Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors

Load Shedding in Network Monitoring Applications

Pere Barlet-Ros

Advisor: Dr. Gianluca Iannaccone
Co-advisor: Prof. Josep Solé-Pareta

A Thesis Presented to the Universitat Politècnica de Catalunya
in Fulfillment of the Requirements for the Degree of

Doctor in Computer Science

Barcelona, December 2008

Acknowledgments

This thesis would not have been possible without the constant help and guidance of my advisor Dr. Gianluca Iannaccone. His research expertise and technical skills have helped me to significantly improve the quality of my research. I also thank Gianluca for providing me with the opportunity to spend two summers working at Intel Research Cambridge and Berkeley, where I developed part of the research work presented in this thesis. I do not forget the day when we discussed the first idea of this thesis on his whiteboard during one of my visits in Cambridge.

I would also like to thank my co-advisor Prof. Josep Solé-Pareta for his confidence in me during this whole time. I greatly appreciate his continuous encouragement and help with any problems I had during the course of this thesis. I also learned many things from Josep, especially about the management aspects of the research, that will certainly be very helpful for me in the future.

I want to express my sincere gratitude to Josep Sanjuàs-Cuxart and Diego Amores-López for their assistance in the development tasks of the load shedding prototype presented in this thesis and for the quality of their work. Their constructive ideas and discussions were always of great value to me. I wish them all the success they deserve in their respective projects.

Several people have also contributed directly or indirectly to this thesis. I am very grateful to Dr. Ricard Gavaldà and Albert Bifet from the LSI department of UPC for helpful discussions on game theory. I would like to thank all the people of the CBA research group, and especially my colleagues Albert Cabellos-Aparicio and René Serral-Gracià, for many constructive conversations that provided me with a different perspective of my research. I am also very thankful to Prof. Jordi Domingo-Pascual for his help and advice whenever I needed it.

I would also like to thank the members of my Ph.D. committee (Prof. Jordi Domingo-Pascual, Dr. Ricard Gavaldà, Prof. Fernando Boavida, Dr. Fabio Ricciato and Dr. Tanja Zseby), the external reviewers (Dr. Fabio Ricciato and Dr. Mikhail Smirnov) and my

pre-defense committee (Prof. Jordi Domingo-Pascual, Dr. Ricard Gavaldà and Dr. David Carrera) for reading the manuscript of this thesis and providing useful comments.

I wish to thank CESCA (Caterina Parals and Carles Fragoso) and UPCnet (José Luis Montero) for allowing me to collect the packet traces used in this work and to evaluate the prototype developed in this thesis in the Catalan Research and Education Network (Anella Científica) and in the UPC access link, respectively.

I also thank NLANR and Endace for giving me the opportunity of doing a research stay in Hamilton (New Zealand) at the beginning of my Ph.D. In particular, I wish to thank Jörg Micheel for his help and hospitality during my stay.

I would like to acknowledge the Intel Research Council for funding three years of this research through a University Research Grant. This work was also funded in part by the Spanish Ministry of Education under contracts TSI2005-07520-C03-02 (CEPOS) and TEC2005-08051-C03-01 (CATARO).

I will always be grateful to my parents and sister for their unconditional support and encouragement. Finally, I would like to express my deepest gratitude to Montse. *Gràcies per fer-me costat durant tot aquest temps i per la teva ajuda, especialment en els moments més difícils.*

Abstract

Monitoring and mining real-time network data streams are crucial operations for managing and operating data networks. The information that network operators desire to extract from the network traffic is of different size, granularity and accuracy depending on the measurement task (e.g., relevant data for capacity planning and intrusion detection are very different). To satisfy these different demands, a new class of monitoring systems is emerging to handle multiple and arbitrary monitoring applications.

Such systems must inevitably cope with the effects of continuous overload situations due to the large volumes, high data rates and bursty nature of the network traffic. These overload situations can severely compromise the accuracy and effectiveness of monitoring systems, when their results are most valuable to network operators.

In this thesis, we propose a technique called load shedding as an effective and low-cost alternative to overprovisioning in network monitoring systems. It allows these systems to handle efficiently overload situations in the presence of multiple, arbitrary and competing monitoring applications. We present the design and evaluation of a predictive load shedding scheme that can shed excess load in front of extreme traffic conditions and maintain the accuracy of the monitoring applications within bounds defined by end users, while assuring a fair allocation of computing resources to non-cooperative applications.

The main novelty of our scheme is that it considers monitoring applications as black boxes, with arbitrary (and highly variable) input traffic and processing cost. Without any explicit knowledge of the application internals, the proposed scheme extracts a set of features from the traffic streams to build an on-line prediction model of the resource requirements of each monitoring application, which is used to anticipate overload situations and control the overall resource usage by sampling the input packet streams. This way, the monitoring system preserves a high degree of flexibility, increasing the range of applications and network scenarios where it can be used.

Since not all monitoring applications are robust against sampling, we then extend our load shedding scheme to support custom load shedding methods defined by end

users, in order to provide a generic solution for arbitrary monitoring applications. Our scheme allows the monitoring system to safely delegate the task of shedding excess load to the applications and still guarantee fairness of service with non-cooperative users.

We implemented our load shedding scheme in an existing network monitoring system and deployed it in a research ISP network. We present experimental evidence of the performance and robustness of our system with several concurrent monitoring applications during long-lived executions and using real-world traffic traces.

Resum

El monitoratge i l'anàlisi de fluxos continus de tràfic són operacions fonamentals per a la gestió i l'operació de les xarxes de computadors. La informació que els operadors de xarxa desitgen extreure del tràfic és de diferent mida, granularitat i precisió segons el tipus de mesura (p.ex. la informació rellevant per a les tasques de planificació de capacitat és molt diferent a la necessària per a la detecció d'intrusions). Amb l'objectiu de satisfer aquestes diferents necessitats, actualment està emergint una nova classe de sistemes de monitoratge de xarxa que permet l'execució de múltiples aplicacions arbitràries de monitoratge de tràfic.

Aquests sistemes han d'enfrontar-se inevitablement als efectes de situacions de sobrecàrrega continues degudes a l'elevat volum de dades, les altes taxes de transmissió i a la naturalesa variable del tràfic de xarxa. Aquestes situacions de sobrecàrrega poden comprometre severament la precisió i l'efectivitat dels sistemes de monitoratge, precisament quan els seus resultats són més valuosos per als operadors de xarxa.

En aquesta tesi es proposa una tècnica anomenada *load shedding* (o despreniment de càrrega) com una alternativa efectiva i de baix cost al sobredimensionament de recursos en sistemes de monitorage de xarxa. Aquesta tècnica permet gestionar eficientment les situacions de sobrecàrrega en la presència de múltiples aplicacions arbitràries de monitoratge que competeixen pels mateixos recursos compartits. Aquest treball presenta el disseny i l'avaluació d'un esquema de *load shedding* predictiu que és capaç de desprendre's de l'excés de càrrega davant de condicions extremes de tràfic i de mantenir la precisió de les aplicacions de monitoratge dins d'uns límits definits pels usuaris finals, mentre que assegura una distribució equitativa dels recursos a aplicacions no cooperatives.

La principal novetat d'aquest esquema és que considera les aplicacions de monitoratge com a caixes negres, amb tràfic d'entrada i cost de processament arbitraris i molt variables. Sense cap coneixement explícit dels detalls interns de les aplicacions, l'esquema proposat extreu un conjunt d'atributs del tràfic d'entrada i construeix un model de predicció en línia de la demanda de recursos de cada aplicació. Aquest model és utilitzat

per anticipar les situacions de sobrecàrrega i controlar l'ús global de recursos mitjançant el mostreig del tràfic d'entrada. D'aquesta manera, el sistema de monitoratge preserva un alt grau de flexibilitat, que incrementa el rang d'aplicacions i escenaris de xarxa en els que pot ser utilitzat.

Donat que no totes les aplicacions de monitoratge són robustes al mostratge de tràfic, també es presenta una extensió de l'esquema de *load shedding* per tal de suportar mètodes de *load shedding* definits pels usuaris finals, amb l'objectiu de proporcionar una solució genèrica per a aplicacions arbitràries de monitoratge. L'esquema proposat permet al sistema de monitoratge delegar de forma segura la tasca de despreniment de càrrega a les aplicacions, i tot i així garantir un servei equitatiu en la presència d'usuaris no cooperatius.

L'esquema de *load shedding* ha estat implantat en un sistema de monitoratge de xarxa existent i desplegat en una xarxa acadèmica i de recerca. Aquesta tesi presenta evidències experimentals del rendiment i la robustesa del sistema proposat, amb diverses aplicacions de monitoratge concurrents, durant execucions de llarga durada i utilitzant traces de tràfic de xarxes reals.

Contents

List of Figures

List of Tables

List of Acronyms

ADSL	Asymmetric Digital Subscriber Line
AMP	Active Measurement Project (NLANR)
API	Application Programming Interface
BPF	Berkeley Packet Filter
BSD	Berkeley Software Distribution
CCABA	Advanced Broadband Communications Center
CDF	Cumulative Distribution Function
CESCA	Centre de Supercomputació de Catalunya
CoMo	Continuous Monitoring
CPU	Central Processing Unit
CQL	Continuous Query Language
DBMS	Database Management System
DDoS	Distributed Denial-of-Service attack
DMA	Direct Memory Access
DoS	Denial-of-Service attack
DSMS	Data Stream Management System
EWMA	Exponentially Weighted Moving Average
FCBF	Fast Correlation-Based Filter
FIFO	First In First Out
FIT	Feasible Input Table (Borealis)
FLAME	Flexible Lightweight Active Measurement Environment
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Provider
LSRM	Load Shedding Road Map (Aurora)

MLR	Multiple Linear Regression
MMFS-CPU	Max-Min Fair Share in terms of CPU Access
MMFS-PKT	Max-Min Fair Share in terms of Packet Access
NE	Nash Equilibrium
NIC	Network Interface Card
NIMI	National Internet Measurement Infrastructure
NLANR	National Laboratory for Applied Network Research
OLS	Ordinary Least Squares
P2P	Peer-to-Peer
PC	Personal Computer
PMA	Passive Measurement and Analysis project (NLANR)
PMC	Performance-Monitoring Counter
PSAMP	Packet Sampling Working Group (IETF)
QoS	Quality of Service
RFC	Request for Comments
S&H	Sample and Hold
SEDA	Staged Event-Driven Architecture
SLR	Simple Linear Regression
SNMP	Simple Network Management Protocol
SONET	Synchronous Optical Networking
SQL	Structured Query Language
STREAM	Stanford Stream Datamanager
SVD	Singular Value Decomposition
SYN	Synchronize Packet (TCP)
TCP	Transmission Control Protocol
TSC	Time-Stamp Counter
UPC	Universitat Politècnica de Catalunya

Chapter 1

Introduction

Network monitoring applications are prone to continuous and drastic overload situations, due to the ever-increasing link speeds, the complexity of traffic analysis tasks, the presence of anomalous traffic and network attacks or simply given the bursty nature of the network traffic. Overload situations can have a severe and unpredictable impact on the accuracy of monitoring applications, right when they are most valuable to network operators. In this thesis, we address the problem of how to efficiently handle extreme overload situations in network monitoring, given that the alternative of overprovisioning monitoring systems to handle peak rates or any possible traffic mix is simply not possible or extremely expensive.

This chapter discusses the motivation behind this dissertation and presents the main challenges involved in the management of overload situations in network monitoring systems. It also introduces the problem space and highlights the main contributions of this thesis. The chapter concludes with an outline of the structure of this document.

1.1 Motivation and Challenges

Data networks are continuously evolving and becoming more difficult to manage. As a result, in recent years network monitoring has become an activity of vital importance for operating and managing data networks. Network operators are increasingly deploying network monitoring infrastructures to collect and analyze the traffic from operational networks in real-time. The information they provide is crucial for the tasks of traffic engineering, capacity planning, traffic accounting and classification, anomaly and intrusion detection, fault diagnosis and troubleshooting, evaluation of network performance, usage-based charging and billing, among others.

However, developing and deploying network monitoring applications is often a complex task. On the one hand, this type of applications has to deal with continuous traffic streams from a large number of high-speed data sources (e.g., 10 Gb/s links) with highly variable data rates (e.g., self-similar traffic [92, 109]). On the other hand, they have to operate across a wide range of network devices, transport technologies, hardware architectures and system configurations. For example, different measurement devices (e.g., standard NICs [129], DAG cards [53], network processors [135, 59], NetFlow-enabled routers [35] or SNMP-based collectors [29]) implement different interfaces to collect and process the network traffic traversing a network monitoring system.

Consequently, the developers of each single network monitoring application have to consider most of these particularities when building their applications, thus increasing their complexity, development times and probability of introducing undesired errors. Therefore, the main complexity (and amount of code) of these applications is often dedicated to deal with these lateral aspects, which are usually common to any monitoring application (e.g., traffic collection, filtering, address anonymization, resource management, etc.).

At the same time, there is an increasing demand for open monitoring infrastructures that allow the measurement community to fast prototype new monitoring applications and share measurement data from multiple network viewpoints in order to test and validate novel traffic analysis methods and to study the properties of network traffic or the behavior of network protocols [36].

In order to address these issues, 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 proposals is to abstract away the inner workings of the measurement infrastructure [40, 69] and allow arbitrary monitoring applications, developed independently by third parties, to run effectively on a shared measurement infrastructure [5, 123, 69, 75]. These systems differ from previous designs in that they are not tailor made for a single specific application, but instead they can handle multiple, concurrent monitoring applications.

The main challenge in these systems is to keep up with ever-increasing input data rates and processing requirements. Data rates are driven by the increase in network link speeds, application demands and the number of end-hosts in the network. The processing requirements are growing to satisfy the demands from network operators of obtaining an ever more detailed representation of the traffic traversing the network to improve the end-user experience and the overall "health" of the infrastructure [64, 107, 106]. For example, there is a growing interest for monitoring applications that require tracking

and inspection of a large number of concurrent network connections for intrusion and anomaly detection purposes [108, 119].

This challenge is made even harder as network operators expect the monitoring applications to return accurate enough results in the presence of extreme or anomalous traffic patterns, when the system is under additional stress. In this context, the ability to adapt in a timely fashion the resource consumption of a monitoring system in front of overload situations is crucial to achieve robustness against extreme traffic conditions or other network anomalies that could be malicious (e.g., DoS attacks or worm infections) or unexpected (e.g., network misconfigurations or flash crowds). During these events, the resource requirements of the monitoring applications could easily overwhelm the system resources leading to unpredictable results, or even interrupted service, right when the measurements are the most valuable to the network operators.

Unfortunately, previous network monitoring system designs do not directly address the serious problem of how to efficiently handle overload situations, when resource demands clearly exceed the system capacity. The alternative of overprovisioning the system to handle peak rates or worst case traffic mixes has two major drawbacks. First, it is not economically feasible in general and can result in a highly underutilized system based on an extremely pessimistic estimation of workload [79]. For example, it would require dimensioning the system buffers to absorb sustained peaks in the case of anomalies or extreme traffic mixes. Second, it would necessarily lead to reduce the flexibility and possible applications of the monitoring system [85].

Load shedding has been recently proposed as an effective alternative to overprovisioning for handling overload situations in other real-time systems [133, 128, 10, 118, 131]. 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. The idea of load shedding originally comes from the field of electric power management, where it consists of intentionally disconnecting the electric current on certain lines when the demand becomes greater than the supply [67].

In this thesis, we address the problem of how to efficiently and fairly shed excess load from an arbitrary set of network 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.

We focus our study on the CoMo (Continuous Monitoring) system [69], a general-purpose network monitoring platform that supports multiple, competing monitoring applications. CoMo provides a common framework that abstracts away the main difficulties of dealing with the different hardware technologies used to collect and process the network traffic. The key differential aspect from previous designs is that the system allows users to express arbitrary monitoring applications using imperative programming languages. CoMo is open source and is publicly available under a BSD-style license [130].

1.2 Problem Space

In a distributed network monitoring infrastructure, there are two possible resource management actions to address overload situations. The first consists of trying to solve the problem locally (e.g., to apply sampling where an overload situation is detected). The second option is to take a global action (e.g., to distribute excess load among the monitors of the infrastructure). If no actions are taken in a timely manner, queues will form increasing response delays and, eventually, the platform will experience uncontrolled packet losses, leading to a severe and unpredictable impact on the accuracy of the results.

Local resource management techniques are needed to manage the available system resources in a single monitor, according to a given policy. For example, such a policy might reduce the response times of monitoring applications or traffic queries, while minimizing the impact of overload situations on the accuracy of the results. Admission control (e.g., rejecting incoming queries) is not an option, since queries already running in the system may also exceed the system capacity. We refer to this problem as the *local resource management problem*.

Given that new network monitoring platforms are distributed systems in nature, global decisions to overcome overload situations can also be made. Global resource

	Static (offline)	*Dynamic (online)*
Local	• Static assignment of queries • Resource provisioning	• **Load shedding** • **Query scheduling**
Global	• Placement of monitors • Placement of queries	• Dissemination of queries • Load distribution

Table 1.1: Resource management problem space

management techniques are used to distribute the monitoring applications among the multiple monitoring systems in order to balance the load of the infrastructure. However, traditional load balancing and load sharing approaches used in other contexts are usually not suitable for network monitoring. The main reason is that neither the incoming traffic nor most applications can be easily migrated to other monitors, since the interesting traffic resides on the network where the monitor is attached to. We refer to this problem as the *global resource management problem*.

On the other hand, some resource management decisions can be made statically (i.e., at configuration time) or dynamically (i.e., at run time). Table 1.1 presents the resource management problem space in the context of network monitoring systems, which can be divided into four dimensions according to whether decisions are made offline or online, and if they are local (i.e., in a single monitor) or global (i.e., involving multiple monitors):

1. The *local static resource management problem* can be divided into two different sub-problems: (*i*) provisioning of system resources (i.e., CPU, memory, I/O bandwidth, storage space, etc.) according to the properties of the network under study (e.g., network bandwidth, traffic characteristics, etc.) and (*ii*) static planning of a fixed set of monitoring applications or queries to be executed in the network monitor.

2. The *global static resource management problem* refers to the placement of both monitors over the network (i.e., where to place the network monitors according to a given budget and/or measurement goals) and the static distribution of monitoring applications or queries over the available monitors (e.g., [126, 18, 77]).

3. The *local dynamic resource management problem* consists of managing the local monitoring applications or queries given the available resources to ensure fairness of service and maximize the utility of the system according to a given policy.

4. The *global dynamic resource management problem* basically refers to how to distribute the load of the platform among the multiple monitors in an effective and efficient manner.

Although resource management techniques have been extensively studied in other contexts, such as operating systems [93, 115, 80], distributed systems [28, 88], real-time databases [3, 114, 105] or multimedia systems [101, 102], network monitoring systems have several particularities that render solutions adopted in other contexts unsuitable. These differences can be summarized as follows:

1. *Arbitrary input.* Traditional resource management techniques have been designed for pull-based systems, where data feed rates can be easily managed, given that the relevant data reside on disk. On the contrary, in network monitoring the input data is the network traffic, which is generated by external sources that cannot be controlled by the monitoring system. Network traffic is highly variable and unpredictable in nature, and typically peak rates are several orders of magnitude greater than the average traffic. Thus, provisioning a network monitoring system to handle peak rates is not possible. However, it is usually during these bursts when the monitoring system is most needed and results are more critical (e.g., to detect network attacks or anomalies). For this reason, network operators are particularly interested in capturing the properties of the traffic during overload situations.

2. *Data rates and volume.* The input rates and volume of data in an online network monitoring system are usually extremely high (e.g., 10 Gb/s). Traditional pull-based systems do not target the high data rates involved in network monitoring. This makes traditional approaches, where data are firstly loaded into static databases, inviable in this scenario.

3. *Arbitrary computations.* On the one hand, the load of monitoring applications heavily depends on the incoming traffic, which is unpredictable in nature. On the other hand, their resource consumption depends on their actual implementation, which is also arbitrary. In particular, new network monitoring systems allow users to express monitoring applications or queries with arbitrary resource requirements (e.g., written in imperative programming languages). As a result, most applications do not have a fixed cost per packet. For example, a worm detection query may be idle for a long period of time until attack traffic appears in the network.

4. *Real-time results.* Several pull-based resource management techniques assume that applications do not have severe real-time requirements. On the contrary, most network monitoring applications require a timely response, whereas some of them may even come with an explicit deadline the monitoring system must assure. For those applications, late results may be useless (e.g., virus and worm detection).

1.3 Thesis Overview and Contributions

This thesis studies the local dynamic resource management problem in the context of network monitoring (see Table 1.1) and addresses the challenges involved in the management of overload situations in network monitoring systems (see Section 1.1).

Recently, several research proposals have also addressed these challenges in different real-time and stream-based systems [128, 10, 118, 54, 87, 85]. The solutions introduced belong to two broad categories. The first includes approaches that consider a pre-defined set of metrics and can report approximate results in the case of overload [54, 87, 85]. The second category includes solutions adopted in the context of Data Stream Management Systems (DSMS) that define a declarative query language with a small set of operators for which the resource usage is assumed to be known [128, 10, 118]. In the presence of overload, operator-specific load shedding techniques are implemented (e.g., selectively discarding some records, computing approximate summaries) so that the accuracy of the entire query is preserved within certain bounds.

These solutions present two common limitations: (*i*) they restrict the types of metrics that can be extracted from the traffic streams, limiting therefore the possible uses and applications of these systems, and (*ii*) they assume explicit knowledge of the cost and selectivity of each operator, requiring a very careful and time-consuming design and implementation phase for each of them. In addition, recent studies have reported poor performance of some DSMS when used for network monitoring purposes [112, 122]. This hinders their deployment in the high-speed networks traditionally targeted by the network monitoring community.

In order to address these limitations, in this thesis we present a novel load shedding scheme for network monitoring systems that: (*i*) it does not require any explicit knowledge of the monitoring applications or the type of computations they perform (e.g., flow classification, maintaining aggregate counters, pattern search), (*ii*) it does not rely on any specific model for the incoming traffic, and (*iii*) it can operate in real-time in high-speed networks. This way, we preserve the flexibility of the monitoring system, enabling fast implementation and deployment of new network monitoring applications.

The core of our load shedding scheme consists of the real-time modeling and prediction of the system CPU usage that allows the system to *anticipate* future bursts in the resource requirements. Without any knowledge of the computations performed on the packet streams, the system infers their cost from the relation between a large set of pre-defined *features* of the input stream and the actual resource usage. A feature is a counter that describes a specific property of a sequence of packets (e.g., number of unique source IP addresses). The intuition behind this method comes from the empirical observation that the cost of a monitoring application is often dominated by the overhead of basic operations used to maintain its state (e.g., adding, updating or searching entries), which can be modeled by considering the right set of simple traffic features. The features we compute on the input stream have the advantage of being lightweight with a deterministic worst case computational cost, thus introducing a minimum delay in the operations of the monitoring system. The proposed scheme automatically identifies those features that best model the resource usage of each monitoring application based on previous measurements of its resource usage and use them to predict the overall load of the system. This short-term prediction is used to guide the system on deciding *when*, *where* and *how much* load to shed.

In the presence of overload, the system can apply several load shedding techniques, such as packet or flow sampling, to reduce the amount of resources required by the applications to run. Previous load shedding designs select the drop locations (i.e., the sampling rate applied to each monitoring application) in such a way that an aggregate performance metric, such as the overall system throughput [127] or utility [128], is maximized. Therefore, each application should provide a utility function to relate the usefulness of its results with the sampling rate being applied. This solution however suffers from serious fairness issues and is not optimal when applied to a non-cooperative environment, where multiple monitoring applications compete for a finite common resource. Thus, it is only suitable for scenarios where the system administrator has complete control over the utility functions or priorities of each application. This problem is well known in the socio-economic literature as the *Tragedy of the Commons* [65].

On the other hand, traditional approaches that allocate an equal share of computing resources or memory to applications [85] can be also unfair, given that different monitoring applications can have very different resource requirements to achieve similar levels of utility or throughput. For example, a simple application that counts the number of packets that traverse a network link would require very few cycles to compute accurate results, while more complex applications, such as signature-based intrusion detection, would require a much larger amount of resources to obtain the same level of accuracy.

The load shedding scheme presented in this thesis is based instead on a packet scheduler that, with minimal information about the accuracy requirements of the monitoring applications (e.g., minimum sampling rate the application can tolerate to guarantee a maximum error in the results), it is able to keep the measurement error within predefined bounds, while ensuring fairness of service in the presence of non-cooperative applications. The main intuition behind its design is that in network monitoring the number of processed packets often exhibits a stronger correlation with the accuracy of a monitoring application than the amount of memory or the number of allocated cycles.

The strategy used by our packet-based scheduler to select the sampling rates has the appealing feature of having a single Nash Equilibrium when the monitoring applications provide correct information about their accuracy requirements. That is, in our system there is no incentive for any non-cooperative application to lie. In contrast, the Nash Equilibrium in those systems that maximize an aggregate performance metric is when all applications lie about their resource requirements and selfishly ask for the maximum amount of resources.

For those queries that are not robust against traffic sampling or that can compute more accurate results using other load shedding mechanisms, we propose a method that allows the monitoring system to safely offload the work of shedding excess load onto the monitoring applications themselves. Similar custom load shedding solutions proposed in other environments [39] require applications to behave in a collaborative fashion, which is not possible in a competitive environment. Our method instead is able to operate in the presence of non-cooperative monitoring applications and to automatically police applications that do not implement custom load shedding methods properly. This is an important feature given that non-cooperative 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).

We have integrated our load shedding scheme into the CoMo monitoring system [69] and deployed it on a research ISP network, where the traffic load and resource requirements exceed by far the system capacity. We present long-lived experiments with a set of concurrent applications that range from maintaining simple counters (e.g., number of packets, application breakdown) to more complex data structures (e.g., per-flow classification, ranking of most popular destinations or pattern search). In addition, we introduced several anomalies into the packet traces to emulate different network attacks to other systems in the network as well as targeted against the monitoring system itself.

Our results show that, with the load shedding mechanism in place, the system effectively handles extreme load situations, while being always responsive and preventing

uncontrolled packet losses even in the presence of non-cooperative monitoring applications and anomalous traffic patterns. The results also indicate that a predictive approach can quickly adapt to overload situations and keep the results of monitoring applications within acceptable error bounds, as compared to alternative load shedding strategies.

In summary, the main contributions of this thesis are as follows:

- We present the design and implementation of a predictive load shedding scheme for network monitoring applications that can efficiently handle extreme overload situations, without requiring explicit knowledge of their internal implementation and cost, or relying on a specific model for the incoming traffic. We show the superiority of our scheme as compared to reactive and other predictive approaches.

- We introduce the design of a packet-based scheduler for network monitoring systems that guarantees fairness of service in the presence of overload situations, while keeping the measurement error of monitoring applications within bounds defined by non-cooperative users. We model our system using game theory and demonstrate that it has a single Nash Equilibrium when all applications provide correct information about their accuracy requirements.

- We present an extension of our load shedding scheme that allows those monitoring applications that are not robust against traffic sampling to provide custom-defined load shedding mechanisms, without compromising the integrity of the monitoring system and still ensuring fairness of service in a non-cooperative environment.

- We implement our load shedding scheme in an existing network monitoring system and deploy it in a research ISP network. We present an extensive performance evaluation of our load shedding scheme when running on both real-world packet traces and long-lived online executions, where the monitoring system faces extreme overload situations. We also show the robustness of our system in the presence of non-cooperative monitoring applications and anomalous traffic patterns.

1.4 Thesis Outline

The rest of this thesis is organized as follows. The next chapter presents the necessary background, including the basic architecture of our network monitoring system as well as the set of monitoring applications and datasets used to validate and evaluate its performance. The following four chapters present the four main contributions of this thesis. Chapter 3 introduces our prediction method, which constitutes the core of our

load shedding scheme, together with a detailed validation and performance evaluation using real-world packet traces. This chapter is based on our work published in [14, 17]. Chapter 4 describes our load shedding scheme in detail and shows how the output of the prediction method presented in Chapter 3 is used to guide the system on deciding *when*, *where* and *how much* load to shed. This chapter also presents the performance of our monitoring system in an operational research ISP network. Most contents of this chapter are based on [13], although it also includes some results from [14, 17]. In Chapter 5, we extend our load shedding scheme to handle non-cooperative monitoring applications and model our system using game theory. Large portions of this chapter are based on [16]. Chapter 6 describes how users can safely define custom load shedding mechanisms in our system, along with an extensive performance evaluation of the complete load shedding scheme with anomalous traffic and selfish applications. This chapter is mainly based on [15]. Finally, Chapter 7 presents in greater detail the related work, while Chapter 8 concludes the thesis and introduces interesting ideas for future work.

Chapter 2

Background

In this chapter, we describe the basic architecture of the network monitoring system that serves as a case study to validate and evaluate the load shedding scheme proposed in this thesis. We also present the set of traffic queries and datasets that are used throughout the various chapters of this thesis to evaluate the different load shedding proposals. We conclude this chapter with the definitions of several basic concepts that are frequently employed in the rest of this document.

2.1 The CoMo System

We chose the CoMo platform (Continuous Monitoring) [69] for developing and evaluating the load shedding techniques proposed in this thesis. The CoMo system is being developed by Intel Research, in collaboration with, among others, the Advanced Broadband Communications Center (CCABA) of the Technical University of Catalonia (UPC).

CoMo is an open-source passive network monitoring system that allows for fast implementation and deployment of network monitoring applications. CoMo has been designed to be the basic building block of an open network monitoring infrastructure that will allow researchers and network operators to easily process and share network traffic statistics over multiple sites.

CoMo follows a modular approach where users can easily define traffic queries as plug-in modules[1] written in the C language, making use of a feature-rich API provided by the core platform. Users are also required to specify a simple stateless filter to be applied to the incoming packet stream (it could be all the packets) as well as the granularity

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

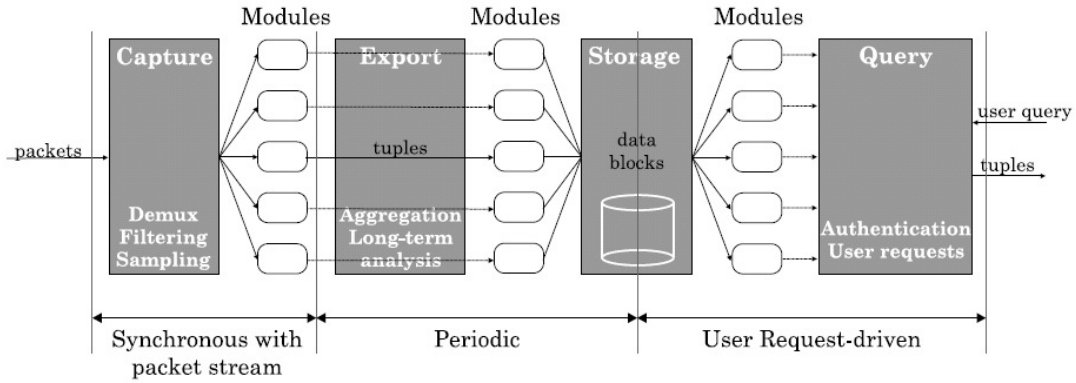


Figure 2.1: Data flow in the CoMo system

of the measurements, hereafter called *measurement interval* (i.e., the time interval that will be used to report continuous query results). All complex stateful computations are then contained within the plug-in module code.

In this section, we describe the high-level architecture of CoMo. Further details about the CoMo system and its architecture are available in [71, 68, 69]. The source code is publicly available at [130].

2.1.1 High-level Architecture

The architecture of CoMo is divided in two main components as shown in Figure 2.1. On the one hand, the *core processes* (gray boxes) control the data path through the CoMo system and perform all management operations common to any monitoring application (e.g., traffic collection, filtering, memory management, storage, etc.). On the other hand, the *plug-in modules* (white boxes) contain the code needed to compute a specific traffic metric or even complex monitoring applications, such as systems for intrusion and anomaly detection, traffic accounting, traffic classification, network performance evaluation, billing and pricing, etc. While the core processes are implemented by a core team of developers and are optimized to operate in high-speed networks, the plug-in modules are written by end users and thus can run sub-optimally or even contain implementation bugs.

Figure 2.1 also illustrates the data flow across the CoMo system. On the left side, CoMo collects the network traffic from one or several links. This traffic is processed by a set of core processes and finally stored onto hard disks. The modules are seen by these processes just as a set of functions that transform the input data streams to

user-defined traffic metrics or processed measurement data as they traverse the system. On the right side, users can retrieve the results of their plug-in modules by querying the CoMo systems.

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, any load shedding scheme for CoMo must operate 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 higher 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 incoming traffic.

2.1.2 Core Processes

The core system is divided into four main processes as illustrated in Figure 2.1. Two basic guidelines have driven the distribution of functionalities among the core processes [71]. First, functionalities with stringent real-time requirements (e.g., packet capture or disk access) are assigned to a single process (*capture* and *storage*, respectively). Other processes instead operate in a best-effort manner (e.g., *query*) or with less stringent time requirements (e.g., *export*). Second, each hardware device is assigned to a single process. For example, the *capture* process is in charge of the network sniffers, while storage controls the disk array. Another important feature of this architecture is the decoupling between real-time tasks and user driven tasks, which allows CoMo to control more efficiently the system resources. This is visualized by the vertical lines in Figure 2.1.

The *capture* process is responsible for traffic collection and filtering. It supports standard NIC cards accessed via the Berkeley Packet Filter [96] and libpcap API [129], dedicated packet capture cards for high-speed links, such as Endace DAG cards [53], raw NetFlow [35] and sFlow [111] streams from routers and switches, and 802.11 wireless devices operating in RF monitor mode. CoMo converts all these incoming data streams in a unified packet stream [69] that is passed through a filter to identify the modules that are interested in processing each packet. Next, the *capture* process delivers the selected

packets to the modules, which process them and update their internal data structures. At each measurement interval, *capture* sends the content of the data structures maintained by the modules to the *export* process. This decouples the real-time requirements of *capture* that deals with incoming packets at line rate from storage and user-driven tasks.

The *export* process is in charge of those long-term analysis tasks that have less time constrains. The behavior of *export* is very similar to that of *capture* with the difference that it handles state information of the modules rather than incoming packets. At each measurement interval, the *export* process receives processed records from *capture* and delivers them to the modules for further processing. As opposed to *capture*, the *export* process does not flush periodically the data, but instead it needs to be instructed by the module. In particular, a module can request the *export* process to store the data or to maintain long-term information.

Finally, the *storage* process is responsible for storing to the hard disk the data maintained by the *export* process when requested by a module, while the *query* process receives user requests for module results, retrieves the relevant data from disk via the *storage* process and returns them to the user.

2.1.3 Plug-in Modules

The monitoring applications in CoMo are provided as plug-in modules written in the C language. A module can be seen as a *filter:function* pair, where the filter selects the packets of interest to the module and the function specifies the action to be performed on the selected packets. The module can also define a *measurement interval* that indicates how frequently the state data maintained by *capture* are flushed to the *export* process.

The filter is provided in the module configuration and executed by the *capture* process for each collected packet, while the function is implemented as a shared object with a set of standardized entry points ("callbacks"). The callback functions are executed in a pre-defined sequence by the *capture*, *export* and *query* core processes. Table 2.1 provides a brief summary of the most representative callbacks.

The `check()` and `update()` callbacks are used by the module to process the packets that matched the filter rule and to update its internal data structures. These data structures are flushed at the end of the measurement interval and their contents are sent to the *export* process in the form of tuples. Then, *export* calls the `export()` function for each received tuple, which is used by the module to maintain long-term information. To indicate which entries have to be stored or discarded in *export*, the module uses the `action()` callback, while the exact information to be stored is provided by the `store()`

Callback	Description	Process
`check()`	Stateful filters	capture
`update()`	Packet processing	
`export()`	Processes tuples sent by *capture*	export
`action()`	Decides what to do with a record	
`store()`	Stores records to disk	
`load()`	Loads records from disk	query
`print()`	Formats records	

Table 2.1: Summary of the callbacks and core processes that call them

callback. Finally, the *query* process uses the `load()` and `print()` callbacks to return the module results when a user request is received.

It is important to observe that the core processes are agnostic to the state that each module computes. Core processes just provide the packets to the modules and take care of scheduling, policing and resource management tasks.

2.2 Description of the Queries

Despite the fact that the actual metric computed by the query is not relevant to the load shedding scheme proposed in this thesis (our system considers all queries as black boxes) we are interested in considering a wide range of queries when performing the evaluation.

In this thesis, we have selected a set of queries that are part of the standard distribution of CoMo. We just modified their source code in order to allow them to estimate their unsampled output when load shedding is performed. In most cases, this modification was simply done by multiplying the metrics they compute by the inverse of the applied sampling rate. We also implemented three additional queries (*uni-dimensional autofocus* [55], *super-sources* [139] and *p2p-detector* [121, 83]) that make use of more complex algorithms and data structures.

Table 2.2 provides a brief summary of the queries.[2] Three queries (*counter*, *application* and *high-watermark*) maintain simple arrays of counters depending on the timestamps of the packets (and port numbers for *application*). The cost of running these queries is therefore driven by the number of packets. The *trace* query stores the full payload of all packets that match a stateless filter rule and therefore the cost depends on the number of bytes to be stored. The query *pattern-search* stores all packets that contain a given string, while *p2p-detector* combines pattern search with the techniques described in [121, 83] to identify those flows belonging to a P2P application. Both queries use the

[2]The source code of the queries used in this thesis is available at `http://como.sourceforge.net`.

Query	Description	Method	Cost
application	Port-based application classification	packet	low
autofocus	High volume traffic clusters per subnet [55]	packet	med
counter	Traffic load in packets and bytes	packet	low
flows	Per-flow classification and number of active flows	flow	low
high-watermark	High watermark of link utilization over time	packet	low
p2p-detector	Signature-based P2P detector [121, 83]	packet	high
pattern-search	Identification of byte sequences in the payload [23]	packet	high
super-sources	Detection of sources with largest fan-out [139]	flow	med
top-k	Ranking of the top-k destination IP addresses [12]	packet	low
trace	Full-payload packet collection	packet	med

Table 2.2: Description of the CoMo queries

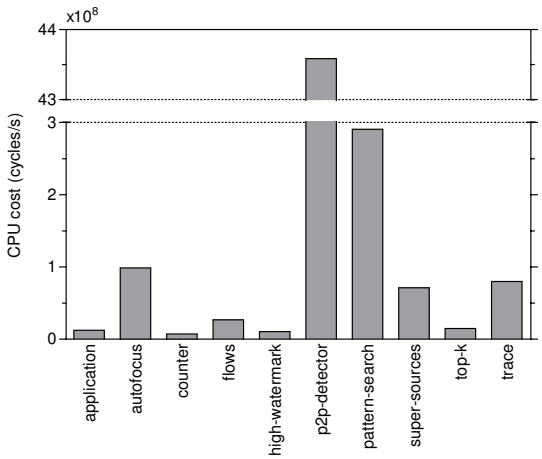


Figure 2.2: Average cost per second of the CoMo queries (CESCA-II trace)

Boyer-Moore algorithm [23] where the cost is linear with the number of bytes processed. The *flows* and *top-k* queries perform a flow classification and maintain a large number of per-flow counters for all the flows (in a way similar to NetFlow) or just the ones that exchanged the largest number of bytes, respectively. The cost of these two queries depends on the number of flows in the packet stream but also on the details of the data structures used for the classification. Finally, the queries *autofocus* (uni-dimensional) and *super-sources* report the subnet clusters that generated more traffic [55] and the source IP addresses with largest fan-out [139], respectively. Figure 2.2 shows the cost of each query when running on the CESCA-II dataset, which is described in Section 2.3.

All queries use packet sampling, with the exception of *flows* and *super-sources* that use flow sampling instead. Packet sampling consists of randomly selecting packets from the input streams with probability p (i.e., the sampling rate), while flow sampling consists

of randomly selecting entire flows, rather than single packets, with probability p.

We believe that the set of queries we have chosen form a representative set of typical uses of a real-time network monitoring system. They present different resource usage profiles (CPU, memory and disk bandwidth) for the same input traffic and use different data structures to maintain their state (e.g., aggregated counters, hash tables, sorted lists, binary trees, bloom filters, bitmaps, etc.). In this thesis, we will show that our load shedding approach is general enough to handle efficiently all these different cases in normal and extreme traffic scenarios.

2.2.1 Accuracy metrics

In this thesis, we use the error of the queries as a performance metric to evaluate the different load shedding proposals. In the case of the *counter*, *flows* and *high-watermark* queries, we measure the relative error in the number of packets and bytes, flows, and in the high-watermark value, respectively. The error of the *application* query is measured as a weighted average of the relative error in the number of packets and bytes across all applications. The relative error is defined as the absolute value of one minus the ratio of the estimated and the actual output of a query, where the actual value in our experiments is obtained from a complete packet trace.

In order to compute the error of the *top-k* query, we use the detection performance metric proposed in [12], which is defined as the number of misranked flow pairs, 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.

The error of the *autofocus* query is defined as the absolute value of one minus the number of clusters in the delta report (see [55]) over the total number of clusters reported by the query. The error of *super-sources* is computed as the average relative error in the fan-out estimations [139], while in the case of the *p2p-detector*, the error is computed as one minus the number of flows correctly identified over the total number of flows.

Finally, the error of the *pattern-search* and *trace* queries is considered to be proportional to the number of processed packets, given that no standard procedure exists to recover their unsampled output from sampled streams and to measure their error. In particular, we compute the error of these two queries as one minus the ratio between the number of processed packets and the total number of packets. Note however 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.

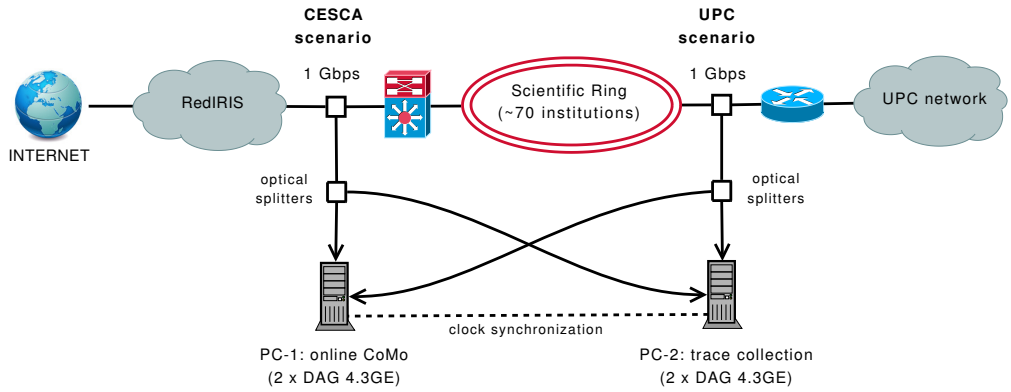


Figure 2.3: Testbed scenario

The selection of queries reflects a typical scenario with two classes of queries. The first class includes queries that compute a specific traffic statistic or metric for which we know exactly how to measure its accuracy (e.g., *application*, *top-k*, etc.). The second class includes queries (e.g., *trace* and *pattern-search*) that do not compute a specific metric but instead perform a particular task on the incoming traffic.

2.3 Datasets

In this section, we present the network scenarios, packet traces and online executions that are used to evaluate the performance of the different load shedding techniques presented in this thesis.

2.3.1 Testbed scenarios

We carried out several experiments in two operational networks. The testbed equipment in both scenarios consisted of two PCs with an Intel® Pentium™ 4 running at 3 GHz, both equipped with a couple of Endace® DAG 4.3GE cards [53] with two network interfaces. Through a pair of optic splitters, both computers received an exact copy of the traffic of the networks under study. Figure 2.3 presents the two network scenarios and the exact location of the capture points.

CESCA scenario

The first scenario consists of a Gigabit Ethernet link that connects the Catalan Research and Education Network (also known as the Scientific Ring) to the global Internet via its

Name	Date/Time	Trace (GB)	Pkts (M)	Bytes (GB)	Load (Mbps) avg/max/min
ABILENE	14/Aug/02 09:00-11:00	34.1	532.4	370.6	411.9/623.8/286.2
CENIC	17/Mar/05 15:50-16:20	3.8	59.5	56.0	249.7/936.9/079.1
CESCA-I	02/Nov/05 16:30-17:00	8.3	103.7	81.1	360.5/483.3/197.3
CESCA-II	11/Apr/06 08:00-08:30	30.9	49.4	29.9	133.0/212.2/096.2
UPC-I	07/Nov/07 18:00-18:30	54.7	95.2	53.0	253.5/399.0/177.8

Table 2.3: Traces in our dataset

Spanish counterpart (RedIRIS). The Scientific Ring is managed by the Supercomputing Center of Catalonia (CESCA) and connects more than seventy Catalan universities and research centers using many different technologies that range from ADSL to Gigabit Ethernet [30]. A trace collected at this capture point is publicly available in the NLANR repository [103]. In this document, we refer to this first scenario as CESCA.

UPC scenario

The second scenario is located at the Gigabit Ethernet access link of the Technical University of Catalonia (UPC), which connects around 10 campuses, 25 faculties and 40 departments to the Internet through the Scientific Ring. Real-time figures of the traffic traversing this Gigabit Ethernet link are publicly available at [74]. In this document, we refer to this second scenario as UPC.

2.3.2 Packet traces

For evaluation purposes, we collected two 30-minute traces from one of the link directions of the CESCA scenario in November 2005 and April 2006. In order to capture the traces, we only used one of the two PCs of our testbed equipment. In the first trace, we only collected the packet headers, while in the second one the full packet payloads were acquired. We refer to these traces as CESCA-I and CESCA-II, respectively. In addition, we captured a third 30-minute unidirectional trace with the entire payloads from the UPC scenario in November 2007. We refer to this trace as UPC-I. Full-payload traces are needed to study those queries that require the packet contents to operate (e.g., *pattern-search* and *p2p-detector*). Details of the traces are presented in Table 2.3.

In order to study our techniques in other environments, we extended our datasets with two anonymized packet header traces collected by the NLANR-PMA project [103], in August 2002 and March 2005. The first one (ABILENE) consists of a OC48c Packet-over-SONET unidirectional trace collected at the Indianapolis router node of the Abilene

Name	Date/Time	Trace (GB)	Pkts (M)	Bytes (GB)	Load (Mbps) avg/max/min
CESCA-III	24/Oct/06 09:00-17:00	155.5	2908.2	2764.8	750.4/973.6/129.0
CESCA-IV	25/Oct/06 09:00-17:00	152.5	2867.2	2652.2	719.9/967.5/218.0
CESCA-V	05/Dec/06 09:00-17:00	138.6	2037.8	1484.8	403.3/771.6/131.0
UPC-II	24/Apr/08 09:00-09:30	47.6	61.3	46.5	222.2/282.1/176.9

Table 2.4: Online executions

backbone (eastbound towards Cleveland). The second trace (CENIC) consists of the first 30 minutes of the traffic collected on the 10 Gigabit CENIC HPR backbone link between Sunnyvale and Los Angeles. Details of these traces are also available in Table 2.3.

In several experiments performed in this thesis we use packet traces for the sake of reproducibility, but all conclusions can be extended to an online system, given that CoMo does not make any distinctions between running online or offline [69].

2.3.3 Online executions

Apart from the offline experiments using packet traces, we also carried out several online executions to experimentally evaluate the online performance of a monitoring system implementing the different load shedding schemes proposed in this thesis.

On the one hand, throughout the thesis, we present the results of three 8 hours-long executions performed in the CESCA scenario. In this case, the first PC of our testbed equipment was configured to run the CoMo monitoring system online, while the second one was used to collect a packet-level trace (without loss), which is used as our reference to verify the accuracy of the results of the queries described in Section 2.2.

In the first execution (CESCA-III), we ran a modified version of CoMo implementing the load shedding scheme presented in Chapters 3 and 4, while in the other two executions (CESCA-IV and CESCA-V) we repeated the same experiment, but using a version of CoMo that implements two alternative load shedding approaches that will be described in Chapter 4. The first two executions were carried out from 9h to 17h on two consecutive days in October 2006. Note that we did not run both experiments at the same time because, although we have two computers, we had to collect an entire trace using the second PC in order to evaluate the accuracy of the queries. The third execution was performed during the same period of time, but in December 2006.

On the other hand, we performed an additional 30-minute online execution in the UPC scenario in April 2008. In this case, the first PC was used to run the load shedding scheme presented in Chapter 6. The duration of all executions was constrained by the

amount of storage space available to collect the packet-level traces (500 GB) and the size of the DAG buffers was configured to 256 MB. Table 2.4 presents the details of all executions.

2.4 Definitions

In this section, we present some definitions of different basic terms that will be used throughout this thesis.

- *Batch:* Set of packets collected during a fixed interval of time defined as *time bin*.
- *Time bin:* Time duration of a batch (i.e., maximum time between the first and last packet of a batch).
- *Batch cost:* Cost of processing a batch by a given query within a particular system (e.g., CPU cycles, memory usage).
- *Response variable:* Batch cost that we want to predict.
- *Batch feature:* Each of the traffic features we can obtain from a batch (e.g., number of packets, bytes, unique IP addresses).
- *Predictors:* Subset of batch features that are used to predict a given response variable.
- *Feature selection:* Algorithm to decide which batch features are useful as predictors of a given response variable.
- *Measurement interval:* Duration of the measurement period defined by the query (i.e., the time bin that will be used to report continuous query results).
- *Load shedding:* It is the process of adjusting the demands to match the available resources according to a given policy. Load shedding is usually implemented by discarding some fraction of the input data.
- *Load shedding scheme:* It is the system in charge of deciding *when*, *where* and *how much* load to shed in order to assure that the monitoring system remains stable during overload situations.
- *Load shedding strategy:* It is the algorithm responsible for selecting *where* to shed a given amount of load (i.e., which queries) when an overload situation is detected.

- *Load shedding mechanism (or method):* It is the technique used to shed excess load. For example, the load shedding scheme proposed in this thesis supports packet and flow sampling as well as custom load shedding methods defined by end users.

- *Load shedder:* It is the component of the load shedding scheme responsible for shedding excess load using one of load shedding mechanisms according to a given load shedding strategy.

Chapter 3

Prediction System

In this chapter, we present our prediction methodology that constitutes the core of the load shedding scheme proposed in this thesis. We also describe the goals and challenges involved in the design of a prediction mechanism for arbitrary network monitoring and data mining applications. We conclude the chapter with an extensive performance evaluation of the prediction accuracy and cost using real-world packet traces and injecting artificially-generated traffic anomalies.

3.1 System Overview

As discussed in Chapter 2, CoMo does not restrict the type of computations that a plug-in module can perform in order to provide the user with maximum flexibility when writing queries. 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 them. This approach allows users to define traffic queries that otherwise could not be easily expressed using common declarative languages (e.g., SQL).

Therefore, any load shedding scheme for such a system must operate only with external observations of the CPU, memory or bandwidth requirements of the modules – and these are not known in advance but only after a packet has been processed.

Our thesis is that the cost of maintaining the data structures needed to execute a query can be modeled by looking at a set of traffic features that characterizes the input data. The intuition behind this thesis 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 such as, for example, creating new entries, updating existing ones or looking for a valid match. We observed that the

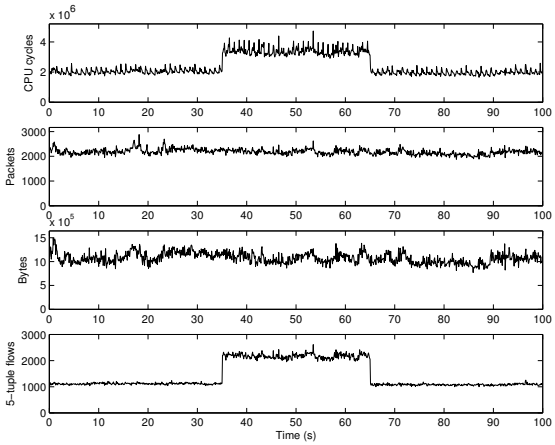


Figure 3.1: CPU usage of an "unknown" query in the presence of an artificially generated anomaly compared to the number of packets, bytes and flows

time spent by a query is mostly dominated by the overhead of some of these operations. Therefore, the cost of a query can be modeled by considering the right set of simple traffic features.

A traffic feature is a counter that describes a property of a sequence of packets. For example, potential features could be the number of packets or bytes in the sequence, the number of unique source IP addresses, etc. In the design of our prediction method we will select a large set of simple features that have the same underlying property: deterministic worst case computational complexity. Later we will describe how a large set of features can be efficiently extracted from the traffic stream (Section 3.2.1).

Once a large number of features are efficiently extracted from the traffic stream, the challenge is in identifying the right ones that can be used to accurately model and predict the query's CPU usage. Figure 3.1 illustrates a very simple example. The figure shows the time series of the CPU cycles consumed by an "unknown" query (top graph) when running over a 100*s* snapshot of the CESCA-I data set (described in Section 2.3), where we inserted an artificially generated anomaly, which simulates a simple attack that unexpectedly increases the number of flows in the traffic. The three bottom plots show three possible features over time: the number of packets, bytes and flows (defined by the classical 5-tuple: source and destination addresses, source and destination port numbers and protocol number). It is clear from the figure that the bottom plot would give us more useful information to predict the CPU usage over time for this query. It is also easy to infer that the query is performing some sort of per-flow classification, hence the higher cost when the number of flows increases, despite the volume of packets and

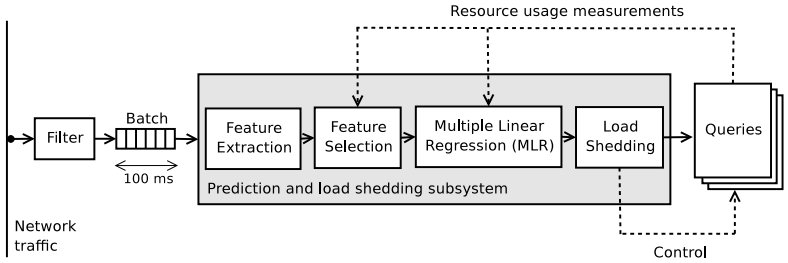


Figure 3.2: Prediction and load shedding subsystem

bytes remains fairly stable.

Based on this observation, we designed a method that automatically selects the most relevant feature(s) from small sequences of packets and uses them to accurately predict the CPU usage of arbitrary queries. This fine-grained and short-term prediction can then be used to quickly adapt to overload situations by sampling the input streams or by providing a summarized view of the traffic data.

Figure 3.2 shows the components and the data flow in the system. The prediction and load shedding subsystem (in gray) intercepts the packets from the filter before they are sent to the plug-in modules implementing the traffic queries. In order to implement the prediction we instrumented the core platform to export some performance metrics.

In this work, we focus only on one resource: the CPU cycles consumed by the queries. As we will show throughout this thesis, the CPU is the primary scarce resource in our monitoring system. However, other system resources are also critical in network monitoring (e.g., memory, disk bandwidth and disk space) and we believe that approaches similar to what we propose here could be applied as well.

The system operates in four phases that are executed online. First, it groups each $100ms$ of traffic in a "batch" of packets.[1] Each batch is then processed to extract a large pre-defined set of traffic features (Section 3.2.1). The feature selection subsystem is in charge of selecting the most relevant features for prediction purposes according to the recent history of each query's CPU usage (Section 3.2.3). This phase is important to reduce the cost of the prediction algorithm, because it allows the system to discard beforehand the features regarded as useless for prediction purposes. This subset of relevant features is then given as input to the multiple linear regression subsystem to

[1]The choice to use batches of 100ms is somewhat arbitrary. Our goal is not to delay excessively the query results but at the same time use a time interval large enough to observe a meaningful number of packets. Indeed an interval too small would add a significant amount of noise in the system and increase the prediction error. Our results indicate that 100ms represents a good trade-off between accuracy and delay. However, this is clearly a function of the input traffic traces we used.

predict the CPU cycles required by the query to process the entire batch (Section 3.2.2). If the prediction exceeds the current allocation of cycles, the load shedding subsystem pre-processes the batch to discard (e.g., via packet or flow sampling) a portion of the packets (Chapters 4, 5 and 6). Finally, the actual CPU usage is computed and fed back to the prediction subsystem to close the loop (Section 3.2.4).

3.2 Prediction Methodology

In this section, we describe in detail the three phases that our system executes to perform the prediction (i.e., *feature extraction*, *feature selection* and *multiple linear regression*) and how the resource usage is monitored. The only information we require from the continuous query is the measurement interval of the results. Avoiding the use of additional information increases the range of applications where this approach can be used and also reduces the likelihood of compromising the system by providing incorrect information about a query.

3.2.1 Feature Extraction

We are interested in finding a set of traffic features that are simple and inexpensive to compute, while helpful to characterize the CPU usage of a wide range of queries. A feature that is too specific may allow the system 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 in order 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 its 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 a batch; (*ii*) the number of new items compared to all items seen in a measurement interval; (*iii*) the number of repeated items in a 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).

No.	Traffic aggregate
1	src-ip
2	dst-ip
3	protocol
4	<src-ip, dst-ip>
5	<src-port, proto>
6	<dst-port, proto>
7	<src-ip, src-port, proto>
8	<dst-ip, dst-port, proto>
9	<src-port, dst-port, proto>
10	<src-ip, dst-ip, src-port, dst-port, proto>

Table 3.1: Set of traffic aggregates (built from combinations of TCP/IP header fields) used by the prediction

For example, we may aggregate packets based on the source IP address and source port number, where each aggregate (or "item") is made of all packets that share the same source IP address and source port number pair. Then, we count the number of unique, new and repeated source IP address and source port pairs.

Table 3.1 shows the combinations of the five header fields considered in this work. Although we do not evaluate other choices here, we note that other features may be useful (e.g., source IP prefixes, other combinations of the 5 header fields or payload-related features). However, we will address the trade-off between the number of features and the overhead of running the prediction in greater detail in Section 3.3.

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). For example, the new items are relevant to predict the CPU requirements of those queries that spend most time creating entries in the data structures, while the repeated items feature may be relevant to queries where the cost of updating the data structures is much higher than the cost of creating entries.

In order to extract the features with minimum overhead, we implement the multi-resolution bitmap algorithms proposed in [57]. 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 smaller memory footprint than linear counting [134] or bloom filters [22]. We dimension the multi-resolution bitmaps to obtain counting errors around 1% given the link speeds in our testbed.

We use two bitmaps for each aggregation level: one that keeps the per-batch unique

count and another that maintains the new count per measurement interval. The bitmap used to estimate the unique items is updated per packet. Instead, the one used to estimate the new items can be updated per batch by performing a bitwise OR with the bitmap used to maintain the unique count, given that the same bits have to be set in both bitmaps. The only difference between them is the moment when they are reset to zero. On the other hand, as mentioned above, it is straightforward to derive the number of repeated and batch-repeated items from the counts of new and unique items respectively by keeping just two additional counters.

3.2.2 Multiple Linear Regression

Regression analysis is a widely applied technique to study the relationship between a response variable Y and one or more predictor variables $X_1, X_2, \ldots, X_p$. The linear regression model assumes that the response variable Y is a linear function of the p X_i predictor variables.[2] The fact that this relationship exists implies that any knowledge we have about the predictor variables provides us information about the response variable. Thus, this knowledge can be exploited for predicting the expected value of Y when the values of the p predictor variables are known. In our case, the response variable is the CPU usage, while the predictor variables are the individual features.

When only one predictor variable is used, the regression model is often referred to as simple linear regression (SLR). Using just one predictor has two major drawbacks. First, there is no single predictor that yields good performance for all queries. For example, the CPU usage of the *counter* query is well modeled by looking at the number of packets in each batch, while the *trace* query is better modeled by the number of bytes in the batch. Second, the CPU usage of more complex queries may depend on more than a single feature.

To illustrate this latter point, we plot in Figure 3.3 the CPU usage for the *flows* query versus the number of packets in the batch. As we can observe, there are several underlying trends that depend both on the number of packets and on the number of new 5-tuples in the batch that SLR cannot consider. This behavior is due to the particular implementation of the *flows* query that maintains a hash table to keep track of the flows and expires them at the end of each measurement interval.

Figure 3.4 shows that the prediction error of the *flows* query by using SLR is relatively large. The spikes in the CPU usage at the beginning of each measurement interval (1s in

[2]It would be possible that the CPU usage of a query exhibits a non-linear relationship with the traffic features. A solution in that case may be to define new features computed as non-linear combinations of simple features. We discuss this issue in greater detail in Chapter 8.

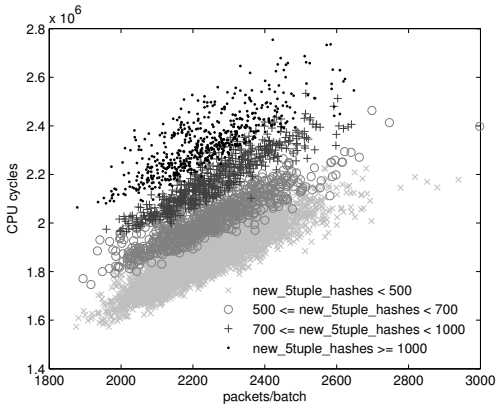


Figure 3.3: Scatter plot of the CPU usage versus the number of packets in the batch (*flows* query)

this example) are due to the fact that when the table is empty, the number of new entries to be created is much larger than usual. This error could be much more significant in presence of traffic anomalies that abruptly increase the number of new entries to be created.

Multiple linear regression (MLR) extends the simple linear regression model to several predictor variables. MLR is used to extract a linear combination of the predictor variables that is maximally correlated with the response variable. The general form of a linear regression model for p predictor variables can be written as follows [41]:

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \cdots + \beta_p X_{pi} + \varepsilon_i, \qquad i = 1, 2, \ldots, n \tag{3.1}$$

where β_0 denotes the *intercept*, $\beta_1, \ldots, \beta_p$ are the *regression coefficients* that need to be estimated and ε_i is the *residual term* associated with the i-th observation. The residual term is an unobservable random variable that represents the omitted variables that affect the response variable, but that are not included in the model.

In fact, Equation 3.1 corresponds to a system of equations that in matrix notation can be written as:

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} = \begin{pmatrix} 1 & X_{11} & \ldots & X_{p1} \\ 1 & X_{12} & \ldots & X_{p2} \\ \vdots & \vdots & & \vdots \\ 1 & X_{1n} & \ldots & X_{pn} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}$$

or simply

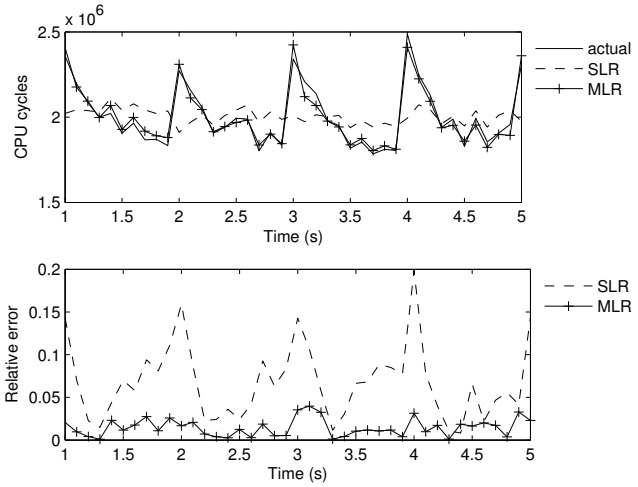


Figure 3.4: Simple Linear Regression versus Multiple Linear Regression predictions over time (*flows* query)

$$Y = X\beta + \varepsilon \tag{3.2}$$

where

- Y is a $n \times 1$ column vector of the response variable observations. We obtain the values of Y by measuring the CPU usage of the previous n batches processed by the query;[3]
- X is a $n \times (p+1)$ matrix resulting from n observations of the p predictor variables $X_1, \ldots, X_p$ (the first column of 1's represents the intercept term β_0). That is, the values of the p features we extracted from the previous n batches;
- β is a $(p+1) \times 1$ column vector of unknown parameters $\beta_0, \beta_1, \ldots, \beta_p$, where $\beta_1, \ldots, \beta_p$ are referred to as the *regression coefficients* or *weights*;
- and ε is a $n \times 1$ column vector of n residuals ε_i.

The estimators b of the regression coefficients β are obtained by the Ordinary Least Squares (OLS) procedure, which consists of choosing the values of the unknown parameters $b_0, \ldots, b_p$ in such a way that the sums of squares of the residuals is minimized. In our implementation, we use the singular value decomposition (SVD) method [113]

[3] In Section 3.3.1, we address the problem of choosing the appropriate value for n.

to compute the OLS. Although SVD is more expensive than other methods, it is able to obtain the best approximation, in the least-squares sense, in the case of an over- or under-determined system.

The statistical properties of the OLS estimators lie on some assumptions that must be fulfilled [41, pp. 216]: (*i*) the rank of X is $p+1$ and is less than n, i.e., there are no exact linear relationships among the X variables (no *multicollinearity*); (*ii*) the variable ε_i is normally distributed and the expected value of the vector ε is zero; (*iii*) there is no correlation between the residuals and they exhibit constant variance; (*iv*) the covariance between the predictors and the residuals is zero. In Section 3.2.3, we present a technique that makes sure the first assumption is valid. We have also verified empirically on the packet traces that the other assumptions hold.

Going back to the example of the *flows* query, Figure 3.4 shows the prediction accuracy when using MLR with the number of packets and new 5-tuples as predictors. However, since queries consist of arbitrary code, the system cannot know in advance which features perform best as predictors for each query. It would be possible to use all the extracted traffic features in the regression, since MLR should be able to find a combination of them that is maximally correlated with the CPU usage. However, as it can be deduced from Equation 3.2, the cost of MLR does not depend only on the amount of history used to compute the linear regression (i.e., n), but also on the number of variables used as predictors (i.e., p). If a large number of predictors is used, the cost of the MLR would increase significantly and it could impose too much overhead to the prediction process. Next section presents a technique that is used to select only the subset of traffic features that is most relevant to predict the cost of a given query.

3.2.3 Feature Selection

Including in the regression model as many predictor variables as possible has several drawbacks [41]. As we mentioned before, the cost of the linear regression increases quadratically with the number of predictors included in the model, while the gain the additional predictors bring usually does not justify their cost. In addition, even if we were able to include in the model all the possible predictors, there would still be a certain amount of randomness that cannot be explained by any predictor. Finally, introducing redundant predictors into the model (i.e., predictors that are linear functions of other predictors) invalidates the no-multicollinearity assumption.[4]

[4]Note that the values of some predictors may become very similar under special traffic patterns. For example, the number of packets and flows can be highly correlated under a SYN-flood attack.

Once a choice of the features to compute on the batch is made, it is important to identify a small subset of features to be used as predictors. In order to support arbitrary queries, we need to define a generic feature selection algorithm. We would also like our method to be capable of dynamically selecting different sets of features if the traffic conditions change during the execution, and the current prediction model becomes obsolete.

Most of the algorithms proposed in the literature are based on a sequential variable selection procedure [41]. However, they are usually too expensive to be used in a real-time system. For this reason, we decided to use a variant of the Fast Correlation-Based Filter (FCBF) [137], which can effectively remove both irrelevant and redundant features and is computationally very efficient.

Our variant differs from the original FCBF algorithm in that we use the *linear correlation coefficient* (Equation 3.3) as a predictor goodness measure, instead of the *symmetrical uncertainty* measure [137], which is based on the information-theoretical concept of *entropy*. The algorithm consists of two main phases:

1. *Selecting relevant predictors:* The linear correlation coefficient between each predictor (X) and the response variable (Y) is computed as follows:

$$r = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}} \tag{3.3}$$

   The predictors with a coefficient below a predefined *FCBF threshold* are discarded as not relevant.[5] This phase has a time complexity that grows linearly with the number of predictors.

2. *Removing redundant predictors:* The predictors that are left after the first phase are ranked according to their coefficient values and processed iteratively to discard predictors that have a mutual strong correlation. Each iteration starts from the first element of the list (i.e., the predictor with the highest linear correlation coefficient) and computes the correlation coefficients between this element and all the remaining predictors. For each pair of predictors, if their relative correlation coefficient is higher than the correlation between the predictors and the response variable (computed in the previous phase), the predictor lower in the list is removed as redundant. Then, the algorithm continues starting again from the second predictor. In each iteration the algorithm can usually remove a significant number of

[5] In Section 3.3.1, we address the problem of choosing the appropriate *FCBF threshold*.

redundant features, giving this phase a time complexity of $O(p \log p)$, where p is the number of predictors in the list.

Finally, the overall complexity of the FCBF is $O(n\, p \log p)$, where n is the number of observations and p the number of predictors [137].

3.2.4 Measurement of System Resources

Fine grained measurement of CPU usage is not an easy task. The mechanisms provided by the operating system do not offer a good enough resolution for our purposes, while processor performance profiling tools [73] impose a large overhead and are not a viable permanent solution.

In this work, we use instead the *time-stamp counter* (TSC) [73] to measure the CPU usage, which is a 64-bit counter incremented by the processor every clock cycle. In particular, we read the TSC before and after a batch is processed by a query. The difference between these two values corresponds to the number of CPU cycles used by the query to process the batch.

Other *performance-monitoring counters* (PMC) [73] could also give us more accurate measurements, but are architecture dependent and behave differently depending on the concrete architecture where the system is executed on.

The CPU usage measurements that are fed back to the prediction system should be accurate and free of external noise to reduce the errors in the prediction. However, we empirically detected that measuring CPU usage at very small timescales incurs several sources of noise:

1. *Instruction reordering:* The processor can reorder instructions at run time in order to improve performance. In practice, the `rdtsc` instruction used to read the TSC counter is often reordered, since it simply consists of reading a register and it does not have dependencies with other instructions. The `rdtsc` instruction at the beginning of the query can be reordered with other instructions that do not belong to the query, while the one at the end of the query can be executed before the query actually ends. Both these events happen quite frequently and lead to inaccuracies in the measurements. To avoid the effects of reordering, we execute a serializing instruction (e.g. `cpuid`) before and after our measurements [73]. Since the use of serializing instructions can have a severe impact on the system performance, we only take two TSC readings per query and batch, and do not take any partial measurements during the execution of the query.

2. *Context switches:* The operating system may decide to schedule out the query process between two consecutive readings of the TSC. In that case, we would be measuring not only cycles belonging to the query, but also cycles of the process (or processes) that are preempting the query. In order to avoid degrading the accuracy of future predictions when a context switch occurs during a measurement, we discard those observations from the MLR history and replace them with our prediction. To measure context switches, we monitor two fields of the `rusage` process structure in the Linux kernel, called `ru_nvcsw` and `ru_nivcsw`, that count the number of voluntary and involuntary context switches, respectively. In some strategic places of our code, we also force the process to be uninterruptible using the `sched_setscheduler` system call and setting the scheduling policy to `SCHED_FIFO` with maximum priority.

3. *Disk accesses:* Disk accesses can interfere with the CPU cycles needed to process a query. In CoMo, a separate process is responsible for scheduling disk accesses to read and write query results. In practice, since disk transfers are done asynchronously by DMA, memory accesses of queries have to compete for the system bus with disk transfers. In Section 3.4, we show how disk accesses have a limited impact on the performance of the prediction system.

We do not take any particular action in the case of other causes of measurement noise, such as CPU frequency scaling or cycles executed in system mode, since we experimentally checked that they usually have much less impact on the CPU usage patterns than the sources of error described above.

It is also important to note that all the sources of error we detected so far are independent from the input traffic. Therefore, they cannot be directly exploited by an external malicious user trying to introduce errors in our CPU measurements to attack the monitoring system.

3.3 Validation

In this section, we show the performance of an actual implementation of our prediction method on several packet traces as well as its sensitivity to the configuration parameters. In order to understand the impact of each parameter, we study the prediction subsystem in isolation from the sources of measurement noise identified in Section 3.2.4. We disabled the disk accesses in the CoMo process responsible for storage operations to

avoid competition for the system bus. In Section 3.4, we evaluate our method in a fully operational system.

To measure the performance of our method we consider the relative error in the CPU usage prediction while executing a set of seven queries over packet traces. Table 3.2 lists the subset of queries from those presented in Table 2.2 used in the validation. The relative error is defined as the absolute value of one minus the ratio of the prediction and the actual number of CPU cycles spent by the queries over each batch.

3.3.1 Prediction Parameters

In our system, two configuration parameters impact the cost and accuracy of the predictions: the number of observations (i.e., n or the "history" of the system) and the FCBF threshold used to select the relevant traffic features. In this particular experiment, we analyze the more appropriate values of these parameters for the CESCA-II trace with full packet payloads (see Table 2.3), but almost identical values were obtained for the other traces.

Number of observations

The cost (in terms of CPU cycles) of the linear regression directly depends on the amount of history, since every additional observation translates to a new equation in the system in (3.2). The accuracy of the prediction is also affected by the number of observations.

In order to decide the appropriate amount of history to keep in our model, we ran multiple executions in our testbed with values of history ranging from $1s$ to $100s$ (i.e., from 10 to 1000 batches). We checked that histories older than $100s$ do not provide us any new relevant information for prediction purposes. Figure 3.5 (left) shows the cost of computing the MLR and the prediction accuracy as a function of the amount of history (each observation corresponds to $100ms$ of traffic), while Figure 3.6 (left) presents the prediction accuracy broken down by query.

As we can see, the cost of computation grows linearly with the amount of history, while the relative error between the prediction and the actual number of CPU cycles spent by the query stabilizes around 1.2% after $6s$ (i.e., 60 observations). Larger errors for very small amounts of history (e.g., $1s$) are due to the fact that the number of predictors (i.e., $p = 42$) is larger than the amount of history (i.e., $n = 10$ batches, $1s$) and thus the no-multicollinearity assumption is not met. Increasing the number of observations does not improve the accuracy, because events that are not modeled by the traffic features are probably contributing to the error. Moreover, a longer history makes

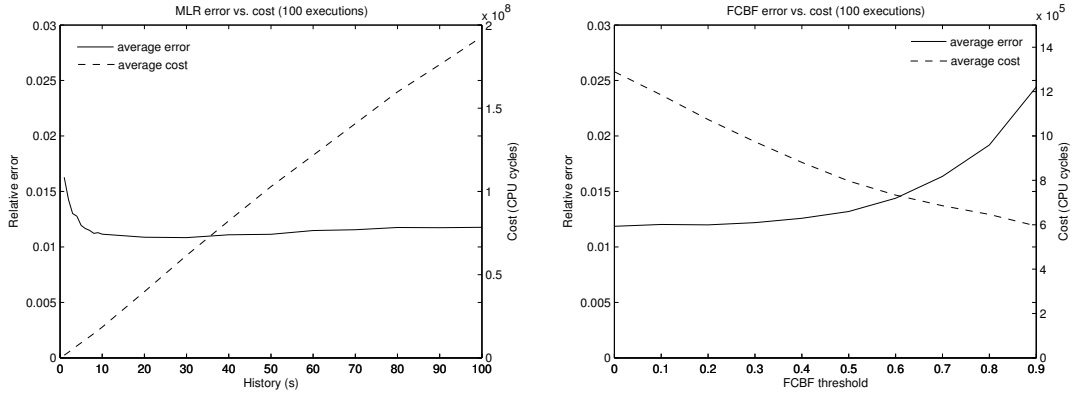


Figure 3.5: Prediction error versus cost as a function of the amount of history used to compute the Multiple Linear Regression (left) and as a function of the Fast Correlation-Based Filter threshold (right)

the prediction model less responsive to sudden changes in the traffic that may change the behavior of a query as well as the most relevant features. In the rest of this chapter, we use a number of observations equal to 60 (i.e., $6s$ of history).

If needed, the recent literature also provides some algorithms that could be easily adapted to automatically adjust the value of n from online measurements of the prediction accuracy [21, 20].

FCBF threshold

The FCBF threshold determines which traffic features are relevant *and* not redundant in modeling the response variable. Large values of this threshold (i.e., closer to 1) will result on fewer features selected.

To understand the most appropriate value for the FCBF threshold, we ran multiple executions in our testbed with values of the threshold ranging from 0 (i.e., all features will be considered relevant but the redundant ones are not selected) to 0.9 (i.e., most features are not selected). Figure 3.5 (right) presents the prediction cost versus the prediction accuracy, as a function of the threshold. The prediction cost includes both the cost of the selection algorithm and the cost of computing the MLR with the selected features. Comparing this graph to Figure 3.5 (left), we can see that using FCBF reduces the overall cost of the prediction by more than an order of magnitude (in terms of CPU cycles) while maintaining similar accuracy.

As the threshold increases, less predictors are selected, and this turns into a decrease in the CPU cycles needed to run the MLR. However the error remains fairly close to the

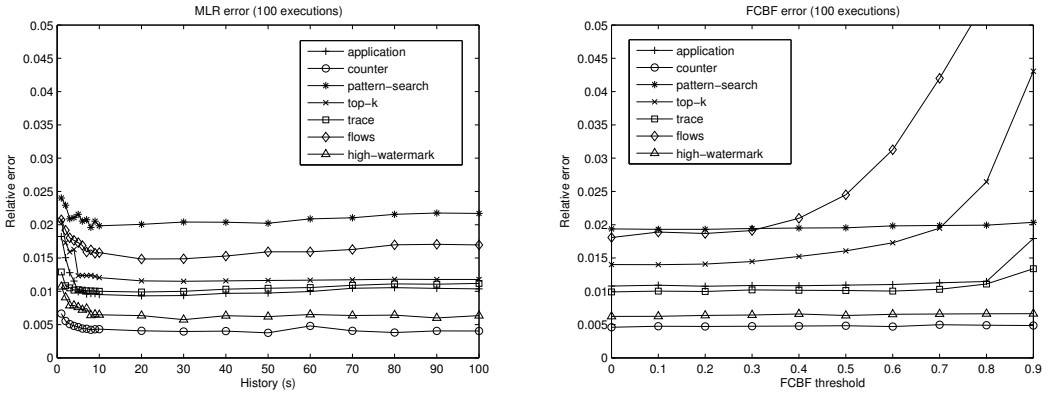


Figure 3.6: Prediction error broken down by query as a function of the amount of history used to compute the Multiple Linear Regression (left) and as a function of the Fast Correlation-Based Filter threshold (right)

minimum value obtained when all features are selected, and starts to ramp up only for relatively large values of the threshold (around 0.6). Very large values of the threshold (above 0.8) experience a much faster increase in the error compared to the decrease in the cost.

Lastly, in Figure 3.6 (right) we plot the prediction accuracy broken down by query, as a function of the FCBF threshold. As expected, queries that can be well modeled with a single feature (e.g., *counter*, *trace*) are quite insensitive to the particular value of the FCBF threshold, while queries that depend on more features (e.g., *flows*, *top-k*) exhibit a significant degradation in the accuracy of the prediction when the FCBF threshold becomes closer to 0.9 (i.e., very few features are selected).

In the rest of this chapter, we use a value of 0.6 for the FCBF threshold that achieves a good trade-off between prediction cost and accuracy for most queries.

3.3.2 Prediction Accuracy

In order to evaluate the performance of our method we ran our seven queries over a set of four traces presented in Table 2.3, namely CESCA-I, CESCA-II, ABILENE and CENIC.

Figure 3.7 shows the time series of the average and maximum error over five executions when running on the packet traces CESCA-I and CESCA-II. The average error in both cases is consistently below 2%, while the maximum error reaches peaks of about 10%. These larger errors are due to external system events unrelated to the traffic that cause a spike in the CPU usage (e.g., cache misses) or due to a sudden change in the

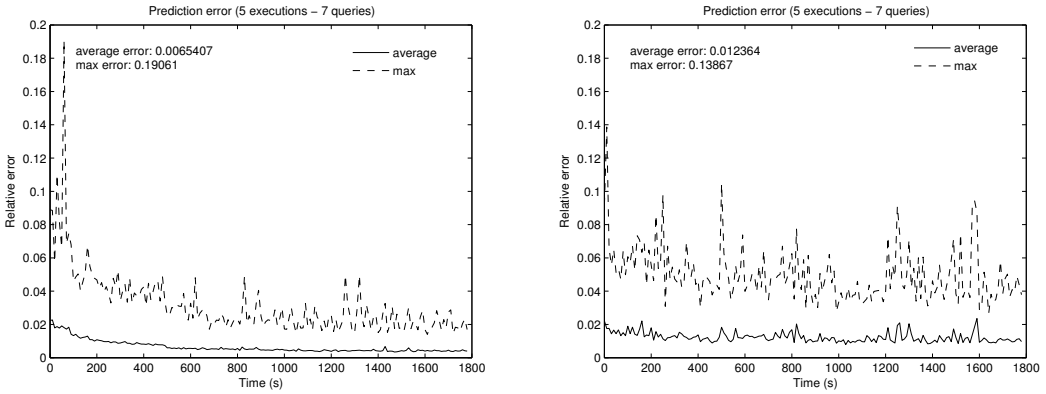


Figure 3.7: Prediction error over time in CESCA-I (left) and CESCA-II (right) traces

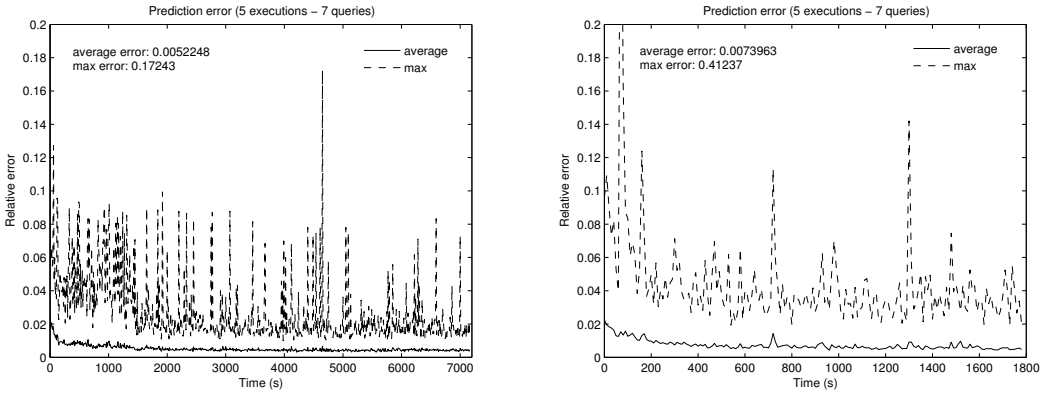


Figure 3.8: Prediction error over time in ABILENE (left) and CENIC (right) traces

traffic patterns that is not appropriately modeled by the features that the prediction is using at that time. However, the time series show that our method is able to converge very quickly.

The trace without payloads (CESCA-I) exhibits better performance, with average errors that drop well below 1%. This is well expected given that the trace contains only headers and the features we have selected allow the system to capture better the queries' CPU usage. Another interesting phenomenon is the downward trend in the maximum and average error. We conjecture that this is due to the kernel scheduler becoming more predictable and thus reducing the likelihood of external system events affecting our method. Similar results are also obtained for the two NLANR's traces (ABILENE and CENIC) as can be observed in Figure 3.8.

In Table 3.2, we show the breakdown of the prediction errors by query. The average

error is very low for each query, with a relatively small standard deviation indicating compact distributions for the prediction errors. As expected, queries that make use of more complex data structures (e.g., *flows*, *top-k* and *pattern-search*) incur the larger errors, but still at most around 3% on average.

It is also very interesting to look at the most frequent features that the selection algorithm identifies as most relevant for each query. Remember that the selection algorithm has no information about what computation the queries perform nor what type of packet traces they are processing. The selected features give hints on what a query is actually doing and how it is implemented. For example, the number of bytes is the predominant traffic feature for the *pattern-search* and *trace* queries when running on the trace with payloads (CESCA-II). However, when processing the trace with just packet headers (CESCA-I), the number of packets becomes the most relevant feature for these queries.

Another example worth noticing is the *top-k* query. In the trace with payloads (CESCA-II) it uses the number of packets as the most relevant predictor. This is an artifact of the particular location of the link where the trace was taken. Indeed, the trace is unidirectional and monitoring traffic destined towards the Catalan network. This results in a trace where the number of unique destination IP addresses is very small allowing the hash table used in the *top-k* query to perform at its optimum with $O(1)$ lookup cost (hence the cost is driven by the number of packets, i.e., the number of lookups). This is not the case for the *flows* query that uses the destination port numbers as well thus increasing the number of entries (thus the lookup and insertion cost) in the hash table.

3.4 Experimental Evaluation

In this section, we evaluate our prediction model in a fully operational system without taking any particular action in the presence of disk accesses. First, we compare the accuracy of our prediction model against two well-known prediction techniques, namely the Exponentially Weighted Moving Average (EWMA) and the Simple Linear Regression (SLR), in order to evaluate their performance under normal traffic conditions (Section 3.4.2). Then, we inject synthetic anomalies in our traces in order to evaluate the robustness of the prediction techniques to extreme traffic conditions (Section 3.4.3). Finally, we discuss the cost of each component in our prediction subsystem, and present the overhead it imposes on the normal operations of the system (Section 3.4.4).

CESCA-I trace (without payloads)

Query	Mean	Stdev	Selected features
application	0.0068	0.0060	*repeated* 5-tuple, packets
counter	0.0046	0.0053	packets
flows	0.0252	0.0203	*new* dst-ip, dst-port, proto
high-watermark	0.0059	0.0063	packets
pattern-search	0.0098	0.0093	packets
top-k	0.0136	0.0183	*new* 5-tuple, packets
trace	0.0092	0.0132	packets

CESCA-II trace (with payloads)

Query	Mean	Stdev	Selected features
application	0.0110	0.0095	packets, bytes
counter	0.0048	0.0066	packets
flows	0.0319	0.0302	*new* dst-ip, dst-port, proto
high-watermark	0.0064	0.0077	packets
pattern-search	0.0198	0.0169	bytes
top-k	0.0169	0.0267	packets
trace	0.0090	0.0137	bytes, packets

ABILENE trace (without payloads)

Query	Mean	Stdev	Selected features
application	0.0065	0.0068	packets
counter	0.0044	0.0063	packets
flows	0.0217	0.0174	*new* dst-ip, dst-port, proto
high-watermark	0.0046	0.0063	packets
pattern-search	0.0116	0.0089	packets
top-k	0.0154	0.0181	*new* src-dst-port, proto, pkts
trace	0.0090	0.0137	packets

CENIC trace (without payloads)

Query	Mean	Stdev	Selected features
application	0.0066	0.0083	packets
counter	0.0064	0.0110	packets
flows	0.0271	0.0341	packets, *new* 5-tuple
high-watermark	0.0058	0.0093	packets
pattern-search	0.0272	0.0248	packets
top-k	0.0218	0.0341	packets, *new* 5-tuple
trace	0.0079	0.0099	packets

Table 3.2: Breakdown of prediction error by query (5 executions)

3.4.1 Alternative Approaches

Exponentially Weighted Moving Average

EWMA is one of the most frequently applied time-series prediction techniques. It uses an exponentially decreasing weighted average of the past observations of a variable to predict its future values. EWMA can be written as:

$$\widehat{Y}_{t+1} = \alpha Y_t + (1-\alpha)\widehat{Y}_t \tag{3.4}$$

where $\widehat{Y}_{t+1}$ is the prediction for the instant $t+1$, which is computed as the weighted average between the real value of Y and its estimated value at the instant t, and α is the *weight*, also known as the *smoothing constant*.

In our case, we can use EWMA to predict the CPU requirements of a particular query at the instant $t+1$, based on a weighted average of the cycles it used in the t previous observations. EWMA has the advantage of being easy to compute, but it only looks at the response variable (i.e., the CPU usage) to perform the prediction. A consequence of this is that EWMA cannot take into account variations in the input traffic to adjust its prediction accordingly. For example, if a batch contains a much larger number of bytes than the previous ones, EWMA will experience large errors for all queries that depend on the number of bytes (e.g., *pattern-search*) and then slowly adapt to the traffic change.

Another example is presented in Figure 3.9. It shows that EWMA is not able to anticipate a significant increase in the CPU requirements of the *counter* query when the number of packets suddenly increases, as it can be observed at time 2.8 and 4.4 seconds. That is, EWMA predicts the effects (i.e., CPU usage) without taking into account the causes (i.e., the number of packets in the batch).

In order to pick the appropriate value of α we ran several experiments on the packets traces. The results presented in this section consider the best prediction accuracy we obtained that correspond to $\alpha = 0.3$, as shown in Figure 3.10.

Simple Linear Regression

SLR is a particular case of the multiple linear regression model, where one single prediction variable is used. The SLR model can be written as:

$$Y_i = \beta_0 + \beta_1 X_i + \varepsilon_i, \qquad i = 1, 2, \ldots, n \tag{3.5}$$

where X is the prediction variable, β_0 is the intercept, β_1 is the unknown coefficient and ε_i are the residuals. As in the case of MLR, the estimator b of the unknown β is

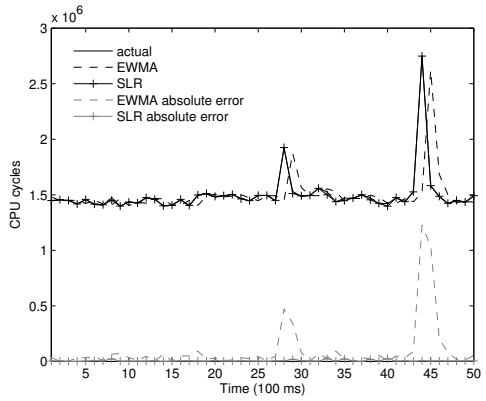


Figure 3.9: EWMA versus SLR predictions for the *counter* query (the 'actual' line almost completely overlaps with the 'SLR' line)

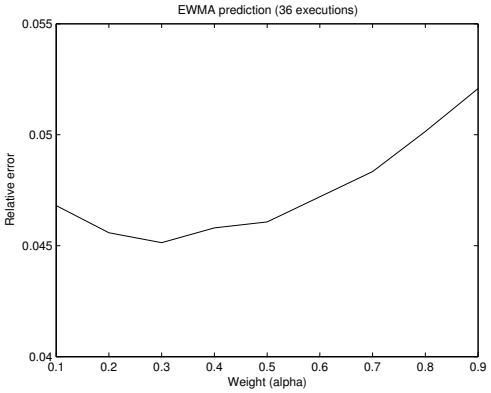


Figure 3.10: EWMA prediction error as a function of the weight α

obtained by minimizing the sum of the squared errors.

For those queries that depend on a single traffic feature, one would expect to obtain similar results to the ones obtained with our prediction model. However, without a feature selection algorithm, the best traffic feature to be used as predictor is not always known, given the lack of explicit knowledge of the queries. In Table 3.2, we show that the most relevant traffic feature for most queries is the number of packets. Thus, in all the experiments presented in this section, we use the number of packets as predictor to perform the regression. The amount of history n is set to $6s$, as in the case of MLR.

In Figure 3.9, we can observe that SLR can anticipate the increase in the CPU requirements of the *counter* query, since its CPU usage only depends on the number of packets (see Table 3.2).

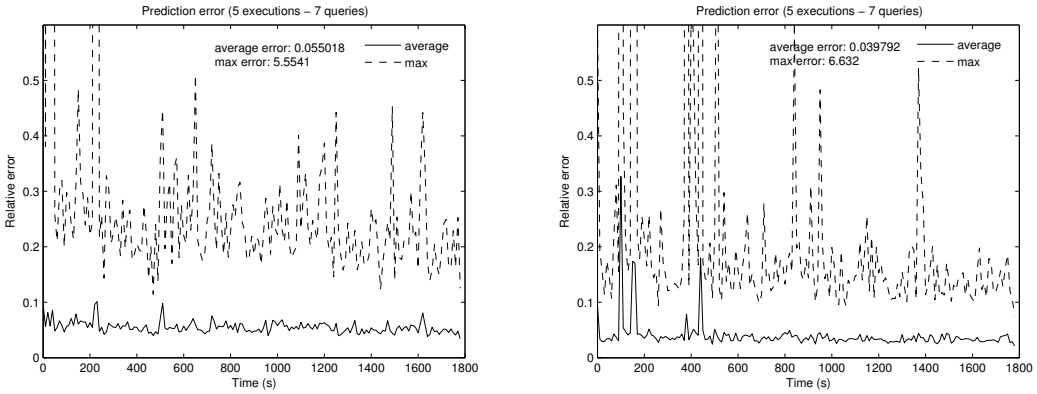


Figure 3.11: EWMA (left) and SLR (right) prediction error over time (CESCA-II trace)

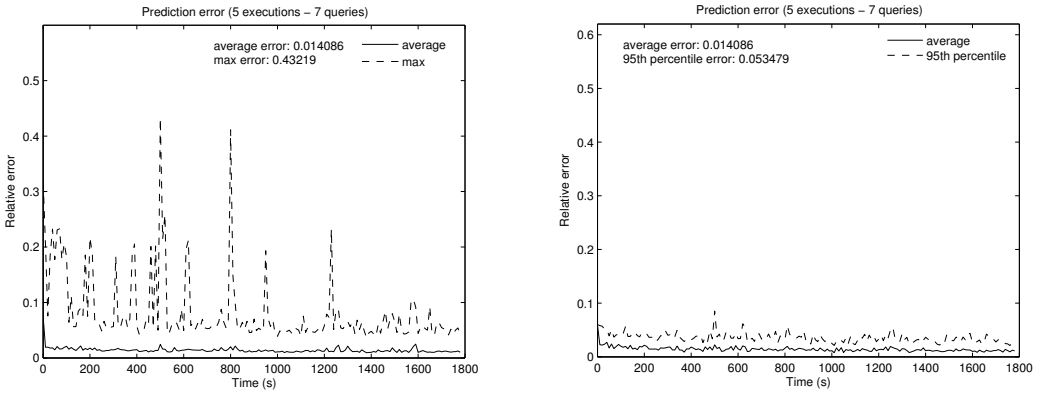


Figure 3.12: MLR+FCBF maximum prediction error (left) and 95^{th}-percentile of the error over time (CESCA-II trace)

3.4.2 Performance under Normal Traffic

Figure 3.11 and 3.12 compare the prediction error of the three methods running over the CESCA-II dataset. Similar results were obtained for the CESCA-I trace. In Section 3.4.3, we also compare the three methods in the presence of traffic anomalies.

The figures show the time series of the average error over 5 executions. The maximum error is computed over an interval of $10s$ and across the 5 executions. Table 3.3 shows the error for each individual query.

All three methods perform reasonably well with average errors around 10% for EWMA and SLR during the entire duration of the trace, and below 3% for MLR. This is expected given the stable traffic conditions that result in relatively stable CPU usage per query. The maximum errors are sometimes relatively large due to the frequent disk

Query	EWMA		SLR		MLR+FCBF	
	Mean	Stdev	Mean	Stdev	Mean	Stdev
application	0.0533	0.0504	0.0254	0.0344	0.0161	0.0179
counter	0.0429	0.0407	0.0050	0.0161	0.0053	0.0119
flows	0.0677	0.0722	0.0579	0.0828	0.0337	0.0335
high-watermark	0.0441	0.0415	0.0077	0.0199	0.0074	0.0139
pattern-search	0.0748	0.3339	0.0805	0.7953	0.0245	0.0592
top-k	0.0504	0.0606	0.0191	0.0386	0.0187	0.0285
trace	0.0563	0.1780	0.0397	0.3579	0.0218	0.0585

Table 3.3: EWMA, SLR and MLR+FCBF error statistics per query (5 executions)

accesses – the trace query stores to disk all packets it receives. However, in Figure 3.12 (right) we show the 95th-percentile of the prediction error, where we can observe the limited impact of disk accesses on the prediction accuracy. Overall, the prediction error for MLR is smaller and more stable than for the other methods.

Inspecting Table 3.3, we can make two observations on the performance of EWMA and SLR. First, with EWMA the error is uniform across queries although the two queries that depend on the number of bytes (i.e., the packet sizes) experience higher variability in the prediction error. This confirms our conjecture that EWMA cannot easily adapt to changes in the traffic mix. The second observation is that SLR performs relatively well over all queries for which the number of packets provides enough information on the CPU usage. However, again for *trace* and *pattern-search*, where the packet size matters, it incurs in larger and more variable errors.

3.4.3 Robustness against Traffic Anomalies

An efficient prediction method for load shedding purposes is most needed in presence of unfriendly traffic mixes. The system may observe extreme traffic conditions when it is monitoring an ongoing denial of service attack, worm infection, or even an attack targeting the measurement system itself. During those events, the query results, even if approximate, are extremely valuable to network operators.

In order to test our prediction system in this type of traffic conditions, we injected synthetic anomalies into our traces. We have generated many different types of attacks to emulate simple such as volume-based denial of service attacks (i.e., an overwhelming number of packets destined to a single target), worm outbreaks (i.e., a large number of packets from many different source and destinations while keeping the destination port number fixed) or attacks against our monitoring system (i.e., attacks that result in

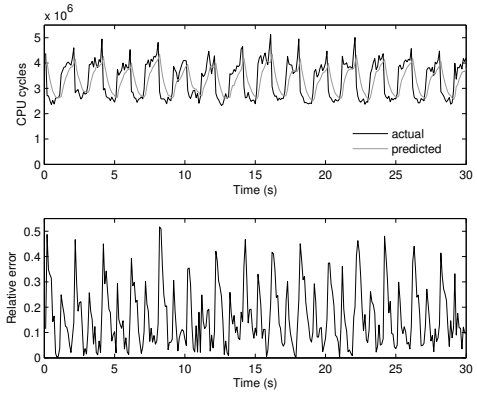


Figure 3.13: Exponentially Weighted Moving Average prediction in the presence of Distributed Denial of Service attacks (*flows* query)

highly variable and unpredictable workloads to the system).

Figures 3.13, 3.14 and 3.15 show the performance of the three methods in the presence of attacks targeting the monitoring system. We injected in the CESCA-II trace a distributed denial of service attack (DDoS) with spoofed source IP addresses and ports, which goes idle every other second to generate a higher variable and difficult to predict workload. The figures show the performance for the *flows* query, which is highly affected by this type of attacks.

In Figure 3.15, we can see that MLR predictions track the actual CPU usage very closely, with errors around the 10% mark (with an average error of 4.77%). MLR can anticipate the increase in CPU cycles, while EWMA (Figure 3.13) is always a little behind, resulting in large oscillations in the prediction error. In the case of SLR (Figure 3.14), since the number of packets does not vary as much as the number of 5-tuple flows, the errors are more stable but persistently around 30% (it converges to the average cost per packet between the anomalous and normal traffic).

We also generated other types of attacks that targeted other queries with similar results. For example, we generated an attack consisting of sending burst of 1500 byte long packets for those queries that depend on the number of bytes (e.g., *trace* and *pattern-search*).

3.4.4 Prediction Cost

To understand the cost of running the prediction we compare the CPU cycles of the prediction subsystem to those spent by the entire CoMo process. Table 3.4 summarizes

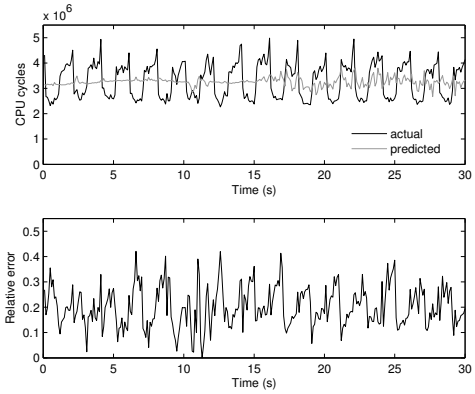


Figure 3.14: Simple Linear Regression prediction in the presence of Distributed Denial of Service attacks (*flows* query)

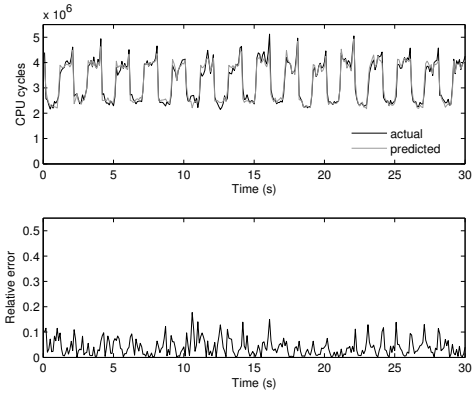


Figure 3.15: Multiple Linear Regression + Fast Correlation-Based Filter prediction in the presence of Distributed Denial of Service attacks (*flows* query)

the results showing the breakdown of the overhead by component.

The feature extraction phase constitutes the bulk of the processing cost. This is not surprising, since several features have to be extracted from every batch (i.e., every $100ms$). Furthermore, our current implementation does not interact with the rest of the CoMo system and incurs additional overhead in order to minimize the modifications in the core platform. An alternative would be to merge the filtering process with the prediction in order to avoid having to scan each packet twice (first to apply the filter and then to extract the features) and to share computations between queries that share the same filter rule.

The overhead introduced by the FCBF feature selection algorithm is only around

Prediction phase	Overhead
Feature extraction	9.070%
FCBF	1.702%
MLR	0.201%
TOTAL	10.973%

Table 3.4: Prediction overhead (5 executions)

1.7% and the MLR imposes an even lower overhead (0.2%), mainly due to the fact that, when using the FCBF, the number of predictors is significantly reduced and thus there is a smaller number of variables to estimate. The use of FCBF allows us to increase the number of features without affecting the cost of MLR.

3.5 Chapter Summary

In this chapter, we presented a system that is able to predict the resource requirements of arbitrary and continuous traffic queries without having any explicit knowledge of the computations they perform. It extracts a set of features from the traffic streams to build an online prediction model with deterministic (and small) worst case computational cost. We implemented our prediction model in a real system and tested it experimentally with real packet traces.

Our results show that the system is able to predict the resources required to run a representative set of queries with small errors, even in the presence of traffic anomalies. In the next chapter, we present the design and implementation of a load shedding scheme that uses the output of this prediction method to guide the monitoring system on deciding when, where and how much load to shed.

One of the current limitations of the system is that it assumes a linear dependency between the CPU usage and the selected features. A solution in that case may be to define new features computed as non-linear combinations of the existing ones. The study of specific network data mining applications that exhibit a non-linear relationship with the traffic features constitutes an important part of our future work. We discuss this aspect in greater detail in Chapter 8.

Chapter 4

Load Shedding System

In this chapter, we provide our answers to the three fundamental questions any load shedding scheme needs to address: (*i*) when to shed load (i.e., which batch), (*ii*) where to shed load (i.e., which query) and (*iii*) how much load to shed (e.g., the sampling rate to apply). We also evaluate the performance of our load shedding scheme, compared to other alternatives, during long-lived executions in a research ISP network. The experimental results show that our load shedding scheme is able to handle extreme overload situations, without introducing undesired packet losses, and that the traffic queries can always complete and return results within acceptable error bounds.

4.1 When to Shed Load

Algorithm 1 presents our load shedding scheme in detail, which controls the Prediction and Load Shedding subsystem presented in Figure 3.2. It is executed at each time bin (i.e., $0.1s$) right after every batch arrival. This way, the system can quickly adapt to changes in the traffic patterns by selecting a different set of features if the current prediction model becomes obsolete.

To decide when to shed load the system maintains a threshold (*avail_cycles*) that accounts for the amount of cycles available in a time bin to process the queries. Since batch arrivals are periodic, this threshold can be dynamically computed as $(time\ bin \times CPU\,frequency) - overhead$, where *overhead* stands for the cycles needed by our prediction subsystem (*ps_cycles*), plus those spent by other CoMo tasks (*como_cycles*), but not directly related to query processing (e.g., packet collection, disk and memory management). The overhead is measured using the TSC, as described in Section 3.2.4. When the predicted cycles for all queries (*pred_cycles*) exceed the *avail_cycles* threshold, excess

Algorithm 1: Load shedding algorithm

Input: Q: Set of q_i queries
b_i: Batch to be processed by q_i after filtering
como_cycles: CoMo overhead cycles
rtthresh, *delay*: Buffer discovery parameters
Data: f_i: Features extracted from b_i
s_i: Features selected for q_i
h_i: MLR history of q_i
ps_cycles, ls_cycles_i, $query\_cycles_i$: TSC measurements

1 $srate = 1$;
2 $pred\_cycles = 0$;
3 **foreach** q_i *in* Q **do**
4 $\quad f_i =$ `feature_extraction`(b_i);
5 $\quad s_i =$ `feature_selection`(f_i, h_i);
6 $\quad pred\_cycles \mathrel{+}=$ `mlr`(f_i, s_i, h_i);
7 $avail\_cycles$ = (time_bin × CPU_frequency) - ($como\_cycles + ps\_cycles$) + ($rtthresh$ - $delay$);
8 **if** $avail\_cycles < pred\_cycles \times (1 + \widehat{error})$ **then**
9 $\quad srate = \frac{\max(0,\, avail\_cycles - \widehat{ls\_cycles})}{pred\_cycles \times (1+\widehat{error})}$;
10 $\quad$ **foreach** q_i *in* Q **do**
11 $\quad\quad b_i =$ `sampling`$(b_i, q_i, srate)$;
12 $\quad\quad f_i =$ `feature_extraction`(b_i);
13 $\quad \widehat{ls\_cycles} = \alpha \times \sum_i ls\_cycles_i + (1-\alpha) \times \widehat{ls\_cycles}$;
14 **foreach** q_i *in* Q **do**
15 $\quad query\_cycles_i =$ `run_query`$(b_i, q_i, srate)$;
16 $\quad h_i =$ `update_mlr_history`$(h_i, f_i, query\_cycles_i)$;
17 $\widehat{error} = \alpha \times \max\left(0, 1 - \frac{pred\_cycles \times srate}{\sum_i query\_cycles_i}\right) + (1-\alpha) \times \widehat{error}$;

load needs to be shed (Algorithm 1, line 8).

We observed that, for certain time bins, *como_cycles* is greater than the available cycles, due to CoMo implementation issues (i.e., other CoMo tasks can occasionally consume all available cycles). This would force the system to discard entire batches, impacting on the accuracy of the prediction and query results. However, this situation can be minimized due to the presence of buffers (e.g., in the capture devices) that allow the system to use more cycles than those available in a single time bin. That is, the system can be delayed in respect to real-time operation as long as it is stable in the steady state.

We use an algorithm, inspired by TCP slow-start [125], to dynamically discover by how much the system can safely (i.e., without loss) exceed the *avail_cycles* threshold. The algorithm continuously monitors the system delay (*delay*), defined as the differ-

ence between the cycles actually used and those available in a time bin, and maintains a threshold (*rtthresh*) that controls the amount of cycles the system can be delayed without loss. *rtthresh* is initially set to zero and increases whenever queries use less cycles than available. If at some point the occupation of the buffers exceeds a predefined value (i.e., the system is turning unstable), *rtthresh* is reset to zero, and a second threshold (initialized to ∞) is set to $\frac{rtthresh}{2}$. *rtthresh* grows exponentially while below this threshold, and linearly once it is exceeded.

This technique has two main advantages. First, it is able to operate without explicit knowledge of the maximum rate of the input streams. Second, it allows the system to quickly react to changes in the traffic.

Algorithm 1 (line 7) shows how the *avail_cycles* threshold is modified to consider the presence of buffers. Note that, at this point, *delay* is never less than zero, because if the system used less cycles than the available in a previous time bin, they would be lost anyway waiting for the next batch to become available.

Finally, as we further discuss in Section 4.3, we multiply the *pred_cycles* by $1+\widehat{error}$ in line 8, as a safeguard against prediction errors, where $\widehat{error}$ is an Exponential Weighted Moving Average (EWMA) of the actual prediction error measured in previous time bins (computed as shown in line 17).

4.2 Where and How to Shed Load

Our approach to shed excess load consists of adaptively reducing the volume of data to be processed by the queries (i.e., the size of the batch).

There are several data reduction techniques that can be used for this purpose. These include filtering (i.e., selection of a subset of packets according to one or more packet attributes), aggregation (i.e., aggregation of the input packets over time across one or several packet attributes) and sampling (i.e., selection of a representative subset of the incoming packets). Since we assume no explicit knowledge of the queries, sampling is the technique that *a priori* should be suitable for most of them, because it permits to retain the measurement detail down to the finest attribute level [48].

In our current implementation, we support uniform packet and flow sampling, and let each query select at configuration time the option that yields the best results. In case of overload, the same sampling rate is applied to all queries (line 11).

We implement packet sampling by randomly selecting packets in a batch with probability p (i.e., the *sampling rate*), while flow sampling randomly selects entire flows with probability p. Thus, the actual number of packets or flows can be simply estimated by

multiplying the number of sampled packets or flows by the inverse of the sampling rate. In order to efficiently implement flow sampling, we use a hash-based technique called *Flowwise sampling* [43]. This technique randomly samples entire flows without caching the flow keys, which reduces significantly the processing and memory requirements during the sampling process. To avoid bias in the selection and deliberate sampling evasion, we randomly generate a new *H3 hash function* [27] per query every measurement interval, which distributes the flows uniformingly and unpredictably. The hash function is applied on a packet basis and maps the 5-tuple flow ID to a value distributed in the range [0, 1). A packet is then selected only if its hash value is less or equal to the sampling rate.

Note that the load shedding scheme based on traffic sampling presented in this chapter has two main limitations. First, using an overall sampling rate for all queries does not differentiate among them. In Chapter 5, we propose a technique that solves this limitation by using different sampling rates for different queries according to external information about their accuracy requirements. Second, there is a large set of imaginable queries that are not able to correctly estimate their unsampled output from sampled streams. In Chapter 6, we propose a method that allows these queries to safely define their own, customized load shedding methods.

4.3 How Much Load to Shed

When the system estimates that the *avail_cycles* threshold is going to be exceeded, excess load has to be shed by reducing the volume of data to be processed. The amount of load to be shed is determined by the maximum sampling rate that keeps the CPU usage below the *avail_cycles* threshold.

Since the system does not differentiate among queries, the sampling rate could be simply set to the ratio $\frac{avail\_cycles}{pred\_cycles}$ in all queries. This assumes that their CPU usage is proportional to the size of the batch (in packets or flows, depending on whether packet or flow sampling is used). However, the cost of a query can actually depend on several traffic features, or even on a feature different from the number of packets or flows. In addition, there is no guarantee of keeping the CPU usage below the *avail_cycles* threshold, due to the error introduced by the prediction subsystem.

We deal with these limitations by maintaining an EWMA of the prediction error (line 17) and correcting the sampling rate accordingly (line 9). Moreover, we have to take into account the extra cycles that will be needed by the load shedding subsystem (ls_cycles), namely the sampling procedure (line 11) and the feature extraction (line 12),

which must be repeated after sampling in order to correctly update the MLR history. Hence, we also maintain an EWMA of the cycles spent in previous time bins by the load shedding subsystem (line 13) and subtract this value from *avail_cycles*.

After applying the mentioned changes, the sampling rate is computed as shown in Algorithm 1 (line 9). The EWMA weight α is set to 0.9 in order to quickly react to changes. It is also important to note that if the prediction error had a zero mean, we could remove it from lines 8 and 9, because buffers should be able to absorb such error. However, there is no guarantee of having a mean of zero in the short term.

4.4 Correctness of the CPU Measurements

In Section 3.2.4, we discussed that measuring the CPU usage at very small timescales incurs several sources of measurement noise. These include context switches, instruction reordering and competition for the system bus. Of these sources of noise, context switches are the most problematic, because we run the monitoring system on a general purpose operating system. Thus, our process can be scheduled out at any time. This could have a severe impact on the CPU measurements used to guide the load shedding procedure and, therefore, on the correctness of the sampling rate computed in Algorithm 1 (line 9), which assumes that the system runs in isolation.

In fact, the CoMo system itself consists of different processes that carry out several tasks besides query processing. These include storage and memory management as well as control of the capture devices. Therefore, there is no way to avoid context switches, if we want the monitoring system to be fully operative. Even if we run the CoMo process responsible of processing the queries with the maximum priority, there would be still no way to avoid context switches among these processes.

Nevertheless, as we experimentally verify in Section 4.5, context switches do not compromise the integrity and performance of our load shedding scheme. Although context switches can occur at any place in Algorithm 1, when they occur while running the rest of CoMo (code not included in Algorithm 1) or executing the prediction tasks (lines 1 to 6), the cycles belonging to the process (or processes) that preempted our process are accounted for as overhead. This is precisely what we need, since we have to discount these cycles from the *avail_cycles* threshold. Even in the case that a context switch occurs after computing the sampling rate (and thus the cycles are not discounted from the *avail_cycles*), they would be discounted in the next time bin, given that in this case the delay for the next time bin would be larger than zero (see line 7).

Another problem may appear if our process is scheduled out while processing a query

(line 15). Although we can recover from this situation, like in the previous case, the MLR history would become corrupt, because cycles belonging to other processes would be considered as cycles actually used by the query. Since this can have a negative impact on the accuracy of future predictions, we discard this observation from the history and replace it with the predicted value.

4.5 Evaluation and Operational Results

In this section, we evaluate our load shedding system in the CESCA scenario (described in Section 2.3), where the traffic load and query requirements exceed by far the capacity of the monitoring system. We also assess the impact of sampling on the accuracy of the queries, and compare the results of our predictive scheme to two alternative systems. We also present the overhead introduced by the complete load shedding procedure and discuss possible alternatives to reduce it further. Finally, we study the impact of traffic attacks on our load shedding scheme.

Throughout the evaluation, we present the results of three 8 hours-long executions (see Table 2.4 for details). In the first one (CESCA-III), we run a modified version of CoMo that implements our load shedding scheme (*predictive*), while in the other two executions (CESCA-IV and CESCA-V) we repeat the same experiment, but using a version of CoMo that implements two alternative load shedding approaches described below. For these experiments, we selected a set of seven queries from those in Table 2.2: *application*, *counter*, *flows*, *high-watermark*, *pattern-search*, *top-k* and *trace*. The *autofocus*, *super-sources* and *p2p-detector* queries are evaluated in Chapters 5 and 6.

4.5.1 Alternative Approaches

The first alternative (*original*) consists of the original version of CoMo, which discards packets from the input buffers in the presence of overload. In our case, overload situations are detected when the occupation of the capture buffers exceeds a pre-defined threshold. Details of this execution are available in Table 2.3 (CESCA-IV execution).

For the second alternative (*reactive*), we implemented a more complex reactive method that makes use of packet and flow sampling. This system is equivalent to a predictive one, where the prediction for a time bin is always equal to the cycles used to process the previous batch. This strategy is similar to the one used in SEDA [133]. In particular, we measure the cycles available in each time bin, as described in Section 4.1, and when the cycles actually used to process a batch exceed this limit, sampling is

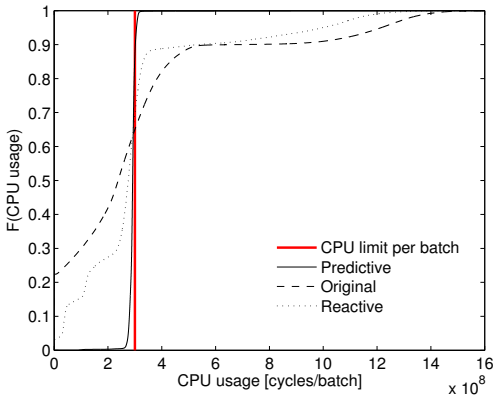


Figure 4.1: Cumulative Distribution Function of the CPU usage per batch

applied to the next time bin. The sampling rate for the time bin t is computed as:

$$srate_t = \min\left(1,\ \max\left(\alpha,\ srate_{t-1} \times \frac{avail\_cycles_t - delay}{consumed\_cycles_{t-1}}\right)\right) \tag{4.1}$$

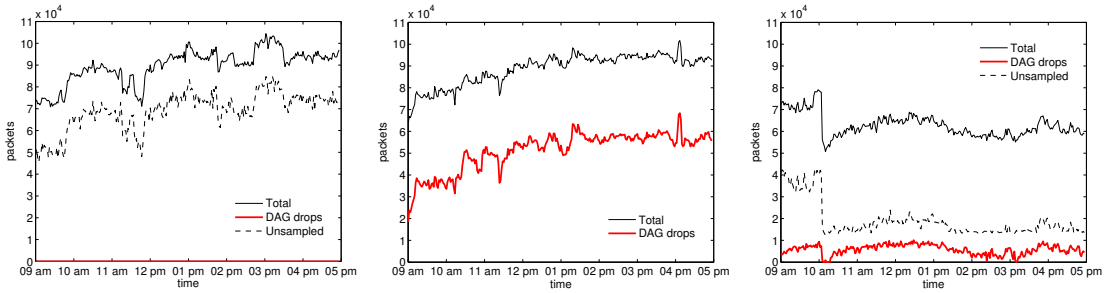
where $consumed\_cycles_{t-1}$ stands for the cycles used in the previous time bin, *delay* is the amount of cycles by which $avail\_cycles_{t-1}$ was exceeded, and α is the minimum sampling rate we want to apply. Table 2.3 (CESCA-V execution) presents the details of this execution.

4.5.2 Performance

In Figure 4.1, we plot the Cumulative Distribution Function (CDF) of the CPU cycles consumed to process a single batch (i.e., the service time per batch). Recall that batches represent $100ms$ resulting in 3×10^8 cycles available to process each batch in our 3 GHz system.

The figure shows that the *predictive* system is stable. As expected, sometimes the limit of available cycles is slightly exceeded owing to the buffer discovery algorithm presented in Section 4.1. The CDF also indicates good CPU usage between 2.5 and 3×10^8 cycles per batch (i.e., the system is rarely under- or over-sampling).

On the contrary, the service time per batch when using the *original* and *reactive* approaches is much more variable. It is often significantly larger than the batch interarrival time, with a probability of exceeding the available cycles greater than 30% in both executions. This leads to very unstable systems that introduce packet drops without control, even of entire batches. Figure 4.1 shows that more than 20% of the batches



(a) *Predictive* load shedding (b) *Original* CoMo (c) *Reactive* load shedding

Figure 4.2: Link load and packet drops during the evaluation of each load shedding method

in the *original* execution, and around 5% in the *reactive* one, are completely lost (i.e., service time equal to zero).

Figure 4.2 illustrates the impact of exceeding the available cycles on the input stream. The line labeled 'DAG drops' refers to the packets dropped on the network capture card due to full memory buffers (results are averaged over one second). These drops are uncontrolled and contribute most to the errors in the query results. The line 'unsampled' counts the packets that are not processed due to packet or flow sampling.

Figure 4.2(a) confirms that, during the 8 hours, not a single packet was lost by the capture card when using the predictive approach. This result indicates that the system is robust against overload.

Figures 4.2(b) and 4.2(c) show instead that the capture card drops packets consistently during the entire execution.[1] The number of drops in the *original* approach is expected given that the load shedding scheme is based on dropping packets on the input interface. Instead, in the case of the reactive approach, the drops are due to incorrect estimation of the cycles needed to process each batch. The *reactive* system bases its estimation on the previous batch only. In addition, it must be noted that traffic conditions in the *reactive* execution were much less adverse, with almost half of traffic load, than in the other two executions (see Table 2.4). It is also interesting to note that when the traffic conditions are similar in all executions (from 9am to 10am), the number of unsampled packets plus the packets dropped by the *reactive* system is very similar to the number of unsampled packets by the *predictive* one, in spite of that they incur different processing overheads.

[1]The values are a lower bound of the actual drops, because the loss counter present in the DAG records is only 16-bit long.

Query	*predictive*	*original*	*reactive*
application (pkts)	1.03% ±0.65	55.38% ±11.80	10.61% ±7.78
application (bytes)	1.17% ±0.76	55.39% ±11.80	11.90% ±8.22
counter (pkts)	0.54% ±0.50	55.03% ±11.45	9.71% ±8.41
counter (bytes)	0.66% ±0.60	55.06% ±11.45	10.24% ±8.39
flows	2.88% ±3.34	38.48% ±902.13	12.46% ±7.28
high-watermark	2.19% ±2.30	8.68% ±8.13	8.94% ±9.46
top-k	1.41 ±3.32	21.63 ±31.94	41.86 ±44.64

Table 4.1: Breakdown of the accuracy error of the different load shedding methods by query ($mean \pm stdev$)

4.5.3 Accuracy

We modified the source code of five of the seven queries executed by the monitoring system in order to allow them to estimate their unsampled output when load shedding is performed. This modification was simply done by multiplying the metrics they compute by the inverse of the sampling rate being applied to each batch.

We did not modify the *pattern-search* and *trace* queries, because no standard procedure exists to recover their unsampled output from sampled streams and to measure their error. In this case, the error should be measured in terms of the application that uses the output of these two queries. In Chapter 6, we present an alternative load shedding mechanism for those queries that are not robust against sampling.

Table 4.1 presents the error in the results of the five queries averaged across all the measurement intervals. We can observe that, although our load shedding system introduces a certain overhead, the error is kept significantly low compared to the two reactive versions of the CoMo system. Recall that the traffic load in the *reactive* execution was almost half of that in the other two executions. Large standard deviation values are due to long periods of consecutive packet drops in the alternative systems. It is also worth noting that the error of the *top-k* query obtained in the *predictive* execution is consistent with that of [12]. Figure 4.3 shows that the overall error of the *predictive* system is below 2% in average, while the error in the two alternative systems is significantly larger.

4.5.4 Overhead

Figure 4.4 presents the CPU usage during the *predictive* execution, broken down by the three main tasks presented in Algorithm 1 (i.e., *como_cycles*, *query_cycles* and $ps\_cycles + ls\_cycles$). We also plot the cycles the system estimates as needed to process all incoming traffic (i.e., *pred_cycles*). From the figure, it is clear that the system is under severe stress because, during almost all the execution, it needs more than twice

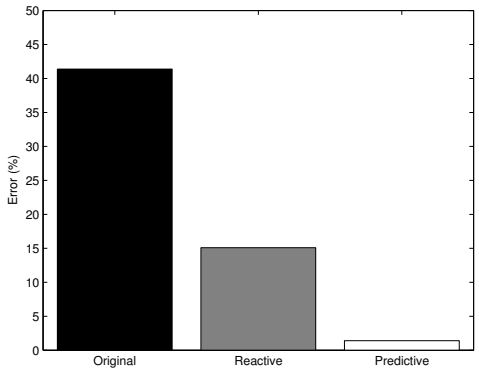


Figure 4.3: Average error in the answer of the queries

the cycles available to run our seven queries without loss.

The overhead introduced by our load shedding system ($ps\_cycles + ls\_cycles$) to the normal operation of the entire CoMo system is reasonably low compared to the advantages of keeping the CPU usage and the accuracy of the results well under control. Note that in Section 3.4.4 the cost of the prediction subsystem was measured without performing load shedding. This resulted in an overall processing cost similar to the *pred_cycles* in Figure 4.4 and, therefore, in a lower relative overhead.

While the overhead incurred by the load shedding mechanism itself (ls_cycles) is similar in any load shedding approach, independently of whether it is predictive or reactive, the overhead incurred by the prediction subsystem (ps_cycles) is particular to our predictive approach. As discussed in Section 3.4.4, the bulk of the prediction cost corresponds to the feature extraction phase, which is entirely implemented using a family of memory-efficient algorithms that could be directly built in hardware [57]. Alternatively, this overhead could be reduced significantly by applying sampling in this phase or simply reducing the accuracy of the bitmap counters.

4.5.5 Robustness against Traffic Anomalies

In this experiment, we show how our load shedding scheme can effectively control the CPU usage under unfriendly traffic mixes by gracefully degrading the accuracy of the queries via traffic sampling. We consider the *flows* query, which tracks the number of active (i.e., for which at least one packet was observed) 5-tuple flows in the packet streams and reports the count every measurement interval.

During 20 seconds (200 batches), we inject a burst of traffic corresponding to a SYN-

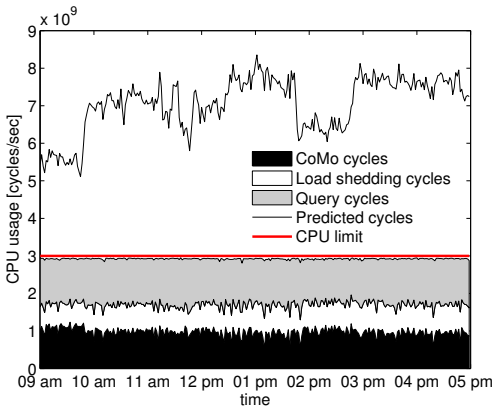


Figure 4.4: CPU usage after load shedding (stacked) and estimated CPU usage (*predictive* execution)

flood attack with spoofed IP source addresses in the CESCA-I (without payloads) and CESCA-II (with payloads) traces in order to force higher CPU usage. To facilitate the representation of the results, we only run this single query and we manually set the *avail_cycles* threshold to 6M and 4M cycles per batch in the CESCA-I and CESCA-II traces, respectively.

The left plot in Figure 4.5 shows the evolution of the CPU usage during the anomaly with and without load shedding (with flow sampling) for the CESCA-I trace. Without load shedding, the CPU cycles increase from 4.5M to 11M cycles during the anomaly (assuming an infinite buffer that causes no packet drops). Instead, when load shedding is enabled, the CPU usage is well under control within a 5% margin of the set target usage.

The right plot in Figure 4.5 shows the query accuracy during the anomaly. To estimate the error in the absence of load shedding, we emulate a system with a buffer of $200ms$ of traffic and 6M cycles available to process incoming traffic. If the CPU usage exceeds 6M, we assume that a queue of packets starts building up until the buffer is full and incoming packets are dropped without control. When load shedding is enabled, the error in the estimation of the number of flows when using flow sampling is less than 1%, while when using packet sampling it is slightly larger than 5%. Without load shedding, the measurement error is in the $35 - 40\%$ range.

Figure 4.6 shows that similar results are obtained for the CESCA-II trace with payloads. The slight differences in both figures are due to the way we generated the attack. In particular, we inserted a SYN-flood attack out of every 5 packets in the original trace. Since the trace with payloads has less packets than the trace with only packet headers

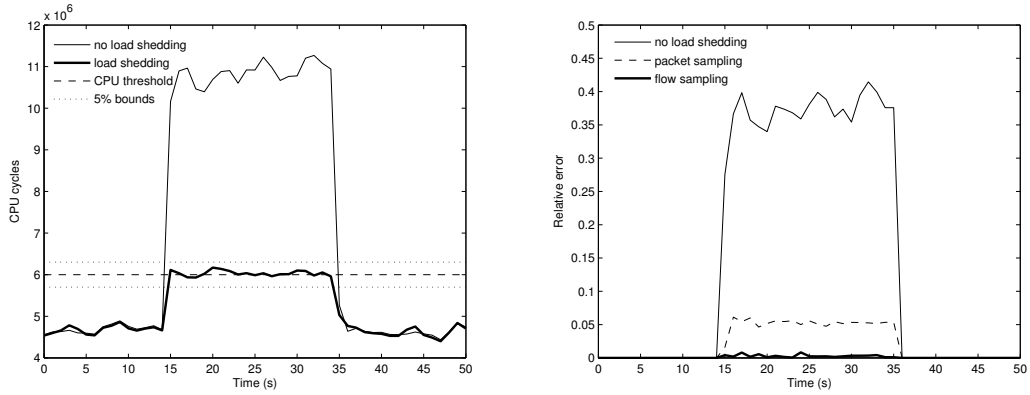


Figure 4.5: CPU usage (left) and errors in the query results (right) with and without load shedding (CESCA-I trace)

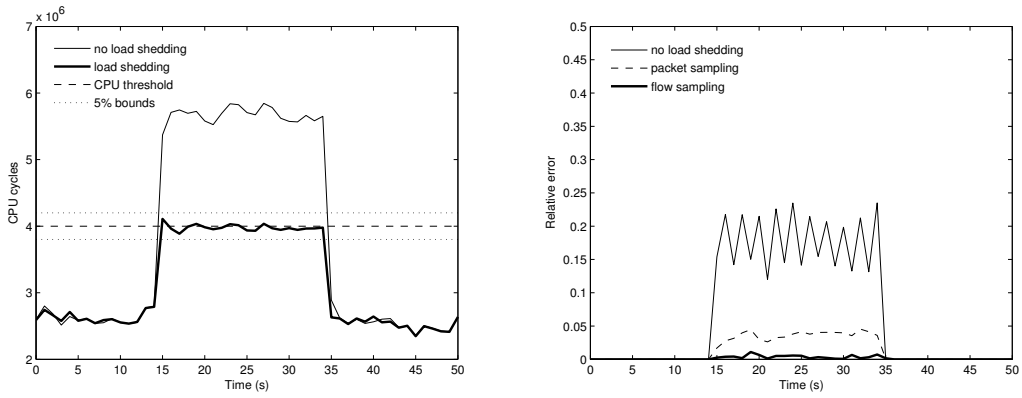


Figure 4.6: CPU usage (left) and errors in the query results (right) with and without load shedding (CESCA-II trace)

(see Table 2.3), it resulted in a less aggressive attack.

4.6 Chapter Summary

The experimental results from an operational network clearly showed that effective load shedding methods are indispensable to allow current network monitoring systems to sustain the rapidly increasing data rates and complexity of traffic analysis methods.

In this chapter, we presented the design and implementation of a predictive load shedding scheme that operates without explicit knowledge of the traffic queries and quickly adapts to overload situations by gracefully degrading their accuracy via packet

and flow sampling. The proposed scheme uses the output of the prediction system described in Chapter 3 to guide the system on deciding when, where and how much load to shed in the presence of overload.

We implemented our load shedding scheme in an existing monitoring system and evaluated its performance and correctness in a research ISP network. We demonstrated the robustness of our method through an 8 hours-long continuous execution, where the system exhibited good CPU utilization without packet loss, even when it was under severe stress. We also pointed out a significant gain in the accuracy of the results compared to two versions of the same monitoring system that use a non-predictive load shedding approach instead, while introducing a tolerable (and deterministic) overhead to the normal operation of the system.

In the following chapters, we present alternative load shedding mechanisms for those queries that are not robust against sampling. We also develop smarter load shedding strategies that allow the system to apply different sampling rates to different queries to further improve the accuracy of the monitoring system during overload situations.

Chapter 5

Fairness of Service and Nash Equilibrium

The load shedding strategy described in Chapter 4 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 of 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., *top-k* flows [12]) 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.

In this chapter, we present the design of a load shedding strategy that supports multiple and competing traffic queries. The main novelty of our approach is that it 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.

5.1 Objectives and Desirable Features

Our main objective is to design a load shedding strategy that solves the limitations discussed above. In particular, we would like our method to exhibit some desirable features, such as:

- *Fairness of service*: In an environment where multiple network traffic queries are competing for the same shared resources, it is important to ensure fairness and avoid starvation of any query. In particular, we would like the system to offer similar levels of accuracy to similar queries.

- *Predictability*: Users should understand how the system will handle their queries in case of overload, according for example to their tolerance to loss or the amount of resources they request to the system. This way, users can write queries in such a way that the impact of load shedding decisions on their results is minimized.

- *Minimal knowledge of queries*: The load shedder should operate with minimum external knowledge of the queries. This reduces the burden on the users and minimizes the probability of compromising the system by providing incorrect information about a query. The system should also enforce that any external information used by the load shedder is correct and free of bias, especially when this information is provided by end users.

- *Scalability and minimum overhead*: The load shedder should be lightweight enough to make quick load shedding decisions, since network monitoring applications usually have real-time expectations. However, load shedding decisions are often an optimization problem, where the cost of finding an optimal solution may increase exponentially with the number of queries. Thus, methods that require polynomial time, even if they only offer near-optimal solutions, would be more desirable than those that require an exhaustive search of the entire solution space.

5.2 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.

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 $q \in Q$

Table 5.1: Notation and definitions

Table 5.1 summarizes the notation used throughout this chapter. For each query $q \in Q$ at time t, $\widehat{d}_q$ and c_q denote the cycles predicted (using the method described in Chapter 3) and those actually allocated by the system, respectively. Let C be the system capacity in CPU cycles at time t.[1] A vector $c = \{c_q \mid q \in Q\}$ of allocated cycles is *feasible* if the following two constraints are satisfied:

$$\forall_{q \in Q} \; c_q \leq \widehat{d}_q \tag{5.1}$$

$$\sum_{q \in Q} c_q \leq C \tag{5.2}$$

The max-min fair share allocation policy is then defined as follows [19]:

Definition 5.1. *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 $c_{q'} > \bar{c}_{q'}$.*

5.2.1 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 permits to keep the load shedding algorithm very simple yet flexible enough to control resource usage.

[1] In Chapter 4 we described how the system capacity is measured. We also showed that it varies over time due to the system overhead and the prediction error in previous time bins.

Minimum constraints however are not considered in the classical definition of max-min fairness. For this reason, we modify the constraint (5.1) of the standard max-min fair policy by the following one in order to introduce the notion of a minimum sampling rate:

$$\forall_{q \in Q} \ (m_q \times \widehat{d_q}) \leq c_q \leq \widehat{d_q} \tag{5.3}$$

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. However, if it exists, it is unique. 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 (5.2) and (5.3), 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 5.3, 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 $\widehat{d_q}$), because given two equivalent queries, the least demanding one will have more chances to run.

5.2.2 Fairness in terms of Packet Access

The strategy described in Section 5.2.1 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 $p = \{p_q \,|\, q \in Q\}$ of sampling rates is *feasible* if the following two constraints are satisfied:

$$\forall_{q \in Q} \; m_q \leq p_q \leq 1 \tag{5.4}$$

$$\sum_{q \in Q} (p_q \times \widehat{d_q}) \leq C \tag{5.5}$$

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

Definition 5.2. *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 5.2.1, 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.

5.2.3 Online 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 [19].

Our algorithm is based on the classical max-min fair share algorithm [19], but it includes the minimum sampling rate constraints. 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:

$$\sum_{q \in Q} (m_q \times \widehat{d_q}) \leq C \tag{5.6}$$

If (5.6) 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). Finally, the sampling rates p_q to be applied to each query $q \in Q$ can be directly computed as the ratio between c_q and $\widehat{d_q}$.

The second phase of the strategy that is fair in terms of packet access consists 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.

5.3 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., $m_q \times \widehat{d_q}$).[2] The objective of a non-cooperative player 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 $-q$

[2]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).

subscript stands for all queries except q).

In particular, according to the strategies described in Sections 5.2.1 and 5.2.2, 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:

$$u_q(a_q, a_{-q}) = \begin{cases} a_q + mmfs_q(C - \sum_{i:u_i>0} a_i), & \text{if } \sum_{i:a_i \leq a_q} a_i \leq C \\ 0, & \text{if } \sum_{i:a_i \leq a_q} a_i > C \end{cases} \tag{5.7}$$

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 5.7 gives us 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 5.7.

Definition 5.3. *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^** [104].

Theorem 5.1. *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 5.3, 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 5.7, 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 [128], 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 a utility function that drops to zero if the sampling rate is less than 1.

5.4 Simulation Results

In this section, we study the differences between the two variants of our load shedding strategy for non-cooperative monitoring applications, namely max-min fairness in terms of access to the CPU ($mmfs\_cpu$) and in terms of access to the incoming packet stream ($mmfs\_pkt$).

We set up a simple simulation environment that allows us to compare both strategies and easily discuss the effects of the level of overload and the minimum sampling rate constraints on the accuracy of the traffic queries. The simulation scenario does not pretend to be representative nor demonstrate the superiority of one strategy over the other, but instead it tries to highlight the differences between them. A comparison of both strategies in a real scenario is presented in Section 5.5.

For ease of exposition, in this experiment we only simulate two types of queries. The first type is a lightweight query (*light*) which is tolerant to low sampling rates. The second type (*heavy*) consists of a more expensive query, but less tolerant to packet loss. In particular, the computational cost of *heavy* is 10 times that of *light*. Given that in a simulated environment we cannot measure the actual accuracy of these queries, we define the accuracy of *light* as $1 - ((1 - p_{light}) \times 0.05)$ when $p_{light} > 0$, and 0 when $p_{light} = 0$, while the accuracy of *heavy* is defined as p_{heavy}. These accuracy functions emulate the behavior of the *counter* and *trace* queries respectively (see Table 2.2), as we show in Section 5.5.

In the simulation, we vary the minimum sampling rate (m_q) of all queries and the overload level (K) 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 (the system capacity is 0). Therefore, the system capacity is computed as $C \times (1 - K)$.

Figure 5.1 shows the difference in the average and minimum accuracy between the packet-based ($mmfs\_pkt$) and the CPU-based ($mmfs\_cpu$) strategies, when running 1 *heavy* and 10 *light* queries concurrently (i.e., the sum of the demands of the *light* and *heavy* queries is equal). While the difference in the average accuracy is negligible (it is almost a flat surface), Figure 5.1 (right) confirms that the packet-based strategy

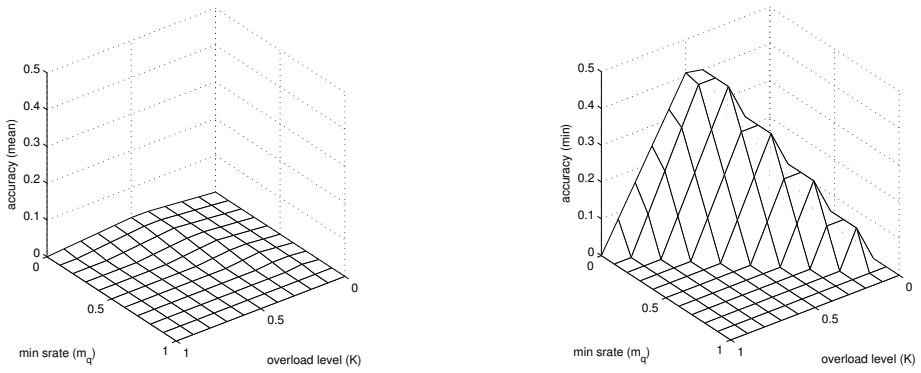


Figure 5.1: Difference in the average (left) and minimum (right) accuracy between the $mmfs\_pkt$ and $mmfs\_cpu$ strategies when running 1 *heavy* and 10 *light* queries in a simulated environment. Positive differences show the superiority of $mmfs\_pkt$ over $mmfs\_cpu$

significantly improves the minimum accuracy, because $mmfs\_cpu$ highly penalizes the accuracy of the *heavy* query. This result indicates that, in terms of accuracy, $mmfs\_pkt$ is significantly fairer than $mmfs\_cpu$. The diagonal from $m_q = 0$ to $K = 0$ shows the point from which both strategies are equivalent, because the *heavy* query is disabled.

Although the simulation scenario was especially chosen to emphasize the differences between both strategies, note however that the same figure for the minimum accuracy would be obtained by defining the same accuracy functions for both queries (*light* and *heavy*), given that the minimum accuracy in the simulation is driven by the most expensive query. Section 5.5 discusses the practical implications of these results.

5.5 Experimental Evaluation

In this section, we evaluate our load shedding scheme in the CoMo platform. We first validate in a real environment the results obtained through simulation. Next, we evaluate the performance of the two variants of our load shedding scheme ($mmfs\_cpu$ and $mmfs\_pkt$) with a diverse set of real traffic queries. We compare both methods to a system that does not implement any explicit load shedding mechanism and to the system presented in Chapter 4, which applies the same sampling rate to all queries. We also compare our solution to the reactive system presented in Section 4.5.

We implemented both strategies in the CoMo platform and performed several experiments using the CESCA-II trace, which contains the full packet payloads. Although sim-

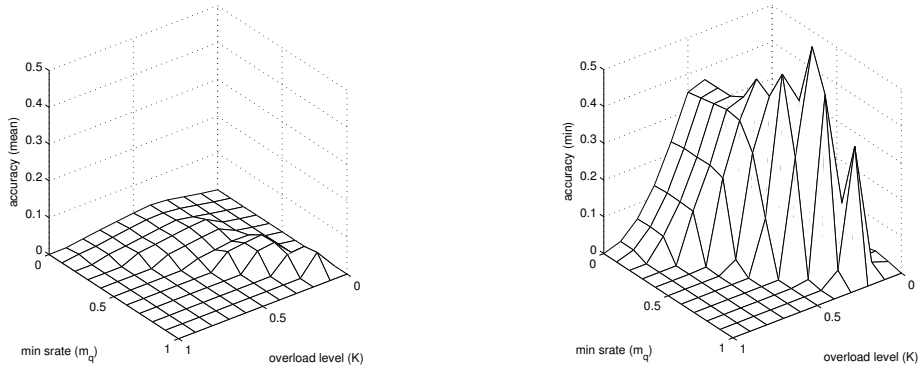


Figure 5.2: Difference in the average (left) and minimum (right) accuracy between the $mmfs\_pkt$ and $mmfs\_cpu$ strategies when running 1 *trace* and 10 *counter* queries. Positive differences show the superiority of $mmfs\_pkt$ over $mmfs\_cpu$

ilar results were obtained using publicly available datasets, such as those in the NLANR repository [103], we present only the results when running the system on a packet trace with the entire payloads in order to evaluate those CoMo queries that require packet contents to operate (e.g., *pattern-search*).

5.5.1 Validation of the Simulation Results

In this experiment, we validate the results obtained through simulation in Section 5.4. We perform 121 executions with a *trace* query plus 10 *counters* in our packet trace (see Table 2.2 for details about the queries). The accuracy of *counter* is computed according to the actual error in the results, given that we can now obtain its actual output from the entire packet trace. Thus, the accuracy is simply defined as $1 - \varepsilon_{counter}$, while the accuracy of *trace* is left as in the simulation (i.e., p_{trace}), since no standard procedure exists to measure the error of this query, as we further discuss in Section 5.5.2. When a query is disabled, its accuracy drops to 0.

Figure 5.2 shows the difference in the average and minimum accuracy between both strategies, which resembles the results obtained through simulation. The differences are explained by the fact that in a real scenario we can measure the actual accuracy of the queries. In addition, the results are now computed as the minimum of the average accuracy per batch over all queries. We cannot plot the minimum per batch because it would lead to unrealistic results, given that the demands of a query vary over time (they depend on the incoming traffic). For example, a real system may be highly overloaded for a certain batch, despite the value of K for that execution being very low.

Query	m_q	Accuracy (mean ±stdev, $K = 0.5$)				
		no_lshed	*reactive*	*eq_srates*	*mmfs_cpu*	*mmfs_pkt*
application	0.03	0.57 ±0.50	0.81 ±0.40	0.99 ±0.04	1.00 ±0.00	1.00 ±0.03
autofocus	0.69	0.00 ±0.00	0.00 ±0.00	0.05 ±0.12	0.97 ±0.06	0.98 ±0.04
counter	0.03	0.00 ±0.00	0.02 ±0.12	1.00 ±0.00	1.00 ±0.00	0.99 ±0.01
flows	0.05	0.00 ±0.00	0.66 ±0.46	0.99 ±0.01	0.95 ±0.07	0.95 ±0.06
high-watermark	0.15	0.62 ±0.48	0.98 ±0.01	0.98 ±0.01	1.00 ±0.01	0.97 ±0.02
pattern-search	0.10	0.66 ±0.08	0.63 ±0.18	0.69 ±0.07	0.20 ±0.08	0.41 ±0.08
super-sources	0.93	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	0.95 ±0.04	0.95 ±0.04
top-k	0.57	0.42 ±0.50	0.67 ±0.47	0.96 ±0.09	0.99 ±0.03	0.96 ±0.07
trace	0.10	0.66 ±0.08	0.63 ±0.18	0.68 ±0.01	0.64 ±0.17	0.41 ±0.08

Table 5.2: Sampling rate constraints (m_q) and average accuracy when resource demands are twice the system capacity ($K = 0.5$)

5.5.2 Analysis of the Minimum Sampling Rates

In general, the minimum sampling rate constraint of a query (m_q) 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 5.2 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. Note that the value of 5% is arbitrary and is used just as an example to evaluate the different methods. For all queries, we measure the relative error as defined in Section 2.2. 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.[3] Table 5.2 also shows that the level of tolerance of most queries to sampling is very different, resulting in very diverse values of m_q.

5.5.3 Performance Evaluation with a Real Set of Queries

In this experiment, we study both strategies by running the set of nine queries listed in Table 5.2. A detailed description of these queries is available in Section 2.2. 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

[3]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.

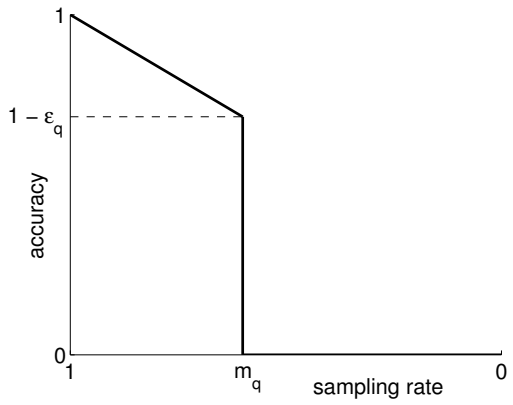


Figure 5.3: Accuracy of a generic query

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 a modified version of the load shedding strategy presented in Chapter 4, which applies the same sampling rate to all queries (*eq_srates*). In this version, when the sampling rate is below the minimum sampling rate of a query, the query is 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 the reactive approach presented in Section 4.5. 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 [133].

Figure 5.3 shows how the accuracy is defined to evaluate the different load shedding alternatives. In particular, the accuracy of a query is computed as $1 - \varepsilon_q$ when $p_q \geq m_q$ and is considered 0 otherwise, where ε_q is the actual error of the query as described in Section 2.2. Note that the values between sampling rate 1 and m_q depend on the actual error of each particular query and, therefore, the accuracy of each query results in different shapes. 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 complex utility

functions needed by other systems (e.g., [128]). Throughout the evaluation, we use the accuracy of the queries as a performance metric to compare the different load shedding 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 5.5.2 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 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 5.3.

Figure 5.4 plots the average and minimum accuracy among all queries when using the minimum sampling rate constraints defined in Table 5.2. The figure shows that the $mmfs\_cpu$ and $mmfs\_pkt$ strategies outperform the three alternative systems. The good performance of the original version of CoMo (no_lshed) 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. Recall that the system capacity is computed as $C \times (1-K)$, where C is experimentally set to the minimum number of cycles that assure that no sampling is applied in our testbed. The figure also confirms that, even with a diverse set of real queries, the $mmfs\_pkt$ strategy significantly improves the minimum accuracy as compared to the $mmfs\_cpu$ strategy, while maintaining a similar average accuracy.

Table 5.2 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 pre-defined 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 5.2), 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.

This is the main reason why the CoMo system implements its own scheduler, and does

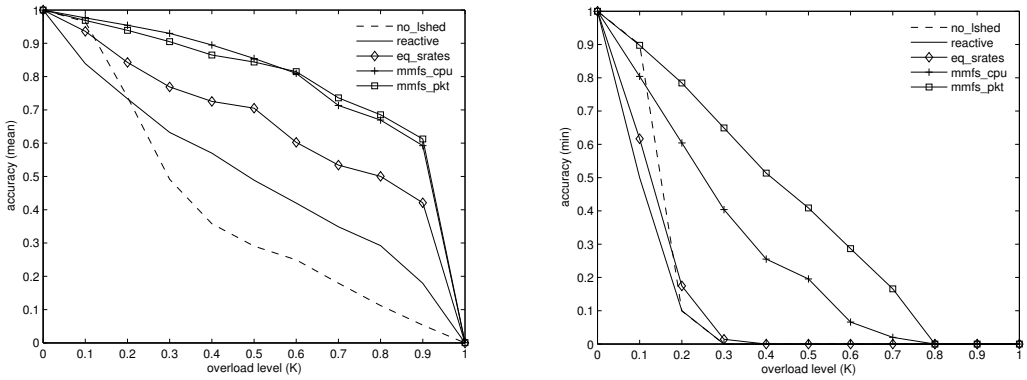


Figure 5.4: Average (left) and minimum (right) accuracy of various load shedding strategies when running a representative set of queries with fixed minimum sampling rate constraints

not leave the responsibility of scheduling the monitoring applications to the Operating System, which is basically designed to be fair in terms of access to the CPU. However, in our current implementation, the strategy to be used can be chosen at configuration time.

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 5.5 the accuracy of the *autofocus* query 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 5.2. 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.

5.5.4 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 Chapter 4, 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 Chapter 4 the traffic features could be recomputed just once, given that all queries always process equal

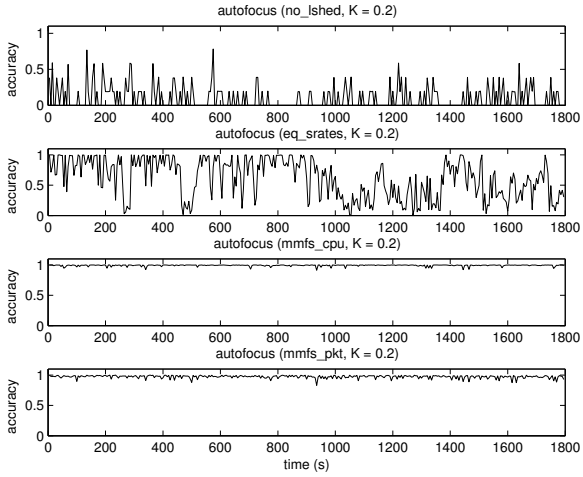


Figure 5.5: *Autofocus* accuracy over time when $K = 0.2$

batches (i.e., an equal sampling rate is applied to all queries).

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 5.5 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 Section 3.4.4.

5.6 Chapter Summary

In this chapter, we presented a load shedding strategy based on a packet scheduler that applies different sampling rates to different queries according to external information about their accuracy requirements. Although this strategy incurs negligible overhead as compared to the alternative of applying an equal sampling rate to all queries described in Chapter 4, our results points out a significant increase in the average and minimum accuracy of the monitoring system during overload situations.

The main novelty of our load shedding strategy is that it ensures that excess load is shed in a fair manner among the queries, even when dealing with non-cooperative users,

given that a single Nash Equilibrium exists when users provide correct information about the accuracy requirements of the queries.

We implemented our load shedding strategy in the CoMo 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 experimental results also indicate that our load shedding strategy based on a packet 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.

In the next chapter, we extend our system to support custom load shedding mechanisms for those queries that are not robust against packet and flow sampling.

Chapter 6

Custom Load Shedding

In previous chapters, load shedding has been implemented by means of traffic sampling. 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 monitoring applications, in this chapter we present an extension of our load shedding scheme that, besides supporting standard sampling techniques, allows 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 guarantee fairness of service in the presence of non-cooperative or 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 performance and robustness of the monitoring system in front of deliberate traffic anomalies and queries that fail to shed load correctly.

6.1 Proposed Method

The load shedding strategy described in Chapter 5 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 Chapter 5, the system stops first those queries with greater resource demands (i.e., those with greater values of $m_q \times \widehat{d_q}$) 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 [56] 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 [118] or different sampling algorithms [56, 116, 37]) 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 queries to provide custom 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.

The intuition behind this idea is that queries can potentially shed load in a more effective and graceful manner than the monitoring system, because they know their actual implementation, whereas the system treats them as black boxes. 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. Thus, these queries will also be able to gracefully reduce their resource requirements and compete therefore, under fair conditions, for the shared resources with those that use one of the sampling methods provided by the core platform.

Even, in the unlikely case that such a custom load shedding mechanism for a given query does not exist, the query could always keep, during overload conditions, the relevant data (or a summary of it) needed to later compute its output, and postpone the actual computations until more resources are available.

6.1.1 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

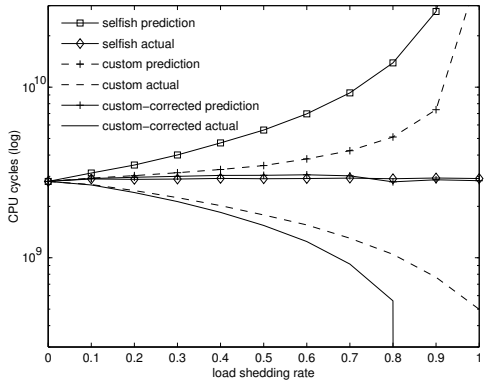


Figure 6.1: Average prediction and CPU usage of a signature-based P2P flow detector query when using different load shedding methods

environments (e.g., [39]) 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 methods 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 (e.g., [98, 5]), we empirically verified that our prediction method inherently penalizes selfish queries by increasing exponentially their predicted cycles ($\widehat{d_q}$) and thus their probability of being disabled.

Figure 6.1 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 query (see Table 2.2, *p2p-detector* query) that does not shed any load, irrespective of the amount of load shedding requested by the core. The figure confirms that $\widehat{d_q}$ 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 $\widehat{d_q}$, as discussed in Section 5.2.

This interesting behavior is consequence of the way we update the history maintained by the Multiple Linear Regression module (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 6.1 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 have no chance to run under overload conditions.

Note that the alternative of recomputing the traffic features used in Chapter 4, 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 (see Section 5.5.4).

6.1.2 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 5.2. Second, it informs the queries about the absolute amount of CPU cycles to be shed, which is simply computed as $r_q \times \widehat{d_q}$.

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 [121, 83]. 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 whether a flow has already been seen, we use a Bloom filter [22].

While this query could use packet or flow sampling instead, Figure 6.2 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.

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 and updating the Bloom filter and scanning the

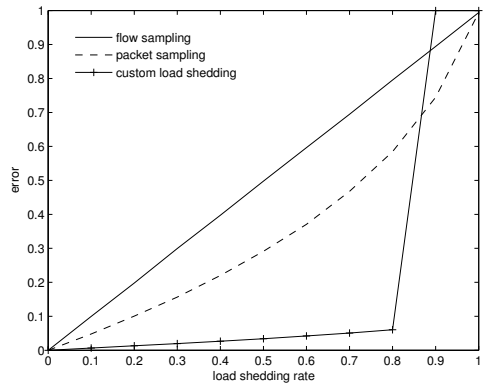


Figure 6.2: Accuracy error of a signature-based P2P flow detector query when using different load shedding methods

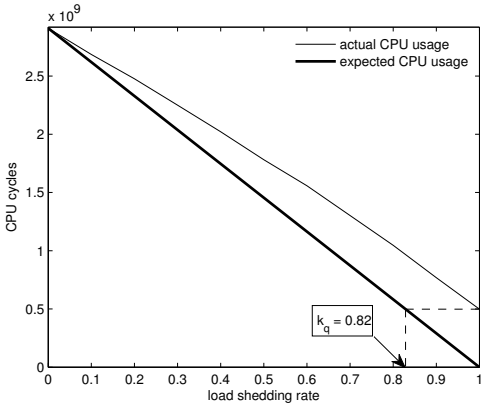


Figure 6.3: Actual versus expected resource consumption of a signature-based P2P flow detector query (before correction)

first packet of each flow.

Figure 6.1 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 6.3 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:

$$r_q = \min\left(1, \frac{r'_q}{k_q}\right) \tag{6.1}$$

where r'_q is the original load shedding rate computed as $r'_q = 1 - p_q$, and k_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 k_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, k_q was obtained empirically as illustrated in Figure 6.3. In particular, k_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 $k_q = 0.82$.

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 $k_q = 1$ and perform the correction by itself.

Figure 6.1 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.

6.1.3 Limitations

The custom load shedding scheme presented in this chapter incurs two different types of limitations. We first discuss some limitations inherent to our load shedding scheme. Next, we present more practical limitations that stem from the current implementation of CoMo.

First, our load shedding scheme assumes a linear relationship between the resource consumption of a query and the load shedding rate. Different relationships, although possible, would indicate that the query does not implement load shedding correctly.

Queries shedding less load than requested are exponentially penalized by our system as explained. On the contrary, inefficiencies in the system arise if queries shed more load than necessary. Both cases result in over-shedding and therefore the integrity of the system is not compromised.

Second, the user must provide the fixed cost (k_q) of those queries using a custom load shedding method. Although, this value could be automatically discovered by the core system from previous observations, this option is not yet available in our current implementation. For example, the system could send a batch with a load shedding rate of zero and then the same batch with a load shedding rate of one, and measure the reduction in the resource consumption of the query to obtain its k_q value.

Finally, the rest of limitations are related to security aspects of the current implementation of CoMo, which is not yet robust against malicious queries that attempt to bring the system down. For example, a query could not return the control to the core system, while processing a batch, entering an endless loop. Similarly, a query could unexpectedly delay its execution to cause the input packet buffers to overflow. Nevertheless, these issues could be easily solved by implementing a timeout clock to preempt the CPU from a query when it significantly exceeds its allocation of CPU cycles.

Other security aspects not fully covered in the current version of CoMo include protection against queries that cause, for example, segmentation faults or other system exceptions, execute system calls bypassing the interfaces provided by the core system, access arbitrary memory regions (including packet buffers and data structures of other queries) or monopolize the system memory.

In order to address these limitations, the next version of CoMo, which is currently under development, will isolate queries in separate sandboxed virtual machines to increase the resilience of the system against such potential threats.

6.2 Validation

In this section, we validate the custom load shedding strategy described in Section 6.1. In particular, we study the performance of a network monitoring system implementing our load shedding strategy compared to our two previous solutions presented in Chapters 4 and 5. We also study the impact of different levels of CPU overload on the accuracy of the traffic queries.

Query	Method	m_q	k_q
application	packet	0.03	-
autofocus	packet	0.51	-
counter	packet	0.02	-
flows	flow	0.03	-
high-watermark	packet	0.10	-
p2p-detector	custom	0.91	0.82
super-sources	flow	0.91	-
top-k	packet	0.50	-
trace	custom	0.95	0.49

Table 6.1: Queries used in the validation

6.2.1 Validation Scenario

In order to validate our load shedding strategy, we implemented it in the CoMo monitoring platform and performed several executions using the UPC-I trace (see Table 2.3). This dataset contains the entire packet payloads, which are needed to study those queries that require the packet contents to operate (e.g., *p2p-detector* query).

Table 6.1 lists the set of nine queries from Table 2.2 used in the validation and presents the load shedding method employed by each query. 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 method already described in Section 6.1, 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 in the presence of overload. The table also presents the values of k_q for these queries, which are computed as described in Section 6.1.

As discussed in Chapter 5, each query can specify at configuration time a minimum sampling rate value (m_q) that will be used by the core system to make scheduling decisions and select the sampling (and load shedding) rates. This value will determine the probability of the query being stopped under overload conditions and the minimum accuracy it can achieve while active.

In a real scenario, m_q should be decided by the end user who submits the query. However, given that the queries in the standard distribution of CoMo do not include this value yet, and in order to provide reasonable values for m_q to validate our proposal, we determined them experimentally using the same methodology described in Section 5.5.2.

Figure 6.4 shows the accuracy curves of three queries (*high-watermark*, *top-k* and a sampling-based version of the *p2p-detector* query) as a function of the sampling rate.

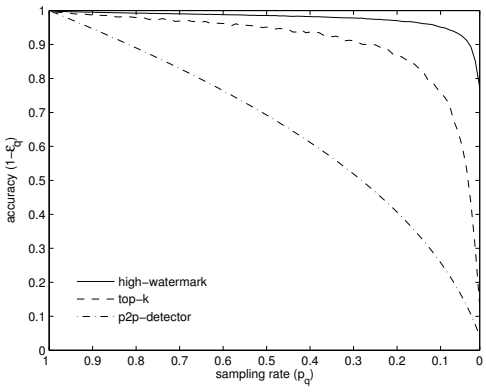


Figure 6.4: Accuracy as a function of the sampling rate (*high-watermark*, *top-k* and *p2p-detector* queries using packet sampling)

Note that while the accuracy of the first two queries drops gradually until a certain point beyond which it degrades drastically, in the case of the sampling-based *p2p-detector* the accuracy decreases almost linearly with the sampling rate, and such a critical point does not exist. This result confirms that this query is not able to gracefully reduce its accuracy using packet sampling, but can do better by using a custom load shedding method, as the one described in Section 6.1.

From the results of these executions, we set the m_q values to the minimum sampling rate that guarantees an average error below or equal to 5% for each query, as discussed in Section 5.5.2. Note 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. Table 6.1 presents the selected values of m_q for those queries that use traffic sampling using the UPC-I trace.

6.2.2 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, 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_srates*) implements the simple load shedding strategy described in Chapter 4, 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 (*mmfs_pkt*) implements

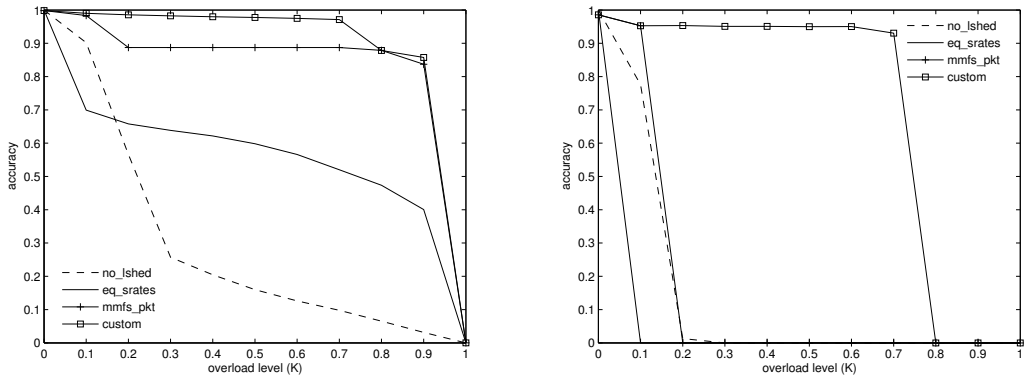


Figure 6.5: Average (left) and minimum (right) accuracy of the system at increasing overload levels

the packet-based strategy presented in Chapter 5. In this strategy, the system tries to satisfy the minimum sampling rate constraints of all queries and eventually distributes the remaining cycles in such a way that the minimum sampling rate among all queries is maximized. In case that the system capacity is not enough to satisfy the minimum requirements of all queries, those with higher minimum demands (i.e., $m_q \times \widehat{d_q}$) are disabled.

Throughout the validation we use the accuracy of the queries as the performance metric to compare the different load shedding alternatives. We define the accuracy of a query $q \in Q$ as a function of its error, as described in Section 5.5.3 (see Figure 5.3).

The left plot in Figure 6.5 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 one 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), as described in Section 5.4. Recall that $K = 0$ 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 (*mmfs_pkt*). This improvement is achieved thanks to the *trace* and *p2p-detector* queries 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 the *p2p-detector* is disabled in the *mmfs_pkt* system. Note that in the *eq_srates* and *mmfs_pkt*

systems, *p2p-detector* and *trace* use packet sampling and their m_q constraints are set to the values shown in Table 6.1.

The right plot in Figure 6.5 shows the minimum accuracy among all queries. The improvement in the minimum accuracy is even 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 than 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.

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 this strategy is not designed to consider the minimum sampling rates, resulting in a large number of violations of the minimum constraints, even when $K = 0.1$.

6.3 Experimental Evaluation

In this section, we evaluate the performance of our custom load shedding strategy in a fully operative network monitoring system with a wide range of queries and traffic scenarios. In particular, we focus the evaluation on studying the robustness of an actual implementation of our load shedding system against traffic anomalies and queries that do not behave properly. In the first case, we inject artificially-generated network attacks that result in a highly variable and difficult to predict workload to stress our prediction system. In the second case, we evaluate the impact of new query arrivals, selfish queries and queries that have implementation bugs on the performance of the monitoring system.

With these experiments we try to verify that, even when delegating the task of shedding excess load onto non-cooperative users, the system is able to achieve robustness and isolation between queries.

In all experiments, the size of the batches is set to $100ms$, the FCBF threshold is 0.6 and the length of the MLR history is configured to $6s$ ($n = 60$), according to the results obtained in Chapter 3. We do not evaluate here other non-predictive alternatives, because the superiority of a predictive approach over other choices (e.g., reactive systems) was already shown in Chapters 4 and 5.

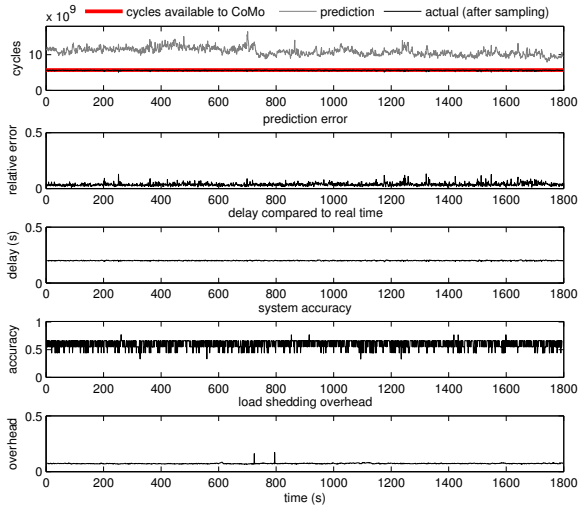


Figure 6.6: Performance of a network monitoring system that does not support custom load shedding and implements the *eq_srates* strategy

6.3.1 Performance under Normal Traffic

We first compare the performance of our custom load shedding method to the system presented in Chapter 4 under normal traffic conditions. Recall that our previous prototype (*eq_srates*) only supports packet and flow sampling and applies an equal sampling rate to all queries in the presence of overload. Instead, the system presented in this chapter (*custom*) supports custom load shedding and applies different sampling rates to different queries according to the packet-based strategy ($mmfs\_pkt$) presented in Section 5.2. In this system, the *trace* and *p2p-detector* queries implement the custom load shedding methods described in Section 6.2, while the rest of the queries use packet or flow sampling.

Figures 6.6 and 6.7 plot 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) for both systems when running the nine queries presented in Table 6.1 on the UPC-I trace. Results are averaged over one second.

The first two plots show the time series of the predicted cycles per second compared to the actual CPU usage of the system 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. That is, in both systems the sum of resource demands for all queries is twice the system

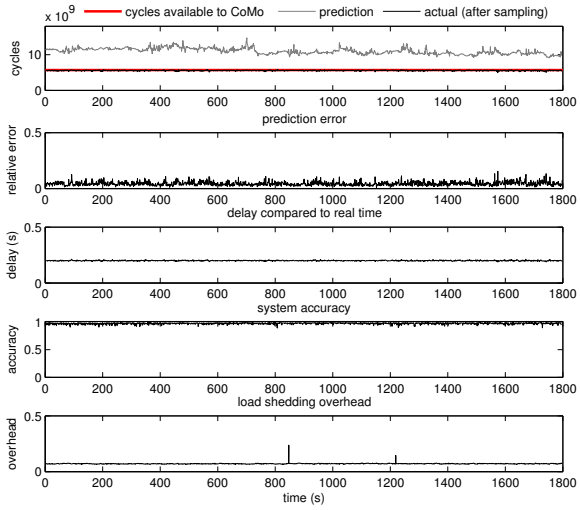


Figure 6.7: Performance of a network monitoring system that supports *custom* load shedding and implements the $mmfs\_pkt$ strategy

capacity.

The plots show that, in both systems, 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 the monitoring system always waits to have an entire batch before passing it to the queries. Recall that batches contain $0.1s$ of traffic. In addition, 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, instead of taking the input traffic from a trace file as in this example.

The figure shows that the delay is almost constant and centered in $0.2s$, indicating that both systems are stable and only require a buffer able to hold two batches, the one that is being processed and the next one. This behavior is well expected given that the CPU is fully utilized and therefore the processing time of a batch is $0.1s$ in average. On the contrary, the delay in an unstable system would increase without bound when using a packet trace. This behavior would be translated into uncontrolled packet losses in a real scenario with a finite buffer.

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. Figure 6.7 shows that the overall accuracy of the *custom* system

is very close to the maximum value of 1, even when the system is highly overloaded. On the contrary, the accuracy of the *eq_srates* system (Figure 6.6) is significantly lower, given that it does not consider the minimum sampling rate constraints when selecting the sampling rates. This results in an allocation that is not optimal according to the accuracy requirements of the queries.

Finally, the bottom plot ('overhead') shows the overhead over time of both load shedding strategies. The overhead is constant (about 7%) in both systems, 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. This result confirms that the load shedding strategy presented in this chapter is able to obtain a significant improvement in the overall system accuracy with similar overhead. The few spikes in the overhead are caused by context switches during the execution of the load shedding procedure, which result in the measurement of additional cycles belonging to the process (or processes) that preempted CoMo.

In the rest of this section, we evaluate the system with anomalous traffic and unexpected or unusual resource usage patterns. In the following examples, we use the same system configuration as in this experiment and omit the figures depicting the load shedding overhead, since we already showed that it is small and constant.

6.3.2 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 monitoring 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, out of every 3 packets 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 6.8 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

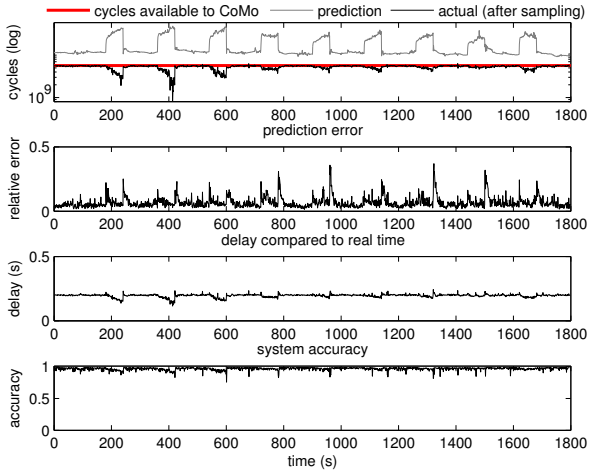


Figure 6.8: Performance of the network monitoring system in the presence of massive DDoS attacks

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. This result indicates that the prediction algorithm is able to quickly detect the anomaly but needs a little longer to forget it.

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 (k_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 6.1. Nevertheless, this situation has no impact on the accuracy of the other queries running on the system, as can be observed in the bottom plot of Figure 6.8.

6.3.3 Effects of Query Arrivals

Another source of instability in our monitoring system is the arrival of new queries. At a given point of time, the resource consumption of the monitoring system can increase significantly due to the arrival of a new query, for which the system does not have any previous observations to predict its resource usage.

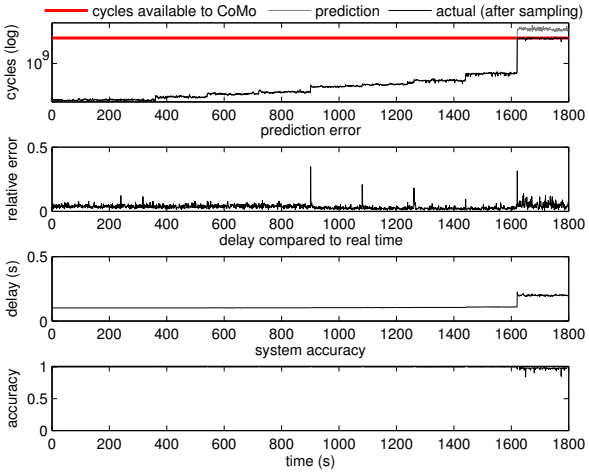


Figure 6.9: Performance of the network monitoring system in front of new query arrivals

Figure 6.9 shows the effects of new query arrivals in our monitoring system. In this experiment, the system is initially running the *trace* query until minute 6, when we start submitting a new query every 3 minutes from those listed in Table 6.1. The system does not start experiencing overload until minute 27, when the *p2p-detector* query is submitted and resource demands become twice the system capacity (note logarithmic scale in the top plot). The figure shows that the system delay is below $0.2s$ until minute 27, and increases up to $0.2s$ when the last query is submitted and the CPU starts to be fully utilized.

Figure 6.9 verifies that the impact of new arrivals on the prediction error and delay is minimal. The spikes in the prediction error during the arrival of some queries are only punctual, which indicates that the system is able to learn the resource patterns of previously unseen queries very quickly.

6.3.4 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 6.1.1 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

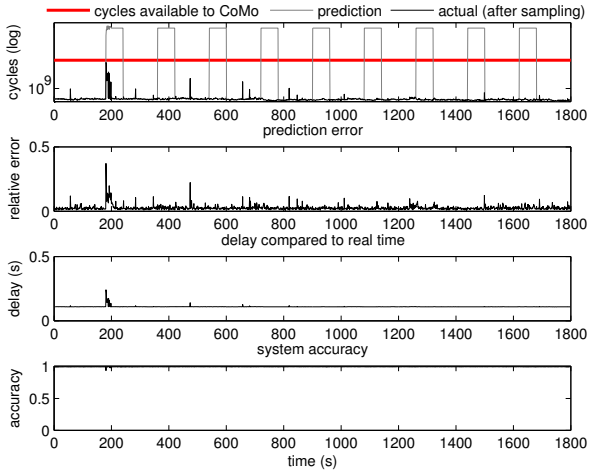


Figure 6.10: Performance of the network monitoring system when receiving a selfish version of the *p2p-detector* query every 3 minutes

of load shedding (r_q) requested by the core platform. We consider a query implementing this custom method as selfish, given that it always tries to obtain a larger share of the system resources than other queries that shed the correct amount of load when requested by the core system.

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 6.1 and it is not experiencing overload. Note that the *p2p-detector* query is the most expensive in Table 6.1, with a cost more than 10 times greater than the rest of the queries.

Figure 6.10 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. The exponential penalization can be easily observed in the predicted cycles depicted in the top plot of Figure 6.10. 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.

6.3.5 Robustness against Buggy Queries

Even if a query is not malicious or selfish in nature, sometimes its resource consumption can increase unexpectedly due to an implementation error. In that case, the network monitoring system must be able to detect the situation and take the necessary actions to make sure that this misbehavior has no impact on the accuracy of other (correct) queries running on the same monitoring system.

In this experiment, we evaluate the impact of queries that have implementation bugs on the overall system accuracy. In particular, we intentionally introduce a bug in the *p2p-detector* query. The bug consists of setting the size of its hash table, which maintains the state of the collected flows, to a very small value. This leads to a large number of collisions that cause a significant increase in the resource consumption of the query, compared to its correct implementation.

As in the previous example, the buggy query is submitted every 3 minutes and withdrawn after 1 minute. In addition, before submitting the new query, the monitoring system is already executing the nine queries listed in Table 6.1, including also a correct version of the *p2p-detector* query. Therefore, unlike in our previous example, the system is already experiencing overload when it receives the buggy query.

Figure 6.11 confirms that the impact of this erroneous query on the accuracy of the other queries running on the monitoring system is minimal. The bottom plot shows the overall system accuracy without including the buggy query. Note that although there are two almost identical *p2p-detector* queries running on the same monitoring system, the system is able to detect the buggy one and stop it in order to guarantee the accuracy of the rest of the queries.

6.4 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 study the online performance of our load shedding scheme in a real scenario where the monitoring system faces continuous and changing overload conditions. We show how, under these adverse conditions, the load shedder is able to gracefully degrade the performance of the monitoring system and minimize the impact of overload on the accuracy of the traffic queries.

In this experiment, we deploy our monitoring system in the UPC scenario presented in Section 2.3 and run online the nine queries described in Table 6.1. We present the results of a 30-minute execution of our monitoring system in an Intel Xeon running at 3

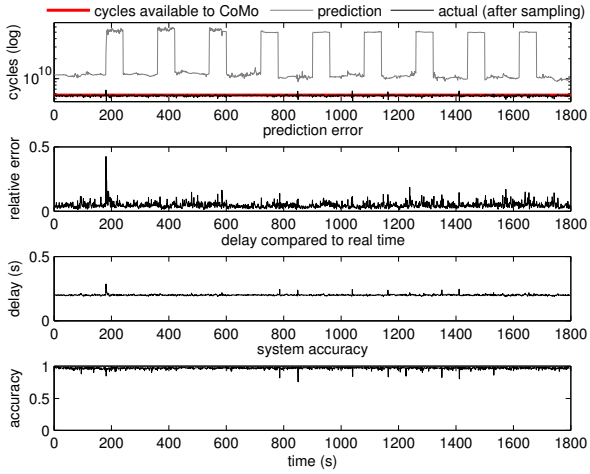


Figure 6.11: Performance of the network monitoring system when receiving a buggy version of the *p2p-detector* query every 3 minutes

GHz. Details of this execution (UPC-II) are available in Table 2.4.

6.4.1 Online Performance

Figure 6.12 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 and number of connections, as can be observed in Figure 6.13. This is an expected behavior given the time of the day at which this experiment was carried out.

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 6.14 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

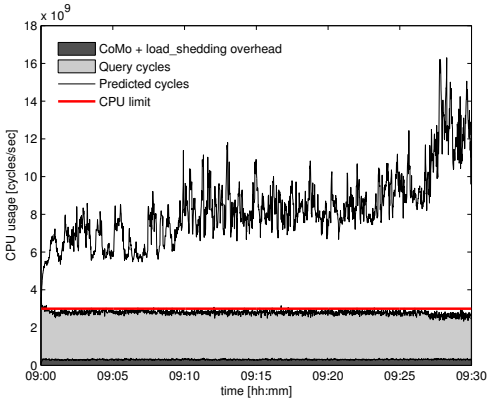


Figure 6.12: CPU usage after load shedding (stacked) and predicted load over time

stopped to avoid uncontrolled packet drops.

As a result, the left plot in Figure 6.13 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 6.13 (left).

The reduction of the overhead in Figure 6.12, compared to the results previously presented in Figure 4.4, is explained by two facts: (*i*) in this experiment we ran the CoMo system on a dual-processor computer and forced the CoMo process responsible of processing the traffic queries to run on a different CPU than the rest of the CoMo processes, using the `sched_setaffinity()` system call. This resulted in a significant reduction of the overhead incurred by the rest of CoMo tasks on the CPU controlled by our load shedding scheme, and (*ii*) we implemented the optimization proposed in Section 5.5.4 to avoid a second full feature extraction on each batch.

Figure 6.14 plots the overall accuracy of the queries over 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 due to buffer overflows. Table 6.2 shows the accuracy broken down by query. We can observe that the average accuracy of most queries is kept above 95% with a small standard deviation, except for the *p2p-detector* query, which is sometimes disabled, especially at the end of the execution, when the traffic conditions are more extreme and its minimum requirements cannot be guaranteed. In the table, we omit the results of the *trace* query, given that no standard procedure exists to measure

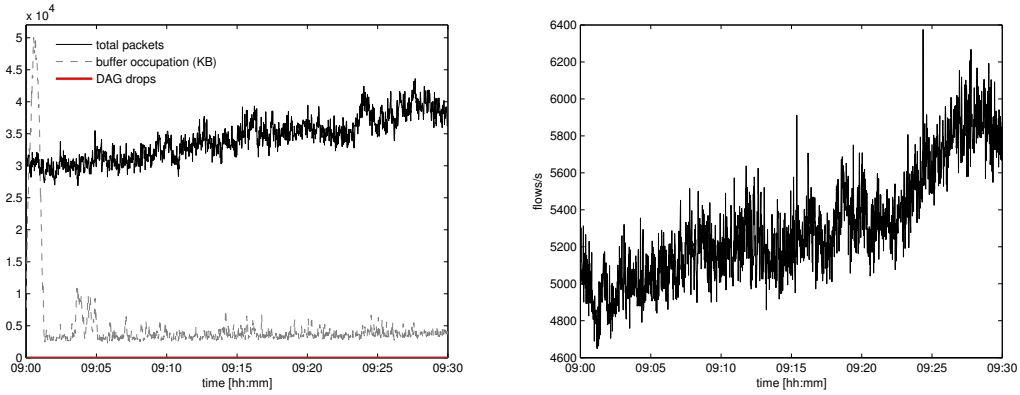


Figure 6.13: Traffic load, buffer occupation and DAG drops (left) and number of new connections (right) over time

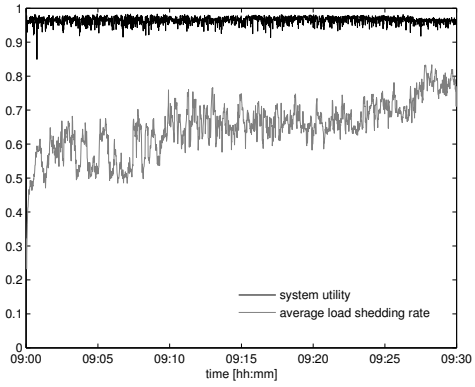


Figure 6.14: Overall system accuracy and average load shedding rate over time

its accuracy, as discussed in Section 2.2. In particular, the load shedding rate of this query was kept during the entire execution below the maximum of $k_q = 0.49$ defined in Table 6.1, with an average load shedding rate $r_q = 0.26$.

6.5 Chapter Summary

In this chapter, we presented an extension of our load shedding scheme 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 than can achieve better accuracy by using other, more appropriate, load shedding methods.

Query	Accuracy
application	1.00 ±0.02
autofocus	0.99 ±0.04
counter	0.99 ±0.01
flows	0.95 ±0.06
high-watermark	0.99 ±0.03
p2p-detector	0.88 ±0.03
super-sources	0.96 ±0.01
top-k	0.96 ±0.06

Table 6.2: Breakdown of the accuracy by query ($mean \pm stdev$)

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 non-cooperative 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 presented in Chapter 3.

This technique completes the design of our load shedding scheme, which now is generic enough to support arbitrary monitoring applications from the point of view of both their resource consumption and the method employed to shed excess load.

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 patterns, selfish users and buggy applications.

Our experimental results show a significant improvement in the average and minimum accuracy of the monitoring system, given that with the extension presented in this chapter those queries that are not robust against sampling can increase their probability of being executed in the presence of overload by implementing a custom load shedding method. This results in increased fairness of service, since now all applications can compete under fair conditions for the shared system resources, regardless of the load shedding method being employed.

Chapter 7

Related Work

The design of mechanisms to handle overload situations is a classical problem in any real-time system design and several previous works have proposed solutions to the problem. Due to the vast amount of existing literature in this field, we limit our review to those techniques of overload management that we consider most relevant to our work. In this chapter, we classify existing solutions by different system areas and cover in greater detail previous load shedding designs, most of them proposed in the context of data stream management systems.

7.1 Network Monitoring Systems

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 [44, 48].

In order to minimize this impact, critical monitoring systems often integrate specialized hardware (e.g., DAG cards [53], SCAMPI adapters [76], router add-ons [81], network processors [135, 59]) or make use of ad-hoc configurations (e.g., [120]) to avoid the inherent hardware limitations of the PC-based architecture for network monitoring [70, 4]. Although these solutions have demonstrated their effectiveness in some scenarios, they present scalability issues that make them viable only as a short-term solution.

Other monitoring systems have opted to significantly reduce their online functionalities in order to operate in high-speed environments (e.g., OC3MON [6], Sprint IPMON [61]). In this case, complex traffic analysis computations are postponed to an offline stage, which significantly limits their usefulness and possible applications.

Recently, several research works have proposed solutions that offer a more robust and predictable behavior in the presence of overload. Traffic sampling is arguably the most widely accepted technique to cope with the high resource requirements imposed on network monitoring systems, and is currently under standardization by the PSAMP working group at the IETF [72]. Duffield reviews in [43] the extensive literature existing in this field. The research on traffic sampling can be divided into two main groups. On the one hand, several works have focused on estimating general traffic properties (e.g., number and average length of flows [46], flow size distributions [47, 66]), on identifying traffic of particular interest to network operators (e.g., elephant flows [99], top-k flows [12, 38]) and on evaluating the impact of sampling on some applications (e.g., flow accounting [140], anomaly detection [95, 94, 24]) from sampled measurements collected using Sampled NetFlow [34] and/or other methods. On the other hand, other research works have proposed efficient sampling techniques to improve the accuracy of certain applications. Thus, these proposals usually require changes on the collection devices. For example, some techniques have concentrated on the detection of large flows in high-speed networks, which account for most Internet traffic [58], (e.g., *sample-and-hold* and *multistage filters* [56], *shared-state sampling* [116]), while others have proposed sampling algorithms that can operate before (e.g., *step sample-and-hold* [37]) or after aggregation (e.g., *threshold sampling* [49, 45], *priority sampling* [48, 50]) to estimate the size of an arbitrary set of flows. Duffield and Grossglauer [51] have also proposed a hash-based technique, called *trajectory sampling*, that allows different monitors to sample the same set of flows.

Our work is complementary to the existing research on traffic sampling, since most of the sampling techniques described above could be directly implemented in our monitoring system either as custom load shedding methods or in the core monitoring platform. For example, the hash-based flow sampling technique supported in our system is based on the trajectory sampling method proposed in [51].

Several network monitoring systems use data reduction techniques, such as packet filtering, traffic sampling, flow aggregation or a combination of them to handle overload situations. The most representative example is arguably Cisco's NetFlow [35]. NetFlow is considered the state-of-the-art technology for network monitoring. It is a widely deployed, general-purpose solution supported in most of today's routers. It extracts pre-defined per-flow information (depending on the version of NetFlow) and periodically reports to a central collection server. In order to handle the large volumes of data exported and to reduce the load on the router, Sampled NetFlow [34] resorts to packet sampling. The sampling rate must be defined at configuration time, and to handle

unexpected traffic scenarios network operators tend to set it to a low "safe" value (e.g., 1/100 or 1/1000 packets). NetFlow input filters [33] also permits to configure different sampling rates to different groups of flows defined by the network administrator.

Adaptive NetFlow [54] allows routers to dynamically tune the sampling rate to the memory consumption in order to maximize the accuracy given a specific incoming traffic mix. Flow Slices [87] uses a combination of packet sampling, sample-and-hold and a variant of threshold sampling to independently control the CPU, memory and reporting bandwidth usage of routers. While the sampling parameters used to control the memory and bandwidth usage are dynamically adapted to runtime conditions, the sampling rate used to control the CPU usage is statically set at configuration time to a conservative value as in Sampled NetFlow. ProgME [138] uses aggregation instead of sampling to control the memory consumption. ProgME is a programmable flow aggregation engine based on the novel concept of flowset. A flowset is an arbitrary set of flows that is defined using a composition language based on set algebra. The main advantage of ProgME is that the memory consumption depends only on the number of flowsets the user is interested in and not on the observed traffic mix, which can significantly reduce the memory consumption for certain types of applications. Keys et al. [85] extend the approach used in NetFlow by extracting and exporting a set of 12 traffic summaries that allow the system to answer a fixed number of common questions asked by network operators. The summaries focus on the detection of heavy hitters. The system deals with extreme traffic conditions and anomalous traffic patterns by gracefully degrading the accuracy of the summaries using adaptive sample-and-hold and memory-efficient counting algorithms.

Several works have also addressed similar problems in the intrusion detection space. For example, Dreger et al. discuss in [42] several modifications to Bro [108], such as dynamically selecting the restrictiveness of the packet filters, to allow Bro to operate in high-speed environments. Gonzalez et al. [63] also propose the inclusion of a secondary path into Bro that implements sampling and filtering to reduce the cost of those analysis tasks that do not require stream reassembly and stateful inspection. However, the capture ratio in the secondary path cannot be adapted to the resource usage.

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 online [35, 34, 33, 54, 87], or (*ii*) they are limited to a pre-defined set of traffic reports or analyses [138, 85, 42, 63].

Our load shedding scheme differs from these previous 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, [85] divides the memory among the various components of the system in equal parts and assumes that their average cost per packet is a constant share of the total CPU usage. This assumption is a direct consequence of their careful design, but does not hold for any arbitrary monitoring application. Conversely, our system is able to deal with arbitrary, non-cooperative monitoring applications, with different (and variable) cost per packet and accuracy requirements.

Similar open network monitoring infrastructures to CoMo have opted instead for more strict and inflexible resource management policies. For example, FLAME [5] is a modular network monitoring platform that allows users to extend the system with arbitrary measurement modules. The main feature of FLAME is that it provides protection mechanisms to ensure that user-defined modules can be executed safely on the monitoring infrastructure. FLAME modules are written in Cyclone [78], a safe dialect of C, and processed by a trusted compiler. In order to deal with competing modules, FLAME bounds their execution time using a cycle counter. A similar solution is adopted by Scriptroute [123], an open distributed platform for active network monitoring. The main difference between Scriptroute and other infrastructures in the active monitoring space, such as NIMI [110], Surveyor [82] or AMP [97], is that it allows users to conduct arbitrary active measurements in the monitoring infrastructure. Measurement tasks are provided as Ruby scripts that are executed on-demand on the Scriptroute servers. In order to allow untrusted users to execute arbitrary code, each Scriptroute script runs on an independent resource-limited sandbox, with no local storage access and limited execution time, memory and bandwidth. Scriptroute servers also implement several checks to avoid the use of the measurement infrastructure to generate network attacks. Scripts that do not meet these policies are aborted.

The resource management techniques proposed in this thesis are significantly different and more complex than those adopted by these systems. The inflexible solution of simply limiting the amount of resources allocated to each application 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.

Recently, some research proposals have tried to bring the flexibility of declarative queries to network monitoring, as an alternative to the complexity of the procedural languages commonly used by network operators and analysts. The most well-known

example is probably the Gigascope project [40]. Gigascope is a proprietary stream database for network monitoring developed at AT&T. In Gigascope, declarative queries are written in GSQL, a restricted subset of SQL extended to operate with data streams. Gigascope has been explicitly designed to operate in high-speed networks. Thus, GSQL queries are first translated into C or C++ code and then broken down into two components. The first component is a low-level subquery that performs a first aggregation of the network traffic, which in some cases can run directly on the hardware NIC. The second component performs more complex processing tasks. This design resembles the architecture of CoMo divided in the *capture* and *export* processes. In Gigascope, users can write new functions and operators in order to implement those tasks that can no be easily expressed in GSQL. This makes it difficult to implement in Gigascope the load shedding schemes used by similar stream-based databases presented in the next section.

Finally, our load shedding scheme is based on extracting features from the traffic streams with deterministic worst case time bounds. Several solutions have been proposed in the literature to this end. For example, counting the number of distinct items in a stream has been addressed in the past by both the database [60, 134, 11, 52] and networking [57] communities. In this work, we implemented the multi-resolution bitmap algorithms for counting flows proposed in [57].

7.2 Data Stream Management Systems

Data management systems that deal with live input streams are becoming increasingly common. These systems are known as data stream management systems (DSMS). Such systems present several particularities, such as their push-based nature and the support for continuous queries, that make the research proposals in the stream database literature very relevant to our work. The survey papers [9, 62] present a good overview of the work in the field of DSMS.

The main limitation of most load shedding approaches in this area is that proposed solutions require the use of declarative query languages with a restricted set of operators, for which their cost and selectivity are assumed to be known. On the contrary, in our context we have no explicit knowledge of the queries and therefore we cannot make any assumption on their cost or selectivity to know when it is the right time to drop records or to decide how much load to shed. This significantly limits the flexibility of load shedding proposals in the field of DSMS to be used for network monitoring purposes, where there is a clear need for supporting arbitrary traffic queries and complex monitoring applications that cannot be easily expressed using standard declarative languages [69].

Although some solutions in the scope of DSMS assume that accurate estimates of the average values of these parameters can be obtained at runtime from historical values, it is also assumed that they change quite infrequently over time. In contrast, in network monitoring, queries can consist of arbitrary code with very different and highly-variable cost per packet due to both implementation issues and the variability of the incoming traffic. For example, a given query can incur different processing overheads depending on whether a specific kind of traffic is found in the input streams (e.g., IDS). On the other hand, the average processing cost per packet of a query that depends on a specific traffic feature (e.g., number of unique flows) can vary significantly from one batch to another, given that different batches can have very different values for these features.

Next, we review the most relevant load shedding proposals in the context of four well-known data stream management systems, namely Aurora, STREAM, TelegraphCQ and Borealis, and discuss their main differences with the load shedding scheme proposed in this thesis. Finally, we also review a control-based load shedding approach that, unlike previous methods, it is explicitly designed to operate with more variable input data rates and processing costs.

7.2.1 Aurora

Aurora is a Data Stream Management System developed at Brandeis University, Brown University and MIT [25, 2], which supports the concurrent execution of multiple continuous queries on several input streams.

In Aurora, queries are provided as a workflow diagram using a graphical interface. Each query consists of a directed acyclic graph built out of a set of eight basic operators. A continuous query accepts push-based inputs (i.e., continuous sequences of tuples) from an arbitrary number of sources and produces a single output. Each query can express its QoS expectations using three different QoS graphs (i.e., utility functions) that describe the relationship between various characteristics of the output and its utility.

The three QoS graphs supported in Aurora are: (*i*) a latency graph, which indicates how the utility of a query drops when an answer is delayed, (*ii*) a loss-tolerance graph that shows how the utility of a query decreases as a function of the rate of dropped tuples, and (*iii*) a value-based graph, which provides information about the importance of the possible values in the output space of a query. In order to simplify the resource management problem, Aurora assumes that (*i*) and (*ii*) have concave shapes. In particular, the loss-tolerance graph of a query is very similar to our concept of a minimum sampling rate.

Aurora concentrates on the processor as the limited resource and all resource management decisions are driven by the QoS functions. In Aurora, the resource management decisions are made independently by two different components of its architecture: the scheduler and the load shedder.

The Aurora scheduler [26] is in charge of the processor allocation during underload conditions and relies on the existence of an independent load shedder to get rid of excess load during overload situations. The scheduler keeps track of the latency of the tuples in the operator queues and schedules the execution of those operators that provide the highest aggregate QoS delivered to queries, according to the latency-based QoS functions. The other two QoS graphs are only used by the load shedder.

In [128], Tatbul et al. present in detail the design of the Aurora load shedder. Load shedding in Aurora is based on the insertion of drop operators into query plans during overload conditions. To detect overload situations, Aurora uses explicit knowledge of the cost and selectivity of each query operator. In the presence of overload, the location of drop operators is selected in such a way that the overall utility of the system is maximized. This renders this solution inviable in our monitoring system, given that when dealing with non-cooperative (selfish) queries the strategy of maximizing an aggregate performance metric can be extremely unfair and lead to severe starvation. Thus, this solution is only suitable in scenarios where the system administrator has complete control over the utility functions, as in the case of Aurora. On the contrary, our load shedding scheme tries to satisfy the minimum accuracy requirements of all queries, while maximizing their minimum sampling rate.

A key aspect of the load shedding approach of Aurora is that, with explicit knowledge of the cost and selectivity of each query operator and the relative proportions of the input rates, the placement of drop operators can be pre-computed offline. This way, the run-time overhead of the load shedding scheme is significantly reduced. To this end, Aurora constructs a static table called Load Shedding Road Map (LSRM) that contains the possible drop locations sorted (in ascending order) by their loss/gain ratio. The loss/gain ratio is a metric that allows Aurora to quantify the loss in accuracy (according to the loss-tolerance QoS graph) compared to the gain in cycles (i.e., the cycles recovered) for each particular drop location and percentage of drop. Since the overload problem in Aurora is a variant of the well-known Fractional Knapsack problem, in order to minimize the loss of utility in the presence of overload, the load shedder can simply follow the greedy approach of applying first those drops with smaller loss/gain ratio in the LSRM.

Implicitly, Aurora assumes that the set of queries is static, and that the cost and selectivity of each query operator and the relative input data rates do not change over

time, since for every change the LSRM needs to be recomputed. In contrast, we support arbitrary queries with different and highly variable cost per packet (i.e., per tuple) and we expect the set of queries running on the monitoring system to change frequently over time. Thus, we cannot make any assumption about the queries nor their input traffic in order to make load shedding decisions.

In Aurora, the value-based QoS graphs are used for semantic load shedding, which consists of dropping tuples based on the importance of their content instead of doing it in a randomized fashion. In this case, it is assumed that the histogram of the output values of a query is available for each interval in the value-based QoS graph. With this information, Aurora estimates a loss-tolerance graph and constructs an approximate filter predicate in order to apply the same method used for random load shedding described above. In our system, it is also possible to implement semantic-based load shedding methods by using the custom load shedding feature presented in Chapter 6.

7.2.2 STREAM

STREAM [100] is a general-purpose stream processing system built at Standford University that supports multiple continuous queries expressed in CQL [7], a declarative stream query language based on SQL. STREAM also provides a graphical interface that allows users to inspect and adjust the system at runtime.

As in the case of Aurora, the load shedding scheme in STREAM [10] is based on dropping tuples by dynamically inserting drop operators into query plans. However, [10] focuses on minimizing the impact of load shedding on the accuracy of the queries, while in Aurora the objective is to maximize the total system utility based on the QoS graphs.

In STREAM, overload situations are detected when the rate at which tuples arrive is greater than the rate at which tuples can be processed. This however requires explicit knowledge of the processing cost per tuple and the selectivity of each query operator, which are estimated from historical values.

In the presence of overload, excess load is shed in such a way that the maximum relative error across queries is minimized. That is, the sampling rate to be applied to each query is selected so that the relative accuracy is the same for all queries. In order to relate sampling rate and accuracy, [10] limits its solution to a single class of queries: sliding window aggregates over data streams (e.g., sum and count). In this case, it is possible to derive probabilistic bounds for the relative error of a query as a function of the sampling rate, which is not possible for more complex queries. However, additional statistics about the average and standard deviation of the values of the input tuples

being aggregated need to be available.

With this information, the system computes the effective sampling rates that guarantee an equal (and minimum) relative error across all queries. Once the sampling rates of each query are available, the system has to find the proper locations of drop operators in the query plans, taking into account that several operators can be shared among multiple queries. For this purpose, [10] uses a simple algorithm that guarantees an optimum placement in terms of processing time, which basically consists of inserting the drop operators as early as possible into the query plans.

The idea of minimizing the maximum relative error across queries is very similar in spirit to the strategy of maximizing the minimum sampling rate used by our load shedding scheme. However, our load shedding scheme is not limited to a single class of queries, but instead has to deal with arbitrary monitoring applications, for which it is impossible to know their relation between accuracy and sampling rate. For this reason, we introduced the notion of a minimum sampling rate, which allows users to specify the minimum sampling rate at which their queries can obtain a minimum acceptable accuracy. With this information, our load shedding scheme tries to guarantee the minimum accuracy requirements of all queries given the available resources, and distributes the remaining cycles in such a way that the minimum sampling rate is maximized. Assuming equal accuracy requirements for all queries, this strategy would be very similar to the idea of minimizing the maximum relative error. However, our strategy is more general in the sense that, apart from supporting arbitrary queries, allows them to specify different accuracy requirements.

Finally, it is also interesting to note that some of the resource management techniques used in STREAM focus on the memory as the primary limited resource. For example, Babcock et al. present in [8] the design of an operator scheduler that minimizes the memory used in the operator input queues subject to a maximum latency constraint, while [124] proposes a load shedding strategy for sliding-window joins that considers the memory to maintain the join state as the limited resource. The design of load shedding techniques for arbitrary network monitoring applications that can consider multiple system resources, such as memory or disk bandwidth, constitutes an important part of our future work, as we discuss in Chapter 8.

7.2.3 TelegraphCQ

TelegraphCQ [32] is a stream query processor developed at UC Berkeley. The software architecture of TelegraphCQ is based on the relational DBMS PostgreSQL, but it

includes large extensions to support streaming data and continuous queries.

In order to deal with overload situations, TelegraphCQ implements an architecture called Data Triage [117, 118]. The main difference between Data Triage and other load shedding schemes, such as those implemented in Aurora [128] and STREAM [10], is that it applies approximate query processing techniques, instead of dropping tuples, in order to provide approximate and delay-bounded answers in presence of overload.

In TelegraphCQ, queries can specify a delay constraint to bind the latency between data arrivals and the generation of the query results. The main mission of the Data Triage architecture is to ensure that the TelegraphCQ query processor meets all the delay constraints.

The architecture of Data Triage is divided in two data paths: a primary data path that performs normal query processing and a secondary data path that uses approximation. When the system detects that there is no enough time to perform full query processing on every tuple, Data Triage sends selected tuples through the secondary path in order to keep the delay within the bounds defined by end users.

In the secondary path, the tuples are first summarized and provisionally stored into a summary data structure. At the end of each time window, the summarized data are processed by a shadow query that uses approximate processing techniques. Finally, the results of the shadow query as well as those of the main query are presented to the user, who can combine them to obtain an approximate version of the query answer.

In order to detect when the delay constraints cannot be satisfied and tuples need to be send through the secondary data path, Data Triage continuously monitors its input queues and uses explicit information about the cost per tuple of the main query, the cost of adding a tuple in the summary data structure and the cost of the shadow query.

The Data Triage framework supports several well-known summarization techniques, such as multidimensional histograms, wavelet-based histograms or random sampling. Whereas the main query can be provisioned for typical data rates, summarization algorithms must be tuned to handle worst case scenarios.

An important feature of TelegraphCQ is that it has built-in support for network monitoring [118]. We argue however that, although the flexibility of declarative queries is very adequate for a wide range of simple network monitoring tasks, more complex applications cannot be easily expressed using common declarative languages (e.g., automated worm fingerprinting [86], application identification [84], anomaly detection [89, 90]) and they require the use of imperative programming languages.

In addition, some preliminary studies have reported very poor performance when using DSMS for network monitoring purposes [112, 122], which hinders their deployment

in the high-speed networks traditionally targeted by the network monitoring community.

Apart from supporting arbitrary monitoring applications, the main difference between our load shedding scheme and the Data Triage approach is that, while [118] assumes that the primary data path can be provisioned to handle the 90th or 95th percentile of the data rates, we focus on more extreme scenarios where the system is continuously overloaded.

Regarding the summarization techniques, our current implementation uses packet and flow sampling, but other methods, such as those supported in Data Triage, can be provided using the custom load shedding approach described in Chapter 6.

7.2.4 Borealis

Recently, several DSMS designs have turned into distributed systems in order to address the intrinsic scalability issues of stream processing systems. Although the most natural way of addressing the resource management problem in a distributed context is by means of load distribution and load balancing techniques (e.g., [136]), some recent works have also proposed load shedding solutions in the context of distributed stream processing systems. Of particular interest is the case of FIT [127], a load shedding scheme for the Borealis system. Borealis [1] is a distributed DSMS developed at Brandeis University, Brown University and MIT, which is based on the query processing engine of Aurora.

In Borealis, queries consist of a chain of query operators that can be located at different processing nodes. The main problem in this scenario is that traditional load shedding schemes designed for centralized DSMS are not appropriate in a distributed setting, since local load shedding decisions made in a particular node can have a significant impact on the descendant nodes in the query path, as illustrated in [127]. Therefore, a load shedding scheme for such a system must consider the requirements of the rest of the nodes in the query path in the decision-making process.

Tatbul et al. [127] model the load shedding problem as a linear optimization problem, with the objective of maximizing the overall weighted throughput of the system given the available resources. Although bottlenecks in a distributed DSMS can also be due to bandwidth limitations, [127] focuses only on the CPU as the limited resource.

The proposed solution consists of generating load shedding plans in advance using explicit information about the cost and selectivity of each query operator. In order to compute the load shedding plans, [127] presents two different approaches: a centralized and a distributed solution. In the centralized approach, a central node receives statistics about the operator costs and selectivities of all nodes. For multiple combinations

of unfeasible input data rates, the coordinator node computes offline the optimal load shedding plans that maximize the total system throughput using a solver tool. In order to reduce the number of possible combinations, several optimizations are proposed, which allow the system to reduce the overhead of computing the optimal solutions, while introducing a given error in the results. Finally, the coordinator node continuously monitors the input data rates and estimates the load of the system at runtime. When an overload situation is detected, the best load shedding plan from those computed offline is selected according to the current input data rates.

In the distributed solution, the information about the load constraints of the nodes is aggregated and sent to parent nodes using a special-purpose data structure called FIT (Feasible Input Table). The FIT data structure of a node basically contains an entry for each combination of feasible input data rates together with its associated score (i.e., weighted throughput). In order to reduce the number of entries, similar techniques to those proposed for the centralized case are used. When a node receives FITs from its children, it merges them with its own FIT and propagates it to its parents, until the root node is reached. With this information, parent nodes can make load shedding decisions on behalf of child nodes in the query path. At runtime, each node monitors its input data rates and, if an overload situation is detected, the FIT entry with the highest score (i.e., weighted throughput) given the incoming data rates is selected, and the input data rates are scaled by inserting drop operators in order to match those in the FIT entry.

Apart from requiring explicit information about the query operators, the main drawback of this solution is the time required to compute the (near) optimal solution in the centralized case, and the volume of data to be exchanged among nodes in the distributed case, which must be done after any change in the query network. This renders this solution inappropriate for dynamic scenarios, such as those described in this thesis, with highly variable processing costs and number of queries.

Although our load shedding scheme is currently limited to centralized network monitoring systems, we are very interested in studying the feasibility of using similar techniques to the one proposed in this thesis in order to address the resource management problem in distributed and highly dynamic scenarios, with limitations on both CPU and bandwidth among nodes.

7.2.5 Control-based Load Shedding

Load shedding designs in Aurora [128] and STREAM [10] are based on an open-loop model that ignores the current status of the system (e.g., occupation of the queues)

when making load shedding decisions, which can result in over- or under-shedding under certain fluctuations in the input data rates, as shown in [131]. To solve this limitation, Tu et al. [131] present the design of a load shedding solution based on feedback control techniques that is able to operate with unpredictable and highly variable data rates.

The load shedding problem in [131] is modeled using control theory, with the aim of satisfying a given target delay with minimum data loss. They present the design of a closed-loop control system that can effectively control the average tuple delay in DSMS using feedback information about their current output delay. The variations in the incoming data rates and processing costs are considered as model disturbances. With this information, the controller can dynamically adjust the amount of load shedding to keep the average tuple delay within the target value given by the system administrator. In order to decide the drop locations in the query network, [131] relies on an existing load shedder in the DSMS or, alternatively, it randomly drops tuples from the queues.

However, the design of the control system requires a model of the system under control (i.e., the DSMS), which describes the response of the system to changes in the inputs. In particular, [131] experimentally derives a model for the Borealis DSMS [1], though it could be easily adapted to other DSMS. The main problem however is that this model relies on a constant parameter that denotes the cost per tuple of the DSMS, which somewhat limits its applicability in scenarios with highly variable processing costs. In addition, this parameter refers to the average cost of the entire query network and therefore should be recomputed on the arrival of any new query.

Although [131] shares similar motivation with our work, the proposed control-based methodology does not solve the problem of highly variable (and unknown) processing costs, typically found in arbitrary monitoring applications. Our load shedding scheme, instead of using a pre-defined model of the entire system, builds an online prediction model of the processing cost of each query based on its relation with a set of features of the input streams. In addition, the advantage of having separate models for each query is that the load shedding scheme can make different load shedding decisions for different queries. On the other hand, [131] addresses only the problem of *when* and *how much* to shed load, while in this thesis we focus also on the questions of *where* and *how* to shed it.

7.3 Other Real-Time Systems

In this section, we review the design of two interesting load shedding solutions proposed in other real-time system areas that we consider of special interest to the problems

addressed in this thesis.

7.3.1 SEDA

In the Internet services space, SEDA [133] proposes an event-driven architecture to develop highly concurrent server applications, such as HTTP servers. In SEDA, applications are built as networks of stages interconnected by queues of events. Each stage implements an event handler that processes a batch of events from its input queue and enqueues resulting events on the input queues of other stages. One of the advantages of this architecture is that it isolates stages from each other and allows them to apply different resource management techniques.

SEDA implements two different resource management approaches as an alternative to standard overprovisioning techniques, such as service replication. On the one hand, the SEDA framework provides generic resource controllers that dynamically adapt some parameters of each stage based on its observed performance, such as the number of threads associated to the stage or the batching factor applied to its input queue [133]. These generic methods are somehow analogous to the sampling techniques provided by our core monitoring platform.

On the other hand, SEDA allows stages to implement their own overload management mechanisms (e.g., load shedding) by giving them direct access to their input queues [132]. This allows stages to make informed resource management decisions tailored for each particular service, which could not otherwise be made by the SEDA framework or the operating system. This solution (and the motivation behind it) is very similar to our custom load shedding approach. However, in SEDA it is implicitly assumed that a server executes a single service (i.e., application) that consists of multiple concurrent stages. Therefore, strict enforcement polices are not needed, since stages always belong to the same application. In contrast, our monitoring system is open to several competing monitoring applications and, thus, the system must ensure that each application sheds the correct amount of load in order to guarantee fairness of service.

SEDA stages implement a reactive load shedding scheme that usually consists of applying admission control to the input queues (e.g., dropping incoming requests) when an overload situation is detected (e.g., the response time of the system exceeds a given threshold). Other solutions are also possible, such as applying different polices to different classes of requests or degrading the quality of the delivered service in the presence of overload. In this thesis, we presented instead a predictive approach that anticipates overload situations. We showed that in network monitoring 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.

7.3.2 VuSystem

Compton and Tennenhouse propose in [39] an interesting collaborative load shedding approach for media-based applications that resembles our idea of custom load shedding. In particular, [39] focuses on the problem of how to dynamically adapt, based on user's priorities, the resource consumption of multiple concurrent video applications that run on a general-purpose operating system without explicit real-time support.

They suggest that it is preferable to let video applications to shed excess load by themselves than relying on the operating system for this task. The main rationale behind this idea is that applications can always shed load in a much more graceful manner than the operating system, which does not have any knowledge of the task carried out by the application. For example, in the presence of overload, a video application could reduce its resolution, window size, color depth or frame rate, while the operating system can only reduce the amount of resources allocated to the application. This is exactly the same intuition behind our custom load shedding approach presented in Chapter 6.

Nevertheless, the load shedding solution proposed in [39] requires applications to behave in a collaborative fashion and to use information about the priorities and number of other applications running on the same system in order to decide *when* and *how much* load to shed. In addition, [39] does not provide a concrete enforcement policy and relies on a social welfare assumption that is not met in our scenario. In order to avoid this issue, [39] argues that applications that do not shed load properly will not be successful in the market, but this solution fails to address the short-term problem.

In this thesis, we proposed instead a simple and lightweight enforcement policy that makes sure that users implement their custom load shedding methods correctly, even in a non-cooperative environment with untrusted applications. In our load shedding scheme, the core system is in charge of providing the application with the information about *when* and *how much* load to shed. This way, the application can shed excess load without knowledge of the current system status nor the rest of applications running on the system.

Chapter 8

Conclusions

This thesis has demonstrated effective methods for handling overload situations that allow network monitoring systems to sustain the rapidly increasing link speeds, data rates, number of users and complexity of traffic analysis tasks, and to achieve robustness against traffic anomalies and network attacks.

We presented the challenges involved in the management of overload situations in network monitoring systems and discussed why this is an interesting and difficult problem. We also argued the increasing interest of network operators and researchers for open network monitoring infrastructures that allow multiple users to execute arbitrary monitoring applications on the traffic streams, which further complicates the resource management problem.

With this basic motivation, we presented the design, implementation and evaluation of a predictive load shedding scheme that can anticipate overload situations and minimize their impact on the accuracy of the monitoring applications by sampling the input traffic streams. The main intuition behind our prediction method comes from the empirical observation that, when dealing with arbitrary applications, for which resource demands are unknown in advance, their cost can be modeled with a set of simple features of the input traffic.

Although predictive schemes are more complex to implement and incur larger overheads, the results presented in this thesis show that it is crucial to anticipate overload situations, as compared to the alternative strategy of reacting to them. Anticipating overload situations allows avoiding uncontrolled packet losses in the system buffers, which can occur at very small time scales and cause a severe and unpredictable impact on the accuracy of the monitoring applications. Our experimental results in a research ISP network showed a reduction of more than one order of magnitude in the error of the

monitoring applications when using our predictive approach.

Another novel feature of the proposed load shedding scheme is that it is suitable for open network monitoring systems, given that it assures that a single Nash Equilibrium exists when applications do not demand more resources than those strictly needed to obtain results with acceptable accuracy. This way, the monitoring system can sustain high levels of fairness and accuracy during overload situations, even when dealing with non-cooperative and competing applications. In particular, the results presented in this thesis showed that our monitoring system was able to maintain an overall accuracy greater than 95% under extreme overload conditions in long-lived executions with several concurrent monitoring applications.

Finally, we discussed that not all network monitoring applications are robust against traffic sampling and, therefore, a load shedding scheme based only on sampling techniques can be unfair to these applications. For this reason, we extended our scheme to support custom load shedding methods provided by end users. A key aspect of our solution is that, without further modifications to the core of our load shedding scheme, it inherently penalizes those monitoring applications that fail to shed excess load in order to preserve the robustness of the monitoring system in the presence of non-cooperative applications.

In summary, the main contribution of this thesis is a generic load shedding framework for network monitoring systems with the following novel features:

- It is able to operate without explicit knowledge of the cost and implementation details of network monitoring applications.
- It is based on an online prediction model that can anticipate overload situations and avoid uncontrolled packet losses in order to minimize their impact on the accuracy of monitoring applications.
- It does not rely on a specific model for the incoming traffic and can dynamically adapt to highly variable traffic conditions and changing resource consumption patterns.
- It is lightweight enough to operate in real-time in high-speed networks since it is based on efficient feature extraction algorithms with a deterministic worst case computational cost.
- It only requires minimal information about the accuracy requirements of the monitoring applications (i.e., the minimum sampling rate) to guide the load shedding procedure, avoiding the use of complex utility functions employed by other systems.

- It is able to operate in an open environment with non-cooperative (competing) applications, given that it guarantees that a single Nash Equilibrium exists when users provides correct information about their accuracy requirements.

- It is based on a packet scheduler that ensures fair access to the packet stream, instead of the classical policy of assuring fair access to the CPU used by typical Operating System task schedulers, resulting in increased accuracy and fairness during overload situations.

- It allows non-cooperative users to safely define custom load shedding methods for those monitoring applications that are not robust against traffic sampling or those that can obtain better accuracy using other load shedding mechanisms not directly provided by the core platform.

- Experimental results show a high degree of robustness against extreme overload situations, traffic anomalies, attacks, previously unseen applications, selfish or buggy monitoring applications, and applications that fail to shed excess load properly for any other reason.

Throughout this document we already pointed out some limitations of our load shedding scheme that open interesting opportunities for future research. We briefly summarize them here.

First, the load shedding scheme presented in this thesis focuses on the CPU as the main resource in network monitoring. However, other system resources, such as memory, disk bandwidth or storage space can also be critical. For example, a network monitoring application performing flow classification can become very greedy in terms of memory consumption during anomalous traffic patterns, such as SYN-flood attacks, massive port scans or DDoS attacks. Although the results presented in this thesis show that a CPU-based load shedding scheme can also prevent overload of other system resources, the design of multi-dimensional load shedding schemes that can consider multiple resources at the same time constitutes an important topic for future research. We believe that similar approaches to the one presented in this thesis could be applied to other system resources as well. In particular, we are currently working on similar predictive techniques to handle overload situations in the intermediate CoMo queues used to export the results of the monitoring applications at each measurement interval.

Second, our predictive scheme assumes a linear dependency between the CPU usage and the selected features. Non-linear relationships in stream processing are unusual given the high performance requirements imposed on this class of applications, which

must be able to operate with a single pass on the incoming stream and implement very efficient algorithms with very low cost per packet. In fact, none of the queries in the standard distribution of CoMo nor any of the more complex applications developed in the framework of this thesis exhibited a non-linear relationship with the traffic features. However, as future work, we intend to study specific network monitoring applications that exhibit non-linear relationships with the set of features we have identified so far. A solution in this case may be to still use linear regression, but define new features computed as non-linear combinations of the existing ones. We are also interested in extending the set of features used in this work to payload-related ones, which may increase the prediction accuracy for those applications that analyze the packet payloads, and with entropy-based features that can capture relevant properties of traffic distributions for prediction purposes. For example, the sample entropy was successfully used in the past for anomaly detection [90]. Fortunately, the recent literature provides us with efficient algorithms to approximate entropies in data streams [31, 91] that could be used in our feature extraction phase.

Finally, our load shedding scheme addresses the problem of how to locally handle (i.e., in a single network monitoring system) rapid and difficult-to-react overload situations. This is an important problem given that some critical computations have very tight real-time constraints and must be performed in the same monitor where the traffic is collected in order to avoid packet losses. However, after shedding excess load, some computations with less time constraints could be distributed across a distributed monitoring infrastructure. In this context, we are currently working on extending our load shedding scheme with effective load balancing and load distribution techniques specifically designed for network monitoring that can efficiently distribute the system load in a distributed network monitoring infrastructure. In this particular problem, other system resources, such as bandwidth between nodes, and other performance metrics, such as query delays, become also critical.

Availability

The source code of the prediction and load shedding system presented in this thesis is publicly available at `http://loadshedding.ccaba.upc.edu` under a BSD open source license. The CoMo system is also available at `http://como.sourceforge.net`.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. of Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB J.*, 12(2), May 2003.

[3] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Trans. Database Syst.*, 17(3), Sept. 1992.

[4] D. Agarwal, J. M. González, G. Jin, and B. Tierney. An infrastructure for passive network monitoring of application data streams. In *Proc. of Passive and Active Measurement Conf. (PAM)*, Apr. 2003.

[5] K. G. Anagnostakis, M. Greenwald, S. Ioannidis, and S. Miltchev. Open packet monitoring on FLAME: Safety, performance, and applications. In *Proc. of IFIP-TC6 Intl. Working Conf. on Active Networks (IWAN)*, Dec. 2002.

[6] J. Apisdorf, K. C. Claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, affordable, high performance staistics collection. In *Proc. of USENIX Large Installation System Administration Conf. (LISA)*, Sept. 1996.

[7] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford University, Oct. 2003.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *VLDB J.*, 13(4), Dec. 2004.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*, June 2002.

[10] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of IEEE Intl. Conf. on Data Engineering (ICDE)*, Mar. 2004.

[11] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of Intl. Workshop on Randomization and Approximation Techniques (RANDOM)*, Sept. 2002.

[12] C. Barakat, G. Iannaccone, and C. Diot. Ranking flows from sampled traffic. In *Proc. of ACM Intl. Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, Oct. 2005.

[13] P. Barlet-Ros, D. Amores-López, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. On-line predictive load shedding for network monitoring. In *Proc. of IFIP-TC6 Networking*, May 2007.

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

[15] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. Custom load shedding for non-cooperative monitoring applications. Technical Report UPC-DAC-RR-2008-50, Technical University of Catalonia, Aug. 2008.

[16] P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. Robust network monitoring in the presence of non-cooperative traffic queries. *Computer Networks*, Oct. 2008 (in press).

[17] P. Barlet-Ros, J. Sanjuàs-Cuxart, J. Solé-Pareta, and G. Iannaccone. Robust resource allocation for online network monitoring. In *Proc. of Intl. Telecommunications Network Workshop on QoS in Multiservice IP Networks (ITNEWS)*, Feb. 2008.

[18] Y. Bejerano and R. Rastogi. Robust monitoring of link delays and faults in IP networks. In *Proc. of IEEE Conf. on Computer Communications (INFOCOM)*, Apr. 2003.

[19] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 2nd edition, 1992.

[20] A. Bifet and R. Gavaldà. Kalman filters and adaptive windows for learning in data streams. In *Proc. of Intl. Conf. on Discovery Science (DS)*, Oct. 2006.

[21] A. Bifet and R. Gavaldà. Learning from time-changing data with adaptive windowing. In *Proc. of SIAM Intl. Conf. on Data Mining (SDM)*, Apr. 2007.

[22] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), July 1970.

[23] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10), Oct. 1977.

[24] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2006.

[25] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, Aug. 2002.

[26] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, Sept. 2003.

[27] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2), Apr. 1979.

[28] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2), Feb. 1988.

[29] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin. A simple network management protocol (SNMP). RFC 1157, May 1990.

[30] CESCA. L'Anella Científica (The Scientific Ring). http://www.cesca.es/en/comunicacions/anella.html.

[31] A. Chakrabarti, K. D. Ba, and S. Muthukrishnan. Estimating entropy and entropy norm on data streams. In *Proc. of Intl. Symp. on Theoretical Aspects of Computer Science (STACS)*, Feb. 2006.

[32] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing of an uncertain world. In *Proc. of Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[33] Cisco Systems. NetFlow Input filters. http://www.cisco.com/en/US/docs/ios/12_3t/12_3t4/feature/guide/gtnfinpf.html.

[34] Cisco Systems. Sampled NetFlow. http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/12s_sanf.html.

[35] Cisco Systems. NetFlow services and applications. White Paper, 2000.

[36] K. C. Claffy, M. Crovella, T. Friedman, C. Shannon, and N. Spring. Community-oriented network measurement infrastructure (CONMI) workshop report. *ACM SIGCOMM Comput. Commun. Rev.*, 36(2), Apr. 2006.

[37] E. Cohen, N. Duffield, H. Kaplan, C. Lund, and M. Thorup. Sketching unaggregated data streams for subpopulation-size queries. In *Proc. of ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*, June 2007.

[38] E. Cohen, N. Grossaug, and H. Kaplan. Processing top-k queries from samples. In *Proc. of ACM Intl. Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2006.

[39] C. L. Compton and D. L. Tennenhouse. Collaborative load shedding for media-based applications. In *Proc. of Intl. Conf. on Multimedia Computing and Systems (ICMCS)*, May 1994.

[40] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of ACM SIGMOD*, June 2003.

[41] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications.* John Wiley and Sons, 1984.

[42] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proc. of ACM Conf. on Computer and Communications Security (CCS)*, Oct. 2004.

[43] N. Duffield. Sampling for passive internet measurement: A review. *Statistical Science*, 19(3), Aug. 2004.

[44] N. Duffield and C. Lund. Predicting resource usage and estimation accuracy in an IP flow measurement collection infrastructure. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2003.

[45] N. Duffield, C. Lund, and M. Thorup. Charging from sampled network usage. In *Proc. of ACM SIGCOMM Internet Measurement Workshop (IMW)*, Nov. 2001.

[46] N. Duffield, C. Lund, and M. Thorup. Properties and prediction of flow statistics from sampled packet streams. In *Proc. of ACM SIGCOMM Internet Measurement Workshop (IMW)*, Nov. 2002.

[47] N. Duffield, C. Lund, and M. Thorup. Estimating flow distributions from sampled flow statistics. In *Proc. of ACM SIGCOMM*, Aug. 2003.

[48] N. Duffield, C. Lund, and M. Thorup. Flow sampling under hard resource constraints. In *Proc. of ACM Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2004.

[49] N. Duffield, C. Lund, and M. Thorup. Learn more, sample less: Control of volume and variance in network measurement. *IEEE Trans. Information Theory*, 51(5), May 2005.

[50] N. Duffield, C. Lund, and M. Thorup. Priority sampling for estimation of arbitrary subset sums. *J. ACM*, 54(6), Dec. 2007.

[51] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, 9(3), June 2001.

[52] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *Proc. of Annual European Symp. on Algorithms (ESA)*, Sept. 2003.

[53] Endace. DAG network monitoring cards. http://www.endace.com.

[54] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *Proc. of ACM SIGCOMM*, Aug. 2004.

[55] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *Proc. of ACM SIGCOMM*, Aug. 2003.

[56] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3), Aug. 2003.

[57] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2003.

[58] A. Feldmann, J. Rexford, and R. Cáceres. Efficient policies for carrying web traffic over flow-switched networks. *IEEE/ACM Trans. Netw.*, 6(6), Dec. 1998.

[59] D. Ficara, S. Giordano, F. Oppedisano, G. Procissi, and F. Vitucci. A cooperative PC/network-processor architecture for multi gigabit traffic analysis. In *Proc. of Intl. Telecommunications Network Workshop on QoS in Multiservice IP Networks (ITNEWS)*, Feb. 2008.

[60] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2), Oct. 1985.

[61] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17(6), Nov. 2003.

[62] L. Golab and T. M. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2), June 2003.

[63] J. M. González and V. Paxson. Enhancing network intrusion detection with integrated sampling and filtering. In *Proc. of Intl. Symp. on Recent Advances in Intrusion Detection (RAID)*, Sept. 2006.

[64] S. Guha, J. Chandrashekar, N. Taft, and K. Papagiannaki. How healthy are today's enterprise networks? In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2008.

[65] G. Hardin. The tragedy of the commons. *Science*, 162(3859), Dec. 1968.

[66] N. Hohn and D. Veitch. Inverting sampled traffic. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2003.

[67] M. Huneault and F. D. Galiana. A survey of the optimal power flow literature. *IEEE Trans. Power Systems*, 6(2), May 1991.

[68] G. Iannaccone. CoMo: An open infrastructure for network monitoring – research agenda. Technical report, Intel Research, Feb. 2005.

[69] G. Iannaccone. Fast prototyping of network data mining applications. In *Proc. of Passive and Active Measurement Conf. (PAM)*, Mar. 2006.

[70] G. Iannaccone, C. Diot, I. Graham, and N. McKeown. Monitoring very high speed links. In *Proc. of ACM SIGCOMM Internet Measurement Workshop (IMW)*, Nov. 2001.

[71] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo white paper. Technical report, Intel Research, Sept. 2004.

[72] IETF PSAMP Working Group. http://www.ietf.org/html.charters/psamp-charter.html.

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

[74] IST LOBSTER sensor at the Technical University of Catalonia (UPC). http://loadshedding.ccaba.upc.edu/appmon.

[75] IST OneLab project. http://www.fp6-ist-onelab.eu.

[76] IST SCAMPI project. http://www.ist-scampi.org.

[77] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. On the placement of internet instrumentation. In *Proc. of IEEE Conf. on Computer Communications (INFOCOM)*, Mar. 2000.

[78] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. of USENIX Annual Technical Conf.*, June 2002.

[79] S. H. S. John A. Stankovic, Chenyang Lu and G. Tao. The case for feedback control real-time scheduling. In *Proc. of Euromicro Conf. on Real-Time Systems (ECRTS)*, June 1999.

[80] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. *ACM SIGOPS Oper. Syst. Rev.*, 31(5), Dec. 1997.

[81] Juniper Networks. Monitoring Services PIC. http://www.juniper.net/products/modules/monitoring_pic.html.

[82] S. Kalidindi and M. J. Zekauskas. Surveyor: An infrastructure for internet performance measurements. In *Proc. of Internet Global Summit (INET)*, June 1999.

[83] T. Karagiannis, A. Broido, N. Brownlee, K. C. Claffy, and M. Faloutsos. Is P2P dying or just hiding? In *IEEE Global Communications Conf. (GLOBECOM)*, Nov. 2004.

[84] T. Karagiannis, D. Papagiannaki, and M. Faloutsos. BLINC: Multilevel traffic classification in the dark. In *Proc. of ACM SIGCOMM*, Aug. 2005.

[85] K. Keys, D. Moore, and C. Estan. A robust system for accurate real-time summaries of internet traffic. In *Proc. of ACM Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2005.

[86] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. of USENIX Security Symp.*, Aug. 2004.

[87] R. R. Kompella and C. Estan. The power of slicing in internet flow measurement. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2005.

[88] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw. Pract. Exper.*, 32(2), Feb. 2002.

[89] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. of ACM SIGCOMM*, Aug. 2004.

[90] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *Proc. of ACM SIGCOMM*, Aug. 2005.

[91] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Perform. Eval. Rev.*, 34(1), June 2006.

[92] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1), Feb. 1994.

[93] C. D. Locke. *Best-effort decision-making for real-time scheduling.* PhD thesis, Carnegie Mellon University, May 1986.

[94] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2006.

[95] J. Mai, A. Sridharan, C.-N. Chuah, H. Zang, and T. Ye. Impact of packet sampling on portscan detection. *IEEE J. Select. Areas Commun.*, 24(12), Dec. 2006.

[96] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of USENIX Annual Technical Conf.*, Jan. 1993.

[97] T. J. McGregor, H.-W. Braun, and J. Brown. The NLANR network analysis infrastructure. *IEEE Commun. Mag.*, 38(5), May 2000.

[98] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. of Intl. Conf. on Multimedia Computing and Systems (ICMCS)*, May 1994.

[99] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. Identifying elephant flows through periodically sampled packets. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2004.

[100] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[101] K. Nahrstedt and R. Steinmetz. Resource management in networked multimedia systems. *IEEE Computer*, 28(5), May 1995.

[102] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. of ACM Symp. on Operating Systems Principles (SOSP)*, Oct. 1997.

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

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

[105] G. Ozsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. Knowledge and Data Eng.*, 7(4), Aug. 1995.

[106] V. Padmanabhan, S. Ramabhadran, S. Agarwal, and J. Padhye. A study of end-to-end web access failures. In *Proc. of ACM Intl. Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, Dec. 2006.

[107] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2005.

[108] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24), Dec. 1999.

[109] V. Paxson and S. Floyd. Wide area traffic: The failure of Poisson modeling. *IEEE/ACM Trans. Netw.*, 3(3), June 1995.

[110] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large scale internet measurement. *IEEE Commun. Mag.*, 36(8), Aug. 1998.

[111] P. Phaal, S. Panchen, and N. McKee. InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks. RFC 3176, Sept. 2001.

[112] T. Plagemann, V. Goebel, A. Bergamini, G. Tolu, G. Urvoy-Keller, and E. W. Biersack. Using data stream management systems for traffic analysis - a case study. In *Proc. of Passive and Active Measurement Conf. (PAM)*, Apr. 2004.

[113] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[114] K. Ramamritham. Real-time databases. *Distrib. Parallel Databases*, 1(2), Apr. 1993.

[115] K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proc. of the IEEE*, 82(1), Jan. 1994.

[116] F. Raspall, S. Sallent, and J. Yufera. Shared-state sampling. In *Proc. of ACM SIGCOMM Internet Measurement Conf. (IMC)*, Oct. 2006.

[117] F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq. In *Proc. of IEEE Intl. Conf. on Data Engineering (ICDE)*, Apr. 2005.

[118] F. Reiss and J. M. Hellerstein. Declarative network monitoring with an under-provisioned query processor. In *Proc. of IEEE Intl. Conf. on Data Engineering (ICDE)*, Apr. 2006.

[119] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of USENIX Large Installation System Administration Conf. (LISA)*, Nov. 1999.

[120] F. Schneider, J. Wallerich, and A. Feldmann. Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware. In *Proc. of Passive and Active Measurement Conf. (PAM)*, Apr. 2007.

[121] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of P2P traffic using application signatures. In *Proc. of Intl. World Wide Web Conf. (WWW)*, May 2004.

[122] J. Søberg, K. H. Hernes, M. Siekkinen, V. Goebel, and T. Plagemann. A practical evaluation of load shedding in data stream management systems for network monitoring. In *Proc. of European Workshop on Data Stream Analysis (WDSA)*, Mar. 2007.

[123] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. In *Proc. of USENIX Symp. on Internet Technologies and Systems (USITS)*, Mar. 2003.

[124] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, Aug. 2004.

[125] W. R. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery algorithms. RFC 2001, Jan. 1997.

[126] K. Suh, Y. Guo, J. Kurose, and D. Towsley. Locating network monitors: Complexity, heuristics, and coverage. In *Proc. of IEEE Conf. on Computer Communications (INFOCOM)*, Mar. 2005.

[127] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, Sept. 2007.

[128] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, Sept. 2003.

[129] tcpdump/libpcap. http://www.tcpdump.org.

[130] The CoMo project. http://como.sourceforge.net.

[131] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *Proc. of Intl. Conf. on Very Large Data Bases (VLDB)*, Sept. 2006.

[132] M. Welsh and D. Culler. Adaptive overload control for busy internet servers. In *Proc. of USENIX Symp. on Internet Technologies and Systems (USITS)*, Mar. 2003.

[133] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM Symp. on Operating Systems Principles (SOSP)*, Oct. 2001.

[134] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2), June 1990.

[135] T. Wolf, R. Ramaswamy, S. Bunga, and N. Yang. An architecture for distributed real-time passive network measurement. In *Proc. of IEEE/ACM Intl. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept. 2006.

[136] Y. Xing, S. Zdoni, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *Proc. of IEEE Intl. Conf. on Data Engineering (ICDE)*, Apr. 2005.

[137] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Proc. of Intl. Conf. on Machine Learning (ICML)*, Aug. 2003.

[138] L. Yuan, C.-N. Chuah, and P. Mohapatra. ProgME: Towards programmable network measurement. In *Proc. of ACM SIGCOMM*, Aug. 2007.

[139] 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.*, 24(10), Oct. 2006.

[140] T. Zseby, T. Hirsch, and B. Claise. Packet sampling for flow accounting: Challenges and limitations. In *Proc. of Passive and Active Measurement Conf. (PAM)*, Apr. 2008.

Appendix A

Publications

A.1 Related Publications

- P. Barlet-Ros, D. Amores-López, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. On-line Predictive Load Shedding for Network Monitoring. In *Proc. of IFIP-TC6 Networking*, Atlanta, USA, May 2007.

- P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Load Shedding in Network Monitoring Applications. In *Proc. of USENIX Annual Technical Conf.*, Santa Clara, USA, June 2007.

- P. Barlet-Ros, J. Sanjuàs-Cuxart, J. Solé-Pareta, and G. Iannaccone. Robust Resource Allocation for Online Network Monitoring. In *Proc. of Intl. Telecommunications Network Workshop on QoS in Multiservice IP Networks (ITNEWS)*, Venice, Italy, Feb. 2008.

- P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. Robust Network Monitoring in the presence of Non-Cooperative Traffic Queries. *Computer Networks*, Oct. 2008 (in press).

Under Submission

- P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, and J. Solé-Pareta. Custom Load Shedding for Non-Cooperative Monitoring Applications. Submitted to *IEEE Conf. on Computer Communications (INFOCOM)*, Aug. 2008.

Technical Reports

- P. Barlet-Ros, G. Iannaccone, J. Sanjuàs-Cuxart, D. Amores-López, and J. Solé-Pareta. Predicting Resource Usage of Arbitrary Network Traffic Queries. Technical Report UPC-DAC-RR-2008-29, Technical University of Catalonia, Dec. 2006.

A.2 Other Publications

- R. Serral-Gracià, P. Barlet-Ros, and J. Domingo-Pascual. Distributed Sampling for On-line SLA Assessment. In *Proc. of IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*, Cluj-Napoca, Romania, Sept. 2008.

- B. Otero, P. Barlet-Ros, S. Spadaro, and J. Solé-Pareta. Mapping Ecology to Autonomic Communication Systems. In *Proc. of Workshop on Autonomic Communications and Component-ware (TACC)*, Budapest, Hungary, Jul. 2008.

- P. Barlet-Ros, V. Carela, E. Codina, and J. Solé-Pareta. Identification of Network Applications based on Machine Learning Techniques. In *Proc. of TERENA Networking Conf.*, Bruges, Belgium, May 2008.

- P. Barlet-Ros, E. Codina, and J. Solé-Pareta. Network Application Identification based on Machine Learning Techniques. *Boletín de RedIRIS*, 1(82-83):40-43, Apr. 2008 (in Spanish).

- R. Serral-Gracià, P. Barlet-Ros, and J. Domingo-Pascual. Coping with Distributed Monitoring of QoS-enabled Heterogeneous Networks. In *Proc. of Intl. Telecommunications Network Workshop on QoS in Multiservice IP Networks (ITNEWS)*, Venice, Italy, Feb. 2008.

- J. Sanjuàs-Cuxart and P. Barlet-Ros. Resource Usage Modeling for Network Monitoring Applications. In *Proc. of Workshop on Execution Environments for Distributed Computing (EEDC)*, Barcelona, Spain, June 2007.

- P. Barlet-Ros, J. Solé-Pareta, J. Barrantes, E. Codina, and J. Domingo-Pascual. SMARTxAC: A Passive Monitoring and Analysis System for High-Speed Networks. *Campus-Wide Information Systems*, 23(4):283-296, Dec. 2006.

- P. Barlet-Ros, J. Solé-Pareta, J. Barrantes, E. Codina, and J. Domingo-Pascual. SMARTxAC: A Passive Monitoring and Analysis System for High-Speed Networks. In *Proc. of TERENA Networking Conf.*, Catania, Italy, May 2006.

- P. Barlet-Ros, H. Pujol, J. Barrantes, J. Solé-Pareta, and J. Domingo-Pascual. A System for Detecting Network Anomalies based on Traffic Monitoring and Prediction. *Boletín de RedIRIS*, 1(74-75):23-27, Dec. 2005 (in Spanish).

- P. Barlet-Ros, J. Solé-Pareta, and J. Domingo-Pascual. SMARTxAC: A System for Monitoring and Analysing the Traffic of the Anella Científica. *Boletín de RedIRIS*, 1(66-67):27-30, Dec. 2003 (in Spanish).

- B. Stiller, P. Barlet-Ros, J. Cushnie, J. Domingo-Pascual, D. Hutchison, R. J. Lopes, A. Mauthe, M. Popa, J. Roberts, J. Solé-Pareta, D. Trcek, and C. Veciana-Nogués. Pricing and QoS. *Quality of Future Internet Services: COST Action 263 Final Report*, Lecture Notes in Computer Science, 2856(1):263-291, Nov. 2003.

Technical Reports

- R. Serral-Gracià, P. Barlet-Ros, and J. Domingo-Pascual. Distributed Sampling for On-line QoS Reporting. Technical Report UPC-DAC-RR-2007-17, Technical University of Catalonia, May 2007.

- P. Barlet-Ros, H. Pujol, J. Barrantes, J. Solé-Pareta, and J. Domingo-Pascual. A System for Detecting Network Anomalies based on Traffic Monitoring and Prediction. Technical Report UPC-DAC-RR-2005-40, Technical University of Catalonia, June 2005.

- C. Veciana-Nogués, P. Barlet-Ros, J. Solé-Pareta, J. Domingo-Pascual. Traffic Accounting and Classification for Cost Sharing in National Research Networks. Technical Report UPC-DAC-RR-2003-24, Technical University of Catalonia, May 2003.