used by typical Operating System task schedulers.

Our ongoing work is focused on extending our current prototype to support other load shedding mechanisms, such as lightweight summaries [20] and custom load shedding mechanisms, for those queries that are not robust against packet and flow sampling. We also plan to extend our methods to address the resource management problem in a distributed network monitoring system. Finally, we are interested in applying similar techniques to other system resources, such as memory, disk bandwidth and storage space.

9 Availability

The source code of the load shedding system presented in this paper is publicly available at `http://loadshedding.ccaba.upc.edu`. The CoMo monitoring system is also available at `http://como.sourceforge.net`.

Acknowledgments

This work was funded by a University Research Grant awarded by the Intel Research Council, and by the Spanish Ministry of Education (MEC) under contract TSI2005-07520-C03-02 (CEPOS) and TEC2005-08051-C03-01 (CATARO). The authors thank the Supercomputing Center of Catalonia for allowing them to collect the packet traces used in this work. We are also grateful to Ricard Gavaldà and Albert Bifet from the LSI department of UPC for useful discussion on game theory. The authors also wish to thank the anonymous reviewers for their helpful comments on an earlier draft of this paper.

References

[1] L. Amini, et al., Adaptive control of extreme-scale stream processing systems, in: Proc. of IEEE Intl. Conf. on Distributed Computing Systems, 2006.

[2] C. Barakat, G. Iannaccone, C. Diot, Ranking flows from sampled traffic, in: Proc. of ACM CoNEXT, 2005.

[3] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, J. Solé-Pareta, Load shedding in network monitoring applications, in: Proc. of USENIX Annual Technical Conf., 2007.

[4] D. Bertsekas, R. Gallager, Data Networks, 2nd ed., Prentice Hall, 1992.

[5] R. S. Boyer, J. S. Moore, A fast string searching algorithm., Commun. ACM 20 (10).

[6] Cisco Systems, NetFlow services and applications, White Paper (2000).

[7] C. Cranor, et al., Gigascope: A stream database for network applications, in: Proc. of ACM Sigmod, 2003.

[8] H. Dreger, A. Feldmann, V. Paxson, R. Sommer, Operational experiences with high-volume network intrusion detection, in: Proc. of ACM Conf. on Computer and Communications Security, 2004.

[9] N. Duffield, Sampling for passive internet measurement: A review, Statistical Science 19 (3).

[10] Endace, http://www.endace.com.

[11] C. Estan, K. Keys, D. Moore, G. Varghese, Building a better NetFlow, in: Proc. of ACM Sigcomm, 2004.

[12] C. Estan, S. Savage, G. Varghese, Automatically inferring patterns of resource consumption in network traffic, in: Proc. of ACM Sigcomm, 2003.

[13] C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links, in: Proc. of ACM Sigcomm Internet Measurement Conf., 2003.

[14] G. Iannaccone, Fast prototyping of network data mining applications, in: Proc. of Passive and Active Measurement Conf., 2006.

[15] Intel, The IA-32 Intel Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Intel Corporation, 2006.

[16] K. Keys, D. Moore, C. Estan, A robust system for accurate real-time summaries of internet traffic, in: Proc. of ACM Sigmetrics, 2005.

[17] L'Anella Científica, http://www.cesca.es/en/comunicacions/anella.html.

[18] NLANR: National Laboratory for Applied Network Research, http://www.nlanr.net.

[19] M. J. Osborne, An Introduction to Game Theory, Oxford University Press, 2004.

[20] F. Reiss, J. M. Hellerstein, Declarative network monitoring with an underprovisioned query processor., in: Proc. IEEE Intl. Conf. on Data Engineering, 2006.

[21] F. Schneider, J. Wallerich, A. Feldmann, Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware., in: Proc. of Passive and Active Measurement Conf., 2007.

[22] N. Tatbul, U. Çetintemel, S. Zdonik, Staying FIT: Efficient load shedding techniques for distributed stream processing, in: Proc. of Very Large Data Bases Conf., 2007.

[23] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, M. Stonebraker, Load shedding in a data stream manager, in: Proc. of Very Large Data Bases Conf., 2003.

[24] Y.-C. Tu, S. Liu, S. Prabhakar, B. Yao, Load shedding in stream databases: a control-based approach, in: Proc. of Very Large Data Bases Conf., 2006.

[25] M. Welsh, D. E. Culler, E. A. Brewer, SEDA: An architecture for well-conditioned, scalable internet services, in: Proc. of ACM Symp. on Operating Systems Principles, 2001.

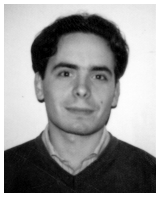
[26] L. Yu, H. Liu, Feature selection for high-dimensional data: A fast correlation-based filter solution., in: Proc. of Intl. Conf. on Machine Learning, 2003.

[27] L. Yuan, C.-N. Chuah, P. Mohapatra, ProgME: Towards programmable network measurement, in: Proc. of ACM Sigcomm, 2007.

[28] Q. Zhao, J. Xu, A. Kumar, Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation, IEEE J. Select. Areas Commun. 24 (10).

Vitae

Pere Barlet-Ros received a M.Sc. degree in Computer Science from the Universitat Politècnica de Catalunya (UPC) in 2003. He is currently an Assistant Professor and Ph.D. candidate at the Computer Architecture Department of UPC. He was also a visiting Ph.D. student at the National Laboratory for Applied Network Research (2004), Intel Research Cambridge (2004) and Berkeley (2007). His research interests are in the fields of network measurements, traffic analysis and evaluation of network performance.



Gianluca Iannaccone received his B.S. and M.S. degrees in computer engineering from the University of Pisa, Italy, in 1998. He received a Ph.D. degree in computer engineering from the University of Pisa in 2002. He joined Sprint as a research scientist in October 2001 working on network performance measurements, loss inference methods, and survivability of IP networks. In September 2003 he joined Intel Research, Cambridge, UK, and recently moved to Intel Research, Berkeley, California. His current interests include network measurements, router architecture design, and network security and troubleshooting.

Josep Sanjuàs-Cuxart is a Ph.D. student at the Computer Architecture Department of the Universitat Politècnica de Catalunya (UPC), where he received a M.Sc. degree in Computer Science in 2006. He was a visiting student at Intel Research Cambridge for six months in 2006 and currently holds a Projects Scholarship at UPC. His research interests are centered around traffic analysis and the design of network monitoring systems.

Prof. Josep Solé-Pareta obtained his M.Sc. degree in Telecom Engineering in 1984, and his Ph.D. in Computer Science in 1991, both from the UPC. In 1984 he joined the Computer Architecture Department of UPC. Currently he is Full Professor with this department. He did a Postdoc stage (1993 and 1994) at the Georgia Institute of Technology. He is co-founder of the UPC-CCABA. His publications include several book chapters and more than 100 papers in relevant research journals (> 20), and refereed international conferences. His current research interests are in Autonomic Communications, Traffic Monitoring and Analysis and High Speed and Optical Networking, with emphasis on traffic engineering, traffic characterization, MAC protocols and QoS provisioning. He has participated in many European projects dealing with Computer Networking topics.