Efficient Algorithms for Passive Network Measurement

Josep Sanjuàs-Cuxart

Advisor: Pere Barlet-Ros

Co-Advisor: Josep Solé-Pareta

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

A thesis presented to Universitat Politècnica de Catalunya in partial fulfillment of the requirements for the degree of

Doctor in Computer Science

February 2012

Abstract

Network monitoring has become a necessity to aid in the management and operation of large networks. Passive network monitoring consists of extracting metrics (or any information of interest) by analyzing the traffic that traverses one or more network links. Extracting information from a high-speed network link is challenging, given the great data volumes and short packet inter-arrival times. These difficulties can be alleviated by using extremely efficient algorithms or by sampling the incoming traffic. This work improves the state of the art in both these approaches.

For one-way packet delay measurement, we propose a series of improvements over a recently appeared technique called Lossy Difference Aggregator. A main limitation of this technique is that it does not provide per-flow measurements. We propose a data structure called Lossy Difference Sketch that is capable of providing such per-flow delay measurements, and, unlike recent related works, does not rely on any model of packet delays.

In the problem of collecting measurements under the sliding window model, we focus on the estimation of the number of active flows and in traffic filtering. Using a common approach, we propose one algorithm for each problem that obtains great accuracy with significant resource savings.

In the traffic sampling area, the selection of the sampling rate is a crucial aspect. The most sensible approach involves dynamically adjusting sampling rates according to network traffic conditions, which is known as adaptive sampling. We propose an algorithm called Cuckoo Sampling that can operate with a fixed memory budget and perform adaptive flow-wise packet sampling. It is based on a very simple data structure and is computationally extremely lightweight.

The techniques presented in this work are thoroughly evaluated through a combination of theoretical and experimental analysis.

Resum

El monitoratge del tràfic de xarxa ha esdevingut una necessitat a l'hora de gestionar i operar xarxes de gran tamany. El monitoratge passiu del tràfic consisteix en l'extracció de mètriques (o qualsevol altra informació d'interès) a partir de l'anàlisi del tràfic que circula per un o més enllaços. L'extracció d'aquesta informació és, però, dificultosa en enllaços troncals de gran ample de banda, degut als enormes volums d'informació implicats, i que es disposa de poc temps per tractar cada paquet de dades. Aquestes dificultats es poden mitigar mitjançant l'ús d'algorismes especialitzats i extremadament eficients, i aplicant tècniques de mostratge del tràfic. Aquesta tesi proposa solucions en aquests dos fronts.

Per la mesura del retard que experimenten els paquets en viatjar al llarg d'una xarxa, es realitzen una sèrie de millores a una tècnica recentment apareguda anomenada Lossy Difference Aggregator, i es proposa una nova estructura de dades capaç obtenir aquesta informació per fluxe anomenada Lossy Difference Sketch.

S'investiga el problema d'obtenir mesures de tràfic sota un model de mesura anomenat *finestra lliscant*, i es proposen dos nous algorismes de funcionament similar, un per estimar el nombre de fluxes actius, i un per filtrar elements del tràfic.

Un element crucial en l'àrea del mostratge de tràfic és la selecció de la freqüència de mostratge. La solució més adequada passa per adaptar aquesta freqüència de forma dinàmica, d'acord amb les característiques del tràfic de xarxa, la qual cosa es coneix com mostratge adaptatiu. Proposem un algorisme anomenat Cuckoo Sampling que opera amb una quantitat de memòria fixa i mostreja els fluxes presents en el tràfic. Aquest algorisme es basa en una estructura de dades molt simple, i té uns requeriments de còmput molt lleugers.

Les tècniques que es presenta en aquest treball són avaluades mitjançant una combinació d'anàlisi teòrica i experimental.

Acknowledgements

This thesis would have been impossible without the close guidance of my advisor Pere Barlet-Ros. His willingness to discuss ideas in depth and his technical insights were fundamental for this work.

My co-advisor Professor Josep Solé-Pareta's door has always been open to me, and his experience and expert advice have contributed in countless ways to this work from the first day.

I would also like to express my gratitude towards Ramana Kompella and Nick Duffield. Our collaboration in delay measurement and flow sampling was very fruitful and a great learning experience.

I am indebted to Gianluca Iannaccone, who offered me the opportunity of interning at Intel Research in Cambridge and Berkeley. He also lead the efforts behind the CoMo passive network monitoring system. A prototype of CoMo has been extensively used in this work, and for this reason I also wish to thank the many other contributors of the project.

Manel Martínez and Marc Solé-Simó contributed important insights to our flow counting algorithm. Ismael Castell-Uroz, Ignasi Paredes-Oliva, Valentín Carela-Español and Jakub Mikians assisted me in various ways. It is also fair to thank Fabio Ricciato and all the anonymous reviewers of conferences and journals whose comments lead to improvements to our work.

I would also like to thank the external reviewers of this thesis Fabio Ricciato and Xenofontas Dimitropoulos, as well as the members of my pre-defense committee Jordi Nin, Leandro Navarro-Moldes and Albert Cabellos-Aparicio, who provided very useful comments to improve this document.

Professor Jordi Domingo-Pascual and Albert Cabellos-Aparicio provided continued support and advice. Diego, Albert, Lori, Valentín, Ignasi, Jakub, Oriol, Ismael, Florin, Oscar, Pedro, Sergio (in no particular order) were great office mates. The weekly lunch time pizza with Xavi was more important to this thesis than one might imagine. I would also like to thank my uni friends (*gràcies, mestres!*), and my closest childhood friends for their support.

Finally, I wish to thank my family. They have always had words of encouragement and unconditional support. My parents and my brother; everything I ever accomplish, I owe to them. My wife Anna, whose support through the ups and downs of this road is the principal reason why I am able to present this thesis. *Moltes gràcies. Sense vosaltres, impossible!*

Thank you all. You have my most sincere gratitude.

Research Institutions and Companies. I thank the Center for Scientific and Academic Services of Catalonia (CESCA) and UPCnet for allowing the collection of network traffic that made this work possible.

This work includes GeoLite data created by MaxMind, available from `http://www.maxmind.com/`. We also acknowledge Ipoque for kindly providing access to their PACE [7] traffic classification engine for research purposes.

Funding. This research was funded by the Spanish Ministry of Science and Innovation under contracts TEC2011-27474 (NOMADS project) and TSI2005-07520-C03-02 (CEPOS), by the *Comissionat per a Universitats i Recerca del DIUE de la Generalitat de Catalunya* (ref. 2009SGR-1140), and by a grant from the Intel Research Council.

Contents

List of Figures

List of Tables

List of Algorithms

LIST OF ALGORITHMS

1

Introduction

As networks grow and become faster, their operation and management becomes more difficult. Network monitoring is nowadays a necessity, and aids in tasks such as fault diagnosis and troubleshooting, evaluation of network performance, capacity planning, traffic accounting and classification, and to detect anomalies and investigate security incidents.

Two broad families of network monitoring techniques exist. Active network monitoring techniques infer network characteristics from edge nodes by injecting probe packets to the network, while passive network monitoring techniques rely instead on per-packet analysis of the traffic that traverses network links. The major advantage that active network monitoring tools present is their ease of deployment, as they do not require any modification of the network, which is measured from the edge. However, they can not directly extract measurements, but only infer them by observing how the network transmits probe packets.

Passive network monitoring requires instead monitors to receive a copy of the traffic that traverses network links. This can be accomplished either by physically replicating the signal in a link (for example, installing an optical splitter) or with the aid of the existing networking hardware, e.g., via port mirroring or leveraging built-in network monitoring capabilities, among which Cisco's NetFlow [3] is arguably the most representative. Either way, the monitoring platform extracts information via per-packet analysis, i.e., by processing every packet that traverses the monitored links.

In principle, through per-packet analysis, any network activity or metric of interest can be captured, measured and reported. However, due to the large number of active connections and the ever-increasing data rates in backbone links, per-packet analysis is very challenging. Fast networks transmit an enormous number of packets per time unit, which leaves very few time to process each packet before the next one arrives. Additionally, long-term buffering is impossible, since it would involve massive data volumes.

In such an environment, even the extraction of seemingly simple metrics is challenging. Consider for example the problem of counting the number of address pairs that are exchanging packets through a particular network link. This could be achieved by maintaining a hash table that stores the address pairs as observed in the packet stream. For every incoming packet, a table lookup would be performed and, if necessary, a new entry would be created for the current packet's source and destination address pair. The number of address pairs then corresponds to the number of entries in the table.

Since the number of active flows and involved address pairs in backbone networks is very large (and potentially enormous), such a table would need to be stored in DRAM. The problem is that DRAM has access times lie within similar orders of magnitude as packet interarrival times [90]. This leaves room for very few memory accesses per packet. If a monitor can not keep up with the incoming packet stream, it will start buffering packets and, soon after, dropping incoming ones. This is extremely problematic, since it unpredictably degrades the accuracy of the measurements.

This example illustrates the main difficulty that passive network monitoring systems face. The extremely fast data rates and large number of flows can easily overwhelm the monitor's resources. This is exacerbated by the fact that network traffic is extremely volatile, and even moreso when attackers can inject malicious traffic (e.g., Denial of Service attacks that artificially introduce a large number of flows). This means that peak monitor load might be orders of magnitude larger than the average (or than expected). As a consequence, provisioning monitors with peak or worst-case load in mind is impractical [31] and, hence, network monitors can be expected to face overload on a regular basis.

Two complementary solutions are commonly applied to reduce the load of network monitors. First, the extraction of information from the packet stream can be done using specialized algorithms that extract basic metrics extremely quickly by slightly sacrificing measurement accuracy. Second, the input traffic can be sampled to reduce the volume of data that measurement algorithms must process. Since this thesis contributes in these fronts, Subsections 1.1 and 1.2 further delve into both topics. Section 1.3 presents the contributions of this thesis, and Section 1.4 provides an outline for the rest of this document.

1.1 Specialized Measurement Algorithms

The first measure to reduce the load of network monitors is to use specialized algorithms that are extremely fast. Network traffic measurement algorithms must present (i) a small per-packet cost and number of memory accesses per packet, and (ii) a light memory footprint, that enables cost-effective implementation in faster (more expensive) SRAM memory modules. Additionally, they have to process the incoming packet stream in one pass, given that the enormous data volumes involved make long term storage impossible. Such algorithms must not be viewed merely as incremental improvements over basic, well-known algorithms and data structures. Instead, they involve fundamentally different approaches, and resort to computing approximate results to meet the demanding requirements of packet processing.

While computing exact results might involve large packet processing costs and memory usage, approximate results can be obtained considerably faster and with a smaller memory footprint. A paradigmatic example of this is the *direct bitmap* presented in [65]. Consider the previous example, where one wishes to compute the number of address pairs that are exchanging packets on a given link. We have established that a naive algorithm based on simply storing address pairs on a hash table requires storing a large number of entries, and several memory accesses per packet in order to manage collisions. The basic idea behind the direct bitmap is not to maintain a full hash table, but, instead, to compute the number of buckets in the hash table that would be used, had one been maintained.

This algorithm maintains only an array of bits, with all positions initially unset. Then, for every packet, the involved address pair is hashed. The hash value, modulus the array's size, indicates a position of the array which is set. If a pseudo-random hash function is used, the original number of address pairs can be estimated from the ratio of unused bits (as explained in [65]). This provides a much more compact representation of the hash table (only one bit per bucket), and requires only a single memory access per packet. The drawback is that results are not exact anymore due to collisions. The actual accuracy depends on the *load* of the data structure, i.e., the ratio of unique entries to positions in the bitmap. The key of this algorithm is that its accuracy is great, even under large loads. For example, this algorithm can count 10^6 unique elements with errors below 1% using only 160,000 positions (i.e., 20KB of memory).

Due to the huge data volumes involved in passive network monitoring, difficulties also arise when the metrics to be computed require the coordination of two (or more) measurement points. A great example of this is packet delay measurement. To compute the amount of time that packets take to traverse the network elements between two measurement points, the only option is to collect measurements in two monitors, exchange packet timestamps, and compare them. The amount of information that has to be exchanged between monitors to compute the final result can then easily become a concern.

For example, in the case of delay measurement, it is not possible for the one of the monitor to simply to transmit the timestamp of every packet to the other one, as this would have a large bandwidth overhead. This problem can be mitigated using sampling or by devising methods to estimate the delay without incurring such large network overheads. An example of an algorithm that overcomes this problem is the Lossy Difference Aggregator [88]. This technique can save several orders of magnitude in transmitted data volume, by using the following key insight. To compute the average delay of a set of packets, one does not need to compare their timestamps individually. Instead, one can compute the sum of packet timestamps in each measurement node (clock synchronization is assumed). By exchanging and comparing these sums, then dividing over the total number of packets, the average delay can be obtained. This scheme saves the huge overhead of sending packet timestamps individually.

The packets that enter a monitor for inspection form a continuous data stream. Most often, the operator of the monitor does not wish to obtain measurements comprising the full lifetime of the monitor. A typical approach to bound measurements in time is to periodically report the current measurements and reset the data structures, i.e., to organize measurements in non-overlapping, back-to-back measurement intervals. Recently, a different measurement model is gaining interest in the research community, which is called the *sliding window model* [48]. Under this model, the monitor can be queried at any time for measurements spanning the last w time units, i.e., a window of w time units that continuously advances in time. This is a more challenging model to implement, since measurement algorithms must track time and expire information as it ages out of the measurement window.

1.2 Input Traffic Sampling

The second, and arguably the most wide-spread measure against overload in network monitors is to sample incoming traffic. By carefully discarding input traffic, monitors can still compute results that preserve certain statistical guarantees (as opposed to dropping packets without control, as happens when a monitor is overloaded).

The accuracy that each metric of interest obtains under sampling depends on two factors. First, obviously, the sampling rate; the more packets that are processed, the better. Second, it depends on the sampling method (Reference [57] presents a survey of sampling techniques). Some statistics are well preserved by sampling packets with uniform probability (what is known simply as *packet sampling*). For example, given a large flow, the original number of packets can be estimated by dividing the resulting number of packets over the sampling rate.

However, others are not tolerant to packet sampling. For example, it is not possible to invert the original number of data flows in the traffic from sampled data. For this reason, other sampling methods exist that preserve certain characteristics of the traffic. For example, flow-wise packet sampling [57] either selects or discards all the packets for a given flow. The accuracy that measurement algorithms achieve under sampling depends on the sampling rate, but also on the sampling scheme being used, since the degree of resistance of a metric to various sampling methods can vary widely.

In order to preserve measurement accuracy, the correct selection of the incoming traffic sampling rates is crucial. In principle, one should choose the highest sampling rate that guarantees that the monitor will not exceed the available resources. However, in practice, sampling methods often require the configuration of a static sampling rate (e.g., Sampled NetFlow [45]). This is harmful to the completeness and accuracy of the measurements, given that network operators tend to select aggressively small (“safe”) sampling rates that guarantee the correct operation of monitors even under anomalies or network attacks.

A more sensible solution is to dynamically adjust incoming sampling rate according to the network conditions and the available resources (for Sampled NetFlow, an adaptive sampling rate selection method is presented in Reference [63]). This way, the monitor can collect more accurate measurements in low (or average) load, and can still run under adverse conditions, i.e., peak load, or in the presence of network anomalies or attacks.

1.3 Objectives and Contributions

Two complementary solutions have been discussed to reduce the load of network monitors: the use of efficient, specialized algorithms to extract network measurements, and to sample input traffic at a dynamic sampling rate that adapts to network conditions. *The main objective of this thesis is to propose solutions for important, open problems in both these fronts*. We next develop this main goal into more specific objectives, and detail the specific contributions of this work. In the area of specialized network monitoring algorithms, we seek the following two objectives:

1. Devise techniques to *efficiently obtain fine-grained packet delay measurements*.

   Applications that are particularly sensitive to network delay are recently gaining popularity, such as algorithmic trading, high-performance computing, media streaming, conferencing, and online gaming.

   Only by obtaining accurate measurements of the delay that packets suffer while traversing a network can the quality of service of such applications be assessed. However, delay measurement techniques are not widely deployed (e.g., within routers). Existing measurement techniques either present too much overhead, depend on specific traffic properties that do not hold in every scenario, or provide coarse measurements.

   The accurate measurement of packet delays at a flow level helps assess the quality of service offered to the users of such delay-sensitive applications. This allows, for example, to measure their performance in a shared infrastructure, for the verification of the service level agreements, or for troubleshooting and to pin-point the cause behind performance penalties. A data center operator (or, even, an Internet Service Provider) can use such measurements to determine whether their network is responsible and even, if so, which links are causing service degradation, and deploy counter-measures (such as increasing the bandwidth, and revising the routing or traffic prioritization policies).

   In this topic, we present the following contributions:

   - An improved analysis of a delay measurement technique called Lossy Difference Aggregator. This analysis allows for improvement of the data structure in two ways. First, it provides a way to optimally parametrize it when certain network performance metrics are known. Second, it provides insights that help parameterize it when these are not known.
   - A novel data structure called Lossy Delay Sketch. The previous technique is not capable of providing measurements per flow, but just overall traffic aggregate measurements. This technique instead measures delay on a per-flow basis, without maintaining per-flow state, and with reasonably small network overhead.

2. Find better mechanisms to *adapt existing specialized measurement algorithms to the sliding window measurement model.*

   Conceptually, network monitoring involves measuring a continuous stream of data. In such a scenario, measurements must be somehow bounded in time. Perhaps the simplest measurement model is collecting information over consecutive, non-overlapping, back-to-back time intervals. This measurement model is easy to implement, as it simply involves periodically flushing results and resetting the data structures. However, it also implies that measurements can be obtained only once per measurement window.

   In contrast, the sliding window measurement allows the operators to query the monitor at *any* moment, to obtain the measurements pertaining to the last w time units. This is akin to having the measurement window to continuously advance in time, as opposed to the previous model, where the window jumps discretely from one time interval to the next.

   Thus, this measurement model provides significantly more flexibility, as monitors can be queried at any time. This means that systems can be built that use large measurement windows, but can be often queried. For example, network anomaly detectors can provide an alarm when the number of flows within the last day is abnormally high. Another example of application is a traffic filtering device (e.g., an intrusion prevention system) that can be instructed to block certain traffic sources for a given amount of time.

   Measurement over sliding windows requires data structures to maintain a sense of time, since information needs to be expired after a pre-defined amount of time has passed. A common approach is to adapt existing specialized algorithms and data structures (e.g., bitmaps or Bloom filters) to the sliding window measurement, by timestamping every piece of information. Then, every data structure entry can be evicted after it ages out of the time window. However, this heavily increases memory requirements.

   In particular, this work contributes:

   - A flow counting algorithm called Countdown Vector that can estimate counts with high accuracy under the sliding window measurement model. This technique adapts an existing flow counting algorithm to this measurement model. The main feature of this algorithm is approximate information expiration, which overcomes the need to store a full time-stamp per data structure entry.

- A traffic filtering scheme called Countdown Bloom filter, based on the Bloom filter data structure, with the particularity that items are automatically evicted from the filter after a pre-defined amount of time has passed. The expiration mechanism is similar to that of the Countdown Vector.

In the area of traffic sampling, our objective is to provision monitors with a *dynamic traffic sampling mechanism* that selects the appropriate sampling rate according to network traffic conditions and the available memory. Such a mechanism must be both *computationally lightweight* to support measurement of high bandwidth network links. Additionally, it must be practical, i.e., *based on simple data structures and algorithms*.

Current adaptive sampling mechanisms are not widely used in practice (e.g., in routers), possibly due to both their computational cost and implementation complexity. Hence, sampling rates must be configured statically. By provisioning a practical dynamic sampling mechanism, deployment in network routers will be possible, thus simplifying their configuration and increasing measurement accuracy. In the area of traffic sampling, this work provides the following contribution:

- A lightweight flow-wise packet sampling algorithm called Cuckoo Sampling that adapts its sampling rate to the traffic conditions, according to a given memory budget. This technique is easy to implement and relies on simple algorithms and data structures that make it very practical for implementation (e.g., within routers).

The evaluation of the performance of the techniques that are presented in this thesis required their implementation and deployment in fast network links. For this reason, we developed a modular network monitoring system and to implemented our techniques. As an additional contribution of this work, we released the source code to the community.

Obtaining access to high speed links in operational networks is not an easy endeavor, principally because network operators are very reluctant to provide access to network traffic due to privacy concerns. For this reason, a secondary objective of this work was to take on the opportunity to perform studies on real network traffic. In this spirit, this work contributes a traffic analysis of HTTP-based file sharing services, the characteristics of which had not been extensively studied in the literature.

1.4 Outline of this Report

Following this introductory chapter, three parts (I to III) form the body of this document. Each of the parts is closely tied to one of the previously described objectives. These parts are meant to be as self-contained as possible, in that they all introduce the topic at hand, discuss the related works, and conclude:

- Part I includes the contributions towards obtaining fine-grained delay measurements. As discussed in the previous section, these include an improvement of the pre-existing Lossy Difference Aggregator (Chapter 2) and a novel data structure called Lossy Delay Sketch to measure per-flow delay (Chapter 3).
- Part II presents measurement techniques over sliding windows. After introducing the specific issues behind measurement over sliding windows (Chapter 4), the Countdown Vector (Chapter 5) and Countdown Bloom Filter (Chapter 6) data structures are presented.
- Part III is centered on traffic sampling. It starts by introducing the challenges behind these techniques, and the existing related works (Chapters 8 and 9). A novel dynamic flow sampling technique called Cuckoo Sampling is then presented in (Chapter 10). This technique is extremely practical, in that it has low cost and is easy to implement.

Part IV concludes the thesis with a summary, and outlines future works. In the appendices that follow, we present the scenario where we have evaluated our proposals, present a traffic analysis based study of HTTP-based file sharing services, and list the peer-reviewed papers that were published as an outcome of this work.

Part I

Delay Measurement

2

Improvements to the Lossy Difference Aggregator

One-way packet delay is an important network performance metric that has been traditionally measured actively with probing traffic, or passively by capturing, transmitting and comparing individual packet timestamps. Recently, a new data structure called Lossy Difference Aggregator (LDA) has been proposed in [88] to estimate this metric more efficiently than these classical approaches.

This chapter presents an independent validation of the LDA algorithm and provides an improved analysis that results in a 20% increase in the number of packet delay samples collected by the algorithm. We also extend the analysis by relating the number of collected samples to the accuracy of the LDA and provide additional insight on how to parametrize it. Finally, the algorithm is extended to overcome some of its practical limitations.

The LDA computes aggregate packet delay measurements. One of its principal limitations is that it does not provide per-flow estimates. In Chapter 3 the more complex problem of obtaining per-flow estimates will be approached by blending the LDA with sketching techniques.

2.1 Introduction

Packet delay is one of the main indicators of network performance, together with throughput, jitter and packet loss. This metric is becoming increasingly important with the rise of applications like voice-over-IP, video conferencing or online gaming. Moreover, in certain environments, it is an extremely critical network performance metric; for example, in high-performance computing or automated trading, networks are expected to provide latencies in the order of few microseconds [88].

Two main approaches have been used to measure packet delays. Active schemes send traffic probes between two nodes in the network and use inference techniques to determine the state of the network (e.g., [33, 44, 110, 125]). Passive schemes are, instead, based on traffic analysis in two of the points of a network. They are, in principle, less intrusive to the network under study, since they do not inject probes. However, they have been often disregarded, since they require collecting, transmitting and comparing packet timestamps at both network measurement points, thus incurring large overheads in practice [108]. For example, [133] proposes delaying computations to periods of low network utilization if measurement information has to be transmitted over the network under study.

The Lossy Difference Aggregator (LDA) is a data structure that has been recently proposed in [88] to enable fine-grain measurement of one-way packet delays using a passive measurement approach with low overhead. The data structure is extremely lightweight in comparison with the traditional approaches, and can collect a number of samples that easily outperforms active measurement techniques, where traffic probes interfering with the normal operation of a network can be a concern.

The main intuition behind this measurement technique is to sum all packet timestamps in the first and second measurement nodes, and infer the average packet delay by subtracting these values and dividing over the total number of packets. The LDA, though, maintains several separate counters and uses coordinated, hash-based traffic sampling [58] in order to protect against packet losses, which would invalidate the intuitive approach. The complete scheme is presented in Section 2.2.

This chapter presents an independent validation of the work presented in [88]. Section 2.3 revisits the analysis of the algorithm. In particular, Section 2.3.1 investigates the effective number of samples that the algorithm can collect under certain packet loss ratios. This work improves the original analysis, and finds that doubling the sampling rates originally suggested maximizes the expectation of the number of samples collected by the algorithm. In Section 2.3.2, we contribute an analysis that relates the effective sample size with the accuracy that the method can obtain, while Section 2.3.3 compares the network overhead of the LDA with pre-existing techniques.

For the case when packet loss ratios are unknown, the original paper proposed and compared three reference configurations of the LDA in multiple memory banks to target a range of loss ratios. In Secs. 2.4 and 2.4.1 we extend our improved analysis to the case of unknown packet loss, and we (i) find that such reference configurations are almost equivalent in practice, and (ii) provide improved guidelines on how to dimension the multi-bank LDA.

Section 2.6 validates our analysis through simulations, with similar parameters to [88], for the sake of comparability. Finally, in Section 2.7 we deploy the LDA on a real network scenario. The deployment of the LDA in a real setting presents a series of challenges that stem from the assumptions behind the algorithm as presented in [88]. We propose a simple extension of the algorithm that overcomes some of the practical limitations of the original proposal.

At the time of this writing, another analysis of the Lossy Difference Aggregator already exists [69]. The authors provide a parallel analysis of the expectation for the sample size collected by the LDA and, coherently with ours, suggest doubling the sampling rates compared to the original LDA proposal. For the case where packet loss ratios are unknown beforehand, their analysis studies how to dimension the multi-bank LDA to maximize the expectation for the sample size. Optimal sampling rates are determined that maximize sample sizes for tight ranges of expected packet loss ratios. Our analysis differs in that we relate sample size with accuracy, and focus on maximizing accuracy rather than sample size. Additionally, our study includes an analytic overhead comparison with traditional techniques, presents the first real world deployment of the LDA and proposes a simple extension to overcome some of its practical limitations.

2.2 Background: the Lossy Difference Aggregator

The Lossy Difference Aggregator (LDA) [88] is a data structure that can be used to calculate the average one-way packet delay between two network points, as well as its standard deviation. We refer to these points as the sender and the receiver, but they need not be the source or the destination of the packets being transmitted, but merely two network viewpoints along their path.

The LDA operates under three assumptions. First, packets are transmitted strictly in FIFO order. Second, the clocks of the sender and the receiver are synchronized. Third, the set of packets observed by the receiver is identical to the one observed by the sender, or a subset of it when there is packet loss. That is, the original packets are not diverted, and no extra traffic is introduced that reaches the receiver.

A classic algorithm to calculate the average packet delays in such a scenario would proceed as follows. In both the sender and the receiver, the timestamps of the packets are recorded. After a certain measurement interval, the recorded packet delays (or, possibly, a subset of them) are transmitted from the sender to the receiver, which can then compare the timestamps and compute the average delay. Such an approach is impractical, since it involves storing and transmitting large amounts of information.

The basic idea behind the LDA is to maintain a pair of accumulators that sum all packet timestamps in the sender and the receiver separately, as well as the total count of packets. When the measurement interval ends, the sender transmits the value of its accumulator to the receiver, which can then compute the average packet delay by subtracting the values and dividing over the total number of packets.

Example. The idea of timestamp aggregation is easily understood from an example. Imagine a network that introduces a certain delay between two measurement points A and B. Let n packets traverse this network from A to B. Let the timestamp of the ith packet be t_i in monitor A, and t'_i in monitor B, with $1 \leq i < n$ and $t'_i > t_i$. Traditional techniques would record each packet timestamp each of the monitors, then send the timestamps captured in A to B, to compare them one by one to obtain the average delay $\sum_{i=1}^{n}(t'_i - t_i)/n$. Note how this requires sending n timestamps over the network, and storing them in memory. The timestamp aggregation approach would instead maintain a *sum* of timestamps in each monitor. In A, we have $s_a = \sum_{i=1}^{n} t_i$, and in B, $s_b = \sum_{i=1}^{n} t'_i$. Then, A would send s_a to B for comparison with s_b, to obtain an equivalent result: $(s_b - s_a)/n = \left(\sum_{i=1}^{n} t'_i - \sum_{i=1}^{n} t_i\right)/n = \sum_{i=1}^{n}(t'_i - t_i)/n$.

This timestamp aggregation approach requires the set of packets processed by the sender and the receiver to be identical, since the total packet counts in the sender and the receiver must agree. Thus, it is extremely sensitive to packet loss. In order to protect against it, the LDA partitions the traffic into b separate streams, and aggregates timestamps for each one separately in both the sender and the receiver. Additionally, for each of the sub-streams, it maintains a packet count. Thus, it can detect packet losses and invalidate the data collected in the corresponding accumulators. When the measurement interval ends, the sender transmits all of the values of the accumulators and counters to the receiver. Then, the receiver discards the accumulators where packet counts disagree, and computes an estimate for the average sampling delay using the remainder.

Figure 2.1 shows a the LDA data structure as will be consider in this work (which slightly differs from the original, as will be explained), together with example measurement data (bor-

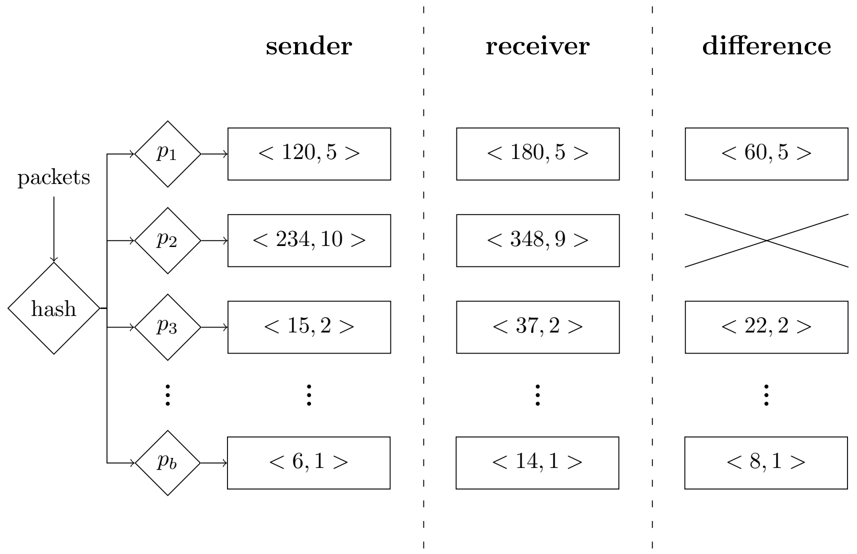


Figure 2.1: Generalized LDA data structure.

rowed borrowed from the original LDA paper). Each of the accumulators must aggregate the timestamps from the same set of packets in the sender and the receiver, i.e., both nodes must partition the traffic using the same criteria. In order to achieve this effect, the same pre-arranged, pseudo-random hash function is used in both nodes, and the hash of a packet identifier is used to determine its associated position in the LDA. In the example, after the sender transmits the accumulators for comparison, the second one is invalidated, as packet counts do not match.

As packet losses grow high, though, the number of accumulators that are invalidated increases rapidly. As an additional measure against packet loss, the LDA samples the incoming packet stream. In the most simple setting, all of the accumulators apply an equal sampling rate p to the incoming packet stream. Again, sender and receiver sample incoming packets coordinately using a pre-arranged pseudo-random hash function [58].

Reference [88] proposes a more complex setting of the LDA, which partitions the accumulators in banks, each one with a different sampling rate. This has several benefits that will be discussed in this chapter. Figure 2.1 depicts a slightly more general variant of the LDA data structure that we will consider in this work. Under this generalized version of LDA, each one of the b accumulators can be set with a potentially different sampling rate p_b.

As an added benefit, the LDA data structure can also be mined to estimate the standard deviation of packet delays using a technique described in Reference [24]. We omit this aspect of the LDA in this work, but the improvements we propose will also increase the accuracy in the estimation of the standard deviation of packet delays.

2.3 Improved Analysis

The LDA is a randomized algorithm that depends on the correct setting of the sampling rates to gather the largest possible number of packet delay samples. The sampling rate p presents a classical tradeoff. The more packets are sampled, the more data the LDA can collect, but the more it is affected by packet loss. Conversely, lower sampling rates provide more protection against loss, but limit the amount of information collected by the accumulators.

This section improves the original analysis of the Lossy Difference Aggregator (LDA) introduced in [88] in several ways. First, it improves the analysis of the expected number of packet delay samples it can collect, which leads to the conclusion that sampling rates should be doubled. Second, it relates the number of samples with the accuracy in an easy to understand way that makes it obvious that undersampling is preferable to sampling too many packets. Third, it compares its network overhead with pre-existing passive measurement techniques. Fourth, it provides a better understanding and provides guidelines to dimension multi-bank LDAs.

2.3.1 Effective Sample Size

In order to protect against packet loss, the LDA uses packet hashes to distribute timestamps across several accumulators, so that losses only invalidate the samples collected by the involved memory positions. Table 2.1 summarizes the notation used in this section. Given n packets, b buckets (accumulator-counter pairs) and packet loss probability r, the probability of a bucket of staying useful corresponds to the probability that no lost packet hashes to the bucket in the $receiver$ node, which can be computed as $(1 - r/b)^n \approx e^{-n\,r/b}$ (according to the law of rare events).

Then, the expectation for the number of usable samples, which we call the effective sample size, can be approximated to $E\,[S] \approx \frac{(1-r)\,n}{e^{n\,r/b}}$. In order to provide additional protection against packet losses, the LDA also samples the incoming packets; we can adapt the previous formulation to account for packet sampling as follows:

$$E\,[S] \approx \frac{(1-r)\,p\,n}{e^{n\,r\,p/b}} \tag{2.1}$$

name	variable	name	variable
n	#pkts	b	#buckets
r	packet loss ratio	μ	average packet delay
p	sampling rate	$\hat{\mu}$	estimate of the avgerage delay

Table 2.1: Notation

Reference [69] shows that this approximation is extremely accurate for large values of n. The approximation is better as n becomes larger and the probability of sampling a packet loss stays low. Note that this holds in practice; otherwise, the buckets would too often be invalidated. For example, when the absolute number of sampled packet losses is in the order of the number of buckets b, it obtains relative errors around 5×10^{-4} for as few as $n = 1000$ packets. Note however that this formula only accounts for a situation where all buckets use an equal fixed sampling rate p, i.e., a single bank LDA. Section 2.4.1 extends this analysis to the multi-bank LDA, while Section 2.6 provides an experimental validation of this formula.

Reference [88] provides a less precise approximation for the expected effective sample size. When operating under a sampling rate $p = \alpha\, b/(L+1)$, a lower bound $E[S] >= \alpha\,(1-\alpha)\,R\,b/(L+1)$ is provided, where R corresponds to the number of received packets and L to the number of lost packets; in our notation, $R = n\,(1-r)$ and $L = n\,r$. Trivially, this bound is maximized when $\alpha = 0.5$. Therefore, it is concluded that the best sampling rate p that can be chosen is $p = 0.5\,\frac{b}{n\,r+1}$.

However, our improved analysis leads to a different value for p by maximizing (2.1). The optimal sampling rate p that maximizes the effective sample size for any known loss ratio r can be obtained by solving $\frac{\partial E[S]}{\partial p} = 0$, which leads to $p = \frac{b}{n\,r}$ (in practice, we set $p = \min(b/n\,r, 1)$). Thus, our analysis approximately doubles the sampling rate compared to the original LDA paper, i.e., sets $\alpha \approx 1$ in their notation, which yields an improvement in the effective sample size of around 20% at no cost. The conclusions of this improved analysis are coherent with the parallel analysis of [69], which also shows that the same conclusions are reached without the approximation in (2.1).

Assuming a known loss ratio and the optimal setting of the sampling rate $p = \frac{b}{n\,r}$, then, the expectation of the effective sample size is (by substitution of p in (2.1)):

$$E[S_{opt}] = \frac{1-r}{r\,e}\,b \tag{2.2}$$

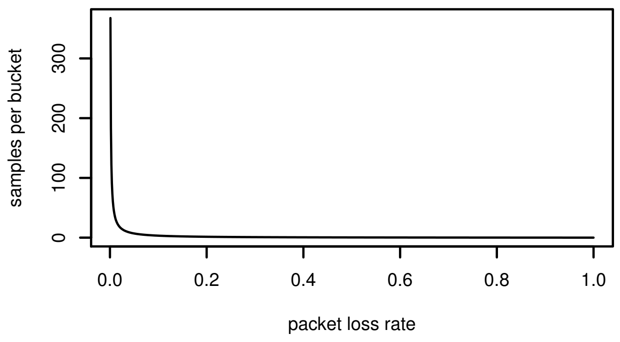


Figure 2.2: Expected number of samples collected per bucket under varying packet loss ratios, assuming an ideal LDA that can apply, for each packet loss ratio, the optimal sampling rate

In other words, given a known number of incoming packets and a known loss ratio, setting p optimally maximizes the expectation of the sample size at $\frac{1-r}{r\,e}$ samples per bucket. Figure 2.2 shows how the number of samples that can be collected by the LDA quickly degrades when facing increasing packet loss ratios. Therefore, in a high packet loss ratio scenario, the LDA will require large amounts of memory to preserve the sample size. As an example, in order to sustain the same sample size of a 0.1% loss scenario, the LDA must grow around 50 times larger on 5% packet loss, and by a factor of around 250 in the case of 20% packet loss.

Recall that this analysis assumes that the packet loss ratios are known beforehand, so that the sampling rate can be tuned optimally. When facing unknown loss ratios, the problem becomes harder, since it is not possible to configure $p = \frac{b}{n\,r}$, given that both parameters are unknown. However, this analysis does provide an upper bound on the performance of this algorithm. In any configuration of the LDA, including in multiple banks, the expectation of the effective sample size will be at most $\frac{1-r}{r\,e}\,b$.

2.3.2 Accuracy

It is apparent from the previous subsection that increasing packet loss ratios have a severe impact on the effective sample size that the LDA can obtain. However, the LDA is empirically shown to obtain reasonable accuracy up to 20% packet loss in [88]. How can we accommodate these two facts? The resolution of this apparent contradiction lies in the fact that the accuracy of the LDA does not depend linearly on the sample size but, instead, the gains in terms of accuracy of the larger sample sizes are small.

The LDA algorithm estimates the average delay μ from a sample of the overall population of packet delays. According to the central limit theorem, the sample mean is a random variable that converges to a normal distribution as the sample size (S in our notation) grows [117]. The rate of convergence towards normality depends on the distribution of the sampled random variable (in this case, packet delays).

If the arbitrary distribution of the packet delays has mean μ and variance σ^2, assuming that the sample size S obtained by the LDA is large enough for the normal approximation to be accurate, the sample mean can be considered to be normally distributed, with mean μ and variance σ^2/S, which implies that, with 99% confidence, the estimate of the average delay $\hat{\mu}$ as the sample average will be within $\mu \pm 2.576 \frac{\sigma}{\sqrt{S}}$ and, thus, the relative error will be below $\frac{2.576\,\sigma}{\mu\sqrt{S}}$.

An observation to be made is that the relative error of the LDA is proportional to $\frac{1}{\sqrt{S}}$, that is, halving the relative error requires increasing sample size by a factor of 4. A point is reached where the return of obtaining additional samples has a negligible practical impact on the relative error.

As stated, the accuracy of the LDA depends on the distribution of the packet delays, which are shown to be accurately modeled by a Weibull distribution in [108], and this distribution is used in the evaluation of the original LDA paper. Figure 2.3 plots, as an example, the accuracy as a function of the sample size (left) and as a function of the loss ratio (right) when packet delays are Weibull distributed with scale parameter $\alpha = 0.133$ and shape $\beta = 0.6$, and 5×10^6 packets per measurement interval (these parameters have been chosen consistently with [88] for comparability). It can be observed that, in practice, small sample sizes obtain satisfactory accuracies. In this particular case, 2000 samples bound the relative error to around 10%, 8000 lower the bound to 5%, and 25 times as many, to 1%.

2.3.3 Overhead

Reference [88] presents an experimental comparison of the LDA with active probing. In this section, we compare the overhead of the LDA with that of a passive measurement approach based on trajectory sampling [58] that sends a packet identifier and a timestamp for each sampled packet. As a basis for comparison, we compute the network overhead for each method per collected sample. Note that, for equal sample sizes, the accuracy of both methods is expected to match, since samples are collected randomly.

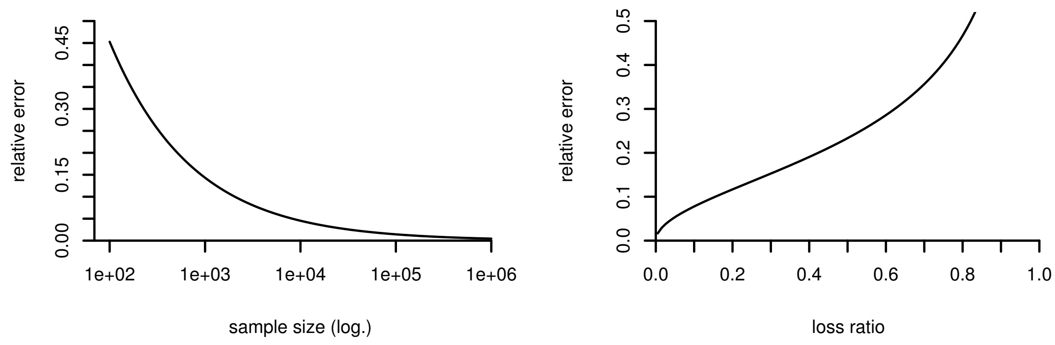


Figure 2.3: 99% confidence bound on the relative error of the estimation of the average delay as a function of the obtained sample size (left) and as a function of the packet loss ratio (right), assuming a 1024 bucket ideal LDA, 5×10^6 packets and Weibull ($\alpha = 0.133$, $\beta = 0.6$) distributed packet delays

Traditional techniques incur an overhead directly proportional to the collected number of samples. For example, an active probe will send a packet to obtain each sample. The overhead of a trajectory sampling based technique is also a constant α bytes/sample. For example, a 32 bit hash of a packet plus a 64 bit timestamp set $\alpha = 12$.

However, as discussed in the previous section, the sample size collected by the LDA depends on the packet loss ratio. Assuming a single-bank, optimally dimensioned LDA, it requires sending $b \times \beta$ bytes (where β denotes the size of a bucket) to gather $\frac{1-r}{r\,e}\,b$ samples. Thus, the overhead of the LDA is $\frac{\beta\,r\,e}{1-r}$ B/sample, and using 64 bit timestamp accumulators and 32 bit counters yields $\beta = 12$.

The LDA is preferable as long as it has lower overhead, i.e., $\frac{\beta\,r\,e}{1-r} < \alpha$ and, thus, $r < \frac{\alpha}{\beta\,e+\alpha}$. The values of α and β will vary in real deployments (e.g., timestamps can be compressed in both methods). In the example, where $\alpha = \beta = 12$, the LDA is preferable as long as $r < \frac{1}{e+1} \approx 0.27$. Figure 2.4 compares the overheads of both techniques in such a scenario, and shows the superiority of the LDA for the lowest packet loss ratios and its undesirability for the highest.

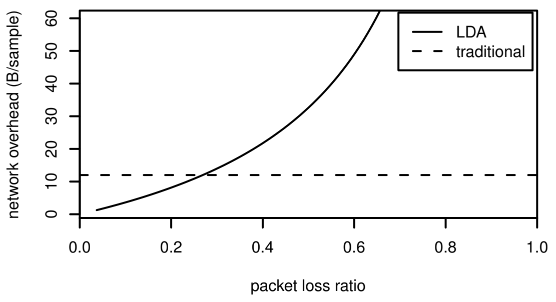


Figure 2.4: Communication overhead of the LDA relative to a traditional trajectory sampling approach, assuming 12 byte per bucket and per sample transmission costs

2.4 Multi-Bank Configuration for Unknown Packet Loss Ratios

It has already been established that the optimal choice of the LDA sampling rate is $p = \frac{b}{n\,r}$, which obtains $\frac{1-r}{r\,e}\,b$ samples. However, in practice, both n and r are unknown a priori, since they depend on the network conditions, which are generally unpredictable. Thus, setting p beforehand implies that, inevitably, its choice will be suboptimal.

What is the impact of over and under-sampling, i.e., setting a conservatively low or an optimistically high sampling rate on the LDA algorithm? We find that undersampling is preferable to oversampling. As explained, the relative error of the algorithm is proportional to $1/\sqrt{S}$. Thus, oversampling leads to collecting a high number of samples on low packet loss ratios, and slightly increases the accuracy on such circumstances, but leads to a high percentage of buckets being invalidated on high loss, thus incurring large errors. Conversely, undersampling preserves the sample size on high loss, thus obtaining reasonable accuracy, at the cost of missing the opportunity to collect a much larger sample on when losses are low, which, however, has a comparatively lower impact on the accuracy.

Figure 2.5 provides a graphical example of this analysis. In this example we consider, again analogously to [88], Weibull ($\alpha = 0.133$, $\beta = 0.6$) distributed packet delays. We compare the sample sizes and accuracy bounds obtained by different configurations of the LDA using a value of p targeted at loss ratios of 5%, 20% and 80%. All LDA configurations use $b = 1024$ accumulators. It can be observed that, in terms of sample size, the conservative setting of p for 80% loss underperforms, in terms of sample size, under the lowest packet loss ratios, but this loss does not imply an extreme degradation in terms of measurement accuracy. On the

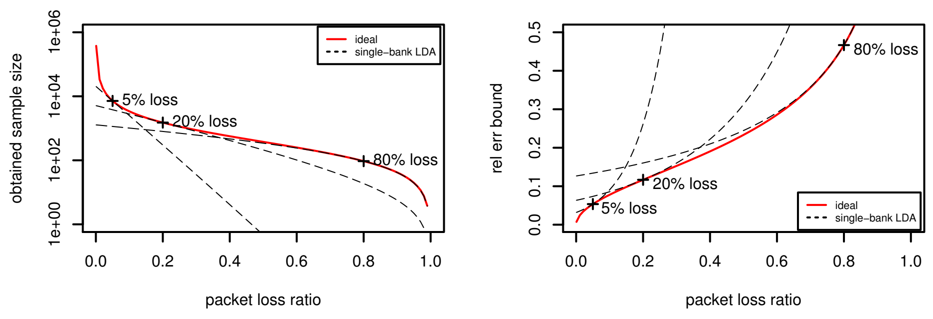


Figure 2.5: Impact on the sample size (left) and expected relative error (right) of selecting a sub-optimal sampling rate

contrary, the more optimistic sampling rate settings achieve better accuracy under low loss, but incur extreme accuracy degradation as the loss ratio grows.

2.4.1 The Multi-Bank LDA

So far, the analysis of the LDA has assumed all buckets have a common sampling rate p. However, as exposed in [88], when packet loss ratios are unknown, it is interesting to divide the LDA in multiple banks. A bank is a section of the LDA for which all the buckets use the same sampling rate. Each of the banks can be tuned to a particular sampling rate, so that, intuitively, the LDA is resistant to a range of packet loss ratios.

The original LDA paper tests three different configurations of the multi-bank LDA, always using equal (or almost) sized banks. No systematic analysis is performed on the appropriate bank sizing nor on the appropriate sampling rate for each of the banks; each LDA configuration is somewhat arbitrary and based on intuition.

We extend our analysis to the most general multi-bank LDA, where each bucket i independently samples the full packet stream at rate p_i (i.e., our analysis supports all combinations of bank configurations and sampling rates). We adapt (2.1) accordingly:

$$E[S] \approx \sum_{i=1}^{b} \frac{(1-r)\, p_i\, n}{e^{n\, r\, p_i}} \tag{2.3}$$

When every bucket uses the same sampling rate, the two equations are equivalent with $p_i = p/b$ (each bucket receives $1/b$ of the traffic and samples packets at rate p). As for the error bound, the analysis from Section 2.3.2 still holds.

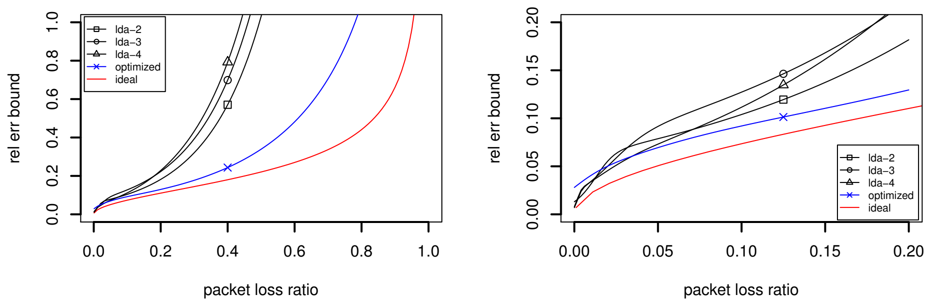


Figure 2.6: Error bounds for several configurations of multi-bank LDA in the 0-1 packet loss ratio range (left) and in the 0-0.2 range (right)

We have evaluated the three alternative multi-bank LDA configurations proposed in the original LDA paper, using the same configuration parameters and distribution of packet delays. Figure 2.6 compares the accuracy obtained by the three configurations. The figure assumes, again, a Weibull distribution for packet losses, with shape parameter $\beta = 0.6$ and scale $\alpha = 0.133$, and a number of packets $n = 5 \times 10^6$. All configurations use $b = 1024$ buckets. The first uses two banks, each targeted to 0.005 and 0.1 loss; the second, three banks that target 0.001, 0.01 and 0.1 loss; the third, four banks that target 0.001, 0.01, 0.05 and 0.1 loss. The figure shows that, in practice, the three approaches (*lda-2*, *lda-3* and *lda-4* in the figure) proposed in [88] perform very similarly, which motivates further discussion on how to dimension multi-bank LDAs. The figure also provides, as a reference, the accuracy obtained by an *ideal* LDA that, for every packet loss ratio, obtains the best possible accuracy (from (2.2)).

We argue that, consistently with the discussion of subsection 2.4, in order to support a range of packet loss ratios, the LDA should be primarily targeted towards maintaining accuracy over the worst-case target packet loss ratio. Using this conservative approach has two benefits. First, it guarantees that a target accuracy can be maintained in the worst-case packet loss scenario. Second, it is guaranteed that its accuracy over the smaller packet loss ratios is at least as good.

However, this rather simplistic approach has an evident flaw in that it does not provide significantly higher performance gains in the lowest packet loss scenarios, where a small number of high packet sampling ratio provisioned buckets would easily gather a huge number of samples. Based on this intuition, as a rule of thumb, 90% of the LDA could be targeted to a worst-case sampling ratio, using the rest of the buckets to increase the accuracy in low packet loss scenarios.

2.4.2 Multi-Bank LDA Optimization

A more sophisticated approach to dimensioning a multi-bank LDA is to determine the vector of sampling rates $< p_1, p_2, \ldots, p_b >$ that performs closest to optimal across a range of sampling rates. We have used numerical optimization to search for a vector of sampling rates such that it *minimizes the maximum difference* between the accuracies of the multi-bank LDA and the ideal LDA across a *range* of packet loss ratios. Additionally, we have restricted the search space to sampling rates that are powers of two for performance reasons [69, 88].

We have obtained a multi-bank LDA that targets a range of loss rates between 0.1% and 20% for the given scenario: 5 million packets, Weibull distributed delays, and 1024 buckets. The best solution that our numerical optimizator has found is, coherently with the previous discussion, targeted primarily to the highest loss ratios. Table 2.2 summarizes the resulting multi-bank LDA. Most notably, a majority (70%) of the buckets use $p_i = 2^{-20}$, i.e., are targeted to a packet loss ratio of 20%, while fewer (around 20%) use $p_i = 2^{-17}$, i.e., are optimized for around 2.6% loss. All buckets combined sample around 0.47% of the packets.

Figure 2.6 shows the result of this approach (line *optimized*) when targeting a range of loss rates between 0.1% and 20% for 5 million packets with the mentioned Weibull distribution of delays. The solution our optimizer found has the desirable property of always staying within below 3% higher relative error than the best possible, for any given loss ratio within the target range. These results suggest that there is little room for improvement in the multi-bank LDA parametrization problem.

In the parallel analysis of [69], numerical optimization is also mentioned as an alternative to maximize the effective sample size when facing unknown packet loss. Optimal configurations are derived using competitive analysis for some particular cases of tight ranges of target packet loss ratios $[l_1, l_2]$. In particular, it is found that both for $l_2/l_1 \leq 2$, and for $l_2/l_1 \leq 5.5$ and a maximum of 2 banks, the optimal configuration is a single bank LDA with $p = \frac{\ln l_2 - \ln l_1}{l_2 - l_1}$. We believe that our approach is more practical in that it supports arbitrary packet loss ratios and it focuses on preserving the accuracy, rather than sample size.

2.5 The Reconfigurable LDA

The core idea behind the Lossy Difference Aggregator (LDA) is, as explained, to aggregate timestamps in order to reduce the network and memory overhead associated to individual

sampling rate	2^{-14}	2^{-15}	2^{-16}	2^{-17}	2^{-18}	2^{-19}	2^{-20}	2^{-21}
#buckets	2	74	6	189	7	27	717	2

Table 2.2: Per-bucket sampling rates respective to the full packet stream of the numerically optimized LDA for the given scenario. Overall sampling rate is around 0.47%

timestamp storage and transmission. This approach, however, requires both monitors to aggregate the same set of packets. Hence, the main difficulty of this approach is how to deal with packet loss. LDA address this difficulty by splitting the packet stream into b sub-streams, and sampling incoming packets at a rate that bounds the number of lost packets that are aggregated. Choosing the best sampling rate to cope with an a-priori known number of packet losses is relatively straightforward, given the analysis of 2.3.1.

The problem of dealing with a more realistic scenario where loss is unpredictable is more challenging. Ref. [88] proposes, for this scenario, a multi-bank configuration of the LDA, where each bank is targeted at a different loss ratio, thus intuitively becoming resistant to a range of losses. In Section 2.4.1, we investigated how to obtain a multi-bank LDA configuration suited to a given scenario using numerical optimization.

In this section, we propose a new variant of the LDA data structure called Re-Configurable LDA (R-LDA) which, at the cost of requiring extra memory usage in the measurement nodes, can achieve an extremely close to optimal accuracy, even in the presence of unknown loss.

2.5.1 The R-LDA Data Structure

It has already been established (Section 2.3.1) that a LDA of b buckets operates optimally when its sampling rate is $p = b/l$ (recall that l is the absolute number of losses during a measurement epoch). The problem is simply that l is unknown a priori, when the sampling rate for the LDA has to be selected. We note, however, that the number of losses can be effortlessly obtained a posteriori, by exchanging and comparing packet counts at each monitor. Can we do better by leveraging this information? We propose a different data structure called Re-Configurable LDA (R-LDA). This data structure, unlike the single- or multi-bank LDA, does not select a fixed sampling rate initially but, instead, reduces the level of timestamp aggregation so that, once l is known, it can build a final LDA that is very close to optimal. That is, it obtains the same result as a single-bank LDA configured with $p \approx b/l$.

The question is then how to store packet timestamps in a way that, later, they can be aggregated into a final LDA of arbitrary packet sampling rate. To accomplish this objective, the R-LDA maintains k banks of b buckets each [1]. When a packet enters the data structure, it is routed to (at most) one of the banks. Every bank has an associated *selection probability* s_i, with $1 \leq i \leq k$. In particular, the ith bank is selected with probability $s_i = 2^{-i}$. In other words, the R-LDA is roughly equivalent to storing k LDAs in memory, with the ith LDA configured with $p_i = 2^{-i}$, the only difference being that the sets of packets aggregated by the banks are disjoint. The first bank then aggregates around 50% of packets, the second, 25% and so on.

Overhead. This approach does not introduce much implementation complexity: given a packet's hash, the number of leading zeros in its binary representation determines the bank that the packet will hit, while the rest of bits can be used to determine its bucket in the selected bank (i.e., requiring a hash of at least k + $log_2 b$ bits). The data structure still requires updating a single bucket per packet. This scheme does however increase memory usage by a factor of k but, as will be discussed later, k does not need to be large to support a large number of losses.

Combination of Banks. The main feature of the scheme we described is that the k banks can be easily combined into a single LDA of b buckets as desired (this paragraph explains *how* to combine banks; we will later discuss *which* should be combined). Since every bank has b buckets, banks can be trivially combined into a single LDA as follows. The ith bucket of the final LDA contains a sum of all timestamps of the ith buckets of the selected banks. Similarly, the packet count of the ith bucket is the sum of packet counts in the ith buckets of the selected banks. It is therefore easy to see that, when banks are combined, the final result is a LDA with sampling rate equal to the sum of the selection probabilities of the involved banks.

Obtaining a close-to-optimal LDA. The question of *which* banks to aggregate to obtain a close-to-optimal LDA is now addressed. Let p_{opt} be the optimal selection of the sampling rate, i.e., $p_{opt} = b/l$. The objective of bank combination is to build, from the available banks, a final LDA with sampling rate p as close to p_{opt} as possible.

The selection of banks that yields the closest approximation to p_{opt} can be found as follows. Start from the binary representation of p_{opt}, truncated to k bits after the radix point. In other words, round p_{opt} down to k bits after the radix point; let the resulting value be p'. Repeat the

[1]Note that this approach differs from the multi-bank LDA, which maintains a total b buckets in memory.

procedure but this time rounding up; let the resulting value be p''. Hence, $p' \leq p_{opt} \leq p''$. A LDA can be obtained for either p' or p'' as follows. The ith bank is selected if the ith bit after the radix point is set. Then, the selected banks are combined into a single LDA. We have found the two closest to p_{opt} sampling rates that this data structure can yield. To choose among the two, we calculate the expectation of the sample size that each one offers from Eq. 2.1, and use the one that yields the largest. Let the selected sampling rate be $p \in \{p', p''\}$, with $p' \leq p_{opt} \leq p''$. Since $p'' - p' = 2^{-k}$, this scheme guarantees that $|p - p_{opt}| \leq 2^{-k}$.

In particular, in the case of no loss, this scheme obtains $p_{max} = 1 - 2^{-k} \approx 1$. That is, in a zero loss scenario, it will only miss an expected $2^{-k}n$ packets (out of a total n). Conversely, the minimum sampling rate the R-LDA can provide is $p_{min} = 2^{-k}$. This implies that an R-LDA with k banks and b buckets can support up to $2^k b$ packet losses. That is, the number of losses that R-LDA supports doubles for every additional bank, and hence the number of banks does not need to be very large.

2.6 Validation

In the previous sections, we derived formulas for the expected effective sample size of the LDA when operating under various sampling rates, and provided bounds for the expected relative error under typical distributions of the network delays. In this section, we validate the analytical results through simulation.

We have chosen the same configuration parameters as in the evaluation of [88]. Thus, this section not only validates our analysis of the LDA algorithm, but also shows consistency with the previous results of [88]. The simulation parameters are as follows: we assume 5 million packets per measurement interval, and Weibull ($\alpha = 0.133$, $\beta = 0.6$) distributed packet delays. In our simulation, losses are uniformly distributed. Note however that, as stated in [88], the LDA is completely agnostic to the packet loss distribution, but only sensitive to the overall packet loss ratio. Thus, other packet loss models (e.g., in bursts [124]) are supported by the algorithm without requiring any changes.

Figure 2.7 (left) compares the expected sample sizes with the actual results from the simulations. The figure includes the three multi-bank LDA configurations introduced in [88], with expected sample size calculated using (2.3), and the *ideal* LDA that achieves the best possible accuracy under each packet loss ratio, obtained from (2.2). This figure validates our analysis

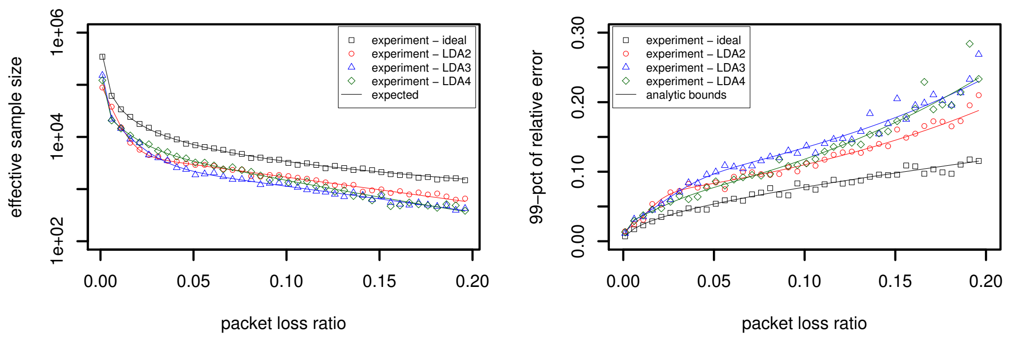


Figure 2.7: Effective sample size (left) and 99 percentile of the relative error (right) obtained from simulations of the LDA algorithm using 5 million packets per measurement interval, and Weibull distributed packet delays

of the algorithm, since effective sample sizes are always around their expected value (while in [88], only a noticeably pessimistic lower bound is presented).

On the other hand, Figure 2.7 (right) plots the 99 percentile of the relative error obtained after 500 simulations for each loss ratio, and compares it to the 99% bound on the error derived from the analysis of Section 2.3.2. The figures confirm the correctness of our analysis for both the effective sample size and the 99% confidence bound on the relative error.

Figure 2.8 shows the performance of the Reconfigurable LDA (R-LDA) described in Section 2.5, with 8 to 10 banks (again, in a scenario with 5 million packets and Weibull-distributed delay). In order to support up to 20% loss, the R-LDA can be configured as follows. At most, 1 million packets will be lost. According to the analysis of Section 2.5, and assuming a target LDA size of 1024 buckets, adding k banks will protect against $1024 * 2^k$ losses. Hence, 10 banks suffice to cover 1 million losses.

In Figure 2.8 (left) we observe the drop in sample size relative to an optimal setting for various loss ratios. It can be observed that, as expected, the more banks, the higher loss ratios that are supported. Only when the loss ratio is beyond the target one, sample size is significantly lower than optimal (notice the scale of the y axis). How does this effect measurement accuracy? Figure 2.8 (right) translates these drops in sample size to the expected accuracy bound. It can be observed that, in practice, these small drops in accuracy are irrelevant in terms of accuracy. It is only when the number of packet loss is well beyond the expected that accuracy degrades (i.e., the relative error bound increases). The figure omits the accuracy of the ideal LDA, because

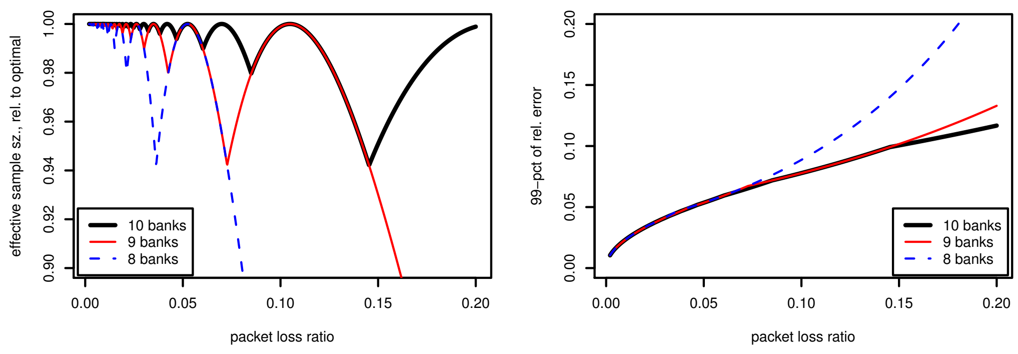


Figure 2.8: Sample size (left) and Reconfigurable LDA

it is visually indistinguishable from the 10-bank R-LDA. This shows how a small number of banks is enough to achieve close-to-optimal accuracy in practical scenarios.

2.7 Experiments

In the previous section, a simulation based validation of our analysis of the LDA has been presented that reproduces that of [88]. In this section we evaluate the algorithm using real network traffic. To the best of our knowledge, this is the first work to evaluate the algorithm in a real scenario.

Our scenario, which is more extensively described in Appendix A, consists of two measurement points: one of the links that connect the Catalan academic network to the rest of the Internet, and the access link of Universitat Politècnica de Catalunya (UPC) to said Catalan academic network. In the first measurement point, a monitor obtains a copy the of the inbound traffic via an optical splitter, and filters for incoming packets with destination address belonging to UPC. In the second measurement point, a monitor analyzes a copy of the traffic that enters UPC.

2.7.1 Deployment Challenges

The deployment of the LDA in a real world scenario presents important challenges. The design of the LDA is built upon several assumptions. First, as stated in [88], the clocks in the two measurement points must be synchronized. We achieve this effect by synchronizing the internal DAG clocks prior to trace collection. Second, packets in the network follow strict

FIFO ordering, and the monitors can inject control packets in the network (by running in the routers themselves) which also observe this strict FIFO ordering, and are used to signal measurement intervals. In our setting, packets are not forwarded in a strict FIFO ordering due to different queueing policies being applied to certain traffic. Moreover, injecting traffic to signal the intervals is unfeasible, since the monitors are isolated from the network under study.

Third, in the original proposal, the complete set of packets observed in the second monitor ($receiver$) must have also been observed in the first ($sender$). In [88], the LDA algorithm is proposed to be applied in network hardware in a hop-by-hop fashion. However, this assumption severely limits the applicability of the proposal; for example, as is, it cannot be used in our scenario, since $receiver$ observes packets that have been routed through a link to a commercial network that $sender$ does not monitor (we refer to these packets as *third party traffic*). This limitation could be addressed by using appropriate traffic filters to discern whether each packet comes from $receiver$ (e.g., source MAC address, or source IP address), but in the most general case, this is not possible. In particular, in our network, we lack routing information, and traffic engineering policies make it likely that the same IP networks are routed differently.

The problem lies in that the LDA counters might match by chance when, in $receiver$, packet losses are compensated by extra packets from the third party traffic. The LDA would assume that the affected buckets are usable, and introduce severe error. We work around this by introducing a simple extension to the data structure: we attach to each LDA bucket an additional memory position that stores an XOR of all the hashes of the packets aggregated in the corresponding accumulator. Thus, $receiver$ can trivially confirm that the set of packets of each position matches the set of packets aggregated in $sender$ by checking this XOR. From a practical standpoint, using this approach makes third party traffic count as losses. We use 64 bit hashes and, thus, the probability of the XORs matching by chance is negligible[1].

2.7.2 Experimental Results

We have simultaneously collected a trace in each of the measurement points in the described scenario, and wrote two CoMo [31] modules to process the traces offline: one that implements the LDA, and another that computes the average packet delays exactly. The traces have a duration of 30 minutes. We have configured 10 seconds measurement intervals, so that the average number of packets per measurement interval is in the order of 6×10^5.

[1]The XORs of the hashes have to be transmitted from $sender$ to $receiver$, causing extra network overhead. Choosing the smallest hash size that still guarantees high accuracy is left for future work.

We have tested 16 different single-bank configurations of the LDA with $b = 1024$ buckets and sampling rates ranging from 2^0 to 2^{-15}. Also, we have used our numerical optimizator to obtain a multi-bank LDA configuration that tolerates up to 80% loss in our scenario. We have omitted the Reconfigurable LDA (R-LDA) from this experimental evaluation, since, as pointed out in the previous Section, it obtains equivalent results to an optimally configured LDA, at the cost of requiring additional memory (in our setting, 10 times as much).

As noted in the previous discussion, third party traffic that is not seen in $sender$ is viewed as packet losses in $receiver$. Therefore, our LDAs operate at an average loss rate of around 10%, which roughly corresponds to the fraction of packets arriving from a commercial network link that $sender$ does not monitor.

Hence, the highest packet sampling ratios are over-optimistic and collect too much traffic. It can be observed in Figure 2.9 (right) that sampling ratios from 2^0 to 2^{-4} lose an intolerable amount of measurement intervals because all LDA buckets become unusable. Lower sampling rates, though, are totally resistant to the third party traffic.

Figure 2.9 (left) plots the results in terms of accuracy, but only including measurement intervals not lost. It can be observed that 2^{-6} and 2^{-7} are the best settings. This is consistent with the analysis of Section 2.3, that suggests using $p = \frac{b}{n\,r} \approx 0.17 \approx 2^{-6}$. The figure also includes the performance of our numerically optimized LDA, portrayed as a horizontal line (the multi-bank LDA is a hybrid of the other sampling rates). It performs very similarly to the best static sampling rates. However, it is important to note that this configuration will consistently perform close to optimal when losses (or third party traffic) grow to 80%, obtaining errors below 50%, while the error bound for the less flexible single bank LDA reaches 400%.

On average, for each measurement interval, the optimized LDA collected around 3478 samples, while transmitting 1024×20 bytes (8 for the timestamp accumulators and the XOR field plus 4 for the counter for each bucket), resulting in 5.8 B/sample of network overhead. A traditional technique based on sampling and transmitting packet timestamps would cause a higher overhead, e.g., if using 8 byte timestamps and 4 byte packet IDs, it would transmit 12 B/sample. Thus, in this scenario, the LDA reduced the communication overhead in over 50%.

2.8 Concluding Remarks

We have performed a validation on the Lossy Difference Aggregator (LDA) algorithm originally presented in [88]. We have improved the theoretical analysis of the algorithm by pro-

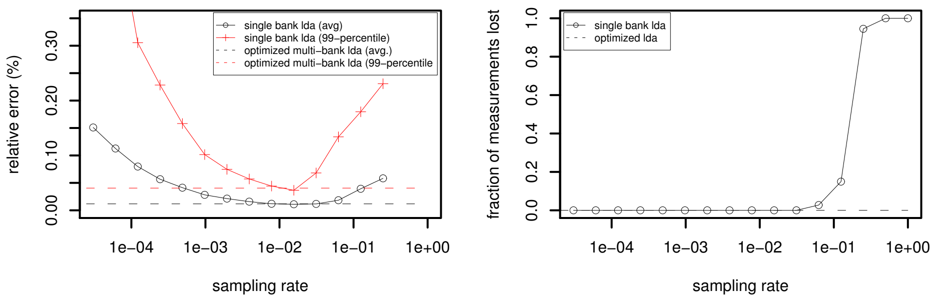


Figure 2.9: Experimental results

viding a formula for the expected sample size collected by the LDA, while in [88] only a pessimistic lower bound was presented. Our analysis finds that the sampling rates originally proposed must be doubled.

Only three configurations of the more complex multi-bank LDA were evaluated in [88]. We have extended our analysis to multi-bank configurations, and explored how to properly parametrize them, obtaining a procedure to numerically search for multi-bank LDA configurations that maximize accuracy over an arbitrary range of packet losses. Our results show that there is little room for additional improvement in the problem of multi-bank LDA configuration.

Additionally, we have introduced a new variant of this data structure called Reconfigurable LDA (R-LDA) that is capable of obtaning a near-optimal LDA for any target range of loss ratios. R-LDA is much simpler to configure than a multi-bank LDA, at the cost of increased memory requirements.

We have validated our analysis through simulation and using traffic from a monitoring system deployed over a large academic network. The deployment of the LDA on a real network presented a number of challenges related to the assumptions behind the original proposal of the LDA algorithm, that does not tolerate packet insertion/diversion and depends on strict FIFO packet forwarding. We propose a simple extension that overcomes such limitations.

We have compared the network overhead of the LDA with pre-existing techniques, and observed that it is preferable under zero to moderate loss or addition/diversion of packets (up to $\sim$25% combined). However, the extra overhead of pre-existing techniques can be justified in some scenarios, since they can provide further information on the packet delay distribution (e.g., percentiles), than just the average and standard deviation that are provided by the LDA.

3

Per-Flow Delay Measurement

Packet delay is a crucial performance metric for real-time, network-based applications. Obtaining per-flow delay measurements is particularly important to network operators, but also computationally challenging in high-speed links. Passive delay measurement techniques have been proposed that outperform traditional active probing in terms of accuracy and network overhead, such as the Lossy Difference Aggregator (LDA) that is discussed in depth in Chapter 2. An important limitation of LDA is that it only provides aggregate measurements for all packets, and can not obtain per-flow delay. Recently, per-flow delay measurement techniques have been proposed that rely on the empirical observation that packet delays across different flows are temporally correlated. However, this assumption is not met in presence of traffic prioritization, load balancing policies, or due to intricacies of the switch fabric.

This chapter presents a novel data structure called Lossy Difference Sketch (LDS) that provides per-flow delay measurements without relying on any specific delay model. LDS obtains a notable accuracy improvement compared to the state of the art with a small memory footprint and network overhead. The data structure can be sized according to target accuracy requirements or to fit a low memory budget.

3.1 Introduction

Packet delay has become a key network performance metric, together with other metrics such as throughput and packet loss. This growth in importance of packet delay is mainly due to the emergence of a new class of network-based applications that demand extremely low end-to-end latency. For instance, algorithmic trading applications require end-to-end latencies to not exceed few microseconds, otherwise they may lose significant amount of revenue in the form of lost arbitrage opportunities [100]. High-performance computing applications form another class of such applications with message latencies directly impacting the amount of time it takes for the job (e.g., weather simulation) to be finished. Finally, modern data center applications have soft real-time deadlines [22] that typically are in the order of milliseconds, but once backend computation requirements are factored in, very little time is left for network delays.

Now, consider a network operator that is running and managing a network environment that supports low-latency applications, such as a data center network. Typically, many data centers host several thousands of machines connected via a network fabric that is often constructed out of commodity networking equipment (e.g., switches and routers). Depending on the requirements (e.g., full bi-section bandwidth), the network is often connected in a multi-rooted tree topology (e.g., a fat-tree) with several thousand switches providing multiple paths between servers for load-balancing purposes. Further, the cluster itself may be shared across several tens to hundreds of customers running tens to hundreds of different applications with potentially very different network usage patterns. Given the complexity stemming from the sheer number of network elements as well as the variety of networking-based applications, it becomes extremely difficult to debug and troubleshoot latency anomalies (such as delay spikes) throughout the network *without* proper latency measurement tools at various points in the network.

Traditionally, such measurements have been obtained using active probing in wide-area ISP networks [33, 44, 110, 125]. However, end-to-end network delays are an order of magnitude smaller in data center networks—order of *microseconds* compared to milliseconds. To capture delay dynamics at such microsecond granularity, high probing frequency (e.g., 10,000Hz [88]) is required, which makes this approach prohibitively expensive in practical scenarios. Further, diagnosing end-to-end delay anomalies requires measurements at various vantage points in the network—ideally, at each pair of interfaces within each switch in the network, so that the root cause can be localized down to a router or a switch. The network operator could then conduct a more extensive analysis, such as study the set of customers or applications that may be routed through that switch to carefully investigate the root cause of the problem.

Unfortunately, native switch/router support for packet delay measurements is sorely lacking. Today, NetFlow and SNMP form the two main measurement solutions that a router supports. Neither, however, focuses on delay measurements. In some environments such as the

London Stock Exchange, operators resort to specialized measurement boxes (e.g., Corvil [4]) that can detect these delays at high fidelity. However, because of the high costs and the hassles of administering a new box in the network, such an approach does not scale well. The complexity of packet latency measurements comes fundamentally from the fact that we cannot easily just store a packet timestamp at two monitoring points, without incurring high storage and communication complexity, since the complexity is linear in the number of packets. It is therefore important to overcome the linear relationship between number of collected timestamps and network overhead for any solution to be scalable.

Recent work [88] proposed the lossy difference aggregator (LDA) to overcome the linear relationship between sample size and network overhead by intelligently aggregating timestamps between the two measurements points. LDA, however, provides only aggregate latency estimates *across* all packets, which may be inadequate for diagnosing customer-specific or application-specific latency issues [92]. As pointed out by prior work [92], flows may exhibit significant diversity in their latency characteristics at a given router, and hence, per-flow measurements are important for network operators. Unfortunately, the problem of measuring per-flow delay is harder in the environments we consider, since the number of flows can be quite large; collecting and exchanging per-flow state becomes prohibitively expensive.

The problem of measuring per-flow delay has been very recently explored in [92, 93]. Both papers exploit the *key observation* that packets exhibit significant temporal delay correlation in specific settings, i.e., packets that are transmitted close in time experience similar delays, even if they do not belong to the same flow. RLI [92], the most recent of the two, exploits this observation to inject simple active probes periodically and uses linear interpolation to estimate per-packet delay. At the downstream monitoring point, these estimated per-packet delays can be aggregated into per-flow latencies using only three counters per-flow.

While the assumption that packets exhibit temporal correlation is valid in a restricted subset of systems, this assumption *does not* hold true in more general scenarios where there is prioritization across packets with two or more parallel queues. For example, many modern routers support different queuing for prioritizing real-time traffic (e.g., VoIP, video) over regular data transmissions (e.g., Web). Thus, in these cases, there exists very little correlation between the delays of packets that end up traversing two different queues. Similarly, in many modern data center networks, packets are routinely load balanced across multiple paths using ECMP—temporal delay correlation may potentially exist across any given path but *certainly not* across paths. Finally, modern switch fabrics (e.g., Clos-network-based switch fabrics used in Juniper's T-Series routers [9]) are often composed of intermediate stages of switching with each packet being sent to a random intermediate location; the latency of a packet through the router may be different depending on the path within the router. (In such switches, packets are re-sequenced back because TCP does not interact well with reordering, but such reordering needs to be only on a per-flow basis.)

Thus, the assumption of temporal delay correlation is not universally applicable; unfortunately, schemes such as RLI will produce grossly inaccurate latency estimates if the assumption does not hold, posing a major hurdle for deploying RLI on a global basis. Switch vendors do not wish to be bothered about the specifics of the deployment scenario, and instead would like to have one scheme that is universally applicable across *all* possible scenarios. Our objective in this work is to accomplish this task. Specifically, we focus on devising a scalable *delay-model-agnostic mechanism* to obtain per-flow latency measurements at microsecond granularity across two measurement points in the network.

We propose a technique called lossy delay sketching (LDS) that essentially combines the model independence nature of LDA with sketching techniques that do not rely on per-flow state to obtain model-free and scalable per-flow delay estimation. LDS essentially maintains a series of hash buckets, with each bucket consisting of a timestamp sum and the number of packets that hash to the bucket (similar to an LDA bucket). In accordance with the spirit of sketching, LDS maps each flow to a random *subset* of buckets, that are potentially shared (partially or fully) by other flows. To minimize the effect of interference, we randomize the fate-sharing by maintaining different banks of buckets, similar to a sketch, with a different hash function.

While the basic idea of blending LDA with sketches makes intuitive sense, several problems must be overcome to design such a data structure. For instance, flows may differ in their delay properties as well as their sizes significantly. It is important to ensure the interference due to collisions does not impact the accuracy of the flow's latency estimates. We present theoretical analysis on determining the size of LDS in order to reduce this interference.

Thus, the main contributions of this work are as follows.

- We propose a new data structure called LDS that *obtains per-flow delay estimates* and that does not rely on delay models (Section 3.2). It blends LDAs that are model independent with sketching techniques that provide per-flow measurements without per-flow state.

- We present a comprehensive theoretical analysis of the data structure and show how to size it to achieve the desired accuracy (Section 3.3).
- We introduce a series of practical enhancements to LDS that allow network operators to fine-tune the data structure for the specifics of an actual deployment scenario (Section 3.4).
- We evaluate LDS with real traffic collected at a large academic network (Section 3.5). Our results indicate that sketching is superior to existing techniques when temporal correlation is not present. Sketching is particularly accurate for large flows, even in the presence of loss. Additionally, the accuracy of a selected subset of flows can be easily incremented.

Finally, Section 3.6 covers the related work in the literature, while Section 3.7 concludes the chapter.

3.2 Delay Sketching

Our main goal is to measure the one-way delay introduced by a network between two measurement points on a per-flow basis. While we can typically support any definition of flow, usually, this will consist of the 5-tuple formed by source and destination IP addresses, originating and destination ports, and protocol. We mainly focus on obtaining per-flow average latency, but we also outline in Section 3.4 how we can obtain second moments as well.

Our architecture is oblivious to what locations exactly constitute the measurement points. Thus, we can imagine obtaining per-flow measurements within a switch or a router across an ingress and egress interface. Alternately, we can obtain measurements across two different routers. Note that both measurement locations are merely viewpoints along the path that packets follow, and do not need to be (although they could be) the emitter or final destination of the traffic. We call the first measurement point *sender*, and the second, *receiver*. We consider the reverse path measurements separately with the *receiver* becoming the *sender* and vice versa.

3.2.1 Assumptions

Single stream. We assume that the sender and receiver observe the same stream of packets. In general, this is highly dependent on the particular scenario. For instance, suppose we consider an ingress (egress) switch interface as the sender (receiver). The receiver (sender) may obtain (transmit) packets from (to) many different ingress interfaces. Thus, we assume there is a simple way to filter out the packets that travel from the sender to the receiver. Note that we do not assume packets flow through a single queue, or even in a FIFO order—just that we have a way to separate out packets that appear at both the sender and the receiver. Within switches, there are often internal headers that contain the port at which they originated and the port to which they are headed to, that we can use for this purpose. Across routers, we can leverage prefix-based filtering to identify the set of packets that travel through one given path (forwarding is prefix-based). Such routers do not need to be co-located or close in terms of network hops.

Packet loss. We assume packets can be lost between the sender and receiver. Depending on the scenario, the packet loss rates may differ significantly. For example, in a financial trading network, we may imagine the network to suffer from minimal packet loss. However, in a real backbone network, packet loss may be slightly more common. Typically, while some amount of loss resilience is required in our data structures, we assume the loss rates are still quite low (say $<1\%$) as TCP may not work well under higher loss rates.

Time synchronization. We also assume the clocks of the sender and receiver are synchronized. This is a common requirement of one-way packet delay measurement techniques [88, 92, 93]. Although techniques have been proposed that do not require clock synchronization (e.g., [103, 111, 132]), removing this assumption was out of the scope of this work. To achieve fine grained precision, packet timestamping clocks can be synchronized to the GPS signal (i.e., using Endace DAG cards), or using the IEEE 1588 protocol [21]. Both these methods are capable of sub-microsecond precision and thus suitable for our needs [54]. (In our evaluation, we obtained traces from a production network that already uses IEEE 1588 protocol to synchronize measurements.)

Embedding timestamps in packets. Similar to prior work [88, 92], we assume that it is *not* possible to embed timestamps within IP packets because existing IP headers do not have a placeholder for timestamps, and it would require significant changes to router forwarding paths and other third-party components making it difficult. We note however that our solutions are important even in the context where router vendors can put a timestamp in a packet, as the number of flows may be still large.

In fact, for ease of exposition, we present a simple data structure called SDS using the *timestamp assumption*, i.e., assuming packets can be embedded with timestamps. We will, however, get rid of this assumption in Section 3.2.3 when we describe our main data structure LDS.

3.2.2 Simple Delay Sketch (SDS)

As mentioned before, we initially assume the sender can embed a timestamp into the packets to be measured for easy exposition of the delay sketching idea. (We will relax this in the next section.) Thus, the receiver can obtain delays for each packet, but still need a scalable mechanism to store per-flow latencies, which is obtained by the data structure we describe in this section. The main idea of our technique is to explore sketching techniques that have been studied before in the literature to obtain measurements without maintaining per-flow state, and requiring very few memory accesses per packet. Such techniques will allow us to compute a comparatively smaller compressed summary of the traffic that allows recovering approximate measurements. We assume measurements are performed in fixed time intervals, which we refer to as *measurement intervals*.

A canonical sketch data structure that we can exploit in our setting is the multi-stage filter [64]. In this data structure, each stage has C associated counters, which are initialized to zero. Then, for each incoming packet, a hash of its flow identifier is used to determine which counter will be updated in the 1st stage. If, for example, one wishes to measure flow sizes, then the packet size is added to that counter. Since every flow always hashes to a particular position, its associated counter can be queried to obtain an upper bound on its size (only an upper bound, since other flows can hash to the same position, i.e., can collide). Additional stages can then be added that are independent replicas of this scheme, thus randomizing collisions. Then, the estimated size of a given flow is the minimum of all of its associated counters in each stage. The Count-Min Sketch [50] is also based on a similar approach.

Our initial idea is to use this sketching technique for per-flow delay measurement. The data structure we propose called Simple Delay Sketch (SDS) contains a series of cells organized in a matrix of R rows and C columns. Each row r has an associated pseudo-random hash function h_r that returns a value in the range $[0, C-1]$. Each cell of the matrix contains a tuple of values $<s,n>$, with s storing the sum and n the number of packets that hash to that cell. The data structure is graphically depicted in Figure 3.1.

Update. When a packet that belongs to a flow with identifier f arrives, for each row r, a position in the matrix is determined using its hash function h_r, which yields position $(r, h_r(f))$. Then, the cells in these positions are updated as follows. The s values of each cell are increased

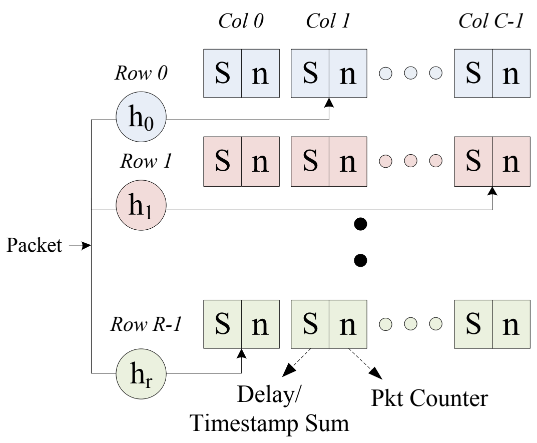


Figure 3.1: Basic data structure. In each cell, s stores the sum and n the number of packets that hash to that cell.

Algorithm 1 SDS – Per-packet operations

```
1: procedure UPDATE STATE($flow$, $\tau$)
2:     for i=1, R do
3:         $j \leftarrow (hash(i, flow)\%C)$                ▷ Compute $i$th hash
4:         $SDS[i][j].S \leftarrow SDS[i][j].S + \tau$
5:         $SDS[i][j].N \leftarrow SDS[i][j].N + 1$
6:     end for
7: end procedure
```

by the delay of the packet (i.e., the current time at *receiver* minus the timestamp embedded in the packet at *sender*), while n values are increased by one (i.e., maintains a count of the packets that hashed to that cell). In other words, s values contain the sum of all packet delays that hit that cell, while n values represent packet counts. Note also that the per-packet cost of this measurement scheme is, for each row, a hashing operation and two counter updates. The full algorithm is described in Algorithm 1.

Delay Estimation. If the data structure were single-row and infinitely large, and the hash functions were perfectly random, each non-empty cell would measure the average delay of the packets of a particular flow. That is, the data structure would be collision-free, since two flows would not hash to the same position. Therefore, to obtain the (exact) average delay of flow with identifier f, one would simply divide the s and n values of cell $(0, h_0(f))$.

Algorithm 2 SDS – Delay estimation algorithm

```
 1: procedure ESTIMATE DELAY($flow$, $SDS$)
 2:     $N_{min} = \infty$
 3:     for i=1, R do
 4:         $j \leftarrow hash(i, flow)$                    ▷ Compute $i$th hash
 5:         if $SDS[i][j].N < N_{min}$ then
 6:             $N_{min} = SDS[i][j].N$
 7:             $S_{min} = SDS[i][j].S$
 8:         end if
 9:     end for
10:     return $S_{min}/N_{min}$
11: end procedure
```

However, in practice, rows cannot be large enough, and there is a non-zero probability of several flows colliding (i.e., hashing to the same cell in one or more rows). When several flows collide, the value obtained by dividing the s and n values is a weighted average of the delays experienced by said flows, where weights correspond to the number of packets of each flow. Therefore, in practice, flow delays can not be exactly obtained, but only estimated from the data structure. To minimize the impact of collisions over measurement accuracy, it is desirable to have a large number of rows. For a given flow, we can obtain one estimate of its average delay for each of the rows, to then choose the one that is least contaminated by other flows.

Several strategies can be devised to obtain the estimates. For example, one could produce a final estimate by combining several cells, such as taking the mean or the median of the available estimates, like other sketching techniques do (e.g., see [50]). In our case, using the average would be problematic for flows that collided with larger ones in *any* of the cells, because the weight they would carry in their estimates would be small. This would invalidate the advantages of provisioning multiple rows to randomize collisions. Likewise, using the median would tend to drag each estimate towards the median delay of all flows.

We find it best to choose the cell that has the lowest n value to produce an estimate. This has two interesting properties. First, if one of the cells is collision free, the algorithm will choose it and, thus, produce an error-free result. Otherwise, it will pick the cell where the fewest amount of packets have collided, i.e., the one where the measured flow carries the highest possible weight. The full algorithm is described in Algorithm 2.

Of course, the cell with the smallest n will not necessarily produce the best estimate among all cells. For example, it may have collided, in one cell, with a large flow that has an extremely similar average delay. However, this strategy always chooses, among all the cells, the one where the measured flow carries the highest possible weight. In Section 3.3, we analyze the accuracy that this data structure offers, and provide guidelines to dimension and parametrize it.

The SDS data structure we presented in this section is a first-cut approach to blending the ideas of LDA with sketching techniques. However, SDS is only applicable if we assume embedding timestamps within packets—an assumption which is hard to achieve in practice, at least in the short term if not in the longer term. We now study a new data structure LDS that relaxes this assumption.

3.2.3 Lossy Difference Sketch (LDS)

In this section, we discuss our main data structure called lossy difference sketch (LDS) to obtain per-flow latency measurements *without* requiring the timestamp assumption. The LDS data structure starts with the basic SDS data structure, and uses the following ideas to make it practical:

1. To get rid of the timestamp assumption, the sender and receiver maintain separate copies of the data structure (described in the previous section), and the sender periodically transmits its copy to the receiver. The receiver then post-processes both sketches to obtain the delays of all packets that hash to each cell (Section 3.2.3.1). We need the sender and receiver to use consistent hashing (same hash function) to ensure packets hash to the same cell in both cases.

2. Since packet losses and reordering can occur between the sender and receiver, this may make the cells inconsistent across the sender and receiver. We detect losses easily since the number of packets does not match across the sender and receiver cells. We detect reordering using a separate field in each cell that stores a stream digest for each cell, as proposed in Chapter 2, and similarly to prior work [94] (Section 3.2.3.2).

3. To handle packet losses, we map packets that belong to the given flow across several contiguous cells in essence forming a virtual LDA for each flow. We also use a stage of sampling to reduce the probability of a packet loss completely corrupting all cells for a given flow. We randomize the set of flows that collide in each row so as to randomize

the fate-sharing. This randomization allows us to minimize the interference of other colliding flows on the estimates for any particular flow (Section 3.2.3.3).

The following subsections will discuss these ideas in detail.

3.2.3.1 Removing the Timestamp Assumption

The main problem when measurement points cannot embed a timestamp in the packets is that we cannot compute the packet delay at *receiver*. Thus, neither can we directly aggregate delays in the data structure as described in Section 3.2.2. However, we can achieve the same effect by using a simple, yet extremely powerful technique introduced in [88]. Reference [88] states that to measure average delay of a set of packets, one does not need to embed a timestamp in the packet or transmit individual timestamps between *sender* and *receiver*. Instead, we can proceed as follows. In both measurement points, compute the sum of all packet timestamps, and maintain a packet count. Then, to compute average delay, compare the aggregate timestamps and divide over the total number of packets.

We leverage this idea in designing LDS as follows: In both measurement nodes, we maintain a sketch as the one described in Section 3.2.2, where each position aggregates the packet timestamps observed at each point, instead of packet delays. Both measurement nodes use the same hashing functions in order to map flows to the same counter positions. Then, at the end of the measurement interval, one of the resulting sketches is sent to the other measurement point. The aggregate timestamps at *sender* are subtracted from those at *receiver*. The result is exactly equivalent to a sketch that aggregates packet delays.

After this step, the average delay of the packets that hit a cell can be simply obtained by dividing its s and n, where s is now the sum of timestamps kept in each cell, instead of the aggregate packet delays. Assume a series of packets $p_1, p_2 \dots p_k$, with timestamps $t_1, t_2, \dots t_k$ at *sender* and $t'_1, t'_2 \dots t'_k$ at *receiver*. The delay of the ith packet is then $t'_i - t_i$. Our data structure calculates $\frac{s}{n} = \frac{\sum_{i=0}^{k} t'_i - \sum_{i=0}^{k} t_i}{k} = \frac{\sum_{i=0}^{k} t'_i - t_i}{k}$, i.e., the average packet delay.

Note that, while we focus on average delay estimation, other useful estimates, including per-flow delay standard deviation, can also be mined from the data structure as described later in Section 3.4.

3.2.3.2 Detecting Losses and Reordering

When packets are lost some of the n fields in *receiver* would be smaller than those obtained by *sender*. This is an important problem, because our data structure relies on the difference of s values at *sender* and *receiver* to calculate average packet delays, as explained in Section 3.2.2; using cells where the set of packet delays aggregated in each measurement point differs introduces severe error [88]. In general, when packet counts n do not match, the set of timestamps aggregated in the corresponding s fields will not be consistent. Thus, in LDS, we do not use such cells for delay measurement.

Packet reordering poses an additional challenge: n fields can match, while the set of aggregated packet timestamps might differ. This is a problem that is analyzed in more detail in [94]; in summary, at the boundaries of measurement intervals, packets might jump to the next (or previous) interval. This, coupled with loss, can easily cause matching packet counts, and mismatching sets of packet timestamps. As introduced in Chapter 2, and further investigated in Reference [94], this problem can be solved by attaching, to each sketch position, a small digest d of the packets that hit such cell, which can be achieved simply with an extra hashing operation as follows.[1]

In LDS, thus, each cell will consist of the additional d field along with s and n for each cell. This value contains a digest of all packets that hashed to a cell. We require digests to be computationally lightweight, and to provide a means to detect loss and packet reorders with high probability. An easy way to achieve this is to cumulatively XOR the hashes of the packet contents. This scheme guarantees that, using b bit hashes, mismatches will be detected with probability $1 - 2^{-b}$.

It is easy to see that, with this basic data structure, no accurate delay estimates can be produced for (i) those flows that experience even a single loss or reordering, and (ii) for those that collide with flows that have experienced such conditions. For such flows, packet digests (and often, n values) in *sender* will not match those at *receiver*, and thus will always be invalidated. We next discuss the mechanism used in LDS to make the flow estimates more robust to packet losses or reordering.

[1]In the final data structure described in Section 3.2.3.4, the hash value provided by h' is used, thus saving this extra hashing operation.

3.2.3.3 Robustness against Loss and Reordering

To make LDS more robust to losses and reordering, we leverage the basic idea used in LDA. (Although LDA is discussed in depth in Chapter 2, we briefly review its main characteristics to make this chapter self-contained.) The main idea of LDA consists of partitioning each flow's packets into k sub-streams, and mapping each packet to one of k different cells *in every row*. To coordinate both measurement nodes, the cell that a given packet will hit is also determined by a hash function, although this time of the full packet instead of only headers, so that successive packets of the same flow are scattered across the k cells.

This way, if one cell is hit by lost or reordered packets, only a subset of packets that belong to the flow are thrown away. The remaining cells to which the flows' packets are mapped will possibly remain intact, thus providing with reasonable estimates for the flow. To reduce the chances of losing cells to losses and reordering, we additionally place a packet sampling stage that will reduce the absolute number of packet losses, but also reduce the number of good estimates (similar to LDA). As analyzed in detail in [69], and further discussed in Chapter 2, the LDA operates optimally when the sampling rate p is set to L/k, where L corresponds to the absolute number of losses in the packet stream. In Section 3.4.2 explores how to configure sampling rates in real deployments.

Now, we have two choices for the k cells. First, we can essentially replace each counter in the SDS data structure with k different cells. Second, we can allocate k (contiguous or random) cells in the counter matrix independently to each flow. If the k cells are contiguous, we can think of them as overlapping virtual LDAs (vLDA) per-flow, but they are not dedicated per-flow. In LDS we use this second approach, since it has the advantage of allowing a larger k without reducing the number of cells, thus increasing the robustness against loss at the cost of introducing extra collisions in the data structure.

3.2.3.4 Final Data Structure

The final data structure LDS is formally described as follows. LDS contains a matrix of R rows and C columns. Each cell of the matrix contains a tuple of values $<s,n,d>$, which keep aggregate timestamps, packet counts, and packet digests respectively. Both the sender and receiver maintain separate LDS copies, that is transmitted by the sender at the end of the measurement interval. Each row r has now *two* associated pseudo-random hash functions h_r and h'_r. While h_r returns a value in the range $[0, C-1]$, h'_r returns a value in the range $[0, k-1]$, where k

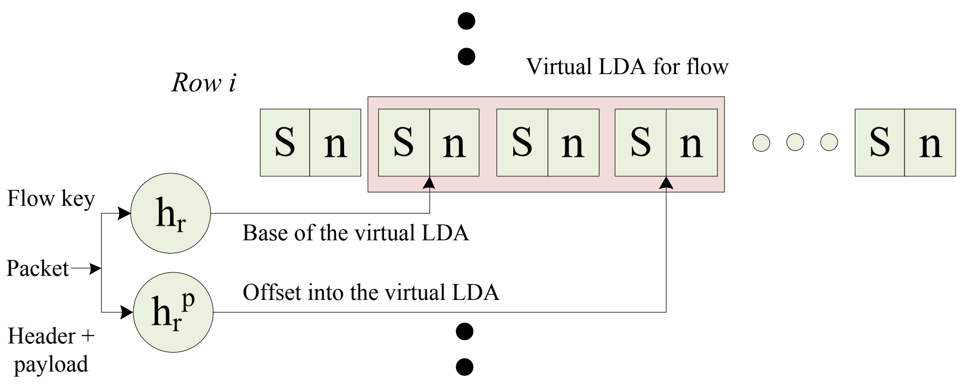


Figure 3.2: Virtual LDA extension to the data structure (d fields are not depicted).

Algorithm 3 LDS – Per-packet operations

```
1: procedure UPDATE STATE($pkt$, $f$, $\tau$)
2:     $ph \leftarrow hash\_packet(pkt)\%k$                ▷ $k$ is vLDA size
3:     for i=1, R do
4:         $fh \leftarrow (hash(i, f))$
5:         $j \leftarrow ((fh + ph)\%C)$
6:         $LDS[i][j].S \leftarrow LDS[i][j].S + \tau$
7:         $LDS[i][j].N \leftarrow LDS[i][j].N + 1$
8:         $LDS[i][j].D \leftarrow LDS[i][j].D \oplus pkthash$
9:     end for
10: end procedure
```

is a configuration parameter of our algorithm that, as we shall see, represents the length of the virtual LDAs in our data structure. Once a packet that arrives at time t hits a cell, its s is increased by t, n is incremented by 1, and d is XORed with the digest of the new packet.

Update. When a packet with payload x and flow identifier f arrives at time t, for each row r, a position in the matrix given by $(r, (h_r(f) + h'_r(x)) \, mod \, C)$ is determined. That is, the flow hash is used to obtain a base position in each row, while the packet payload's hash determines an offset to that position in the range $[0, k-1]$. Thus, the packets of a given flow are randomly distributed among the neighboring cells. To coordinate this randomization between sender and receiver, they both use, again, the same pre-arranged (consistent) hash functions. Figure 3.2 presents a diagram of this scheme, while Algorithm 3 formally describes this algorithm.

This scheme has the advantage of, while not introducing the full overhead of embedding a LDA in each cell, still obtains its advantages, by spreading the packets of each flow across several cells thus gaining protection against loss or reordering. If a flow experiences losses, they will invalidate some, but not necessarily all of the counters, which gives the algorithm a chance to recover its delay. As will be discussed in Section 3.3, this feature increases the amount of collisions. Therefore, to support the same number of flows, it still has to be larger than the basic data structure presented in Section 3.2.

Delay Estimation. When producing an estimate of a given flow, all associated usable vLDA cells are initially selected. After this step, the question of how to estimate flow delays arises. The algorithm now has to choose among the usable cells to produce an estimate. In the event that, for a flow, none of its cells are invalidated, it has $R\,k$ cells that can produce delay estimates (for each row, all cells of the flow's vLDA).

Again, several strategies could be used to select which cells are going to be used for estimation. For example, one could aggregate all usable cells of each row into a single one, thus obtaining one candidate delay estimation per row and, like in the previous data structure, choose the one with the least amount of packets.

Such a strategy is impractical, because it unnecessarily gives up the advantages of having the packets spread across several positions in the data structure. Instead, it is beneficial to selectively discard specific positions with high measurement interference.

We thus adopt the following strategy. First, among all cells, we choose the one with the smallest number of packets. Assuming that each vLDA cell contains $1/k$th of the packets of the measured flow, this is the cell that has experienced least colliding packets. Let the number of packets aggregated in this cell be n. Then, from the rest of the cells, we select those that contain, at most, $n\,(1+\alpha)$ packets, where α is a configuration parameter that reflects a maximum percentage of tolerable interference. Too large an α leads to the inclusion of interfering packets, while setting it too small discards valid samples. We empirically found $\alpha = 0.1$ to represent a good trade-off between these two factors in our setting.

Then, the resulting set of cells are aggregated to produce a final estimate. Since, within each vLDA, packets are randomly distributed across cells, the possibility of double counting packets exists. Note that this is not problematic for the measurement of average delay.

The pseudocode in Algorithm 4 captures our delay estimation mechanism more formally. Here $L1$ and $L2$ are the sender- and receiver-side LDSes. Note that k refers to the configured

Algorithm 4 LDS – Delay estimation algorithm

```
 1: procedure DELAY ESTIMATION($f$, $L1$, $L2$)
 2:     $N_{min} = \infty$
 3:     for i=1, R do
 4:         $fh \leftarrow hash(i, f)$
 5:         for $j = fh, (fh + k)\%C$ do                    ▷ $k$ is vLDA size
 6:             if L1[i][j].N == L2[i][j].N &&
 7:                 L1[i][j].D == L2[i][j].D then
 8:                 $cell = \{L2[i][j].S - L1[i][j].S, L1[i][j].N\}$
 9:                 $S = S \bigcup cell$                    ▷ Stores all valid cells
10:                 $N_{min} = min\{N_{min}, cell.N\}$
11:             end if
12:         end for
13:     end for
14:     for $cell \in S$ do
15:         if $cell.N < (1 + \alpha)N_{min}$ then
16:             $S_{sum}+ = cell.S$
17:             $N_{sum}+ = cell.N$
18:         end if
19:     end for
20:     return $S_{sum}/N_{sum}$
21: end procedure
```

vLDA size for each flow. Also, we assume both $L1$ and $L2$ have already been updated with the same hash functions $hash(i)$, $i = 1 \ldots R$.

3.3 Analysis

In this section, we analyze our data structure and provide guidelines on how to dimension it in order to obtain the desired level of accuracy. We start with the analysis of SDS and then extend it to the more general case of LDS.

3.3.1 Simple Delay Sketch

One could hope to dimension the data structure so that measurements are error-free with high probability, i.e., flows are highly likely to be free from collision in, at least, one of the cells.

Unfortunately, this would require a great deal of memory since, for example, when using a single row with as many counters as flows, the probability that a given flow is collision free is only e^{-1} (as in a standard Bloom filter with a single hash function). In order for this probability to grow beyond 95%, we require a number of counters that is well above of the number of flows we want to measure.

As explained, in our data structure, colliding flows cause interfering measurements, and estimates produced by each row are an average delay weighted by the number of packets of the colliding flows. In other words, in practice, larger flows tend to have less error, since they will most often collide with small flows, rather than larger ones.

In order for a flow to be accurately measured, the total number of packet in flows that collide with it must be sufficiently small so as not to significantly impact its delay estimate. Specifically, we say that a flow suffers *only small collisions* if the number of colliding packets is no more than some threshold x. To compute the probability Q of small collisions we consider first the number of flows colliding with a given flow, and then the number of packets that they bring. Letting K_i denote the probability of i colliding flows, and S_i the probability that these i flows bring no more that x packets in total, then $Q = \sum_{i=0}^{n} K_i \, S_i$, assuming n background flows. By definition, $S_0 = 1$, since a flow is always correctly measured when it is free from collision.

We aim to dimension the data structure in a way that the probability Q of only small collisions stays high. Assuming a uniform distribution of hash values, K_i is probability of obtaining i items under the Binomial distribution $B(n, 1/C)$, yielding a mean number n/C of colliding flows. To compute S_i we need to assume some distribution of flow sizes. We will assume Pareto distributed flow sizes. This distribution is often used to model flow sizes (e.g., [70, 109]) and can be fitted to the characteristics of the network data we use in the evaluation in Section 3.5. The sub-exponential property of the Pareto distribution implies that, if T_k is the sum of the sizes of k flows, then $\Pr\left[T_k > x\right] \approx k \Pr\left[T_1 > x\right]$ for large x at fixed k. In other words, when T_k is very large, this tends to be because one entry in the sum is very large, not because several are moderately large. Using this property, we obtain $Q \approx \sum_{i=0}^{n} K_i(1 - iP) = 1 - \frac{n}{C} * P$, where $P = \Pr[T_1 > x]$ under the (fitted) Pareto distribution. The accuracy of approximation increases for small n/C and large x.

Using this analysis, we can adapt the size of our data structure to obtain the desired probabilistic accuracy bound. For example, using Pareto parameters that match our traffic (see

Section 3.5), we obtain a probability $Q \approx 91\%$ of small collisions comprising at most 50 packets, with half as many counters as flows ($n/C = 2$), structured in 1 row. This means that, for example, flows with 1000 or more packets have a 91% probability of small collisions comprising no more than 5% of their packets. Incidentally, a numerical computation of Q without the subexponential approximation differed by a few tenths of absolute percent in this example.

The formulation of this example illustrates a key requirement to measure a flow accurately: not only must it suffer only small collisions but the flow itself must be large. (How large depends on the delay distribution.) To simplify the analysis, we stipulate a large flow to be one with at least x packets, where x is the threshold total packets for small collisions. With this formulation, we call a flow *survivable* in storage if it is both large, and suffers only small collisions. What then, is the maximal storage capacity of survivable flows? Suppose n flows are stored. The average number of large flows is nP and so the average number of survivable flows is $nPQ = nP(1 - nP/C)$. This expression is maximized at $n = C/(2P)$, yielding $C/4$. This is reminiscent of the collision free capacity C/e of the standard Bloom filter. The difference is that the proposed structure can store up to $C/4$ *survivable* flows out of a potentially far larger $C/(4P)$ that are presented for storage.

In fact we do not expect the operating regime to accommodate the maximal number of flow because the probability of *large* (i.e., not small) collisions is $1 - Q = nP/C = 1/2$. We now investigate operating regimes with rare large collisions in the generality of multiple $R \geq 1$ rows. We assume the primary design aim is to limit the probability of large collisions, with a secondary aim of maximizing the number of survivable flows under that constraint. In the case of multiple rows, a large flow is survivable if it has small collisions in at least one row. With R rows, the total resources C are divided up evenly between rows, and so substituting C/R for C in Q, the relevant survival probability is $Q(R) = 1 - (1 - Q)^R = 1 - (nP/CR)^R$. For a cleaner analysis it is convenient to change variables from n to $z = nP/C$, which can be thought of as the offered load of large flows per unit storage. Then $Q(R) = q(z, R) := 1 - (zR)^R$.

As a function of R for fixed z, $q(z, R)$ is maximized at $R = 1/(ez)$. But only $R \geq 1$ are physical. (In this analysis we omit consideration of integrality; in practice we round to an integer at the end). Thus $\max_{R \geq 1} q(z, R) = q(z) := q(z, \max\{1, 1/(ez)\})$. $q(z)$ is a decreasing function of z which takes the value $1 - z$ for $z > 1/e$ (corresponding to $R = 1$) and $1 - e^{-1/(ez)}$ for $z \leq 1/e$ (corresponding to the case $R > 1$). Assuming we wish a small probability $\varepsilon < 1/e \approx 0.63$ of large collisions, then we should be in the small $z < 1/e$ regime, so we would want to chose z such that $\varepsilon > e^{-1/(ez)}$, which corresponds to the choice $R = -\log(\varepsilon)$, modulo discretization, then making sure the offered load z is less than $z_{max} = -1/(e\log(\varepsilon))$. The relative gain of allowing multiple rows can be seen as follows: under the constraint $R = 1$, achieving the same bound on the probability of large collisions would require $z = 1 - q(z, 1) \leq \varepsilon$. Hence allowing $R > 1$ allows us to increase the offered load by a ratio $-1/(e\varepsilon \log(\varepsilon)) > 1$ for target $\varepsilon < 1/e$. Conversely, maintaining the same load achieves a dramatic reduction in the frequency of large collisions. In the previous example $z = nP/C = 0.0875$, so we are in the regime $z \leq 1/e$, leading to optimal $R = 4.20$. Rounding to the nearest integer $R = 4$, we obtain $q(4, z) = 0.9850$, as compared with the previous $q(1, z) = 0.91$.

3.3.2 Lossy Difference Sketch

The introduction of the Virtual LDAs in the LDS has several side effects. The principal consequence of further spreading flow packets across the data structure is that fewer positions remain unused and, more importantly, more collisions occur. However, this is to some extent compensated by the fact that every flow is spread across k positions, and, thus, collision randomization is higher. In this section, we will investigate how these factors change the previous analysis.

The Virtual LDA divides up the packets of a flow amongst k locations in each of R rows. Accurate estimation of a given flow depends on having only small collisions in at least one of these locations. Thus we adapt our notion of survivability as follows for general k: A given flow is *survivable* if it is large (the number of packets exceeds some value x) while at the same time suffers only small collisions (of size no more than x/k) at at least one of the Rk locations it occupies in the Virtual LDA.

In this section we examine a simplified model of the Virtual LDA that admits an extension to the analysis of Section 3.3.1 to approximate the probability of survivability. This shows that, from the collision survivability point of view, the Virtual LDA is no worse than the multirow data structure described in Section 3.3.1, and is actually expected to be better. This property, coupled with the superior loss resilience of the Virtual LDA, recommends it as the better choice.

Our model and analysis are as follows. For a specific flow, let U_ℓ be the number of its packets hashed to a location ℓ, and V_ℓ the number of packets from all other flows that are mapped to that location. The estimation algorithm first determines the location ℓ of minimal $U_\ell + V_\ell$. Since we are concerned principally with the case that the specific flow has some large number u of packets, our first simplification is to ignore the sampling variability amongst the

U_ℓ and approximate the U_ℓ as taking the same value (i.e., the average u/k). Thus the problem of minimizing $U_\ell + V_\ell$ is thus reduced to that of minimizing the V_ℓ.

Because the locations allocated to a given flow in a row are contiguous, the V_ℓ are in general dependent, because if packets from a background flow hash to location ℓ, the other packets from the same flow are more likely to collide at a neighboring location ℓ'. This dependence leads to positive correlations amongst the V_ℓ, meaning that the joint probability of collisions occurring at all locations of a flow is greater than the product of the marginal probability of collisions occurring at each site. Conversely, the corresponding survival probability is bounded below by that of a model where collisions are independent: it is conservative to use this as our second simplification. Thus we model the distribution K_i of the number of colliding flows as a Bernoulli $B(nk, R/C)$ random variables.

For our final simplification, we note that under our Pareto model, the probability that the number $T(k)$ of packets sampled from a background flow to each of the k locations in a row exceeds a level x obeys $\Pr[T(k) \geq x] \approx \Pr[T_1 \geq kx]$ for large x, where T_1 the length of the background flow; see [115]. Coupled with the subexponential approximation for sums of flow lengths, we approximate the probability of a small number of packets (at most x/k) due to i colliding flows at some site ℓ as $S_i = \Pr[V_\ell < x/k | \text{i colliding flows}] \approx 1 - iP$ where $P = \Pr[T_1 > x]$ under the (fitted) Pareto distribution.

Thus, under our simplifications, the probability of small collisions is $Q(Rk) = 1 - (nkRP/C)^{Rk}$. The optimization and dimensioning strategy is then immediate by comparison with Section 3.3.1: (i) choose a value k based on targets for loss resilience; (ii) for a given small target survivability rate $1-\varepsilon$, calculate the number of rows by rounding $\max\{1, -\log(\varepsilon)/k\}$ to the nearest integer.

3.4 Practical Enhancements

In this section, we introduce a series of enhancements to the LDS data structure that make it more practical for real deployment. We start by defining a mechanism to boost the accuracy for a selected subset of flows of particular interest to network operators. We then investigate how to parametrize the sampling rates to support a wide range of loss ratios. Finally, we note that the data structure contains additional information that can be used to mine other interesting metrics, including per-flow packet loss and heavy hitter detection.

3.4.1 Weighting of Flows

As formally analyzed in Section 3.3, the LDS intrinsically produces better estimates for large flows. This is due to the fact that, when flows collide, estimates are average delays of said flows, weighted by the amount of packets.

However, often times, small flows are of interest (a notable example are DNS flows, which usually consist of only one packet per direction). On the other hand, operators might be particularly interested in measuring a particular set of flows with higher accuracy. For example, one could increase the accuracy for certain subnetworks where critical services are hosted, or where troubleshooting activities call for precise examination of network delays (a practical use case for flow weighting is presented in Section 3.5).

We provide a mechanism to raise the accuracy of flows at will. This can be very simply accomplished by weighting flows according to some pre-defined policies driven by the operator's desires. Such policies can define a flow's weight according to any information present on packet headers. The default weight for non-policed flows is defined as 1 for simplicity.

These weights are taken into consideration straightforwardly by slightly modifying the update procedure. When a packet of a flow f arrives, its headers are examined and a weight w is determined according the existing weighting policies. Then, a cell of each row is selected as explained in Section 3.2.3.4. For each of the cells, their s value is increased by w times the packet timestamp, while n values are increased by w (recall that, previously, s values were increased by the timestamp and n values by 1).

No modification is required to the estimation procedure. When all flow weights are equal, the estimates are identical to those of Section 3.2.3.4. Otherwise, flows are weighted by their number of packets times their weight. It shall be noted, however, that this extra accuracy will always come at expense of the accuracy of the estimates for flows with lesser weight. Thus, it is not advisable to heavily increase the weight of a large percentage of flows, as it will dramatically reduce the accuracy for all the others. We will analyze the effect of weighting from a practical standpoint in Section 3.5.

3.4.2 Multi-Bank LDS

As explained in Section 3.2.3.3, to maximize the collection of delay samples in front of packet loss, each vLDA should sample the incoming packet stream at rate L/k, where L corresponds to its associated number of losses and k to its length. However, in a real scenario, the absolute number of losses that each flow will experience is unpredictable, which raises the question of how to set the sampling rate. On the one hand, a reasonable amount of loss has to be supported, which calls for low sampling rates. On the other, aggressive sampling will miss small flows and fail to collect enough packets for those that do not experience loss.

Inspired by [88], we propose dividing the counters of the LDS in several banks, and have each bank sample the incoming packets at a different rate. This way, at least one of the banks will suit the actual loss rate of each flow.

A natural way to divide counters in banks is to set a different sampling rate on a per-row basis. In the evaluation provided in Section 3.5, we show that configuring one row for worst-case loss scenarios, and maintaining increasingly higher sampling rates in the other rows, does not sacrifice accuracy in normal scenarios with low loss, while still providing protection against high loss. When a flow surpasses the target worst-case threshold, the LDS will be unable to provide delay estimates for that flow. In such a case, however, the data structure can provide an estimate for its number of lost packets, as will be explained in Section 3.4.3.

This variant of the LDS requires very few changes to the algorithms detailed in Section 3.2.3. Now each row has an associated sampling stage, which is also implemented using pseudo-random hashing to coordinate measurement nodes, while the estimation procedure only needs to be modified to be aware of the sampling rate that each cell has applied. In particular, packet counts need to be inverted before deciding which LDS cell will be selected to produce a final estimate. After the cell selection procedure, delay estimates are produced normally.

3.4.3 Mining Other Estimates

The LDS data structure can be mined to extract additional information of practical interest to network operators. Firstly, the data structure can provide per-flow delay variance estimates by examining the differences across the delays recorded in buckets dedicated to a given flow. The procedure to obtain this estimate was originally proposed and is thoroughly described in [88]. Our data structure has a comparatively smaller number of buckets per flow, but the same method could be applied to obtain rough delay variance estimates. This additional estimate can be extremely useful in practice to detect unexpected delay variations, such as jitter or delay peaks.

If we ignore the s fields and focus only on the n fields of each cell, the data structure behaves similarly to a Count-Min Sketch [50]. Consequently, an estimate of the length of a given flow can be obtained as follows. For each row, aggregate all vLDA cells to obtain a packet count. Then, take the minimum of such values. This is the final estimate. It can be easily shown that this estimate is, at best, error free, and, in the presence of collisions, it can only be greater than the actual value. This is an interesting property for certain problems and, especially, for heavy hitter detection. The accuracy of this technique is analyzed in greater detail in [50].

Simply by attaching another counter to each cell, where packet sizes are aggregated, we can also estimate flow sizes in bytes. Both these new counters and the existing could be used for heavy hitter detection in terms of bytes or packets respectively. Additionally, one could obtain crude estimates for the average packet sizes. We divide the aggregate delay over the number of packets to estimate average flow delays. Likewise, total flow sizes could be used to obtain a per-flow average packet size. Finally, we note that per-flow packet loss can be obtained by comparing the n fields of our data structure as collected in *sender* and *receiver*.

3.5 Evaluation

With the objective of evaluating the LDS data structure in a realistic scenario, we deployed two network monitors in an operational network. For the sake of reproducibility, we collected a packet delay trace, rather than directly processing live traffic. We then ran a series of experiments using LDS and two state of the art techniques that will be described in Section 3.5.1. We note, however, that all the traffic measurement procedures, including LDS and both reference techniques, were fast enough to run on-line and, therefore, the results we present are completely equivalent to live traffic analysis.

The measurement scenario, which is more thoroughly presented in Appendix A, consists of two measurement points. The first monitor was deployed in the link that connects Catalan research and education institutions to the rest of the Internet. The second was located in the access link of one of such institutions: Universitat Politècnica de Catalunya. We obtained a copy of the traffic that traversed both links in both directions and used Endace DAG cards [6] to simultaneously capture packets in both measurement points. We synchronized DAG clocks using the PTP protocol, which reportedly provides sub-microsecond accuracies [54] and thus is accurate enough for fine grained delay measurement.

We then wrote a CoMo module [31] to analyze the trace and extract, for each packet, its flow identifier, packet identifier, and exact one-way delay. The trace averages 27Kpkts/s and contains around 7.76 million packets that belong to approximately 146000 flows.

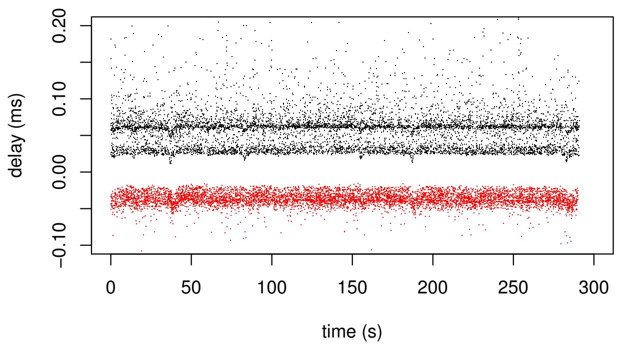


Figure 3.3: Timeseries of a sample of the packet delays, with outbound delays portrayed as negative values.

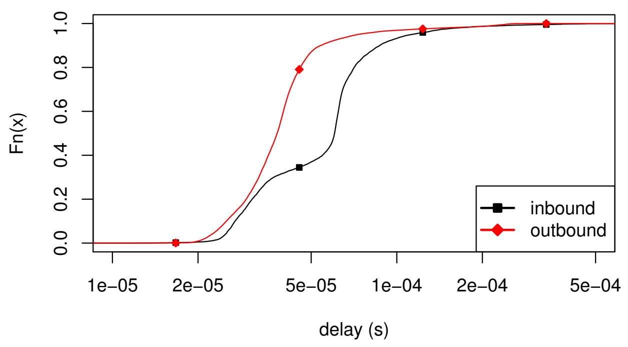


Figure 3.4: CDF of the delays for inbound and outbound packets.

Figure 3.3 shows a time series of a sample of the packet delays for each traffic direction. Two features are apparent from this figure that make inbound traffic (i.e., destined towards the University network) more interesting. First, inbound packet delays present higher variability. Second, two delay modes are clearly appreciable in the inbound traffic, as can be also confirmed in the CDF of the packet delays presented in Figure 3.4. Therefore, unless otherwise noted, the experiments presented next in the evaluation are performed on the inbound traffic.

3.5.1 Comparison with Existing Methods

The objective of this section is to compare the accuracy of LDS with the state-of-the-art on per-flow delay measurement. We choose the NetFlow Multi-Point Estimator (MPE) [93] and the Reference Latency Interpolation (RLI) [92] as representatives of a recently introduced class of techniques that exploit temporal delay correlation to refine measurements from a few samples.

The Multi-Point Estimator is conceived as an extension to NetFlow, and requires routers to

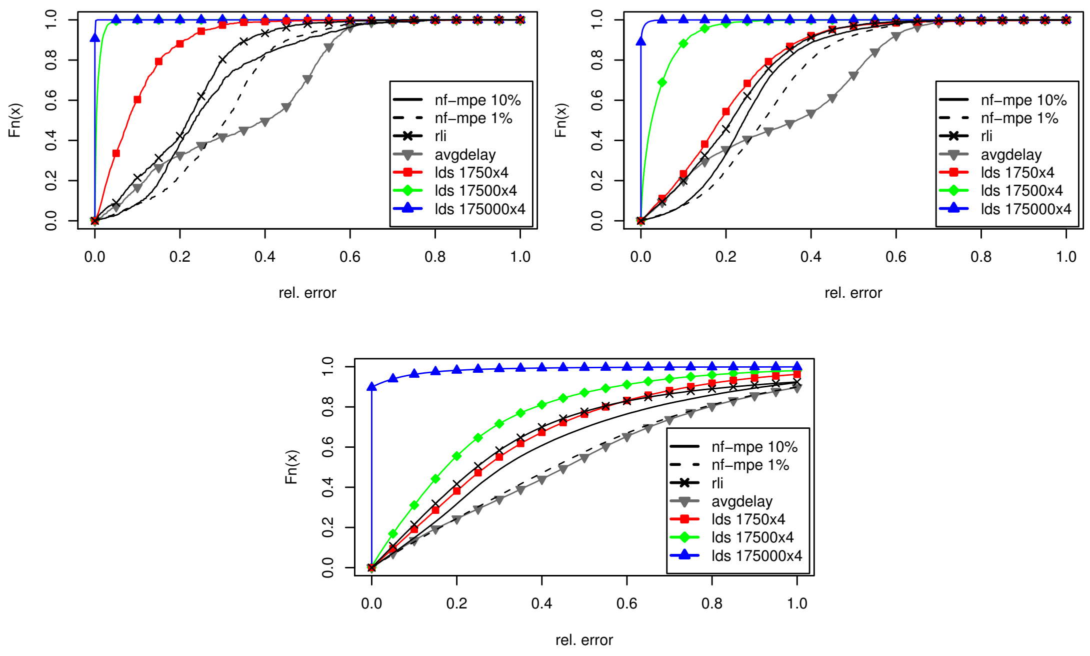


Figure 3.5: CDF of the relative error of various measurement methods for flows with > 1000 pkts (left), with > 100 pkts (right) and all flows (bottom).

use coordinated sampling. Under this assumption, for each sampled flow, there exist two delay samples from which to estimate the flow delay (NetFlow records include a timestamp of the first and last packet). Additionally, based on the empirical observation that packets that travel close in time experience similar delays, the method can interpolate the delay between these two samples using the NetFlow records of other flows that start or end within the duration of the measured flow. The main difference between MPE and RLI is that, while MPE relies on a modified version of NetFlow, RLI injects active probes to obtain the necessary delay samples and assumes that packets between two probes experience the same delay.

We evaluate LDS with three different configurations: one that provisions half as many counters as flows ($n/C \approx 2$) (to obtain a configuration that, as will be discussed, is comparable with MPE), while the other two are 10 times larger and smaller than this reference LDS. Consistently with the example in Section 3.3, we structure the sketch in 4 rows, which yields a sketch of 17500×4 counters for the first configuration. Given that loss is negligible in our scenario, we set the vLDA length $k = 1$, and the sampling rate $p = 1$. We analyze the impact of loss in detail in Section 3.5.2.

Figure 3.5 plots the CDF of the relative error obtained by each flow in the traffic, for different flow sizes. The figure includes the accuracy of MPE with a sampling rate of 1% and 10%,[1] RLI with 1KHz probing, and a simple method that estimates the delay of each flow to be the average delay of all packets. The figure shows that LDS greatly outperforms both MPE and RLI. The increase in accuracy compared to MPE can be explained by two primary causes. First, MPE completely misses a large number of small flows (e.g., more than 50% with 10% sampling). For these flows, we estimate their delay as the average delay of all packets, instead of simply assigning an error of 1. In contrast, LDS can always obtain an estimate for all flows. Second, for the flows it does collect, it interpolates the delays using other flows, but in our case these are not necessarily correlated, as can be observed in Figure 3.4. While RLI outperforms MPE, its accuracy is also far from LDS, especially for large flows, and requires significantly more memory and state maintenance.

Figure 3.5 (left) shows that, as predicted by the analysis, large flows are very accurately measured. A still notable accuracy for flows of 100 or more packets is also observed in the middle plot. Figure 3.5 (right) shows the per-flow accuracy for all flows, including also those with less than 100 packets. According to the analysis in Section 3.3, these flows are not considered to be survivable, since they tend to experience large collisions. These flows however only account for 20% of the packets in our trace. Even in this case, the accuracy of LDS is consistently above the state of the art. This result shows that the estimate of LDS for unsurvivable flows is in practice more accurate than just using the average delay of all packets.

LDS also features better memory usage. For example, with 10% sampling, MPE captures around 70000 flows, so (generously disregarding the fact that NetFlow stores flow keys) it consumes roughly as much memory as the 17500×4 LDS. Thus, with the same memory budget, LDS clearly outperforms MPE in terms of measurement accuracy. Note also that, even when LDS uses 10 times less memory than MPE ($C = 1750 \times 4$), it obtains significantly higher accuracy, especially for medium sized to large flows. LDS also outperforms RLI, which requires even more memory than MPE, since it maintains per-flow state.

Figure 3.6 (top) plots the median and 95-percentile of the relative error of flows binned by size. Consistently with the analysis of Section 3.3, the figure shows that larger flows are more accurately measured. Our method compares extremely favorably to both MPE and RLI. Only with 7,000 counters we obtain significant improvements for flows larger than 1,000 packets.

[1]Note that MPE uses sampling to control the memory usage, while for LDS sampling is only a measure against packet loss.

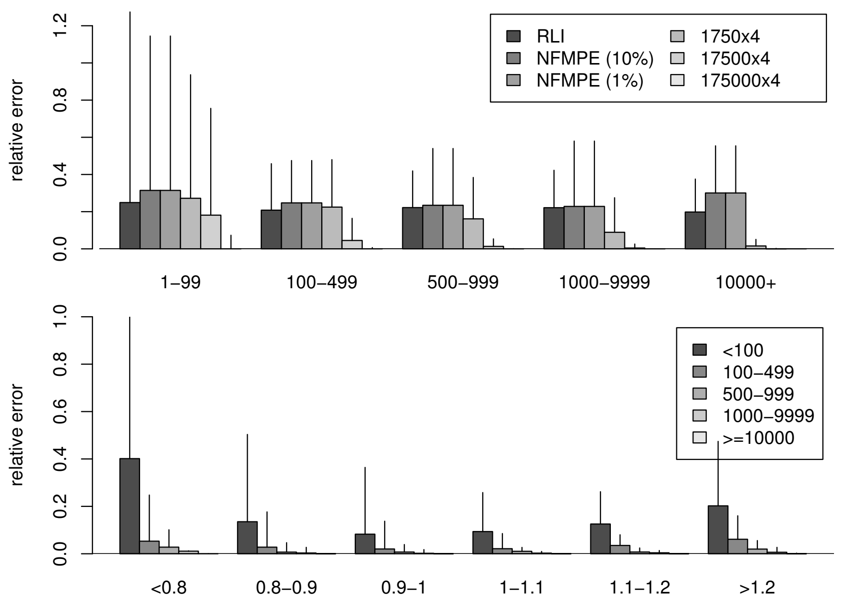


Figure 3.6: Median and 95-pct of relative error, binning flows by number of packets (top) and number of packets and average delay (relative to avg. flow delay; bottom).

When using 70,000 counters, which take about 1MB, flows with 500 to 999 packets obtain 1.2% median relative error, which falls to 0.4% and 0.06% for the larger size bins.

Figure 3.6 (bottom) bins flows both by size and average delay with a fixed sketch size of 70,000 counters. The figure shows how the errors are slightly larger as delay deviates in extreme values for the mean. This happens because smaller flows show more extreme values, but interferences tend to drag measurements toward the mean.

3.5.2 Measurement under Packet Loss

We now analyze the effect of packet loss to our data structure. Under small loss rates, it is desirable to keep the vLDA length parameter k small, since increasing it increments the number of collisions in the sketch. However, increasing k provides higher protection against loss. Additionally, sampling helps contain loss, since a large number of losses in a single flow can potentially invalidate the k counters.

We wish to dimension our data structure to support a given maximum number of losses per flow. The main intuition behind this approach is that, when flows experience a large amount of losses, performance degradation is more a consequence of loss than delay; thus, delay mea-

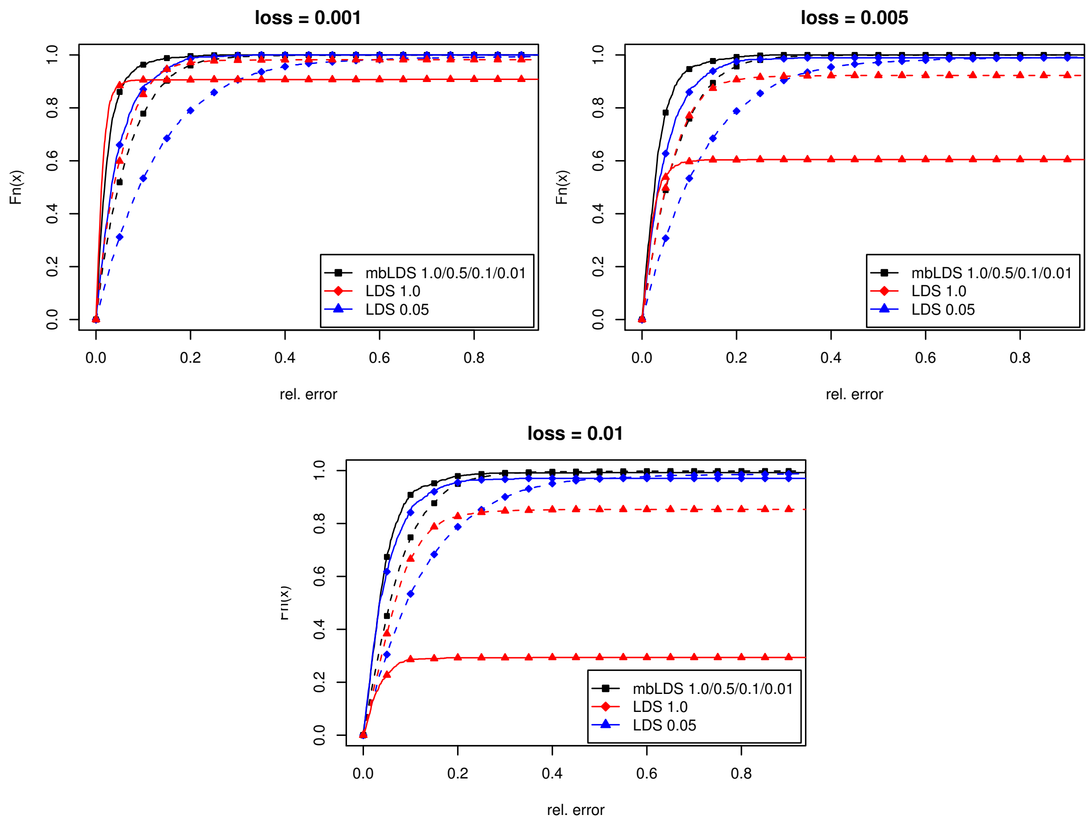


Figure 3.7: CDF of the relative of several LDS parametrizations under varying uniform loss rates. Solid lines represent flows with at least 10^3 packets, while dashed lines, flows with more than 100 packets.

surements cease to be meaningful (note that LDS can be mined to estimate per-flow loss, as explained in Section 3.4.3).

In this experiment, we arbitrarily set a target number of losses of 500 packets per flow. However, we also wish the LDS to be able to capture a large sample size if losses are much lower. Thus, we use a multi-bank configuration of LDS, as described in Section 3.4.2. We provide 4 rows with $k = 5$ vLDA buckets, like in the previous scenario, increase the size of each row of the sketch by a factor of k, and set $\alpha = 0.1$. We then pick suitable packet sampling rates for each row. With $k = 5$, each vLDA cell needs to support 100 losses, according to our target number of losses. This means that the sampling rate should be set to 0.01 to support this worst-case loss. Then, we wish the rest of the banks to tolerate lower loss in order not to sacrifice the accuracy of LDS in the normal case. We set the rest of the banks with increasing sampling rates of 0.1, 0.5 and 1 to tolerate up to 50, 10 and 5 losses respectively. For comparative purposes, we also set two LDS with a fixed sampling rate in all rows of 0.05 and 1.

Since, in our scenario, losses are negligible, we introduce random, uniform loss to test such configurations. Consistently with the assumptions made in Section 3.2.1, we perform 3 series of experiments with loss rates 0.1%, 0.5% and 1%. It should be noted that uniform loss is one of the most harmful loss models to LDS, for three main causes. First, losses are spread among a large number of flows, instead of being contained within a few. Second, the absolute number of losses that hit each bucket heavily varies according to the lengths of the involved flows. Recall from Section 3.2.3 that the optimal sampling rate for each bucket depends on its absolute number of losses. Third, this loss model penalizes large flows, which are precisely those that our method can measure most accurately.

Figure 3.7 shows the results we obtained. We start by noting that neither 5% nor 100% sampling single-bank LDSs perform satisfactorily. The former maintains its accuracy under higher loss, but is too conservative and underperforms on low loss. Conversely, 100% sampling is too optimistic and does not offer protection against loss. Thus, its cells become too quickly invalidated under increasing loss, causing measurements to be lost.

In contrast, the multi-bank LDS performs consistently well under all loss rates. This desirable behavior is a consequence that, in all three scenarios, most flows experience losses that are well tolerated by at least one of the banks. Therefore, very seldom a flow invalidates all of its buckets, and accurate measurements are always produced.

The accuracy of LDS in Figure 3.7 is still above that of RLI and MPE without loss (as presented in Section 3.5.1). However, MPE and RLI are more robust to loss. Hence, in scenarios with high loss and temporal correlation, we expect MPE and RLI to be a better choice.

3.5.3 Flow Weighting

We now test a more realistic use case of our technique. We envision a data center that hosts network services for a series of customers. Not all customers, though, are equally sensitive to network delay. We group the hosted services in three classes. First, bronze customers are not overly concerned about packet delays in the data center. For example, those could include bulk data transfer applications, such as backup, static web content serving, e-mail relaying, or computing intensive tasks.

Second, silver customers are somewhat dependent on network delay, but they do not require strict compliance of low-delay QoS requirements. A class that would fit these well are interactive services, such as remote shells, highly interactive web applications (e.g., Google is known to seek low delay to enhance the user's browsing experiences of AJAX-powered web applications), or web services for third party applications.

Finally, gold customers host applications that are extremely sensitive to delay, and wish to closely track the QoS of the services they are offering. Perfect examples for this class of applications involve multi-media streaming, audio/video conferencing, or remote gaming. Financial services such as automated trading could also fit this category, although, given that, in their case, low-delay data transmission is critical, they are unlikely to be hosted in shared infrastructure.

Since we do not have access to network traffic from a data center that hosts such applications, we adapt our scenario as follows. We randomly assign each flow to one of the categories. Bronze customers take 90% of the flows; silver customers, 9%, and gold customers, 1%. This approach ensures that the results are not an artifact of flow sizes, since large flows tend to be more accurately measured. In a real setting, these weights can be adjusted to the specific characteristics of the traffic under measurement.

We experiment with different sizings of the data structure, and various weights for each of the customer classes. Figure 3.8 shows the result of a series of experiments. We have tested two reference configurations. One that uses 40,000 counters (first row), an another that uses 100,000 counters and provides greater accuracy (second row). The first configuration fits in 625KB, and the second, in around 1.5MB. As for flow weights, we have tested three different configurations, which can be observed in each column: assigning weights of 1, 25 and 100 (first); 1, 10 and 100 (second), and, 1, 50 and 2500 (third).

The figure shows the relative error for the full set of flows (solid lines) and only for flows with more than 100 packets (dashed lines). Besides the error of each customer class, the figure also shows, as a reference, the result of applying no weighting. An important observation to be made is that flows from classes that have higher weights obtain significantly greater accuracy. The accuracy boost greatly depends on the actual weights; for example, when gold customers carry weight 2500, they obtain extreme accuracy. However, this penalizes the accuracy of bronze customers. In this case, we have ensured that the accuracy of bronze customers is not highly penalized, because few flows belong to higher priority classes. If, otherwise, the number of flows in each class was more balanced, the only option to increase accuracy for the flows of a given class without significantly diminishing that of a lower priority class would be to increase the sketch size. In other words, this method is only applicable to increase the accuracy of a small subset of flows.

3.6 Related Work

One-way packet delay has been measured both using passive and active schemes. Active monitoring methods (e.g., [33, 44, 110, 125]) are based on injecting probe traffic in the network under study, and inferring one-way delay from the delays incurred by such probes. In contrast, passively monitoring network delays has been traditionally accomplished by recording packet timestamps in two measurement points, and exchanging such timestamps for comparison. Because these techniques generate huge data volumes, they require aggressive sampling to reduce the overhead. Further, packet sampling has to be coordinated across nodes, since the timestamps recorded at both measurement points must correspond to an equal subset of all packets. This effect can be achieved using consistent hashing, i.e., using same pre-arranged hash function—an idea used before in trajectory sampling [58]).

More recently, LDA [88] has been proposed as a mechanism to overcome the linear relationship between sample size and overhead. LDA has been further analyzed in Chapter 2 and Reference [69]. Since we borrow some of the ideas of LDA, we have discussed this idea in great length. The problem of obtaining per-flow latency estimates in a scalable fashion, which is exactly the problem we attempted to solve in this work, has received recent attention [92, 93]. [93] proposes modifying NetFlow [3] to allow measurement of one-way delay. If NetFlow samples packets using consistent hashing, the first and last timestamp fields of NetFlow records can be used to obtain two delay samples of a given flow that can be refined from using samples from other flows in between these two timestamps. The core idea of temporal delay correlation forms the basis for Reference Latency Interpolation [92], that we already discussed in detail.The biggest difference between RLI and our work is that we do not assume temporal correlation of packet delays. Removing the dependence on this assumption is beneficial in many ways, as explained in Section 3.1.

Besides packet delay measurement techniques, sketching is also very relevant to this work. Most relevant to us are two similar data structures: Multi-Stage Filters [64], which are designed for elephant flow detection, and the Count-Min Sketch [50], which can provide per-flow estimates with probabilistic accuracy guarantees. Other sketching techniques are reviewed and compared in [49].

3.7 Concluding Remarks

We have presented a sketch-based data structure capable of producing per-flow one-way delay estimates. Although sketching naturally produces the best estimates for larger flows, this data structure can enhance the accuracy of arbitrary flows. For measurement in networks with packet loss, we have combined our sketching technique with a recently appeared data structure called Lossy Difference Aggregator.

State-of-art techniques rely on temporal correlation of delays to produce their estimates. However, in practice, routers can use various queueing policies for different kind of traffic, which greatly reduces the effective of said techniques. In our evaluation, we show how our technique achieves higher accuracy than such techniques when using a similar amount of memory, even in the presence of packet loss.

We have also presented a practical deployment scenario where our technique and its ability to improve measurement for arbitrary flows could be very useful. In particular, our technique could very well cater a data center with shared resources, where various applications present diverse degrees of dependency on network delay. In such a scenario, our technique can be used to obtain extremely precise measurements for the most critical applications, while still providing an acceptable degree of accuracy for other applications.

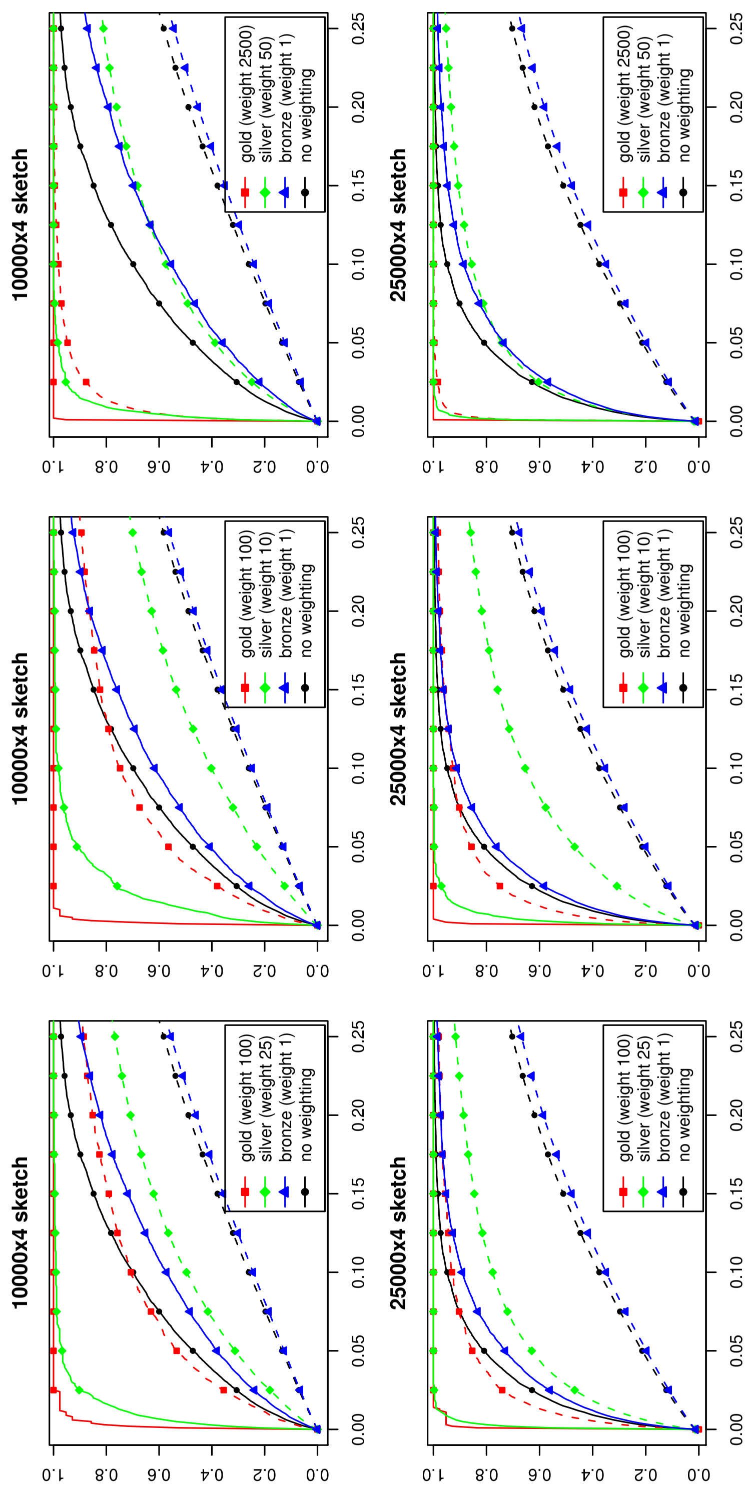


Figure 3.8: CDFs of the relative error with various flow weights and sketch sizes. Solid lines correspond to flows with more than 100 packets, while dashed lines include the full set of flows.

Part II

Measurement over Sliding Windows

4

Introduction

From an abstract point of view, network packets form a continuous data stream that is fed to the monitor. The operator of such a monitor is usually not interested on measurements on the full lifetime of the data stream. On the contrary, measurements need to be bounded in time. Perhaps the most widely adopted approach to this issue is to collect measurements in consecutive, back-to-back time bins of a certain pre-defined length. This model is easy to implement: simply, data structures need to be queried and re-initialized periodically. We call this approach the *discrete window* measurement model.

More complex, but often more useful is the *sliding window* measurement model [48]. Under this model, the monitor can be queried at any point of time for measurements that span the last w time units. That is, measurements are not simply collected over consecutive time bins. Instead, they are collected and reported over a continuously advancing (sliding) time window. This calls for more complex algorithms and data structures that must be aware of time and expire information as it ages out of a measurement window.

The sliding window model is gaining interest in the networking and database communities, given the streaming nature of many current data sources (e.g., network traffic, sensor networks or financial data). Under this paradigm, queries process streaming data, instead of static databases, and compute metrics over time windows that advance *continuously* (e.g., the number of active flows during the last 5 minutes). A new class of systems based on this measurement model is emerging in the database [28, 76] and network monitoring [51, 82, 114] communities in order to support continuous queries over sliding windows.

An interesting introduction to the field of data stream research can be found in [104], while a more informal discussion that motivates this research area is [80].

In this part of the thesis, we investigate the problems of performing two basic traffic analysis tasks under this measurement model. First, we revise the problem of counting the number of active flows in the traffic; while several specialized algorithms have been proposed for this particular problem, fewer have addressed extracting measurements over a sliding window. Second, we provide a novel solution to traffic filtering over a sliding window, by adapting the well-known Bloom filters to this measurement model. Both algorithms we propose are based on the same idea: to attach, to each entry in the data structures, a small integer counter that is slowly decremented.

5

Counting the Number of Active Flows

The design of efficient algorithms to count the number of active flows in high-speed networks has recently attracted the interest of the network measurement community [65, 71, 86]. Counting the number of flows in real-time is particularly relevant to network operators and administrators for network management and security tasks. For example, this metric is the basis of most network intrusion detection systems to detect port scans and DoS attacks.

However, the naive solution of tracking flows using a hash table is becoming unfeasible in high-speed links. First, it requires several memory accesses per packet with the overhead of creating new flow entries and handling collisions. Second, this solution uses large amounts of memory, since it requires storing all flow identifiers. The number of concurrent flows present in high speed networks is very large, and can be well over one million in current backbone links [66]. Therefore, hash tables must be stored in DRAM, which has an access time greater than current packet interarrival times. For example, access times of standard DRAM are in the order of tens of nanoseconds, while packet interarrival times can be up to 32 ns and 8 ns in OC-192 (10 Gb/s) and OC-768 (40 Gb/s) links respectively. Thus, flow counting algorithms must be able to process each packet in very few nanoseconds to be suitable for high-speed links.

In order to reduce the large amount of memory required to store flow tables, most routers (e.g., Cisco NetFlow [3]) and network monitoring systems [31] resort to packet sampling. However, it has been shown that packet sampling is biased towards large flows and tends to underestimate the total number of flows [59].

Recently, several probabilistic algorithms have been proposed to efficiently estimate the number of flows in high-speed networks [65, 71, 86]. These algorithms share the common approach of using specialized data structures to approximately count the number of flows, which need a very small amount of memory, as compared to the traditional approach of keeping per-flow state, while requiring one or very few memory accesses per packet. This drastic reduction in the memory requirements allows the storage of these data structures in fast SRAM, with access times below 10 ns. These techniques can therefore be implemented in router line cards or, in general-purpose systems, the data structures can reside in cache memory.

The direct bitmaps technique is the basis of most probabilistic algorithms to estimate the number of flows. This technique was first proposed by Whang et al. [130] in the database community and popularized in the networking community by Estan et al. in [65], which also presents several variants of the direct bitmaps that require less memory. One of the variants, called multiresolution bitmaps, obtains similar accuracy as Loglog counting [62]. Giroire's proposal [74] is to estimate the count from the minimum of the hashed values. These and other techniques are compared in [102]. A remarkable conclusion from this study is that, from a practical standpoint, direct bitmaps offer the best tradeoff between complexity and accuracy.

The basic idea behind direct bitmaps is to use a small vector of bits (i.e., bitmap). For each packet, a hash of the flow identifier is computed and the corresponding bit is set in the bitmap. At the end of a measurement interval, the number of flows can be simply estimated according to the number of non-set bits and the collision probability [130]. The main problem of these algorithms is that they can only operate over fixed, non-overlapping measurement intervals and, therefore, cannot obtain continuous estimates of the number of flows.

Datar et al. [53] analyze the basic problem of counting the number of ones within the last N elements of an arbitrary stream composed of zeros and ones. They present an algorithm based on a technique called Exponential Histograms that can provide an approximate solution to this problem. Using this basic algorithm as a building block, they approach related problems such as calculating sums, averages, maximum and minimum values. For the distinct count problem, they propose adapting a bitmap-based counting technique to the sliding window model by storing timestamps instead of bits in each bitmap position.

Kim and O'Hallaron [86] propose a technique that addresses the flow counting problem using this approach, called Timestamp Vector (TSV). TSV is based on the original direct bitmap algorithm and consists of replacing the bitmap for a vector of timestamps. However, the main limitations of TSV are that (i) it requires a significantly larger amount of memory compared to the original direct bitmap (a 64-bit timestamp for each vector position) and (ii) the cost of the queries is linear with the size of the vector, which renders this solution impractical in scenarios where a continuous estimate of the number of flows is needed.

We propose a new algorithm called Countdown Vector (CDV), which we describe in the next sections, to estimate the number of flows over sliding windows. The basic idea behind our method is the use of a vector of small timeout counters, instead of full timestamps, that are decremented independently of the per-packet update and query processes. Our algorithm requires less resources (i.e., CPU and memory) than existing solutions, and has $O(1)$ query cost. This way, a network monitoring system can implement our method using less memory, and can react faster to changes in the number of flows (e.g., network anomalies or attacks), since queries can be issued more frequently than in previous proposals. Another interesting advantage of our technique over other alternatives is that it is possible to degrade the accuracy of the estimates according to a given CPU and memory budget.

5.1 Background

Direct Bitmaps. Since our algorithm is based on direct bitmaps, we first review this technique in greater detail. We also describe the Timestamp Vector algorithm (TSV), which we use to compare the accuracy and overhead of our method. Additionally, we propose a simple improvement of the TSV that significantly reduces its memory requirements under certain conditions.

A flow is defined as a sequence of packets that share equal values for a subset of the TCP/IP header fields. Typically, a flow is identified by the 5-tuple that consists of the source and destination IP addresses and ports, and protocol field. However, the flow counting methods we review and propose are agnostic to the particular definition of a flow.

Like the naive algorithm of using a hash table to track flows, direct bitmaps are based on hashing, but do not create one table entry per flow. Instead, they just record whether a position in the hash table would be used or not (hence the name bitmap). The algorithm uses a pseudo-random hash function (e.g., [41]) to evenly distribute the positions corresponding to each flow identifier.

One could think of extrapolating the number of ones in the bitmap as the number of flows in the original dataset. However, this would not be correct, since there is a non-negligible probability that several flow identifiers collide (i.e., hash to the same bitmap position), given the reduced size of the bitmap. While the bitmap cannot be used to extract the exact count of flows, instead, an estimate can be obtained from the bitmap size (b) and the number of zeros in

b:	Number of positions of the vector or bitmap.
c:	Value to which counters are initialized.
f:	Time between queries in a jumping window model.
n:	Number of flows in the traffic during a time window.
$\hat{n}$:	Estimation of the number of flows n.
s:	Time between counter updates.
w:	Length of the time window.
z:	Number of positions with value 0 in the bitmap or the vector or bitmap.

Table 5.1: Notation

the bitmap (z) as shown in [130]:

$$\hat{n} = b\, ln \left(\frac{b}{z} \right) \tag{5.1}$$

We refer to the process of obtaining an estimate of the number of flows as the process of *querying* the bitmap. Table 5.1 summarizes the notation used throughout this section.

The principal advantage of this technique is that, with a small amount of memory (especially when compared to the naive algorithm), the number of flows can be estimated with high accuracy. A second important characteristic of this approach is that the accuracy can be arbitrarily increased or reduced by the bitmap size. To correctly dimension the bitmap, the appropriate size must be chosen so that the expected amount of unique elements can be estimated within the desired error bounds, as explained in [130].

To give the reader an idea of how little memory this algorithm requires, it suffices to state that this technique can count 10^6 elements by using around 20KB with errors below 1% [130].

Timestamp Vector. As just discussed, direct bitmaps are a very efficient technique to count the amount of flows in the network traffic. However, in order to provide meaningful values when monitoring a network traffic stream, measurement statistics must be bounded in time, i.e., must correspond to a particular *time window*. Direct bitmaps do not incorporate a sense of time and thus must be periodically reset to avoid lifetime flow counting.

Periodically querying and resetting a direct bitmap would provide flow counts over consecutive, non-overlapping windows. While there is value to this application of the technique, it imposes the restriction that queries must be aligned with bitmap resets. In contrast, in the *sliding window* model, queries can arrive at any time. This requirement implies that old information must be removed as newer arrives, in order to continuously maintain the data structures to be able to provide an estimate at any time.

A straightforward solution to adapt the direct bitmaps to the sliding window model is the Timestamp Vector algorithm [86]. Instead of a bitmap, a vector of timestamps is now used. When a packet hashes to a particular position, its timestamp is set to the timestamp of that packet. Using this vector, when a query for a time window of w time units arrives at time t, the number of flows can be estimated by using Equation 5.1, where in this case z corresponds to the number of positions with timestamp less than $t - w$, and b to the number of positions in the vector. Note that this requires a full traversal of the vector for each query.

5.2 Extension of the Timestamp Vector

The principal limitation of the Timestamp Vector technique is the increase in the amount of memory that it requires. Timestamps in network monitoring are typically 64 bits long, e.g. as provided by libpcap [84] or Endace DAG cards [6]. Hence, the final size of the vector is increased by a factor of 64 compared to the original bitmap.

A relaxation of the sliding window model is the *jumping window* model [75], where the window does not advance continuously, but discretely in fractions of the measurement window. When operating under this model, we propose the following improvement over the Timestamp Vector algorithm that significantly reduces its memory requirements. Instead of full timestamps, only the fraction of the window where the packet arrived is stored. This idea can be implemented using $log_2 (w/f + 1)$ bits per position when measuring a window of w time units with query periods of f. For example, with a window of 30 s and queries every 1 s, the original Timestamp Vector will require 64 bits per position, while this approach would require only 5 bits per position, using below 10% of the memory. Note however that, using this extension, the Timestamp Vector can only be queried every f time units.

One limitation of both variants of the Timestamp Vector algorithm is, thus, their additional memory requirements. The jumping window variant reduces memory usage by limiting the time where queries may be performed. The second disadvantage of this scheme is that, for each query, one full traversal of the array is required to calculate the number of positions whose timestamp is older than $t - w$.

5.3 Countdown Vector

In this section we present our technique for flow counting over sliding windows. Our scheme is, like the TSV algorithm, an adaptation of the direct bitmaps to the sliding window model. We start by outlining the basic intuition behind the technique we propose.

The main difficulty when adapting the direct bitmap to the sliding window model is to remove old information from the data structure as time advances. Let us start by defining an ideal algorithm which would precisely calculate z in a sliding window of w time units. Recall that b (vector size) and z (number of zeros in it) suffice to estimate the number of flows in the traffic using Equation 5.1. A vector of b positions could be used, with the values set to w time units every time a packet hashed to the corresponding position. Every time unit, all the positions of the vector with non-zero value would be decremented by one. The count of positions with counter value zero would correspond to z in order to estimate the number of flows seen in the time window.

In order to obtain a perfect resolution, this scheme would require defining the time unit to the maximum resolution of the system clock. This ideal scheme would therefore be very costly in terms of both memory and CPU. First, a high resolution counter would have be stored in each vector position, thus increasing the overall memory required to store the vector. Second, all of the counters would need to be updated for each time unit. These additional costs make this technique infeasible as described, especially when it would achieve results equivalent to the TSV.

The technique that we propose is very similar to the ideal algorithm just described but, instead, using small integer counters in each position. Using small values requires less memory and calls for a low counter decrement frequency for counters to reach zero after w time units, in exchange of introducing small inaccuracies. In the following paragraphs we describe our technique with detail. The key to the effectiveness of our algorithm is that surprisingly low values suffice to achieve good accuracy to estimate network flow counts. In practice, then, we require less memory and introduce lower overhead compared to the original TSV algorithm and to its extension presented in Section 5.2.

Algorithm. Our algorithm starts by allocating a vector of counters, all of which are initialized to zero. Our algorithm can then be seen as divided in two concurrent processes: the first updates

Algorithm 5 CDV – Initialization and packet-synchronous operations

```
1: $z \leftarrow b$
2: $vector \leftarrow \{0, \ldots, 0\}$
3: $start\_continuous\_maintenance\_procedure()$
4: for all packet $p$ do
5:     $key \leftarrow hash(p.flow\_identifier)$
6:     if $vector[key \bmod b] = 0$ then                ▷ update count of zeros
7:         $z \leftarrow z - 1$
8:     end if
9:     $vector[key \bmod b] \leftarrow c$
10: end for
```

Algorithm 6 CDV – Continuous maintenance procedure

```
1: $s \leftarrow \frac{w}{b \times (c - \frac{1}{2})}$
2: $i \leftarrow 0$
3: loop
4:     sleep for s time units
5:     if $vector[i] = 1$ then                ▷ update count of zeros
6:         $z \leftarrow z + 1$
7:     end if
8:     $vector[i] \leftarrow max(0, vector[i] - 1)$
9:     $i \leftarrow (i + 1) \bmod b$
10: end loop
```

the vector for each packet, while the second is in charge of decreasing the counters at a fixed rate.

The first process is described in Algorithm 5 and runs synchronously with the packet stream. When a packet hashes to a position, we store a maximum value c in the corresponding position. This process remains the same independently of the time window that is being measured.

In contrast, the second process, which is described in Algorithm 6, performs a continuous maintenance of the vector. It decreases one counter every s time units, advancing one position in the vector at every step. The desired time window w plays a role in this data structure maintenance process, where it conditions the speed at which counters are decreased.

To determine s we proceed as follows. Since the packet arrivals and the maintenance are

Algorithm 7 CDV – Query procedure

```
1: return $b \times ln\,(b/z)$
```

independent processes, and each flow hashes to a random position, on average, the first decrement of a counter (after it is set to w by the first process) will happen after $b/2$ counter updates. Afterwards, the counter will be decremented after b additional counter updates. Therefore, on average, counters reach zero after $b/2 + b\,(c-1) = b\,(c-1/2)$ updates. Since this time must correspond to the time window w, we calculate s the following way:

$$s = \frac{w}{b\,\left(c - \frac{1}{2}\right)} \tag{5.2}$$

Both the first and the second processes maintain the count of positions of the vector with value zero, updating the value of z as values of the vector are modified. This has the advantage that query operations run in constant time $O(1)$, by simply applying Equation 5.1, as described in Algorithm 7.

Our algorithm is then governed by the following configuration parameters: (i) the desired measurement window w, (ii) the size of the vector b, and (iii) the maximum values to which counters are set c.

The precision of our algorithm increases with larger b and c values. Larger values of b (vector size) make the estimation error of Equation 5.1 decrease, as explained in further detail in [130]. On the other hand, increasing c also has a positive impact on the accuracy of the method, since, the larger c is, the more our algorithm approaches the ideal algorithm explained in the previous subsection.

However, larger values of c increase both the memory and CPU requirements of our algorithm, since, the higher c is, the more space is required to store the counters, and the higher the counter decrease frequency, i.e., s decreases, according to Equation 5.2.

Our algorithm has two sources of error. First, the approximation introduced by the original technique upon which ours builds, the direct bitmaps. The second source of error is introduced by the fact that old information is expired after w time units only *on average*. Inevitably, some counters will, on the worst case, be set to c right after the maintenance process has decremented the corresponding position, and will thus will reach zero after $c * b * s = \frac{c}{c-1/2}w$ time units. Conversely, others will reach zero after $\frac{c-1}{c-1/2}w$ time units. In the worst case a counter will be inaccurate only during this small period of time. This explains why the accuracy increases with larger values of c, which tighten these bounds around w.

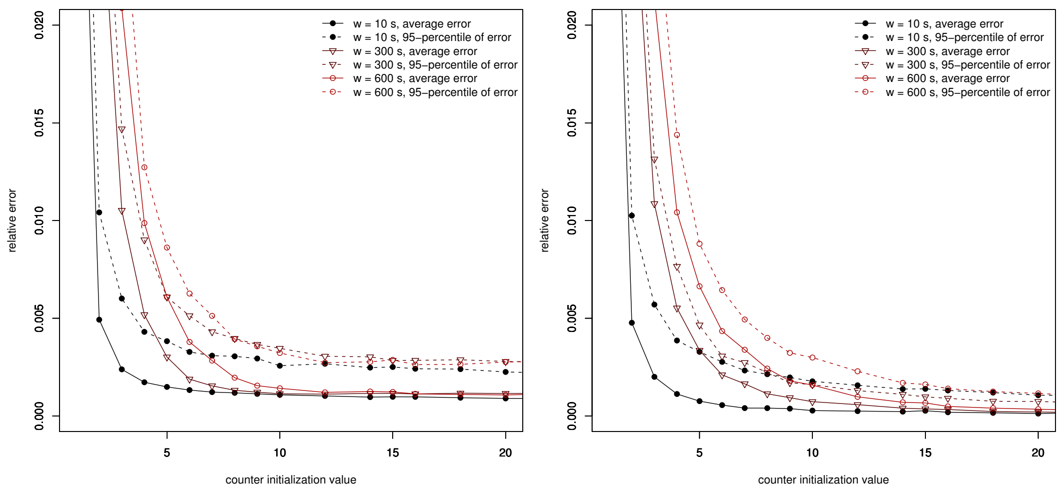


Figure 5.1: Error of the Countdown Vector algorithm (left) and error of the Countdown Vector algorithm compared to the estimates of the Timestamp Vector algorithm (right).

In order to choose a value for b, the tables provided by [130] can be looked up to determine the an appropriate vector size for the expected number of flows in the traffic. In the next section we analyze the impact of c over the accuracy of the method and show the overhead reduction of our method compared to both variants of the Timestamp Vector.

5.3.1 Evaluation

In order to obtain sensible results, we have tested our technique using real traffic. We collected a 30-minute packet-level traces in November 2007 at the access link of the Universitat Politècnica de Catalunya (we refer the reader to Appendix A for a more thorough presentation of the measurement scenario). The trace accounts for 106M packets with an average data rate of 271.6 Mbps. The average number of flows is around 50000 in a ten seconds window, and 1.8 million in a 10 minute window.

Figure 5.1 (left) shows the results of running our algorithm with window sizes of 10, 300 and 600 seconds with different counter initialization values, using a fixed vector size. The size of the vector has been chosen according to [130] so that the average number of flows for the largest window can be counted with errors below 1%. Each point in the figure corresponds to a full pass on aforementioned trace, querying the algorithm every second, and shows either

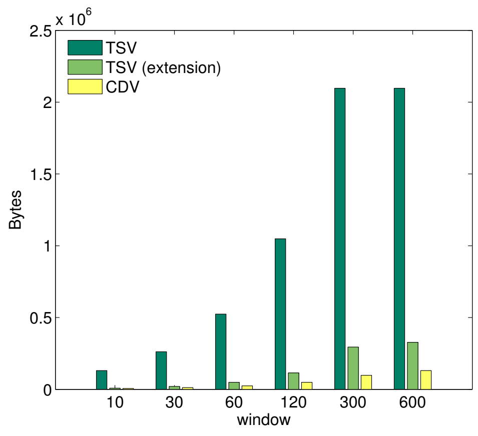
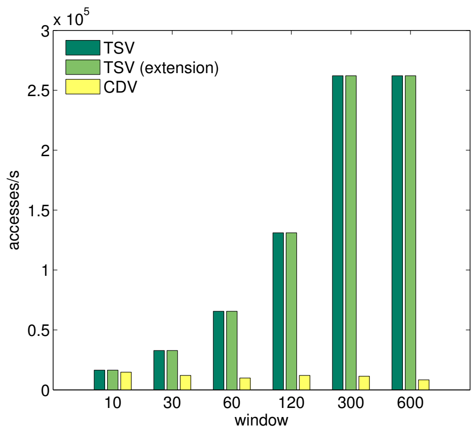


Figure 5.2: Memory consumption for the three algorithms with varying window sizes.



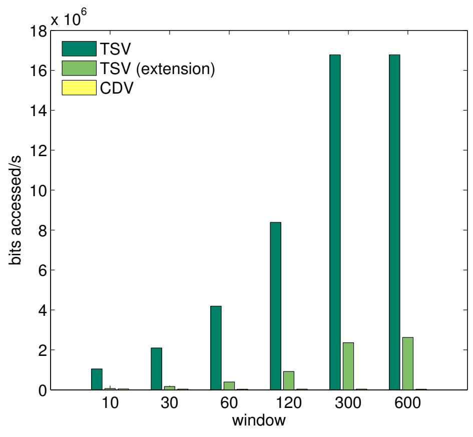


Figure 5.3: Number of memory accesses per second (left) and number of bits accessed per second (right) for the three algorithms with varying window sizes.

the average relative error or the 95th percentile of the relative error compared to a precise calculation of the number of flows.

As expected, for every window size, the relative error decreases as c increases. It is interesting to observe that, while the error is intolerable for very small values of c (5 and below), when it reaches values as small as 10 the error stabilizes, and does not decrease significantly beyond that point, even for a window as large as 600 s. This observation explains the great overhead savings that our technique shows in comparison to the Timestamp Vector, as we show in the next paragraphs. We have observed that the error for these values of c is almost equal to that of the Timestamp Vector algorithm. This source of error can be imputed to the underlying method of estimation of the direct bitmaps (see Equation 5.1). Figure 5.1 (right) shows the error of our method relative to the values obtained using the Timestamp Vector algorithm and confirms this observation.

We now examine the cost of the Countdown Vector algorithm and compare it to that of the Timestamp Vector. Figures 5.2 and 5.3 summarize a different set of executions of the algorithms using the same trace. To obtain realistic overhead calculations, in this case we dimension the vectors for both TSV and CDV appropriate for the observed number of flows in each time window, using [130], bounding the error introduced by the estimation formula to 1%. We dimension our variant of the TSV for a 1 second query frequency. For performace reasons, our implementation restricts the vector sizes to powers of two; we choose the smallest suitable sizes.

Our algorithm has, besides the size of the vector, an additional parameter: the counter initialization value (c). We have run the experiments with various c values, and have chosen, for each time window, the smallest that obtains at most 0.1% more relative error than the Timestamp Vector algorithm.

Figure 5.2 shows the high memory savings that our variant of the TSV introduces, at the expense of reducing the query frequency to only one second. In contrast, the CDV we propose further reduces the memory to roughly one half compared to our TSV variant, without introducing such a restriction.

We compare the cost of the algorithms using the number of memory accesses per second, assuming one query every second, and including the maintenance cost in the case of the CDV. However, we omit the cost of updating the vectors for every packet, since this cost is common to all of the algorithms. It suffices to state that it is only in the order of 50000 accesses per second, which roughly corresponds to the average packet rate in our trace. In Figure 5.3 (left), it can be observed that the cost of the Timestamp Vector grows with the size of the vector, since it has to be traversed for every query. In contrast, the cost of the Countdown Vector remains small.

This figure is unfair to our variant of the TSV since, while the number of memory accesses are equal to those of the original TSV, these are accesses to smaller chunks of memory. Depending on the architecture, then, specific optimizations could be employed to improve its performance (e.g., read several positions with a single memory access). To correct this, we also present the cost in terms of bits accessed per second in Figure 5.3 (right).

The cost of both variants of the TSV grows proportionally to the vector size, since full vector traversals per query are required. Vectors grow with window sizes, since higher flow counts have to be obtained. In contrast, the CDV's query cost is constant; in our algorithm, the bulk of the cost is in the maintenance phase. However, since very low counter values can obtain an accuracy very similar to the TSV variants, counter decrements are performed at a low frequency, therefore incurring significantly lower costs.

6

Traffic Filtering

In this section we focus on the problem of traffic filtering over a sliding window. From an abstract point of view, we are interested in providing a set of data structures and algorithms that can answer the question “is this item a member of our filtered set of items?”, with the particularity that items are automatically evicted from the set after w time units.

The abstract problem of set membership has many potential applications. For example, a network operator might desire to temporarily block connection attempts from an attacker to slow down bruteforcing attacks [106]. Another scope of application is network caches, where cache items are expected to be considered as expired after a certain amount of time since their insertion.

The problem of set membership has been traditionally addressed using extremely efficient data structures called Bloom filters [32] that have been widely applied to other networking problems [40]. They allow for an extremely compressed representation, thus being suitable to reside on fast (more expensive) memory modules, and their update and query operations run with constant time and very few memory accesses. However, the problem of set membership over a sliding window has received less attention. To our best knowledge, only one paper exists [89] that mentions this problem and adapts Bloom filters to this measurement paradigm. In the data streaming literature, Stable Bloom filters have been proposed [55] as a modification of standard Bloom filters to detect duplicates in a huge stream of data with limited memory. In this scenario, the main problem is that Bloom filters can quickly fill and become useless for duplicate detection.

We propose a different extension for Bloom filters to operate over sliding windows that is considerably more memory efficient than the existing. In Section 6.1, we review the necessary

symbol	meaning
b	number of positions of the filter
z	fraction of unset positions in the filter
h	number of hash functions
$hash_i$	ith hash function, i in $[0, h-1]$
w	configured measurement window (TSBF and CDBF)
c	counter initialization value (CDBF specific)
s	sleep time between counter decrements (CDBF specific)

Table 6.1: Notation

background on Bloom filters and the mentioned existing proposal to extend them for measurement over sliding windows. In Section 6.2, we present our proposal, which we evaluate in Section 6.3.

6.1 Background

Set Membership and Bloom Filters. It would be trivial to compute set membership using standard data structures and algorithms. For example, one could implement a hash table to track inserted items. Every item insertion would create a new entry in the hash table. Then, set membership of a given item could be determined trivially by a hash table lookup.

However, this naive solution would have three main problems [65]. First, maintaining a large number of entries would require a great deal of memory. Second, both insertion and membership query operations would require several memory accesses, due to hash table collisions. Third, the large memory requirements would make it prohibitively expensive to hold the data in fast SRAM memory; cheaper DRAM memory accesses are too slow for nowadays' backbone link speeds.

Bloom filters [32] are extremely lightweight data structures that can very efficiently compute set membership. We only briefly describe this data structure, since it is very well known and widely used in networking algorithms; Table 6.1 describes the notation we use throughout the rest of this document.

A Bloom filter is an array of b bits (bitmap), initially all set to 0, and h different pseudorandom hash functions. To insert an element to a Bloom filter, one proceeds as follows. First, the item is hashed with the h hash functions. The output of the hash functions determine h

different positions in the bitmap. These positions in the bitmap are set to 1. To perform a set membership query operation for a given item, again, its h positions are determined using the hash functions. If all associated bits are set, membership is reported. Conversely, if any of these bits are unset, the element is reported not to be a member of the set.

This data structure presents clear advantages. First, it features an extremely compressed representation of the set. This, in turn, makes it cheaper to store it in fast memory. Third, a fixed, small number of memory accesses per operation is necessary. However, these advantages come at the price of no longer being to precisely determine set membership. It is easy to see that several items can collide in the same positions. Thus, it might wrongly be concluded that an element is present in the set, causing a *false positive* (reporting membership for a non-member). The converse error (false negative, i.e., reporting non-membership for a member) can not occur. In practice, many applications can tolerate a small probability of false positive. Additionally, this probability depends on the size of the bitmap, and can thus be arbitrarily reduced.

The Time-Stamp Bloom Filter. So far, we have introduced the Bloom filter as a space-efficient data structure for set membership testing. However, in this work we are interested in the more complex operation of set membership querying over sliding windows. Under such a model, as explained in Section 6, inserted items automatically leave the set after w time units. An alternative way to formulate this problem is as follows. An item can be flagged at any time. Then, the algorithm has to determine whether a given item has been flagged within the last w time units (i.e., belongs to the set of flagged items).

The standard Bloom filter does not provide any means to expire old entries as they age out of a desired time window. A way to address this limitation could be to periodically reset the data structure. The obvious drawback of this approach is that the filter would expire all items simultaneously, but not over a continuously advancing time window.

A way to provide a Bloom filter with a mechanism to expire old information presented in [89] is as follows. Instead of an array of bits, the data structure now holds an array of timestamps. When inserting a packet, the corresponding positions are set to the current packet's timestamp (Algorithm 8). Then, when querying the Bloom filter, positions where the timestamp falls out of the measurement window are considered to be unset (Algorithm 9). That is, only positions whose associated item (or items) have been inserted during the time window will be

Algorithm 8 TSBF – Packet insertion operations

```
1: for $i \leftarrow 1, h$ do
2:     $pos \leftarrow \texttt{hash}_i(p.identifier)$
3:     $filter[pos \bmod b] \leftarrow p.timestamp$
4: end for
```

Algorithm 9 TSBF – Query operations

```
1: for $i \leftarrow 1, h$ do
2:     $pos \leftarrow \texttt{hash}_i(p.identifier)$
3:     if $now - filter[pos \bmod b] > w$ then
4:         return NOT-PRESENT
5:     end if
6: end for
7: return PRESENT
```

reported as members (barring false positives). This approach was, to the best of our knowledge, first proposed in [89].

We call this data structure the Time-Stamp Bloom Filter (TSBF). While it has the advantage of tracking set membership over a sliding window, it has the problem of increasing the size of the data structure by a factor of the timestamp size. Thus, we give up one of its main advantages, which is its compactness.

6.2 The Countdown Bloom Filter

So far, we have presented a fairly straightforward extension of the Bloom filter data structure that is capable of tracking set membership over a sliding window called Time-Stamp Bloom Filter (TSBF). Its main disadvantage is however that, compared to a standard Bloom filter, it requires significantly more memory. In this section we propose our technique, which we call CountDown Bloom Filter (CDBF) to track set membership over a sliding window model in a way that requires considerably less memory, yet, as will be shown, obtains similar accuracy.

In essence, what is required for the Bloom filter to be able to expire information is that, w time units after each position being set, it is reset to zero. For example, when setting a position, we could initialize it to w, then continuously decrement it as time advanced. In order to obtain close to perfect accuracy, the time unit should be set to the maximum resolution of the system clock. This scheme would be unfeasible in practice for two important reasons: first, every cell

Algorithm 10 CDBF – Packet insertion operations

```
1: for $i \leftarrow 1, h$ do
2:     $pos \leftarrow \texttt{hash}_i(p.identifier)$
3:     $filter[pos \bmod b] \leftarrow c$
4: end for
```

Algorithm 11 CDBF – Query operations

```
1: for $i \leftarrow 1, h$ do
2:     $pos \leftarrow \texttt{hash}_i(p.identifier)$
3:     if $filter[pos \bmod b] = 0$ then
4:         return NOT-PRESENT
5:     end if
6: end forreturn PRESENT
```

of the filter would need to be large enough to contain a high resolution counter, thus increasing the total memory requirements; second, all of the counters would need to be updated at every clock pulse.

The algorithm we propose emulates this effect in a more practical way. Instead of timestamps, each position now holds a small integer counter. When setting a position, its counter is initialized to a fixed value. Then, with a certain fixed frequency, positions are slowly decremented. We will later discuss how to choose a counter decrement frequency that ensures that each item inserted in the Bloom filter is evicted after w time units *on average*. In other words, compared to the TSBF, our algorithm no longer expires information precisely. However, as will be seen, this source of error is small, and this disadvantage is offset by the memory savings that the CDBF scheme achieves.

Algorithm. The algorithm starts by allocating a vector of counters (the Bloom filter) initialized to zero. An insertion hashes the item and sets the corresponding positions to a predefined value c (Algorithm 10). The query process just checks if all the positions associated to an item are non-zero (Algorithm 11).

Additionally, the algorithm launches a background process that continuously decrements non-zero counters, which we call *maintenance procedure*. The key of our algorithm is this procedure, which sequentially decrements every counter at a fixed rate that ensures that, on average, each item expires after w time units.

Algorithm 12 CDBF – Maintenance procedure

```
1: $s \leftarrow w\, b^{-1} \left(c - 1 + \frac{1}{z(h+1)}\right)^{-1}$
2: $head \leftarrow 0$
3: while true do
4:     sleep for $s$ time units
5:     if $filter[head] > 0$ then
6:         $filter[head] \leftarrow filter[head] - 1$
7:     end if
8:     $head \leftarrow (head + 1) \bmod b$
9: end while
```

The maintenance procedure, which is described in detail in Algorithm 12, works in steps. Every step, it advances its head and checks the associated counter, and if it is not zero, it decrements it. Then, it sleeps for s time units before executing the next step. A very important detail of this procedure is then rate at which it decrements counters, which is driven by the amount of time it sleeps between steps.

The objective of this procedure is that the rate at which counters are decremented ensures items are expired after w time units on average. Let us initially consider an empty Bloom filter. When an item is inserted, it hashes to h positions, which are all set to c. Let us denote by P the (multi)set of positions to which said item has hashed (with $|P| = h$). Since hash functions are pseudo-random, the distance between $head$ and the next position from P that the maintenance procedure will hit is, on average, $b/(|P| + 1) = b/(h + 1)$ (where b is the size of the vector).

After such first update has been made, said position will be decremented exactly after b more steps. The counter will then reach zero after $(c - 1)$ additional decrements. Thus, the counter reaches zero after $b/(h + 1) + b\,(c - 1)$ steps, causing the item to be evicted from the set. We want these steps to take w time units, so that items are expired at the correct rate. We would therefore obtain that the time the maintenance procedure has to sleep between steps is $s = \frac{w}{b\left(c - \frac{h}{h+1}\right)}$. The b and h parameters can be calculated as described in the original Bloom filter proposal [32], and depend on the desired false positive probability and the number of keys that will be inserted.

However, usually, the Bloom filter will not hold a single item. An optimally operating Bloom filter has around half of its bits set. Intuitively, this means that, when the maintenance procedure decrements the first position of P it encounters, it will cause the item to be evicted only with probability 0.5. In this case, the item will be expired, again, with probability around 50%, when the second position of P is decremented, etcetera.

We can generalize this reasoning to any Bloom filter load and use the properties of the negative binomial distribution to find that, on average, the z^{-1}th position of P will cause the item to be evicted, where z corresponds to the ratio of *unset* positions in the Bloom filter. For example, if the load is 0%, the first decrement evicts the item, while if its load is 50%, on average, it is the second position that causes the evicton. On average, the $1/z$th position of P will be located in position $\frac{b}{z(h+1)}$ of the Bloom filter.

Thus, we can modify the previous formula to account for collisions as follows:

$$s = \frac{w}{b\left(c - 1 + \frac{1}{z(h+1)}\right)}$$

Observations. As discussed, our algorithm evicts items from the measured set after w time units *on average*. This implies that, unlike the original Bloom filter and the Time-Stamp Bloom filter, it can cause both false positives (due to collisions in the Bloom filter, or because an element took longer than expected to expire), and false negatives (only because an element was expired too early).

This is a consequence of the extra source of approximation that our algorithm presents. The Time-Stamp Bloom filter expires information precisely, and its only source of error is caused by the underlying Bloom filter, which, as explained, only answers queries correctly with a certain (albeit high) probability.

However, as we will show in the evaluation section, this extra source of error is small. On the other hand, by performing approximate information expiration, our algorithm achieves notable memory savings. In our experiments, 6 bit counters perform extremely accurately, where the Time-Stamp Bloom filter requires full timestamps.

On the other hand, our algorithm provides a mechanism to arbitrarily control the precision of information expiration. Just like the false positive rate of a Bloom filter can be improved by making it larger, our algorithm can expire information more precisely by increasing its counter initialization value c. This will increase memory usage (by a factor of $log_2\,(c+1)$ compared to the original Bloom filter), and counter decrement frequency, which will increase the accuracy of item eviction.

It is also noteworthy that our algorithm's query and insertion mechanisms require only exactly h hashing operations and memory accesses (just like regular Bloom filters). Additionally, the maintenance procedure's cost is constant and can be determined beforehand, according to a predefined memory and CPU budget.

6.3 Evaluation

In this section, we present an evaluation of the Countdown Bloom Filter (CDBF) using real network traffic. The objective of this evaluation is two-fold. First, we analyze the error introduced by approximate information expiration, show that it decreases as counter sizes increase, and that small counter sizes suffice to obtain notable accuracy. Secondly, we show that, when using equal amount of memory, CDBF is more accurate than TSBF, given that, since counters are smaller, we can provision a larger Bloom filter, which increases overall accuracy.

6.3.1 Trace and Methodology

For these experiments, we collected a 30 minute trace at the access link of the Universitat Politècnica de Catalunya (UPC). We refer the reader to Appendix A for a more extensive description of our measurement scenario. The trace we collected averages around 138 Kpkt/s and just short of 700 Mb/s and includes both link directions.

Besides the CDBF, we implement the Time-Stamp Bloom Filter (TSBF) to compare against an algorithm that does not perform approximate information expiration, but is still based on Bloom filtering. Finally, we also implement a precise filter based on hash tables, which serves as a ground truth to measure the error of both Bloom filter based algorithms.

In order to compare the filtering accuracy of both algorithms, we must define an accuracy metric. With this objective in mind, we present the following hypothetic scenario. We assume that the UPC network is protected by a firewall and an intrusion detection system (IDS) that is continuously observing the traffic and reporting attacks. For those attacks where the IDS is confident enough, we attempt to slow down the attackers by adding a temporary filtering rule to the firewall that drops their traffic.

Since intrusion detection is not the focus of this study, we randomly select packets and flag them as attacks. Packets are independently selected with uniform probability of 1%. This has the advantage that it is extremely simple to implement and that the results are not tied to particularities of attack patterns or a small set of network flows.

Then, for each flagged packet, we insert the sender's IP address to the filter for 10 seconds. We choose a rather small time-out with the objective of rendering our tests sensitive to the filter expiration accuracy. The firewall then queries the filter for every packet to determine whether it has been sent by an attacker, and drops it if so. The accuracy metric we use is given by the filtering errors in the firewall. False positives occur when the firewall drops packets from non-attackers, and false negatives, then it accepts packets from attackers.

6.3.2 Counter Size

In this subsection we explore the counter initialization value c that should be configured for the CDBF algorithm to perform close to the TSBF when using the same Bloom filter size b. The shorter the counter values, the smaller the CPU overhead and the more memory is saved, but expiration accuracy decreases.

Figure 6.1 shows the effect of various counter initialization values, compared to the precise expiration mechanism of the TSBF. Each point in the plot corresponds to the average error (including both false positives and negatives) throughout an entire pass on the trace.

Most importantly, the figure shows that small counter values are enough to achieve a close-to-TSBF accuracy. Recall that our method can not perform better than TSBF under equal filter size b, since the latter features precise information expiration.

In particular, for the smaller Bloom filter, $c = 3$ already shows great performance (using only 2 bits per position). Larger filter sizes can benefit from larger c, although for all of them, $c = 63$, which uses 6 bits per position, obtains good accuracy.

We interpret this results as follows. As c grows, information expiration becomes increasingly precise. However, the error quickly becomes dominated by the false positives of the Bloom filter, rather than expiration error. Thus, approximate expiration has little practical effect after c grows past a reasonably small value.

6.3.3 CPU and Memory Cost

The Count-Down Bloom filter introduces an additional overhead in terms of memory accesses to decrement counters so that filter entries are evicted as they fall out of the measurement window. In this subsection, we compare the CPU and memory costs of both algorithms.

In our example scenario, every packet causes a query to the Bloom filter. Additionally, insertions cause another access to the Bloom filter. Since insertion operations are comparatively infrequent, we ignore them in this subsection. In Table 6.2, we compare the memory requirements of both methods.

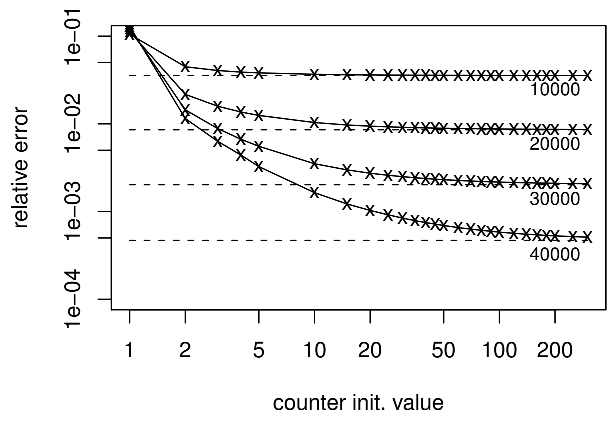


Figure 6.1: Relative error obtained by the CDBF with various filter sizes and counter initialization values. The errors obtained by equally sized (in terms of Bloom filter positions) TSBF are portrayed as dashed lines.

#pos	#hfunc	#acc/s (pkt)	c	#acc/s (mnt)	TSBF	CDBF
10 K	2	276 K	3	3 K	78.1KB	2.44KB
20 K	5	690 K	15	30 K	156KB	9.76KB
30 K	8	1104 K	31	93 K	234KB	18.3KB
40 K	11	1518 K	63	252 K	312KB	29.3KB

Table 6.2: Memory accesses per second for the per-packet operations (CDBF and TSBF) and maintenance procedure (CDBF specific) for various filter configurations, and memory requirements.

For the four sample Bloom filter sizes, we have chosen a c value that, according to Figure 6.1, obtains a close accuracy to that of the TSBF. Then, we have calculated the overhead of the maintenance procedure, considering a measurement window of $w = 10$ seconds. The table shows that the maintenance procedure has a negligible to moderate cost in terms of number of memory accesses.

However, the memory savings are notable in all cases. This means that our data structure is more suitable for deployment in fast memory modules, which are more costly than slow DRAM. The extra cost in terms of memory accesses is then easily offset, since our data structure can be deployed in faster memory. Alternatively, given a fixed memory budget, since every position occupies fewer bits, our data structure can hold a Bloom filter with more positions.

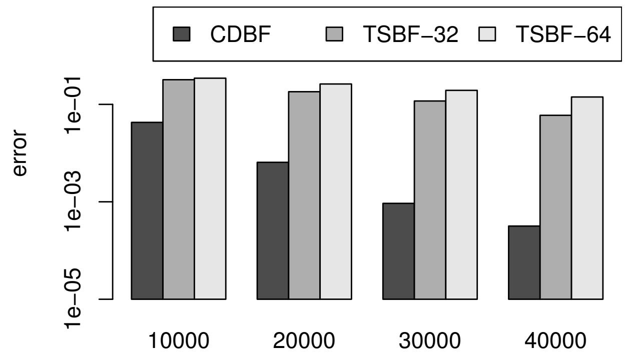


Figure 6.2: Comparison of the performance of the CDBF vs TSBF when using a fixed memory budget. For the TSBF, we use standard 64-bit timestamps and smaller 32-bit ones. Note the logarithmic vertical axis.

6.3.4 Performance under a Fixed Memory Budget

In this subsection we compare the performance of both data structures using a fixed memory budget. Again, we take as a reference the Bloom filters with 10,000, 20,000, 30,000 and 40,000 positions. Using Table 6.2, and taking the Count-Down Bloom filter sizes as a reference, we configure Time-Stamp Bloom filters that use an equivalent amount of memory. Since TSBF requires more bits per Bloom filter position, this results in smaller filters than those of the CDBF. We compare CDBF against TSBF of 64 and 32 bit timestamps.

Figure 6.2 shows the results obtained with this comparison. The results we obtain clearly highlight that, even though our data structure has an additional source of error, it uses memory much more efficiently and obtains notably higher filtering accuracy. In particular, the experiments show that, in practice, expiration precision and Bloom filter form a trade-off worth exploring. By lowering expiration precision, our algorithm fits a larger Bloom filter in the same amount of memory, which, in turn, has the effect of increasing overall accuracy.

7

Concluding Remarks

We have proposed two algorithms for measurement over sliding windows. The basic idea behind our scheme is to use a vector of timeout counters that expires old information in an approximate fashion, rather than simply timestamping every entry in the data structures. We used this approach to adapt existing algorithms to this novel measurement model.

We have presented an algorithm called Countdown Vector based on direct bitmaps that efficiently calculates the number of flows present in the network traffic over sliding windows. Our scheme introduces a continuous maintenance cost but, unlike previous proposals, can be queried in constant time. We have performed an evaluation using real traffic, and compared the cost in terms of memory and CPU to the state of the art Timestamp Vector algorithm. The Countdown Vector shows comparable accuracy with significatly lower costs.

The problem of determining if a packet has been originated from a set of filtered addresses has been traditionally addressed using Bloom filters. To evict filtered items after a given window, methods already existed based on replacing the bitmap of the Bloom filter for a vector of timestamps. We have presented an algorithm called Countdown Bloom Filter that features approximate information expiration. In the experiments, we have observed that small counter values suffice to achieve a filtering accuracy close to that achieved by precise expiration. Therefore, our method has the advantage of reducing memory requirements compared to the full timestamp storage approach, while obtaining similar accuracy. Another way to look that these results is that, comparing to precise estimation, our method can fit a larger number of positions in the same amount of memory, which increases filtering accuracy.

Our findings show that expiration accuracy exhibits a tradeoff worth exploring. We observe that, by decreasing expiration accuracy, we can increase the size of the data structures, and obtain a net gain in accuracy. We expect this approach to be applicable to adapt other measurement data structures to the sliding window measurement model. For example, this scheme can be easily applied to the algorithms proposed by Estan et al. in [65]. On the other hand, while we have performed an empirical analysis using network traffic traces, it would also be an improvement to obtain error bounds that apply to other scenarios.

Part III

Traffic Sampling

8

Introduction

As explained in Chapter 1, two complementary approaches exist in order to cope with overload in network monitors. Parts I and II have introduced resource efficient algorithms for traffic analysis. In this part, we turn our attention to traffic sampling as a means to reduce the load of the monitors.

As networks grow more complex and hard to manage, the deployment of devices that monitor network conditions has become a necessity. Network monitoring can aid in tasks such as fault diagnosis and troubleshooting, evaluation of network performance, capacity planning, traffic accounting and classification, and to detect anomalies and investigate security incidents. However, network traffic analysis is challenging in high-speed data links. In current backbone links, incoming packet rates leave very little time Additionally, storing all traffic is inviable; usually, operators only record traffic aggregates on a per-flow basis, as a means to obtain significant data volume reduction.

A paradigmatic example and, arguably, the most widespread flow-level measurement tool is NetFlow [3], which provides routers with the ability to export per-flow traffic aggregates. However, in today's networks, one can expect the number of active flows to be very large and highly volatile. Under anomalous conditions, including network attacks such as worm outbreaks, network scans, or even attacks that target the measurement infrastructure itself, the number of active flows can rise by orders of magnitude. Thus, not only must the router be able to process each packet very quickly, but must also maintain a potentially enormous amount of state. As a consequence, provisioning monitors for worst-case scenarios is prohibitively expensive [31].

The most widely adopted approach both to prevent memory exhaustion and to reduce packet processing time is to sample the traffic under analysis. For example, Sampled NetFlow [3] is a standard mechanism that samples the incoming traffic on a per-packet basis. Sampled NetFlow requires the configuration of a fixed (static) sampling rate by the network operator. The main problem of such an approach is that operators tend to select "safe" parameters that ensure network devices will continue to operate under adverse traffic conditions. As a result, the sampling rates are set with the worst-case scenario in mind, which harms the completeness of the measurements under normal conditions.

Several works have addressed the problem of dynamic packet sampling rate selection, which overcomes the drawbacks of setting static sampling rates by adapting to network conditions (e.g., [31, 63, 87]). Most notably, Adaptive NetFlow [63] maintains a table of active flows; when the table becomes full, the algorithm lowers the sampling rate and updates all table entries as though packets had been initially sampled at the resulting (lower) rate; flows for which the packet count becomes zero are discarded.

However, adaptive sampling schemes, including Adaptive NetFlow, are still not widely used. For example, Cisco's NetFlow still relies on static sampling. We believe that the main reasons for this are that existing adaptive sampling schemes are too costly in terms of CPU requirements, and rely on complex data structures and algorithms, which makes them less attractive for implementation in networking hardware (we review the related work in Section 9, while Section 9.1 presents Adaptive NetFlow in greater detail).

In this work, we turn our attention to flow-wise packet sampling [57] (also known as flow sampling), which allows us to find an elegant solution to the problem of adaptive sampling. We present a novel measurement scheme which we have named *Cuckoo Sampling* (Section 10) that performs aggregate per-flow network measurements and, when the state required to track all incoming traffic exceeds a memory budget, maintains the largest possible random selection of the incoming flows, i.e., *under overload, performs flow sampling at the appropriate rate*. Our algorithm can cope with the extreme data rates of today's fast network links. The data structure is extremely efficient both when the traffic conforms to the available memory budget, but also under overload, when flow sampling is necessary.

We provide analytic (Section 10.5) and experimental (Section 10.6 and Section 10.7) evidence of the efficiency of our proposal. An important finding of this work is that Cuckoo Sampling is both easier to implement and less resource demanding than adaptive packet sampling. Our scheme has smaller peak costs, can smoothly discard excess flows, and relies on very simple data structures and algorithms. Unlike Adaptive NetFlow and other alternative data structures, it exhibits an expected constant per-packet cost, i.e., its cost is independent of the memory budget. Additionally, our algorithm is very easy to parametrize, and is independent of the traffic profile, especially in front of attacks or aggressive traffic patterns. For all these reasons, we believe it to be more practical for hardware implementation within routers.

9 Related Work

A classical solution to reduce the load of network monitors is traffic sampling. Several sampling methods have been proposed in the literature; a review of the most relevant can be found in [57]. Perhaps the simplest and most widely used is uniform random packet sampling, which is easy to implement by generating a random value for each packet arrival, and can guarantee a reduction in the per-packet processing cost. However, it does not proportionally reduce the amount of memory when computing per-flow aggregates [63], which forces operators to set low sampling rates.

Trajectory sampling [58] provides a means to coordinate the sampling across several monitors along a packet path, using a pre-arranged pseudo-random hash function. Sampling decisions are based on the hash values of each packet; all monitors use the same hash function, so that packet selection is coordinated.

Flow-wise packet sampling [57], also known as flow sampling, requires each data flow to be either completely sampled or discarded. Achieving this effect might look challenging, but can also be effortlessly done using hash-based sampling. For every packet, a hash of the flow identifier is computed. If this value is below a certain threshold, the flow is sampled.

Each sampling method preserves certain characteristics of the input traffic. Sampling mechanisms should be considered to be complementary [57]. Packet sampling biases data collection towards large flows, and makes it very hard to recover per-flow statistical properties, such as the original flow size distribution [81]. On the other hand, flow sampling preserves flow aggregates, but is less accurate for applications such as volume based traffic accounting, or for heavy hitter detection, due to the heavy-tailed nature of flow size distributions [58]. Several studies analyze the impact of sampling on the accuracy of other monitoring applications, e.g., flow accounting [134] and anomaly detection [38, 98, 99].

Sampling methods have been proposed for specific traffic metrics, such as the number and average length of flows [59] or flow size distribution [60], and to detect flows of particular interest [29, 47, 64]. Another family of sampling algorithms operate after packets have been aggregated by routers. Ref. [61] proposes a sampling method to select, under hard memory constraints, a representative subset of the NetFlow records exported by a router, but this solution is not applicable for traffic sampling within routers themselves.

Statically setting sampling rates is problematic, since they are usually selected with worst-case traffic conditions in mind. An alternative solution is to adapt sampling rates according to traffic conditions. The most relevant related work to our solution is Adaptive NetFlow [63]. Given its relevance to our work, we devote Section 9.1 to introduce this proposal in detail. Flow Slicing [87] is a recently appeared technique that combines Sample and Hold [64] and packet sampling in a way that can simultaneously control memory usage and the volume of output results. The problem of adaptively choosing sampling rates in a system running multiple monitoring applications has been investigated in [31].

The abstract problem of sampling a pre-defined number of items from a set is called Reservoir Sampling [128]. Initially, all elements are sampled, until the reservoir is fully populated. Then, every new element replaces a randomly chosen element of the reservoir with a certain probability, so that every item is equally likely to be selected. Our technique can be seen as an efficient design of Reservoir Sampling to collect per-flow aggregates.

The algorithm we present stores items in an array and, as packets arrive, can relocate items to alternative positions within the array. This is reminiscent of the way the Cuckoo Hashing [107] data structure operates; hence, we have named our technique Cuckoo Sampling. Note however that this is merely anecdotal, since both data structures operate very differently.

9.1 Adaptive NetFlow

As has been explained, statically setting sampling rates is harmful for measurement accuracy, since it forces operators to choose sampling rates with worst-case scenarios in mind. A more sensible solution is to adapt sampling rates to traffic conditions. Adaptive NetFlow (ANF) is a proposal to implement adaptive packet sampling in routers that export traffic information via NetFlow records. Given its close relationship with our work, we devote this section to introduce it in more detail.

ANF initially samples all packets, and starts collecting flow aggregates in a table. The core idea behind ANF is that, when the memory becomes full, the packet sampling rate for future packets is lowered and, simultaneously, all existing flow entries are modified as though they had been initially sampled at the resulting rate, a procedure that is known as *renormalization*.

The objective of renormalization is to delete flow entries for which packet counts reach zero, thus freeing space for new entries. This process is repeated as necessary as new flows arrive, and runs in parallel with regular packet processing. Naive renormalization would involve binomial random number generation, which is costly. ANF tackles this issue by proposing a method that achieves similar results with a single coin flip per stored flow.

The choice of the new sampling rate is critical since, for a given sampling rate, the fraction of flows that will be discarded depends on the distribution of the traffic. For this reason, ANF also maintains a histogram of flow sizes. Given a target fraction of flows to discard f, and the flow size distribution, the new sampling rate can be computed. This implies that (e.g., if traffic measurement is about to end) the algorithm might unnecessarily discard up to f of its samples. Of course, this problem could be mitigated by provisioning the monitor with additional memory. We argue, though, that flow aggregate collection data structures must not only require a small number of memory accesses per packet, but should also be memory efficient to allow line-speed monitoring with fast (and therefore, more expensive) memory modules.

10

Cuckoo Sampling

In this chapter, we present an extremely simple data structure that is capable of performing adaptive flow sampling with a given memory budget. The main novelty of our proposal is the algorithm to update the data structure, which is also very simple, and requires considerably lower cost than Adaptive NetFlow.

10.1 Measurement of a Single Flow

Let us start with the assumption that we have memory for exactly one flow. Thus, we wish to devise an algorithm that randomly selects one flow from the traffic, i.e., operates with a reservoir of size 1. Under such a setting, reservoir sampling takes new items with probability $1/n$, where n corresponds to the number of arrivals so far. However, this scheme would require tracking active flows (e.g., in a hash table) in order to discern if a packet belongs to a new flow that can replace the existing one.

We can easily obtain a similar result without storing per-flow state by using hash-based sampling, with the intention of storing the flow with lowest hash. For every incoming packet, a pseudo-random hash of its flow identifier is calculated and compared against that of the currently stored flow. If it is smaller, the existing flow is discarded in favor of the newer. If it is larger, the packet is discarded. Finally, if it matches, the flow information is updated. As long as the hash function is perfectly pseudo-random, and has a large enough range, it is easy to see that the algorithm will select a flow randomly, given that all flows have exactly the same probability of obtaining the lowest value.

10.2 Arbitrary Reservoir Size

As explained, devising an algorithm that randomly selects one flow from the traffic is simple and computationally lightweight. However, network operators wish to obtain as much data as possible, according to the memory budget of the measurement device. Under normal operating conditions, this budget will be sufficient to track most active flows. However, the data structure must be robust to (possibly, sudden) increases in the number of flows well beyond the available memory budget. It is in such a scenario where we wish the data structure to perform random flow sampling at the appropriate rate.

We propose an initial step towards that goal by trying to apply the basic algorithm outlined in Section 10.1 and extending it to multiple flow measurement. One could hope to extend it straightforwardly to a reservoir of size r as follows. Maintain as many instances of the previous scheme as the memory budget allows. Then, use the hash function h on the flow identifier id to randomly pick an instance $i = h(id) \mod r$. Each instance then runs the algorithm described in Section 10.1 independently.

This approach presents important advantages. Firstly, it is based on a very simple data structure. Traditional hash tables or other kind of dictionaries, including those based on Cuckoo Hashing [107], require collision management. Secondly, its per-packet cost is constant, unlike those reservoir sampling algorithms that rely on maintaining secondary data structures to perform flow selection (e.g., a heap), where cost depends on the reservoir size. Third, unlike Adaptive NetFlow, it does not require periodic maintenance to renormalize entries, which would cause spikes in the overall cost.

Despite these advantages, this scheme is not effective because, when operating under normal conditions (i.e., no overload), it wastes a significant amount of memory. In particular, when the number of incoming flows is in the order of the available memory budget, i.e., the memory is well dimensioned to handle incoming flows, around $e^{-1} \approx 36.8\%$ of the memory remains unused due to collisions, as will be further discussed in Section 10.5. This approach would therefore require provisioning an additional $\approx 60\%$ of memory than would initially seem necessary.

10.3 The Complete Algorithm

In this section, we present the complete data structure and packet processing algorithm that form the core of our proposal. As explained, the main problem of the scheme outlined in the previous section is that a large amount of memory is wasted due to collisions precisely when the monitor is correctly dimensioned.

Our data structure is composed of an array of b buckets and $k + 1$ pseudo-random hash functions. Each bucket contains a flow hash, and the attached flow information, possibly including the flow identifier and aggregate statistics, such as the total number of packets or bytes, or any number of NetFlow equivalent fields. The per-packet operations are as follows (the full algorithm can be observed in Algorithm 13).

When a packet arrives, its flow identifier id is hashed by $k + 1$ pseudo-random hash functions. Functions $h_1..h_k$ have range $[0, b-1]$, and determine k positions in the array of counters. Additionally, hash function h determines what we call *the flow's hash value*, which must always be stored in the bucket where the flow will reside.

The k positions in the array of counters are verified. If a matching flow identifier is found, the corresponding entry is updated. Otherwise, the first empty position, if any, is used. When a new entry is created, the flow's hash $h(id)$ is recorded in the flow's bucket.

As more flows enter the data structure, the data structure starts facing overload, and the probability of finding an unused position decreases. When none of the k positions are empty, the algorithm performs the following procedure. First, it checks which of the k positions holds the largest hash. Let that position be L. Then, it proceeds to compare the hash stored in bucket b_L against $h(id)$.

If the stored hash is smaller than the current flow's, it means that the k positions where the current flow hashes all hold smaller hashes. Analogously to the scheme presented in Section 10.1, the current packet is discarded, and its flow will never have the chance to enter the data structure, as will be discussed later.

If it is not smaller, the packet will enter the data structure, and take position L. However, the flow stored in this position can not be simply discarded, as will be explained in Section 10.4. Instead, we attempt to re-locate the flow recursively. That is, again, we determine its k possible positions, and repeat the previous scheme. Note that this might, in turn, trigger additional relocation of other flow entries (we will show that the number of necessary relocations is very small in Section 10.5).

Algorithm 13 Cuckoo Sampling algorithm.

```
1:  function InsertAggregate($flow$, $value$)
2:      for i=1, K do
3:          $p \leftarrow h_i(flow)$
4:          $info \leftarrow table[p]$
5:          if $info$ undefined then                          ▷ found empty spot
6:              $table[p] \leftarrow \langle h(flow), flow, value\rangle$
7:              return
8:          end if
9:          if $info.hash = h(flow)$ then                    ▷ flow found, update
10:             $table[p].value$ += $value$
11:             return
12:         end if
13:         if $i = 1$ or $info.hash > max.hash$ then
14:             $max \leftarrow info$
15:             $maxp \leftarrow p$                           ▷ track largest hash
16:         end if
17:     end for
18:     if $h(flow) > max.hash$ then
19:         return                                  ▷ flow hash even larger, discard
20:     end if
21:     $table[maxp] \leftarrow \langle h(flow), flow, value\rangle$             ▷ insert
22:     InsertAggregate($max.flow, max.value$)                    ▷ relocate
23: end function
```

The main advantage of this scheme is that it greatly increases worst-case memory usage, since it provides greater opportunities for flows to occupy unused memory positions, compared to the previous scheme described in Section 10.2, as will be shown in Section10.5.

10.4 The Flow Sampling Guarantee

An alert reader might question the need to recursively relocate flows when another one with a lower hash value enters the data structure. Why can the data structure not simply discard the older flow? Let position p, occupied by a flow with identifier id, have been claimed by a flow that has a smaller hash value. The replaced flow hashed to the set of positions $P =$

$\{h_1(id), .., h_k(id)\}$, with $p \in P$. Then, consider the case that $h(id)$ is not the largest of the hashes stored in positions P, which, incidentally, is a common case. What would happen if the algorithm just discarded the existing flow, instead of relocating it?

When a new packet of flow id arrived, the algorithm would determine positions P. Since $h(id)$, as we previously assumed, is not the highest among P, the algorithm would determine that the packet should enter the data structure by replacing the now worst hash of P. Therefore, a new entry for flow id would be created. However, this entry would not aggregate all the packets of id, which clearly violates the objective of either sampling or discarding *entire* flows.

In other words, the recursive relocation procedure guarantees that, when a flow is discarded, it will never be considered for re-inclusion in the data structure. Additionally, since the decision to include or reject flows is based on pseudo-random hashes of flow IDs, the selection of flows is also pseudo-random. Therefore, under overload, the data structure performs random flow sampling.

10.5 Analysis and Parametrization

10.5.1 Memory Efficiency

We start by analyzing the memory efficiency of the algorithm with b buckets, $k = 1$, i.e., with a single hash function, and n incoming flows. The probability that a given bucket is never hit by a flow is $(1 - 1/b)^n \approx e^{-n/b}$. Thus, the expectation of the number of measured flows is $E[m] = b(1 - e^{-n/b})$. Of course, the data structure can only measure up to b flows. We can define the memory efficiency ratio of the data structure as $m/min(n, b)$. This expression is minimized when $n = b$, which renders e^{-1} buckets unused. That is, around 37% of the memory is wasted.

By introducing additional hash functions, this worst case is mitigated. Every additional hash function provides opportunities for flows to occupy alternative buckets, in case of collision with a flow with lower hash. We can provide a lower bound on the expectation of the number of used buckets from a simplified model of the algorithm.

In this model, if all k hashes of a previously unseen flow key map to a occupied location, the flow is immediately discarded without attempt at relocation (or, equivalently, the algorithm does not attempt to relocate evicted flows). It is easy to see that the actual algorithm can only obtain equal or higher memory usage.

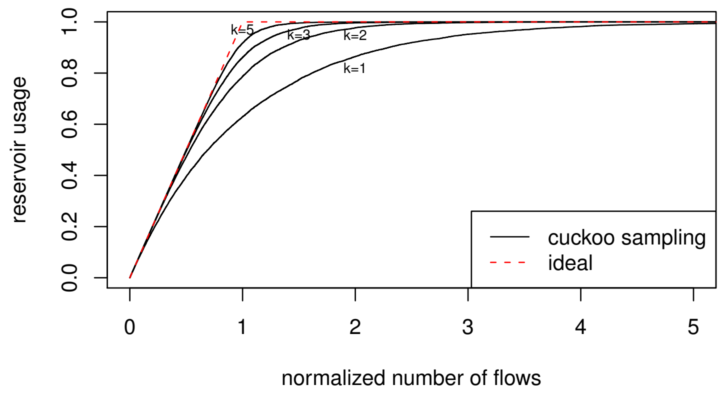


Figure 10.1: Reservoir usage obtained using cuckoo sampling with different number of hash functions.

Suppose n distinct keys have arrived, and let $X_n \leq n$ be the number of these that are stored. Then $\{X_n : n \geq 1\}$ is a pure birth process indexed by "time" n, with $X_1 = 1$, and transitions:

$$X_n \mapsto X_{n+1} = \begin{cases} 1 + X_n, & \text{with probability } 1 - p(X_n) \\ X_n, & \text{with probability } p(X_n) \end{cases}$$

where $p(x) = (x/b)^k$ is the probability that all k hash functions maps to an occupied location. The "lifetime" in each level x of X, i.e., the number of additional unseen keys before X increments, is then geometrically distributed $1/(1-p(x))$. Thus the average total "time" taken to get to a level z (i.e. the average number of distinct keys that result in z slots being occupied) is:

$$T(z) = \sum_{x=0}^{z-1} 1/(1 - p(x)).$$

In this approximation, we can then exhibit the store utilization as function of the number of distinct keys by plotting points with coordinates $(T(z), z/b)$ for $z = 1, 2, ..., b$.

The benefit introduced by each additional hash function diminishes, as can be observed in Figure 10.1. The figure presents the percentage of occupied buckets including relocations, as a function of the number incoming of flows. To avoid tying the figure to any particular case, we normalize the number of flows by dividing over the reservoir size.

10.5.2 Cost

Collecting traffic aggregates is computationally inexpensive; the bulk of the cost comes from managing the data structures and performing the necessary memory accesses. Hence, we measure the CPU cost of our algorithm in terms of memory accesses, which we analyze in this subsection.

When it runs out of empty buckets, our algorithm starts to recursively relocate items in the array of buckets. How large is this cost? Let us consider an array that is fully populated, i.e., it contains no unused buckets. This is clearly a worst case: if the array has empty positions, it presents more opportunities to cut the chain of successive relocations.

We consider the algorithm starts at recursivity level 1, and wish to calculate the expectation of the number of recurses that the algorithm will perform. Let P_i be the probability of advancing past recursivity level i, once it has been reached. In recursivity levels $i \geq 2$, the algorithm has already visited $x_i = i\,(k-1)+1$ buckets in the previous levels. In the current level, it is trying to relocate the largest hash it has encountered so far into one of the new $k-1$ positions (one position is shared with the previous level). Another relocation will be triggered if, in the new level, an even larger hash is found, which happens with probability $P_i = \frac{k-1}{x_i+k-1}$ for $i \geq 2$. Let us assume an also pessimistic $P_1 = k/(k+1)$, which corresponds to the case where the array is populated with random hash values that have never been replaced; P_1 can only be smaller in practice.

Then, the probability that the algorithm performs exactly i recurses is $R_i = (1-P_i)\prod_{j=1}^{i-1} P_j$, and the expectation for the number of recurses is $\sum_{i=1}^{\infty} iR_i$. It can be shown that each of the iR_i terms is smaller than $1/i!$ and, thus, the sum is smaller than e. Therefore, we can conclude that the expected number of relocations per flow arrival is a constant smaller than e.

We are more interested in the number of memory accesses, rather than number of recurses, but this number follows from our analysis. Every recurse requires k memory accesses, except when a free bucket is found. Hence we require an expected $e\,k$ accesses per flow arrival (i.e., first packet of each flow), and only k memory accesses for both (i) packets belonging to discarded flows and (ii) successive packets of sampled flows. Hence, in the worst-case scenario, where each packet belongs to a new flow (i.e., a DoS attack such as a SYN flood [95]), the per-packet cost of our measurement scheme is below $e\,k$.

Since, as explained earlier, small values of k are already practical to enhance memory usage, k can be considered to be a constant, just like e is. Therefore, our algorithm can be considered to be able to process packets linearly to the number of packets. Note also that this bound on the per-packet algorithm's cost has the remarkable property that it does not depend on the reservoir size. This feature clearly differentiates this measurement scheme from alternative approaches and, especially, Adaptive NetFlow. As for memory usage, our algorithm is linear to the reservoir size.

A series of optimizations could be considered to reduce the number of hash functions (i.e., memory accesses per packet) as the number of incoming flows severely outgrows the reservoir size. Two possible optimizations in such case would be as follows. First, we could periodically compute the largest stored flow hash. This way, packets belonging to a flow with higher hash could be immediately discarded, without checking stored hashes. Second, we could revert to $k = 1$. This would require moving each flow aggregate to the position indicated by the first hash function as new flows take its other positions. Note however that, inevitably, the algorithm would need to start with $k > 1$ to guarantee high worst-case memory usage. We argue that such optimizations, though useful, would have very limited benefit in practice, since the hardware of the monitor should still have to be dimensioned for the initial k. We further elaborate on this argument in Section 10.6.

We end by noting that, as revealed by the analysis, the probability of entering successive recursivity levels greatly diminishes. This means that, in an implementation of the technique, the number of relocations can be artificially cut, while incurring an arbitrarily small risk of damaging the measurements. The probability of performing i or more successive relocations is below $\sum_{j=i}^{\infty} 1/j!$, e.g., for 9 relocations, below $\approx 3 \times 10^{-6}$. This is desirable for ease of implementation with a hardware pipeline.

10.6 Simulation Results

The evaluation of our proposal is divided in two sections. In this one, we rely on synthetic traffic to understand the performance of our technique in a simple scenario. In Section 10.7, we will evaluate it with real traffic and present a few use cases that show how our technique preserves flow aggregates.

In this section we analyze the methods under synthetic traffic by generating a large number of 1-packet flows, which is a worst-case scenario for collecting traffic aggregates. Generally, successive packet arrivals are not overly interesting, since the sampling decision has already been taken, and have smaller cost. Additionally, this scenario mimics an extreme DoS attack, and puts the measurement algorithms under maximal stress. We set an example configuration with reservoir size 10000, and generate 5 times as many 1-packet flows.

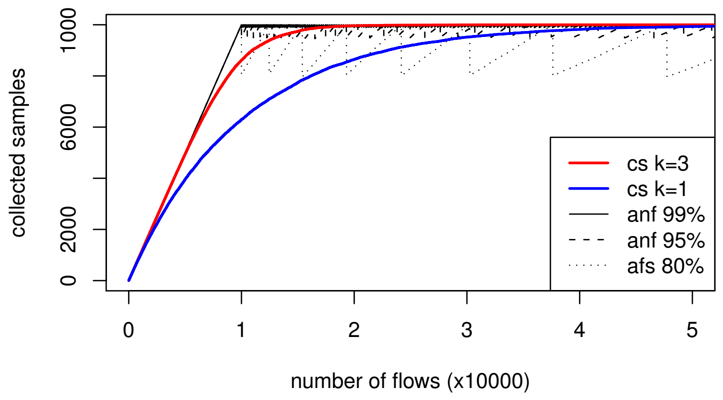


Figure 10.2: Sample size obtained with several configurations.

We have tested several configurations of Cuckoo Sampling (CS) with k hash functions (referred to as CS-k), and Adaptive NetFlow (ANF) with different α values (ANF-α), with $1-\alpha$ corresponding to the fraction of flows that ANF evicts in every renormalization (ANF is thoroughly described in Section 9.1). Our implementation of ANF assumes perfect knowledge of the flow size distribution, i.e., we do not implement nor count the necessary memory access to maintain a histogram of flow sizes. As for the flow table, we use a standard hash table with as many buckets as the reservoir size.

Figure 10.2 shows the sample size we obtained. CS-3 achieves higher than ANF-80% worst-case memory usage (on the 10^4th flow), but clearly outperforms it as more flows arrive. The figure includes CS-1 as a reference. Note that CS-3 performs as well as ANF-99% after 2×10^4 flow arrivals and, as will be shown later, has much lower CPU cost. For clarity of presentation, this figure excludes CS-5, but its behavior can be predicted from Figure 10.1.

We next analyze the number of memory accesses that each configuration requires. Figure 10.3 presents the CDF of the number of memory accesses of a few reference configurations. CS-3 outperforms ANF-80%, while having similar worst-case memory usage.

However, the CDFs fail to capture an important particularity of ANF, which is that its cost spikes when the maintenance procedure starts. Figure 10.4 shows the average memory accesses of incoming packets, binned in groups of 1000. The cost of ANF decreases in the long term. We argue that this is irrelevant, since a network monitor will require either the processing power to absorb the peak, or a large buffer capable of absorbing incoming packets while others are being processed. In contrast, CS does not present spikes, so it will demand less from the CPU, and will require almost no buffering. Note the logarithmic axis: ANF-99% peaks at 107;

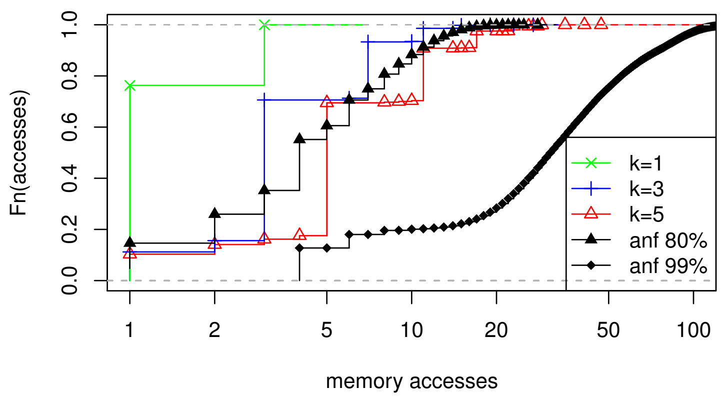


Figure 10.3: CDF of the number of memory accesses with several values of k, reservoir of size 10^4 and 5 times as many flow arrivals.

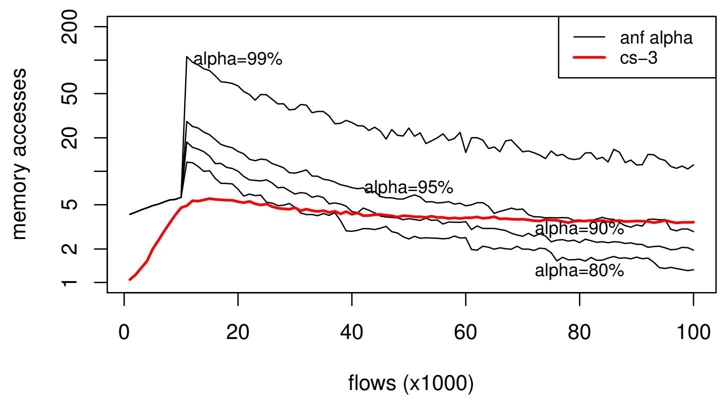


Figure 10.4: Average cost per flow in bins of 1000 flows.

ANF-95% at 28, while CS-3 is around 5.

Figure 10.5 analyzes the required necessary buffer to absorb any cost peaks, according to the processing capabilities of the monitor, expressed in number of memory accesses that can be performed per packet arrival. To provide a fair basis for comparison, we have implemented the normalization process of ANF as a background process, i.e., packets do not need to wait until normalization is complete to enter the data structure. The figure shows that CS is less resource demanding. For example, CS-5 severely outperforms ANF-99%, since it only buffers 8 packets using 11 accesses/pkt, whereas ANF-99% requires 124 accesses/pkt to achieve the same buffer usage (note the logarithmic axes). As discussed in Section 10.5.2, the number of relocations can be safely capped, which would reduce buffer usage for CS even further.

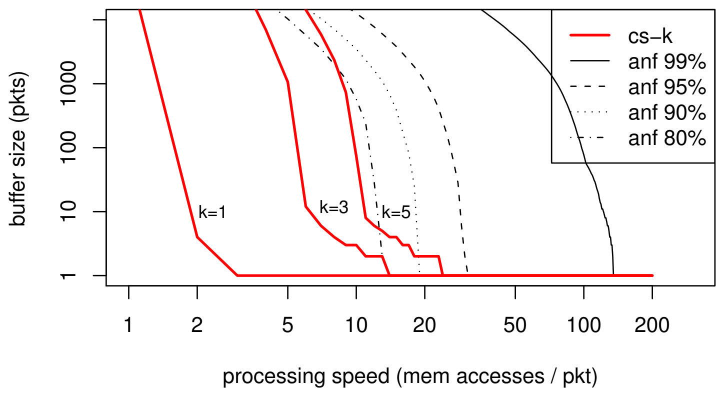


Figure 10.5: Buffer size required to sample 10^4 flows as a function of the processing speed.

10.7 Experiments with Real Traffic

In this section, we present experiments with real traffic. We deployed a series of CoMo [31] modules at the Gigabit access link of Universitat Politècnica de Catalunya–BarcelonaTech. We refer the reader to Appendix A for a more thorough description of our measurement scenario. Our experiments last 30 minutes of traffic, with around 250 million packets spread across just short of 7.9 million flows.

10.7.1 Cost

As a reference, for each flow, we computed the 5-tuple along with total number of packets and bytes (without sampling). We simultaneously ran Cuckoo Sampling with reservoirs of $r_1 = 10^5$ and $r_2 = 10^4$ flows, with $k = 3$. Average cost and its standard deviation were 2.94 and 0.40 for r_1, and 2.99 and 0.13 for r_2 (r_1 has lower cost because more packets belong to sampled flows and cause $\leq k$ memory accesses).

Note that the cost of CS is smaller than in the simulation results, where we assumed an extremely aggressive traffic mix of only one-packet flows. In this case, the average cost is slightly below the number of hash functions. This is due to the fact that a fraction of the packets find a matching flow in the first or second access. This fraction of packets is larger for r_1, since the reservoir is larger than r_2; hence its cost is slightly smaller.

Consistently with the analysis of Section 10.5 and the simulation results, very seldom do packet arrivals require a large number of memory accesses. Figure 10.6 shows the histogram of the number of relocations (note the logarithmic axis). For both r_1 and r_2, less than 13 packets

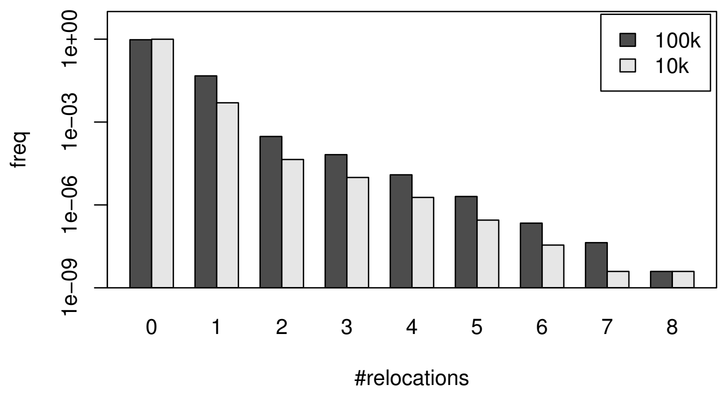


Figure 10.6: Histogram of the number of relocations triggered by packets with the two reference reservoir sizes.

triggered more than 6 relocations. Thus, a hardware pipeline could only implement 6 relocation steps with a negligible impact in the measurements.

We also ran our experiments with our implementation of Adaptive NetFlow (ANF). As has been established in Section 10.6, the cost of ANF varies with the number of incoming flows, and the monitor must be provisioned to absorb its peak cost. We computed the number of memory accesses per packet, as well as the periodic normalization. To include the cost of the normalization procedure, we uniformly spread it across the packets that arrive before the next normalization is triggered. The period of peak load then corresponds to the packets that arrive between the first and the second normalization.

Under normal traffic conditions, with reservoir size 10^5 and $\alpha = 0.95$, ANF had a worse average cost of 6.04 memory accesses per packet during this peak load period. However, as observed in Section 10.6, ANF's cost is sensitive to aggressive traffic profiles. To illustrate this issue, we ran several experiments replacing part of the traffic for a distributed denial of service attack. With a DoS using 25% of the bandwidth, this peak cost grew to 12.53; with 50%, to around 20. In both these scenarios, the average cost for Cuckoo Sampling stayed below 3 accesses per packet.

10.7.2 Collecting Traffic Aggregates

Figure 10.7 shows an example application of Cuckoo Sampling to estimate two kinds of flow aggregates. First, we compute the percentage of flows on each TCP port. The left figure presents the 50 ports with a highest amount of traffic, in decreasing order (*full traffic* line). We

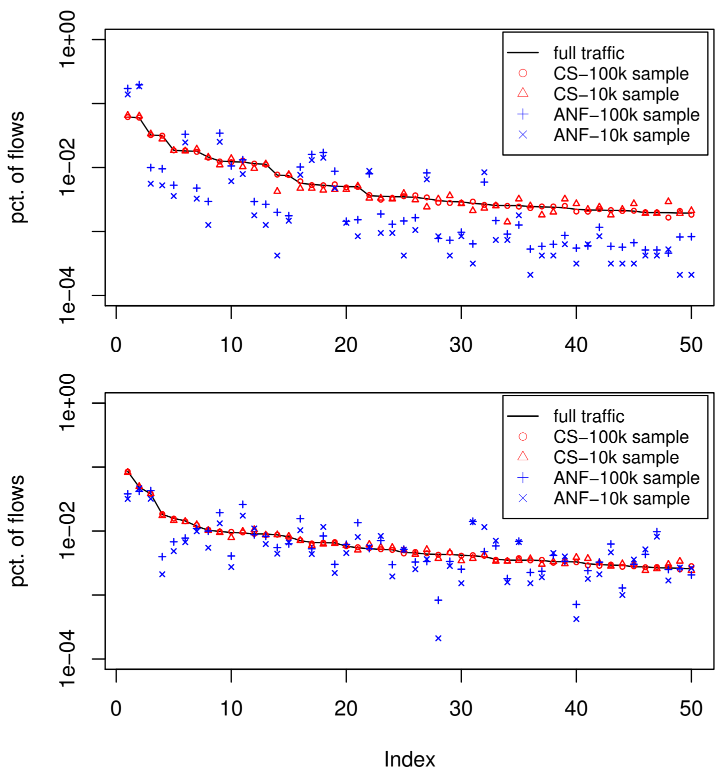


Figure 10.7: Percentage of flows of the ports with largest amount of flows (left) and of the subnetworks with largest amount of flows (right).

then estimate the percentage of flows using the results of gathering a sample 10^5 and 10^4 flows using both Cuckoo Sampling (CS) and Adaptive NetFlow (ANF).

Since CS is based on flow sampling, CS-based estimates are close to the real values, with the estimates obtained from the larger sample being more accurate. On the contrary, results from ANF are more error prone, given the biases introduced by packet sampling. In particular, points associated to ports where flows are larger tend to be overestimated (e.g., the first corresponds to HTTP traffic), and conversely, those that tend to be used by small ports are under-represented (e.g., the third point, which corresponds to DNS).

Figure 10.7 (right) presents the result of a similar experiment. This time, we group flows by internal class C subnetwork. Again, we plot the 50 subnets with largest amount of traffic, and show how these values can be correctly estimated from CS, but ANF is more error prone. Again, this is due to the biases introduced by packet sampling.

However, flow sampling methods should be considered complementary, and have to be

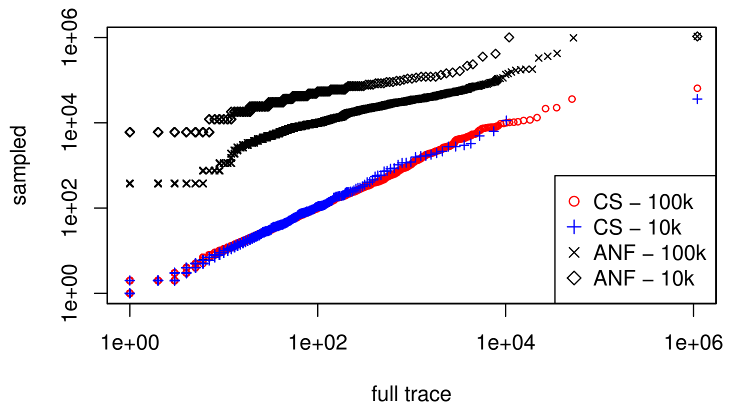


Figure 10.8: QQ plots comparing actual and sampled distribution of the number of packets per flow.

chosen depending on the particular application or metric that the monitor is calculating. Of course, packet sampling based methods such as ANF would estimate packet counts more precisely than flow sampling based ones.

It is well known that flow distribution estimation using packet sampling is a complex problem [97]. The main issue is that flow size distributions usually exhibit a long tail. Then, packet sampling biases measurements towards long flows, because it tends to always capture large flows and miss small ones. Figure 10.8 compares the actual flow size distribution against the one obtained after sampling by our method. As expected, our method correctly preserves flow size distribution, whereas ANF does not.

10.7.3 Anomaly Detection and Extraction

Another use case for our technique is anomaly detection and extraction. Recently, Ref. [37] proposed the use of a technique called frequent itemset mining to efficiently identify and present anomalous flows. Since operational networks carry large volumes of traffic, it is impossible for a human expert to review the full set of flows to discern which correspond to attention worthy events events, such as network attacks or other kinds of anomalies. An approach to ease this problem is to partition the traffic in clusters that encompass a large number of flows, and present this data for manual examination.

We use the publicly available [34] frequent item set mining tool SaM based on the Split and Merge algorithm [35], and run it using the set of TCP flows. As output, we obtain a reference data set where each cluster has at least 1% of the total flows. We then compare the results

src	sport	dst	dport	proto	#flows	rank	rank in sample			
							flow samp.		pkt samp.	
							10^5	10^4	10^5	10^4
79.205.92.135	*	171.52.0.0/16	22	TCP	3.8%	1	1	1	7	–
5.135.48.203	*	171.52.0.0/16	1433	TCP	3.1%	2	3	4	29	–
171.52.0.0/16	1433	5.135.48.203	*	TCP	3.1%	3	2	2	51	–
171.52.175.235	*	116.23.115.107	80	TCP	2.8%	4	5	3	1	4
116.23.115.107	80	171.52.175.235	*	TCP	2.8%	5	4	5	2	5
138.0.0.0/8	*	171.52.0.0/16	*	TCP	2.5%	6	6	6	6	–
66.233.148.83	*	171.52.0.0/16	32000	TCP	2.5%	7	8	7	–	–
171.52.0.0/16	*	138.0.0.0/8	*	TCP	2.4%	8	9	10	4	–
*	*	171.52.0.0/16	80	TCP	2.4%	9	7	–	3	–
171.52.87.0/24	*	*	*	TCP	2.3%	10	10	12	–	–

Table 10.1: Traffic clusters that encompass the largest amount of flows, with anonymized IP addresses

obtained with the full set of flows against those obtained from the $r_1 = 10^5$ and $r_2 = 10^5$ flow sample, collected via both CS and ANF.

The results we obtained can be observed in Table 10.1, which shows the 10 largest (in number of flows) traffic clusters reported by the algorithm with the full sample. IP addresses have been anonymized with a prefix-preserving anonymization algorithm [131]. The table includes the ranks when running the tool from a sample of the flows. Missing entries mean that the frequent itemset mining tool has failed to report a cluster.

The algorithm was able to identify all but one of the clusters from flow-sampled traffic. Note that the samples are quite small; the effective sampling rate for r_1 was around 1.27%, and for r_2, around 0.12%. Thus, the time required to extract the clusters was under 0.1 seconds for the smallest sample and around 1 second for the larger one, compared to around 85 seconds to process the full set of flows.

When running the algorithm on the flow data reported by Adaptive NetFlow, the algorithm is unable to identify the vast majority traffic clusters. This is a well expected result that is due to the fact that ANF relies on packet sampling, which tends to miss many small flows, and is skewed towards large flows.

11

Concluding Remarks

Dynamically adjusting sampling rates is crucial to extract as much information as possible from the input traffic, without exceeding the monitor's resources. The literature provided a packet-sampling based methods (most notably, Adaptive NetFlow) that followed this approach. However, adaptive packet sampling schemes have not been widely implemented, possibly due to their complexity in terms of both implementation and hardware requirements.

In this work, we turned our attention to flow-wise packet sampling, and presented a novel technique called Cuckoo Sampling that performs adaptive flow sampling. We propose a simple randomized data structure that has very small (constant) per-packet cost and is very easy to parametrize. Compared to previous approaches, it is based on an extremely simpler algorithm that can be expressed in 23 lines of pseudo-code and, most importantly, requires fewer hardware resources than adaptive packet sampling. A very notable feature of this algorithm is that its per-packet cost is independent of the size of the flow store.

Additionally, we have shown that it is suitable for implementation with a hardware pipeline. We have analyzed the method using both synthetic and real traffic to verify that the method behaves as predicted by the theoretical analysis. Our experiments included an extremely challenging traffic profile of 1-packet flows that mimicked a distributed denial of service attack, as well as regular traffic. As a conclusion of this work, we believe this method to be very practical and, in particular, to be suitable for implementation within routers.

Availability

An implementation of Cuckoo Sampling can be downloaded from `http://people.ac.upc.edu/jsanjuas/cuckoo_sampling/index.html`.

Part IV

Conclusion

12

Conclusions and Future Work

Passive network monitoring presents important challenges due to the extreme data rates of today's networks, which leave very little time to process each packet. As a consequence, network monitors are usually overloaded and require careful resource management to maximize their usefulness.

Network monitors need to carefully address the problem of overload, since they otherwise miss incoming packets, which degrades their measurement accuracy unpredictably. In this work, we have discussed two complementary approaches to mitigate this problem. First, the use of extremely efficient, specialized algorithms, that can capture metrics of interest with very little cost, while slightly reducing measurement accuracy. Second, to sample input traffic.

In particular, in the area of specialized algorithms, we have contributed in Part I a series of improvements to the Lossy Difference Aggregator (LDA) and a new data structure called Lossy Difference Sketch (LDS). The contributions we have presented on LDA seem to leave very little room for improvement in the configuration of this data structure. On the other hand, LDS allows obtaining per-flow measurements with great accuracy, especially under small loss. LDS represents a first step towards sketch-based delay measurement. Both LDA and LDS (which is based on LDA) suffer from a common problem: their small tolerance of packet loss. Devising an aggregation mechanism that is less sensitive to packet loss would be a great improvement.

In Part II, this thesis report presents a series of specialized algorithms that improve the state of the art in the problem of measurement over sliding windows. In particular, the problems of flow counting (Chapter 5) and traffic filtering (Chapter 6) have been investigated. Operating under the sliding window model poses an additional difficulty: how to give data structures a sense of time, so that they can expire information as it ages out of the measurement window. We have investigated a solution based on, rather than simply time-stamping data structure entries, attaching low resolution expiration timers. This idea allows us to adjust the precision of expiration. The main advantage of reducing expiration accuracy is that it allows to store more data structure entries, which in turn can increase accuracy. Further investigation of this tradeoff would be interesting and is left as future work. Our solution is similar to attaching timestamps with degraded accuracy. Comparing both approaches would also be an interesting piece for future work.

The other approach (besides the use of efficient, specialized algorithms) to deal with overload in network monitors is to reduce the amount of traffic to be processed by sampling it. Then, a crucial problem is how to choose the sampling rate. Statically selecting a sampling rate is undesirable, as it must be chosen with worst-case load in mind, which can be orders of magnitude large than the average. In this line, we have presented a flow-wise sampling algorithm called Cuckoo Sampling (Part III) that automatically adjusts the sampling rate according to a given memory budget and traffic conditions.

An algorithm already existed in the literature that was capable of providing adaptive sampling called Adaptive NetFlow. However, in practice, NetFlow capable hardware does not implement adaptive sampling, but operators are expected to select a static sampling rate. Possibly, this is due to the technical difficulties and overhead (even if reasonably small) of Adaptive NetFlow. Cuckoo sampling is based on a much simpler data structure and algorithm, and might ease the path towards the implementation of adaptive sampling in networking hardware.

Currently, uniform random packet sampling is the most widely used method (e.g., in NetFlow). We conjecture that this is because of two reasons: due to its simplicity of implementation, and because it is perceived to better preserve certain properties of the traffic. Investigating these would be an interesting piece of future work. In any case, Cuckoo Sampling dispells the notion that flow-wise packet sampling (flow sampling) is computationally expensive. In the case of adaptive sampling, it can for now be considered to be simpler to implement and computationally lighter than uniform random packet sampling.

Network monitoring can be considered to be far from mature as a research area. The main problem faced by the network measurement community is the lack of publicly available network traffic traces from operational networks. These public data sets are indispensable to ensure reproducibility of experiments and to allow for independent validation and cross-comparison of research results, especially considering the empirical nature of this research area. Nevertheless, most research works on network monitoring are performed and validated using different, undisclosed data sets. This breaks one of the basic principles of the scientific method, which is experiment reproducibility and full disclosure of experimental data [126]. Therefore, the rigorousness of this research area falls behind more traditional research fields, such as physics or biology, where reproducibility and independent validation of experimental results are a basic requirement.

Important barriers have to be overcome in order to increase the rigorousness of this scientific area. First, the mere acquisition of reference data sets is extremely challenging from a technological viewpoint due to the ever-increasing network speeds. Second, sociological reasons discourage their publication, primarily due to privacy concerns. Both the technological and sociological barriers are currently perceived to be insurmountable, which prevents the use of common data sets in scientific works and impedes experiment reproducibility.

A clear example of the necessity of sharing data sets can be found in the fields of traffic classification [105] and anomaly detection [27, 91], which are two classic research areas in network monitoring. In both these research areas, the performance of existing methods can be compared only when using identical traffic data. For example, several traffic classification studies try to infer which application generated each packet. In this case, a common data set labeled with the application responsible for each packet (commonly known as ground truth) is indispensable to validate and compare the performance of different traffic classifiers.

Although traffic data sets are vital to networking research, currently few public traffic traces from representative networks are available to the research community. We have identified the main reasons behind this lack of public traffic traces, which can be summarized as follows:

- Privacy concerns. This is the main reason why so few public traces exist. While anonymization methods exist, these destroy information that is vital for some research (e.g., traffic classification works often require access to packet payloads).
- Lack of incentives. Publication of network data sets is not mandatory in our research community, also mainly due to the privacy concerns that are perceived to be insurmountable.
- Technical difficulties. Network data collection in backbone networks requires specialized hardware that is costly and hard to operate.

A major consequence of this lack of public traces is that many research works are evaluated with private data sets. Therefore, the credibility of research in the field of network monitoring overly relies on the scientific community bona fide.

Some initiatives have attempted to provide public traffic data repositories [5, 10, 13, 16, 18, 20, 101]. However, due to the aforementioned reasons, these traces are of short duration (e.g., few hours at most) and are collected in small networks with a single network viewpoint. Usually, traces are obtained by researchers in academic networks, which makes it difficult to generalize results since they can be biased towards local network properties [72]. Moreover, the traffic anonymization process applied to public traces discards information that is indispensable in certain research works. For example, packet payloads are always removed from public datasets for obvious privacy reasons, but they are required for traffic classification research [105] (e.g., to build a ground truth) or signature-based intrusion detection [112, 116].

A way to solve these issues is to promote new data sharing models that allow for reproducible network research, while strictly observing the associated privacy issues. This model needs to be easy to implement and follow, especially from the point of view of the researcher; it must allow for agile development and be flexible to accommodate any experiments, and it also has to provide an easy means to verify that relevant privacy policies are observed.

The following data sharing model might suit these needs. Instead of publicly releasing anonymized traces, the members of the research community would provide, as a service to fellow researchers, access to a system capable of running their experiments from network traces (or even, live traffic). Instead of sharing the data, researchers interested in accessing network traces would send the computing tasks towards the data. Then, the host of the traces could send the results back to the researcher, after verifying that the privacy of the network data is not compromised (either by revising the source code submitted by the researcher, or by analyzing the output of the experiment).

Of course, this data sharing model would need to be automatized to the maximum degree, so that it does not become too burdensome for researchers, who already work under tight schedules. The standardization of a general purpose packet processing system would be an important first step toward this goal. Such system, for which the CoMo architecture is a great fit, should support a large number of packet capturing devices, and network trace formats. It should also provide an easy means to implement any monitoring task, and provide an interface for the host of network data sets to verify that all privacy policies are observed.

Once a new data sharing model exists and is widely accepted, either the use of available data sets, or the release of private data sets could be set as a standard requirement for publication in our area.

All the techniques presented in this document have been implemented on a modular network monitoring system, and deployed in operational network links (the deployment scenario is presented in Appendix A). As a final contribution of this Ph.D. work, we release the source code of this prototype system, together with an implementation of the techniques explained in this thesis. This software can be found in `http://monitoring.ccaba.upc.edu/mocome/`.

Bibliography

[1] Adobe Flash. http://www.adobe.com/. 155

[2] Alexa. http://www.alexa.com. 166, 176

[3] Cisco NetFlow. http://www.cisco.com/web/go/netflow. 1, 67, 75, 103, 104

[4] Corvil. http://www.corvil.com/. 39

[5] CRAWDAD: A Community Resource for Archiving Wireless Data At Dartmouth. http://crawdad.cs.dartmouth.edu/. 134

[6] Endace: DAG network monitoring cards. http://www.endace.com. 59, 79

[7] ipoque Protocol and Application Classification Engine. http://www.ipoque.com/products/pace-application-classification. ii, 161

[8] JDownloader. http://www.jdownloader.org. 161, 171

[9] Juniper Networks T series Core Routers Architecture Overview. www.juniper.net/us/en/local/pdf/whitepapers/2000302-en.pdf. 40

[10] MAWI Working Group Traffic Archive. http://tracer.csl.sony.co.jp/mawi/. 134

[11] MaxMind GeoLite Country. http://www.maxmind.com/app/geoip_country. 177

[12] Megaupload. http://www.megaupload.com/. 156

[13] NLANR Project. http://www.nlanr.net/. 134

[14] RapidShare AG. http://www.rapidshare.com/. 156

[15] RapidShare news. http://www.rapidshare.com/news.html. 169

[16] RIPE Routing Information Service data. http://www.ripe.net/projects/ris/rawdata.html. 134

[17] Team Cymru. http://www.team-cymru.org. 177

[18] The Internet traffic archive". http://ita.ee.lbl.gov/index.html. 134

[19] WinRAR archiver. http://www.rarlab.com/. 174, 176

[20] WITS: Waikato Internet Traffic Storage. http://wand.cs.waikato.ac.nz/wits/. 134

[21] IEEE/ANSI 1588 standard for a precision clock synchronization protocol for networked measurement and control systems, 2002. 42

[22] ALIZADEH, M. AND GREENBERG, A. AND MALTZ, D.A. AND PADHYE, J. AND PATEL, P. AND PRABHAKAR, B. AND SENGUPTA, S. AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of ACM SIGCOMM* (2010), pp. 63–74. 38

[23] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009. 169

[24] ALON, N. The Space Complexity of Approximating the Frequency Moments. *Journal of Computer and System Sciences 58*, 1 (Feb. 1999), 137–147. 17

[25] ANDERSON, P. What is Web 2.0?: ideas, technologies and implications for education . *JISC Technology and Standards Watch* (2007), 2–64. 155

[26] ANTONIADES, D., MARKATOS, E., AND DOVROLIS, C. One-click hosting services: a file-sharing hideout. In *Proceedings of ACM SIGCOMM IMC* (2009). 157, 159, 162, 167, 169, 173, 174, 177

[27] AXELSSON, S. Intrusion detection systems: A survey and taxonomy. Tech. rep., 99-15. Department of Computer Engineering, Chalmers University, 2000. 133

[28] BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. Models and issues in data stream systems. *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2002). 73

[29] BARAKAT, C., IANNACCONE, G., AND DIOT, C. Ranking flows from sampled traffic. In *Proceedings of ACM CoNEXT* (2005), no. i. 108

[30] BARFORD, P., BESTAVROS, A., BRADLEY, A., AND CROVELLA, M. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web 2*, 1 (1999), 15–28. 157, 158

[31] BARLET-ROS, P., IANNACCONE, G., SANJUÀS-CUXART, J., AMORES-LÓPEZ, D., AND SOLÉ-PARETA, J. Load shedding in network monitoring applications. In *Proceedings of USENIX Annual Technical Conference* (2007). 2, 32, 59, 75, 103, 104, 108, 121, 151, 161

[32] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (July 1970), 422–426. 87, 88, 92

[33] BOLOT, J. Characterizing end-to-end packet delay and loss in the internet. *Journal of High Speed Networks 2*, 3 (1993), 289–298. 14, 38, 67

[34] BORGELT, C. SaM frequent item set mining tool. `http://www.borgelt.net/sam.html`. 124

[35] BORGELT, C., AND WANG, X. SaM: A split and merge algorithm for fuzzy frequent item set mining, 2009. 124

[36] BORGNAT, P., DEWAELE, G., FUKUDA, K., ABRY, P., AND CHO, K. Seven years and one day: sketching the evolution of Internet traffic. In *Proceedings of IEEE INFOCOM* (Apr. 2009). 158

[37] BRAUCKHOFF, D., DIMITROPOULOS, X., WAGNER, A., AND SALAMATIAN, K. Anomaly extraction in backbone networks using association rules. In *Proceedings of ACM SIGCOMM IMC* (2009). 124

[38] BRAUCKHOFF, D., TELLENBACH, B., WAGNER, A., MAY, M., AND LAKHINA, A. Impact of packet sampling on anomaly detection metrics. In *Proceedings of ACM SIGCOMM IMC* (2006). 108

[39] BRISCOE, B. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review 37*, 2 (2007), 63–74. 170

[40] BRODER, A., AND MITZENMACHER, M. Network applications of bloom filters: A survey. *Internet Mathematics 1*, 4 (2004), 485–509. 87

[41] CARTER, J. L., AND WEGMAN, M. N. Universal classes of hash functions. *Journal of Computer and System Sciences 18*, 2 (1979). 77

[42] CATLEDGE, L. Characterizing browsing strategies in the World-Wide web. *Computer Networks and ISDN Systems 27*, 6 (Apr. 1995), 1065–1073. 157, 158

[43] CHA, M., KWAK, H., RODRIGUEZ, P., AHN, Y., AND MOON, S. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *Proceedings of ACM SIGCOMM IMC* (2007). 156, 157, 158

[44] CHOI, B.-Y., MOON, S., CRUZ, R., ZHANG, Z.-L., AND DIOT, C. Practical delay monitoring for ISPs. *Proceedings of ACM CoNEXT* (2005). 14, 38, 67

[45] CISCO SYSTEMS: SAMPLED NETFLOW. `http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/12s_sanf.html`. 5

[46] CLAFFY, K., BRAUN, H., AND POLYZOS, G. Tracking Long-term Growth of the Nsfnet. *Communications of the ACM 37*, 8 (1994), 34–45. 158

[47] COHEN, E., GROSSAUG, N., AND KAPLAN, H. Processing top-k queries from samples. *Computer Networks 52*, 14 (Oct. 2008), 2605–2622. 108

[48] COHEN, E., AND STRAUSS, M. J. Maintaining time-decaying stream aggregates. *Journal of Algorithms 59*, 1 (Apr. 2006), 19–36. 4, 73

[49] CORMODE, G., AND HADJIELEFTHERIOU, M. Methods for finding frequent items in data streams. *The VLDB Journal 19*, 1 (Dec. 2009), 3–20. 68

[50] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (Apr. 2005), 58–75. 43, 45, 58, 59, 67

[51] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (2003). 73

[52] CUEVAS, R., KRYCZKA, M., CUEVAS, A., KAUNE, S., GUERRERO, C., AND REJAIE, R. Is Content Publishing in BitTorrent Altruistic or Profit-Driven. In *Proceedings of ACM CoNEXT* (2010). 156, 159

[53] DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing 31*, 6 (2002), 1794. 76

[54] DE VITO, L., RAPUANO, S., AND TOMACIELLO, L. One-way delay measurement: State of the art. *IEEE Transactions on Instrumentation and Measurement 57*, 12 (2008), 2742–2750. 42, 59

[55] DENG, F., AND RAFIEI, D. Approximately detecting duplicates for streaming data using stable bloom filters. *Proceedings of ACM SIGMOD International Conference on Management of Data* (2006). 87

[56] DERI, L., AND FUSCO, F. Exploiting commodity multicore systems for network traffic analysis, 2011. 151

[57] DUFFIELD, N. Sampling for Passive Internet Measurement: A Review. *Statistical Science 19*, 3 (Aug. 2004), 472–498. 5, 104, 107

[58] DUFFIELD, N., AND GROSSGLAUSER, M. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking 9*, 3 (June 2001), 280–292. 14, 17, 21, 67, 107

[59] DUFFIELD, N., LUND, C., AND THORUP, M. Properties and prediction of flow statistics from sampled packet streams. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop* (2002). 75, 108

[60] DUFFIELD, N., LUND, C., AND THORUP, M. Estimating flow distributions from sampled flow statistics. In *Proceedings of ACM SIGCOMM* (2003). 108

[61] DUFFIELD, N., LUND, C., AND THORUP, M. Flow sampling under hard resource constraints. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (June 2004), 85. 108

[62] DURAND, M., AND FLAJOLET, P. Loglog counting of large cardinalities. In *Annual European Symposium on Algorithms* (2003). 76

[63] ESTAN, C., KEYS, K., MOORE, D., AND VARGHESE, G. Building a better NetFlow. In *Proceedings of ACM SIGCOMM* (Oct. 2004). 5, 104, 107, 108

[64] ESTAN, C., AND VARGHESE, G. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants , Ignoring the Mice. *ACM Transactions on Computer Systems 21*, 3 (2003), 270–313. 43, 67, 108

[65] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. *Proceedings of ACM SIGCOMM IMC* (2003). 3, 75, 76, 88, 100

[66] FANG, W., AND PETERSON, L. Inter-AS traffic patterns and their implications. In *Proceedings of IEEE GLOBECOM* (1999). 75

[67] FELDMANN, A., REXFORD, J., AND CÁCERES, R. Efficient policies for carrying Web traffic over flow-switched networks. *IEEE/ACM Transactions on Networking 6*, 6 (1998), 673–685. 157, 158

[68] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. 155, 176

[69] FINUCANE, H., AND MITZENMACHER, M. An improved analysis of the lossy difference aggregator. *ACM SIGCOMM Computer Communication Review 40*, 2 (Apr. 2010), 4. 15, 19, 26, 49, 67

[70] FRED, S., BONALD, T., PROUTIERE, A., RÉGNIÉ, G., AND ROBERTS, J. Statistical bandwidth sharing: a study of congestion at flow level. In *Proceedings of ACM SIGCOMM* (2001). 53

[71] FUSY, E., AND GIROIRE, F. Estimating the number of Active Flows in a Data Stream over a Sliding Window. In *Proceedings of the 4th SIAM Workshop on Analytic Algorithms and Combinatorics* (2007). 75

[72] GARCÍA-DORADO, J. L., HERNÁNDEZ, J. A., ARACIL, J., LÓPEZ DE VERGARA, J. E., MONSERRAT, F. J., ROBLES, E., AND DE MIGUEL, T. P. On the Duration and Spatial Characteristics of Internet Traffic Measurement Experiments. *IEEE Communications Magazine*, November (2008), 148–155. 134

[73] GARRETT, J. J. Ajax: A new approach to web applications, 2005. 155, 157, 158

[74] GIROIRE, F. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics 157*, 2 (Jan. 2009), 406–427. 76

[75] GOLAB, L., DEHAAN, D., DEMAINE, E. D., LOPEZ-ORTIZ, A., AND MUNRO, J. I. Identifying frequent items in sliding windows over on-line packet streams. *Proceedings of ACM SIGCOMM IMC* (2003). 79

[76] GOLAB, L., AND ÖZSU, M. Issues in data stream management. *ACM Sigmod Record 32*, 2 (June 2003), 5–14. 73

[77] GUHA, S., DASWANI, N., AND JAIN, R. An experimental study of the skype peer-to-peer voip system. In *Proceedings of IPTPS* (2006), vol. 6. 157, 159

[78] GUMMADI, K. P., DUNN, R. J., SAROIU, S., GRIBBLE, S. D., LEVY, H. M., AND ZAHORJAN, J. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of ACM SOSP* (2003). 158

[79] GUMMADI, P. K., SAROIU, S., AND GRIBBLE, S. D. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *ACM SIGCOMM Computer Communication Review 32*, 1 (Jan. 2002), 82. 157

[80] HAYES, B. The Britney Spears Problem. `http://www.americanscientist.org/issues/pub/the-britney-spears-problem`. 73

[81] HOHN, N., AND VEITCH, D. Inverting sampled traffic. In *Proceedings of ACM SIGCOMM IMC* (Feb. 2003), vol. 14, ACM. 107

[82] IANNACCONE, G., DIOT, C., GRAHAM, I., AND MCKEOWN, N. Monitoring very high speed links. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop* (2001). 73

[83] IANNACCONE, G., DIOT, C., MCAULEY, D., MOORE, A., PRATT, I., AND RIZZO, L. The CoMo white paper. Tech. rep., Intel Research Cambridge, Tech. Rep. IRCTR-04-017, 2004. 151, 152

[84] JACOBSON, V., LERES, C., AND MCCANNE, S. *libpcap*. Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release June 1994. Currently available at `http://www.tcpdump.org`. 79

[85] KARAGIANNIS, T., RODRIGUEZ, P., AND PAPAGIANNAKI, K. Should internet service providers fear peer-assisted content distribution? In *Proceedings of ACM SIGCOMM IMC* (2005). 159, 179

[86] KIM, H.-A., AND O'HALLARON, D. Counting network flows in real time. In *Proceedings of IEEE GLOBECOM* (2003). 75, 76, 79

[87] KOMPELLA, R. R., AND ESTAN, C. The power of slicing in internet flow measurement. In *Proceedings of ACM SIGCOMM IMC* (New York, New York, USA, 2005), ACM Press, p. 1. 104, 108

[88] KOMPELLA, R. R., LEVCHENKO, K., SNOEREN, A., AND VARGHESE, G. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *Proceedings of ACM SIGCOMM* (2009). 4, 13, 14, 15, 17, 18, 19, 20, 21, 23, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 38, 39, 42, 47, 48, 58, 67

[89] KONG, S., HE, T., SHAO, X., AN, C., AND LI, X. Time-out Bloom filter: a new sampling method for recording more flows. *Information Networking. Advances in Data Communications and Wireless Networks* (2006), 590–599. 87, 89, 90

[90] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (June 2004), 177. 2

[91] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing network-wide traffic anomalies. In *Proceedings of ACM SIGCOMM* (Oct. 2004). 133

[92] LEE, M., DUFFIELD, N., AND KOMPELLA, R. R. Not all microseconds are equal: fine-grained per-flow measurements with reference latency interpolation. In *Proceedings of ACM SIGCOMM* (2010). 39, 42, 60, 67

[93] LEE, M., DUFFIELD, N., AND KOMPELLA, R. R. Two Samples are Enough: Opportunistic Flow-level Latency Estimation using NetFlow. In *Proceedings of IEEE INFOCOM* (2010). 39, 42, 60, 67

[94] LEE, M., GOLDBERG, S., KOMPELLA, R. R., AND VARGHESE, G. Fine-grained latency and loss measurements in the presence of reordering. In *Proceedings of ACM SIGMETRICS* (2011). 46, 48

[95] LEMON, J., ET AL. Resisting syn flood dos attacks with a syn cache. In *Proceedings of BSDCon* (2002). 117

[96] LI, W., MOORE, A. W., AND CANINI, M. Classifying HTTP traffic in the new age. In *ACM SIGCOMM Poster Session* (2008). 156, 158

[97] LOISEAU, P., GONÇALVES, P., GIRARD, S., FORBES, F., AND VICAT-BLANC PRIMET, P. Maximum likelihood estimation of the flow size distribution tail index from sampled packet data. In *Proceedings of ACM SIGMETRICS* (2009). 124

[98] MAI, J., CHUAH, C. N., SRIDHARAN, A., YE, T., AND ZANG, H. Is sampled data sufficient for anomaly detection? In *Proceedings of ACM SIGCOMM IMC* (2006). 108

[99] MAI, J., SRIDHARAN, A., CHUAH, C. N., ZANG, H., AND YE, T. Impact of packet sampling on portscan detection. *IEEE Journal on Selected Areas in Communications 24*, 12 (Dec. 2006), 2285–2298. 108

[100] MARTIN, R. Wall street's quest to process data at the speed of light. `www.informationweek.com/news/infrastructure/showArticle.jhtml?articleID=199200297`. 38

[101] MCGREGOR, T., ALCOCK, S., AND KARRENBERG, D. The RIPE NCC internet measurement data repository. In *Proceedings of Passive and Active Measurement Conference* (2010). 134

[102] METWALLY, A., AGRAWAL, D., AND ABBADI, A. Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic. In *Proceedings of International Conference on Extending Database Technology* (2008). 76

[103] MOON, S., SKELLY, P., AND TOWSLEY, D. Estimation and removal of clock skew from network delay measurements. In *Proceedings of IEEE INFOCOM* (1999). 42

[104] MUTHUKRISHNAN, S. *Data Streams: Algorithms And Applications*. Now Publishers Inc, 2005. 73

[105] NGUYEN, T., AND ARMITAGE, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials 10*, 4 (2008), 56–76. 133, 134, 161

[106] OWENS, J., AND MATTHEWS, J. A Study of Passwords and Methods Used in Brute-Force SSH Attacks. Available on-line: `http://people.clarkson.edu/~owensjp/`. 87

[107] PAGH, R. Cuckoo hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144. 108, 112

[108] PAPAGIANNAKI, K., MOON, S., FRALEIGH, C., THIRAN, P., AND DIOT, C. Measurement and analysis of single-hop delay on an IP backbone network. *IEEE Journal on Selected Areas in Communications 21*, 6 (Aug. 2003), 908–921. 14, 21

[109] PARK, K., GITAKE, K., AND CROVELLA, M. On the relationship between file sizes, transport protocols, and self-similar network traffic. *Proceedings of International Conference on Network Protocols* (1996). 53

[110] PAXSON, V. *Measurements and analysis of end-to-end Internet dynamics*. PhD thesis, 1997. 14, 38, 67

[111] PAXSON, V. On calibrating measurements of packet transit times. In *ACM SIGMETRICS Performance Evaluation Review* (1998), vol. 26, ACM, pp. 11–21. 42

[112] PAXSON, V. Bro: a system for detecting network intruders in real-time 1. *Computer Networks 31*, 23-24 (Dec. 1999), 2435–2463. 134

[113] POUWELSE, J., GARBACKI, P., EPEMA, D., AND SIPS, H. The bittorrent p2p file-sharing system: Measurements and analysis. *Peer-to-Peer Systems*, Oct (2005), 205–216. 157, 159

[114] Reiss, F., and Hellerstein, J. Declarative network monitoring with an underprovisioned query processor. In *Proceedings of International Conference on Data Engineering* (2006). 73

[115] Robert, C. Y., and Segers, J. Tails of random sums of a heavy-tailed number of light-tailed terms. *Insurance: Mathematics and Economics 43*, 1 (2008), 85 – 92. 56

[116] Roesch, M. Snort–Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA* (1999). 134

[117] Rohatgi, V. *Statistical inference*. Dover Pubns, 2003. 21

[118] Saroiu, S., Gummadi, P., Gribble, S., and Others. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking* (2002). 157, 158

[119] Schneider, F., Agarwal, S., Alpcan, T., and Feldmann, A. The new web: characterizing AJAX traffic. In *Proceedings of Passive and Active Measurement Conference* (2008). 156, 157, 158

[120] Schulze, H., and Mochalski, K. Internet study 2007. `http://www.ipoque.com/resources`. 158

[121] Schulze, H., and Mochalski, K. Internet study 2008-2009. `http://www.ipoque.com/resources`. 158, 162, 164

[122] Schulze, H., and Mochalski, K. P2P Survey 2006. `http://www.ipoque.com/resources`. 158

[123] Sen, S., and Wang, J. Analyzing Peer-To-Peer Traffic Across Large Networks. *IEEE/ACM Transactions on Networking 12*, 2 (Apr. 2004), 219–232. 157, 158

[124] Sommers, J., Barford, P., Duffield, N., and Ron, A. Improving accuracy in end-to-end packet loss measurement. In *Proceedings of ACM SIGCOMM* (2005). 29

[125] Sommers, J., Barford, P., Duffield, N., and Ron, A. Accurate and efficient SLA compliance monitoring. In *Proceedings of ACM SIGCOMM* (2007). 14, 38, 67

[126] TREFIL, J. *Encyclopedia of science and technology*. Taylor & Francis Group, 2001. 133

[127] TUTSCHKU, K. A measurement-based traffic profile of the eDonkey filesharing service. In *Proceedings of Passive and Active Measurement Conference* (2004). 157, 159

[128] VITTER, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software 11*, 1 (Mar. 1985), 37–57. 108

[129] VON AHN, L., BLUM, M., HOPPER, N. J., AND LANGFORD, J. CAPTCHA: Using hard AI problems for security. In *Proceedings of Eurocrypt* (2003). 168

[130] WHANG, K.-Y., AND VANDER-ZANDEN, B. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems 15*, 2 (June 1990), 208–229. 76, 78, 82, 83, 85

[131] XU, J., FAN, J., AND AMMAR, M. Prefix-preserving IP address anonymization: Measurement-based security evaluation and a new cryptography-based scheme. In *Proceedings of IEEE Network Protocols* (2002). 125

[132] ZHANG, L., LIU, Z., AND HONGHUI XIA, C. Clock synchronization algorithms for network measurements. In *Proceedings of IEEE INFOCOM* (2002). 42

[133] ZSEBY, T. Deployment of sampling methods for SLA validation with non-intrusive measurements. In *Proceedings of Passive and Active Measurement Workshop* (2002). 14

[134] ZSEBY, T., HIRSCH, T., AND CLAISE, B. Packet sampling for flow accounting: Challenges and limitations. *Proceedings of Passive and Active Measurement Conference* (2008). 108

Appendix A

Measurement Scenario

The techniques presented in Parts I to III required deployment in operational, fast data links in high-speed operational networks. In this chapter, we present the scenario where the previous techniques have been deployed and evaluated. In the next chapter, we will present a series of measurements collected in this network.

The measurements have been collected at the Catalan National Research and Education Network (NREN), called Anella Científica, and at one of the institutions that are connected to this network: Universitat Politècnica de Catalunya – BarcelonaTech (UPC). Annela Científica is managed by the Center for Scientific and Academic Services of Catalonia (CESCA). This network connects UPC, together with many other Catalan research and education institutions, to the Internet via its Spanish counterpart, called RedIris. This Catalan NREN services upwards of 90 research and education centers, including several universities and a super-computing center, and with many different technologies that range from ADSL to Gigabit Ethernet. We estimate the number of users of the NREN to be in the order of a few hundred thousand. As for UPC's network, it connects 10 campuses, 25 faculties and 40 departments to the Internet, and services around 50,000 users.

We obtained access to two measurement points, one in each of these networks. We obtained access to the uplink of UPC, that connects it to Anella Científica via a full-duplex Gigabit Ethernet link. As for Anella Científica, we were given access its 10 Gigabit Ethernet link to RedIris, which connects UPC and Catalan research and education institutions to the rest of the Internet. Figure A.1 graphically depicts our measurement scenario. Unfortunately, the operators of either of the networks did not provide further details on the architecture of their network, which, in any case, are not particularly relevant to our experiments.

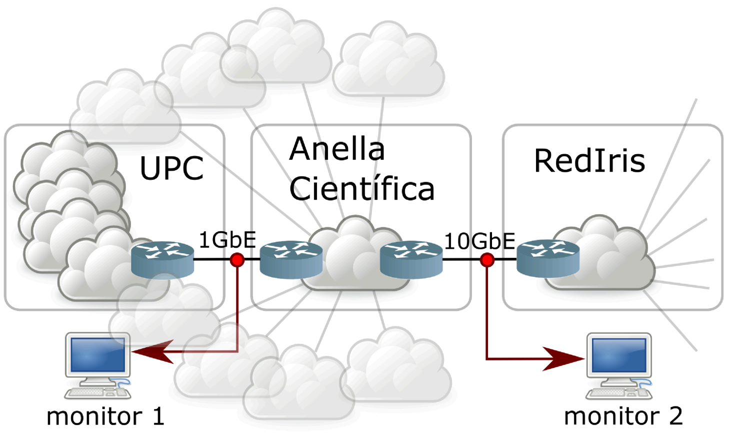


Figure A.1: Measurement Scenario

The monitoring setup was similar in both scenarios. In both cases, an optical splitter was deployed, which provided a physical means to replicate the traffic for analysis. Then, a copy of the traffic was sent to a monitoring box. These were based commodity PC hardware, except for the fact that they were equipped with specialized network cards for traffic analysis. In particular, Endace DAG cards were used in both measurement points. The monitor deployed at UPC was provisioned with a 1Gbps Endace DAG 4.3 card, that has two ports (one to monitor each direction of the link). In the case of Anella Científica, two 10Gbps DAG 5.2SXA cards were required, one per traffic direction.

The experiments presented in this document were performed using traces captured from these two monitors. This was for two primary reasons. First, it allowed for experimental reproducibility, and it provides a way to test many measurement algorithms (or configuration parameters thereof) concurrently and offline. The second reason is of a more practical nature, and given by the fact that the author of this thesis was not the sole user of the monitoring infrastructure. Hence, it was more convenient to collect a trace and later process it. Note however that all the algorithms are meant to run online, rather than from packet traces.

The experiments in Part I required timestamp synchronization of the two monitoring points, using the PTP protocol. This enabled the measurement of packet delays for the evaluation of both the improvements over the Lossy Difference Aggregator (presented in Chapter 2) and the Lossy Difference Sketch (Chapter 3). The monitor of Anella Científica was only used in these experiments. For the rest, including the evaluation of the techniques for measurement over sliding windows (part II) and Cuckoo Sampling (part III), only the UPC measurement point was used, as it was technically less challenging to deal with a 1Gbps link rather than a 10Gbps one.

Apart from the results presented in this document, we performed a traffic characterization study. However, to protect the privacy of the users of these networks, these results are not included in this document.

A.1 Implementation of a Network Monitoring System

Network monitoring techniques or traffic analysis studies require processing live traffic or network activity traces collected at high-speed backbone links. The design, implementation and deployment of network monitoring tools presents several challenges. Reference [83], which is next summarized, presents such challenges and proposes an architecture for a *general-purpose passive network monitoring system* called CoMo.

Currently, network monitoring software applications are designed to run stand-alone. This results in replication of effort, moreso given that designing and implementing monitoring applications is, as discussed throughout this document, challenging.

The CoMo system abstracts out various details of the underlying hardware architecture, the operating system, and packet collection mechanism. Monitoring applications are then implemented as pluggable modules that run on top of this architecture. By following this approach, monitoring applications need to be implemented only once, but can run on various hardware, from top-end to tiny sensors, and can use all the packet capturing mechanisms that systems can provide, including standard network interface cards, specialized monitoring devices (such as Endace DAG cards) or network card drivers (such as TNAPI [56]), or even NetFlow and sFlow data sources.

Additionally, this architecture allows the core to provide solutions to the challenges involved in high-speed network monitoring, which are automatically inherited by all applications implemented on top of it. The principal one is the management of the load of monitoring applications. CoMo offers several pre-defined input traffic sampling methods (including both packet sampling and flow-wise sampling), and provides mechanisms to allocate a fair share of computing resources to all applications running on top of it [31].

An open source software release of a prototype of CoMo was released by a development team led from Intel Research. This prototype was however inadequate for the needs of this research project. First and foremost, the official release enforced a rigid structure for the monitoring application modules, which were not allowed to e.g., allocate memory to use arbitrary data structures. An alternative version of the prototype was in development which did not present such limitations. However, it could not be considered stable for release, since it included complex features that demanded heavy development efforts to be fully implemented, and great debugging efforts to be considered stable, such as the support for application modules being written in higher level languages or to spread the load of the monitoring system across several computing nodes.

In the framework of the work behind this thesis, an alternative prototype of CoMo was developed that stood in a middle ground. The architecture and even a large part of the source code of this software tool heavily borrowed from CoMo and its development version. Unneeded features were stripped, so that the implementation effort required to obtain a stable release became manageable as a (mostly) solo project. This tool will be made available at `http://monitoring.ccaba.upc.edu/mocome/`.

We do not take credit for the architecture of CoMo, which is described in Reference [83]. We next provide an overview of the architecture of monitoring applications for reference. Application modules are broken in three distinct stages that run independently:

- **capture.** This stage of the application module processes input packets and generates intermediate results in the form of data "tuples" (containing any kind of information on the packet stream). No long-term aggregation or processing is performed. The main aim is to reduce the data volumes, so that the next stage can run faster.

- **analysis.** Receives tuples from capture, and performs long term aggregation. If possible, all computation-intensive steps should be pushed to this stage (a notable exception are tasks that require access to the packets, such as pattern matching in packet payloads). This stage generates the final results of the application modules, which are stored to disk.

- **query.** This stage is in charge of converting the records of the application to any desired formats; usually, to human readable text.

Example module application. We next describe an example module application called "top addresses". This application obtains a list of the addresses that sent the largest amount of data through the network link(s) being monitored. The source code is of this example application is of limited interest in the scope of this document, but it can be found in the release of this software tool. The source code has been thoroughly commented, so that the interested developer can understand every detail.

Capture. In this stage, the input packets are transformed to tuples of the form <source address, number of bytes>. This way, the packets are processed very quickly and discarded from the buffers. This stage performs short-term aggregation in relatively small time intervals.

Analysis. This stage performs long-term aggregation. It receives the tuples from capture, and maintains the count associated to each address for the pre-configured time interval. When the interval finishes, it dumps all entries to an array, sorts it, and stores the top-N addresses with largest counts.

Query. This final stage simply outputs the records in human-readable or machine-readable form. For example, a human readable output would be as follows:

```
address #1 a.b.c.d X GB
address #2 e.f.g.h Y GB
```

A machine readable format could be JSON, which would be of the form:

```
{ "a.b.c.d": X, "e.f.g.h": Y }
```

Appendix B

HTTP Measurements

It is commonly believed that file sharing traffic on the Internet is mostly generated by peer-to-peer applications. However, a preliminary web traffic analysis we performed on a backbone link of a research and education institution (that wished to remain anonymous) showed that HTTP based file sharing services are also extremely popular. We analyzed the traffic of this link for three months, and observed that a large fraction of the inbound HTTP traffic corresponds to file download services, which indicates that an important portion of file sharing traffic is in the form of HTTP data. Since this measurement study is not a central part of this Ph.D. thesis, we chose to include it as an Annex.

B.1 Introduction

The Hypertext Transfer Protocol (HTTP) [68] is the most popular and well-known application-level protocol in the Internet, since it is the basis of the World Wide Web. The HTTP protocol follows the classic client-server architecture, and it was originally designed to transfer content, typically in Hypertext Markup Language (HTML) format, from a web server to a user running a web browser.

However, many alternative applications and services based on the HTTP protocol have recently emerged, such as video streaming or social networking, and have rapidly gained extreme popularity among users, mostly thanks to the explosion of the Web 2.0 [25] and the development of technologies such as Asynchronous JavaScript and XML (AJAX) [73] and Flash [1]. Given this increasing popularity, recent research works have been entirely devoted to the study of the particular characteristics of the traffic generated by these novel web-based sources [43, 96, 119].

Traditionally, file sharing has never stood out among the multiple and diverse usages of the HTTP protocol given that, from a technological point of view, peer-to-peer (P2P) protocols are superior. However, while P2P accounts for a large percentage of the total Internet traffic, HTTP-based file sharing has recently gained popularity and is already responsible for a significant traffic volume, even comparable to that of P2P applications. Several popular *one-click file hosting* services, also known as *direct download* (DD) services, such as RapidShare [14] and Megaupload [12], are mostly contributing to this phenomenon.

Unlike P2P file sharing, DD services are based on the traditional HTTP client-server model. These sites offer a simple web interface open for anyone to upload any file to their servers. The sites then host the content and provide the uploader with a Uniform Resource Locator (URL), which can be freely shared without restriction (e.g., on public forums) thus enabling massive file sharing. These links are often called *direct download links* (DDL), to emphasize the difference with the links to P2P content, where downloads are usually queued and, therefore, do not start immediately.

Even though they rely on a simpler technological model, DD services offer a competitive advantage over P2P in that they are over-provisioned in terms of bandwidth whereas, in P2P applications, speeds are constrained by the availability of the content and the upstream bandwidth of the peering nodes. In order to generate revenue, one-click file hosting services rely on advertisements to users and, more interestingly, offer *premium accounts* that can be purchased to increase download speeds and bypass many of the limitations imposed by the sites to non-premium users.

In contrast, revenue in the P2P file sharing communities is generated by web sites where popular content is announced. For example, BitTorrent search engines and trackers rely on mechanisms including advertisements, users' donations and even spreading malware to generate revenue [52]. However, under this paradigm, individuals have no direct monetary incentive to host content nor, more importantly, the possibility of increasing their download speeds via payment.

Our study reveals that DD services are far more popular than expected. In the analyzed network, DD traffic constitutes more than 22% of HTTP traffic, and around 25% of all file sharing traffic. We also observed that the two most popular DD services, Megaupload and RapidShare, are in the Top-3 list (Top-1 and 3 respectively) in terms of served bytes. Perhaps surprisingly, we also found that a significant amount of users paid for premium membership accounts for unlimited, faster downloads, which is a market that the P2P community fails to exploit.

The popularity of one-click file hosting services was not apparent until very recently and, despite the large associated transfer volumes, little is known about them. The main objective of this work is to gain an understanding of the main characteristics of DD traffic, and the usage patterns of such services. Such an understanding can aid network managers in deciding whether to limit the access to such sites, understanding if traffic shaping policies would be effective or technologically easy to implement, and its potential impact on nowadays' networks.

To the best of our knowledge, so far, only one work exists [26] that specifically targets one-click file hosting traffic. The study compares DD traffic to P2P, and analyzes the infrastructure of one of the major DD providers (RapidShare). The detection of premium accounts is heuristic, based on the observation that premium users obtain more bandwidth. Our study extends the existing to another network environment, and complements its results by including another major provider (Megaupload), featuring an accurate premium account detection, and providing an analysis of the download speeds obtained by users and the kind of contents available on DD sites. We also discuss differences between HTTP and P2P based file sharing traffic that are of relevance to network managers. We find that DD tends to use more transit link bandwidth, since it does not exploit locality of user interest on files. Another important finding is that DD traffic often abuses Transmission Control Protocol (TCP) congestion control mechanisms to gain an unfairly large share of network resources under congestion.

In this work, we analyzed the HTTP traffic of a large research and education network during three months without interruption (from March to June 2009). Our measurement-based study mainly focuses on DD services and spans four major dimensions: traffic properties (Section B.3.3), user behavior (Section B.3.4), downloaded content (Section B.3.5), and server infrastructure and geographical distribution of the traffic (Section B.3.6).

B.2 Related Work

The study of the Internet traffic is an important research topic. While several works have been devoted to the analysis of traffic workloads of the most popular Internet applications during the last years (e.g., [30, 42, 43, 67, 73, 77, 79, 113, 118, 119, 123, 127]), comparatively few have studied the long term evolution of the Internet traffic. One of the most recent ones analyzed the traffic of a trans-oceanic link from 2001 to 2008 [36]. The study confirms that HTTP and P2P are currently the most dominant applications on the Internet. They have progressively replaced other protocols and applications, such as File Transfer Protocol (FTP) and e-mail, that carried a large part of the Internet traffic at the beginning of the nineties [46].

Ipoque has released three of the most extensive Internet traffic studies existing so far, which cover years 2006 [122], 2007 [120] and 2008-2009 [121]. While the 2006 and 2007 editions of the study highlighted P2P as the most dominant application on the Internet in terms of traffic volume, the 2008-2009 study reported a clear growth of HTTP traffic compared to P2P, which was mostly attributed to file sharing over HTTP.

Many applications run nowadays over HTTP, such as web-based e-mail clients, instant messengers, video streaming or social networks. Therefore, they have also been the main subject of study of several research works. For example, a measurement-based analysis of YouTube, which is currently the most popular video streaming service, was presented in [43], while [119] examined the particular traffic characteristics of several popular AJAX [73] based applications. Other previous works were devoted to characterize the behavior of web users (e.g., [30, 42, 67]), which is crucial for the proper design and provisioning of web-based services or to enable network optimizations.

A preliminary analysis of the application breakdown that runs over HTTP was presented in [96], which used two short traces of HTTP traffic collected in 2003 and 2006 at a research institution network. The study shows an increasing trend in the use of HTTP for activities other than traditional web browsing.

P2P protocols are widely used for file sharing, and are considered one of the main sources of traffic on the Internet. Therefore, they have been extensively studied in the literature. In [123], an analysis of the P2P traffic from a large Internet Service Provider (ISP) was presented. Traffic features similar to those studied in this work were analyzed for P2P applications, such as the number of hosts, traffic volumes, duration of connections and average bandwidth usage. Several measurement studies have concentrated in particular P2P networks. Ref. [118] focuses on Gnutella and Napster traffic. While both are P2P file sharing protocols, Gnutella is purely decentralized, and Napster relies on a centralized file location service. A significant fraction of peers are found not to share, and a high degree of peer heterogeneity is reported. An analysis of Kazaa traffic [78] attempts to model its workload, which is found to be very different to that of the Web. Currently popular P2P protocols have also been studied, including

BitTorrent [113] and eDonkey [127]. Finally, Skype [77] is another widespread P2P application that provides Voice over Internet Protocol (VoIP) services. Ref. [85] shows that P2P file sharing can significantly save bandwidth in transit links when locality is exploited, while a recent study investigates the incentives that drive content publication specifically in BitTorrent trackers [52].

While research works have already studied in depth the traffic of several HTTP and P2P applications, so far, only one work [26] specifically targets DD traffic and, in particular, one of the major one-click file sharing services (RapidShare) through a combination of passive and active measurements. Ref. [26] also features an interesting comparison with BitTorrent download rates, an analysis of RapidShare's load balancing policies, and an examination of the direct download links available in popular content indexing sites.

Our study complements the results of [26] using passive measurements on the traffic of a large academic ISP, with thousands of users of such services. Compared to [26], our study features a larger network scenario, accurate detection of premium accounts (instead of heuristic), and includes the top two DD services: Megaupload and RapidShare. Additionally, our study provides a deeper analysis of the data rates attained by users that reveals that premium users obtain higher download speeds abusing TCP congestion control mechanisms, and analyzes a large sample of the contents that were downloaded from DD sites.

B.3 Measurements

In this chapter, we present the measurements we have obtained. These are grouped in four sections, that analyze traffic volumes, the behavior of the users, the contents that are downloaded from these services, and the server infrastructure. We start by presenting the network scenario and the measurement methodology.

B.3.1 Network Scenario

The data presented in this work have been collected at the access link of a large research and education network. We avoid disclosing the name of the organization to preserve the privacy of the users of this network. Our measurement point averaged 442 Mbps including both incoming and outgoing traffic.

Table B.1 summarizes the most relevant features of the analyzed traffic. We have continuously ran our analysis between March 20 and June 20 2009. During these 3 months, our traffic

feature	value
Duration of study	3 months (March 20 to June 20 2009)
Traffic	148 TB in, 294 TB out
Avg. bw. usage	156 Mbps in, 310 Mbps out
Estimated p2p traffic	50.32 TB in, 235 TB out
HTTP traffic	78.85 TB in, 30.30 TB out
(DD only)	17.71 TB in, 0.89 TB out
HTTP queries	307.40 M in, 1321.56 M out
(DD only)	0 in, 7.59 M out
# Internet domains accessed	2.7 M

Table B.1: Principal features of the analyzed traffic

analysis software has tracked well over 1.6 billion HTTP requests and responses. Around 81% of these requests were made by clients inside the network under study.

In order to reduce the impact of local particularities of our measurement scenario, we restrict our study to outgoing HTTP queries and incoming responses. We thus choose to ignore HTTP traffic served by the monitored network, as well as the corresponding requests, in the rest of this study.

Our data will still be biased by the geographical location and the profile of the users of this network. Removing such bias would require an open worldwide measurement infrastructure, which is currently not available to conduct this kind of research. Our measurements are thus necessarily constrained to the scenario where the data has been collected.

It is important to note that this study cannot be done with publicly available traces for two primary reasons. First, public traces are usually anonymized and, thus, accessed web servers cannot be recovered. Second, such traces do not contain the HTTP headers, which are needed in this study.

B.3.2 Measurement Methodology

In this work, we have employed a passive traffic analysis approach. While the measurement methodology in itself is not a novelty of this work, we include several details that help understand how the measurement data was gathered. We were provided with access to a copy of the traffic that traverses the link between the described network and the Internet. In order to set up a monitoring node and quickly develop software tools for traffic analysis, we have used the CoMo [31] general-purpose passive network monitoring system. In particular, we have developed a set of CoMo modules that anonymously track and analyze all HTTP traffic.

In order to identify direct download (DD) traffic[1], we have compiled a large list of one-click file hosting domains that contains 104 entries, gathered by analyzing the source code of JDownloader [8] (a popular download manger that has support for several DD services) and browsing Internet forums where direct download links are published.

As will be shown, two download services outstand in popularity: Megaupload and RapidShare. For both these services, we have instrumented our measurement code to detect whether downloads correspond to paid visits by analyzing HTTP and HTML headers. However, we do not capture information that links such traffic to individuals.

In order to gain perspective on the proportion of file downloads, we also track the rest of the traffic. However, we do not collect detailed statistics. We limit data collection to the aggregate number of bytes per unique combination of IP protocol, source port and destination port in intervals of 10 minutes. This data is used to calculate the proportion of HTTP traffic compared to the total, as well as to compare the data volumes of HTTP traffic to that of P2P activity.

Traffic classification is technologically complex and requires access to packet payloads to obtain accurate results, while we only collect HTTP headers. This is an active topic of research (e.g., see [105]), and the application of sophisticated methods for accurate, on-line P2P identification was outside the scope of this work. Our analysis is centered on measuring file hosting services, and we only require an estimation of P2P traffic volumes for comparative purposes. Therefore, we elect to identify P2P with a deliberately simple well-known ports approach. We use the Internet Assigned Numbers Authority (IANA) list of ports to clearly identify non P2P traffic, and assume the rest corresponds to P2P.

Our approximate approach has two primary sources of inaccuracy. First, P2P traffic can mask as regular traffic running on well-known ports and, in particular, on TCP port 80. Second, not all traffic running on unknown ports is necessarily P2P. To verify the significance of these sources of inaccuracy in our network, we obtained access to 13 full-payload, bidirectional traces from the same network under study, which we fed to Ipoque's state-of-the-art Protocol and Application Classification Engine (PACE) traffic classifier [7]. According to PACE, only 5.7% of the identified P2P traffic used well-known ports, and only around 0.6% specifically TCP port 80. Conversely, 91.61% of the classified traffic not involving well-known ports was

[1]Throughout this work, we use the terms direct download and one-click file hosting to refer to HTTP-based file sharing services interchangeably.

P2P. Thus, it can be concluded that our approach provides reasonable traffic volume estimates in our scenario.

Privacy Concerns. Although the HTTP headers of each request and response are analyzed in order to extract relevant information such as the domain name being accessed, the URL being queried or the file sizes, we preserve the privacy of the individual users by implementing the following policies. First, all collected traffic is processed and discarded on-the-fly and only highly aggregated statistics are stored to disk. Second, the IP addresses of the hosts in the network under study are always anonymized prior to the analysis. We do not keep information that permits the recovery of original IP addresses or that can link individual users or hosts to their network activities.

Representativeness of the Data Set. Our dataset is inevitably, as almost any traffic analysis work, biased by the scenario where the data has been collected. A large majority of the traffic in our setting is generated by students. This might slightly overestimate the popularity of DD. However, in terms of traffic volume, we believe that our findings are still representative to a high degree, since independent traffic analysis highlight the growth of HTTP traffic, both in absolute terms and, in particular, relative to P2P. In particular, ref. [121] cites direct download as one of the main causes of this growth, together with the rise in popularity of video streaming. In this network, no traffic filtering policies are applied to P2P or HTTP traffic that can bias the results. In the cases where we observe a regional bias in the data set, we indicate it explicitly in the text. Our findings are consistent with prior work and, in particular, with the findings of [26].

B.3.3 Traffic Volumes

Figure B.1 presents a profile corresponding to one week of the overall traffic we have observed. As expected, the traffic profile is extremely time dependent. The traffic spikes during work hours, but it is significantly lower at nights and during weekends. A residential network would most likely show a slightly different profile, spiking outside work hours, but still low during nights.

The majority of inbound traffic, especially during working hours, corresponds to HTTP (53.2%). Note however the different profile of outgoing traffic, where peer-to-peer (P2P) is clearly dominant, taking 80.0% of the bytes. Our intuitive explanation for this fact is that,

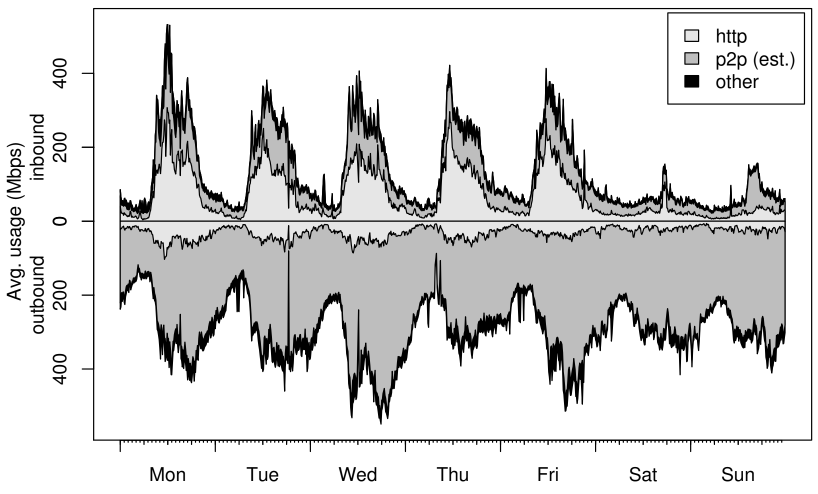


Figure B.1: Traffic profile for one week

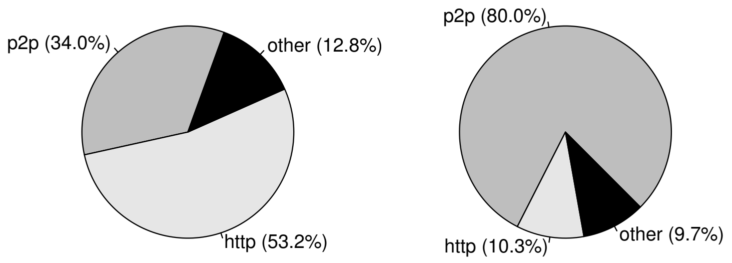


Figure B.2: Application breakdown for the inbound (left) and outbound (right) traffic

since this network has a huge uplink capacity, P2P nodes can serve large volumes of traffic, but cannot achieve comparable downstream speeds, given that downloads are constrained by the capacities of remote nodes (e.g., ADSL lines). In the downlink, P2P represents a comparatively smaller 34.0% of the incoming data. Figure B.2 presents the protocol breakdown of the traffic.

Downloads from one-click file hosting sites are taking 22.46% of all incoming HTTP traffic. Therefore, the common hypothesis that all file sharing is in the form of P2P traffic leads to a severe underestimation of the incoming data volumes that correspond to file sharing. Under the assumption that all P2P traffic corresponds to file sharing, 24.91% of all incoming file sharing traffic is in the form of HTTP responses. Note that this is an estimate, given that not all P2P traffic necessarily corresponds to file sharing, and that we perform heuristic P2P detection, as explained in Section B.3.2.

The observed data volumes are not particular to our scenario, and have been also recently

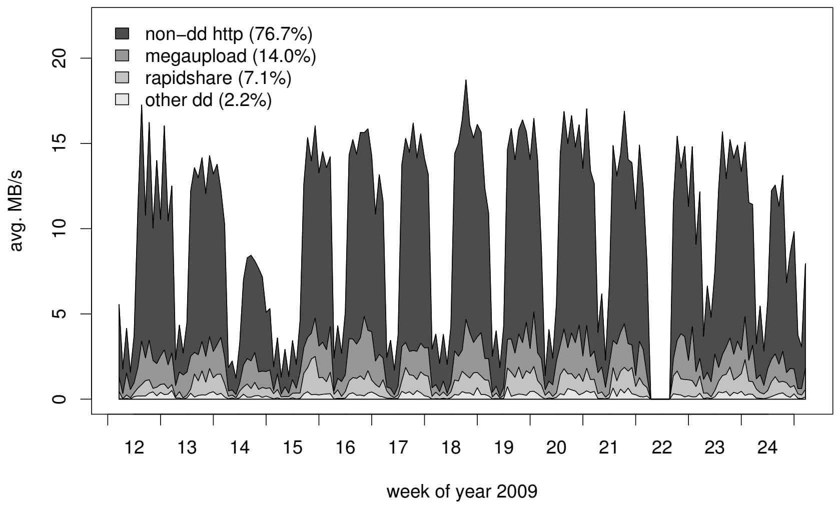


Figure B.3: Aggregate HTTP traffic volume (12-hour averages)

observed in commercial networks [121], where an important increase of HTTP, together with a decrease of the relative volumes of P2P traffic, is reported.

Figure B.3 shows a time series of the volume corresponding to the HTTP traffic we have analyzed, including both requests and replies, averaged across intervals of 12 hours. As expected, traffic volume significantly decreases during weekends, where most users of the network are away from the campuses. Week 14 also shows a significant drop in the traffic volumes, since it corresponds to a vacation period in the country where the network under study is located. During week 22, our network monitor suffered a service cut during approximately three days, and we therefore lack data for the corresponding time period. Finally, the downward trend at the end of the measurement period is a consequence of the diminishing academic activities associated to the end of semester. The figure also shows that RapidShare and Megaupload account for most HTTP file sharing traffic which, in turn, constitutes a large portion of all HTTP.

As discussed in Section B.3.1, we focus our analysis of HTTP traffic to only outgoing requests and incoming responses. By ignoring HTTP traffic served by the monitored network, we avoid obvious biases in our results (e.g., the list of the most popular domains would be dominated by those served locally).

Table B.1 showed that our traffic analysis software tracked approximately 1321 million outgoing HTTP requests. Figure B.4 (top) shows information about the top Internet domains in terms of served volume of data over HTTP. The key observation that motivated this study was

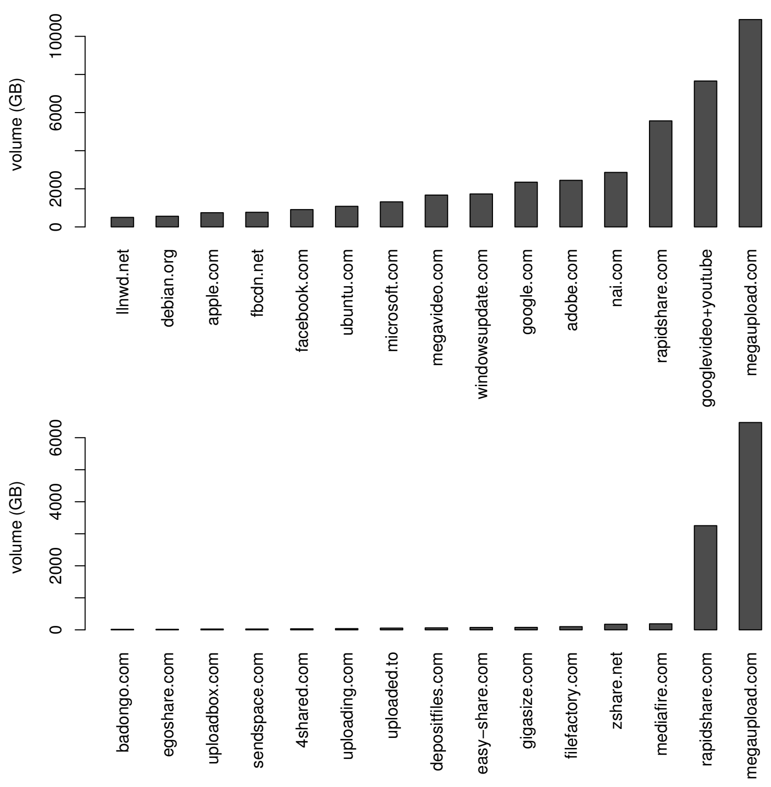


Figure B.4: Top 15 sites by served volume (top) and by served volume including only file hosting domains (bottom)

that the first and third top domains correspond to one-click file hosting services (Megaupload and RapidShare), even above domains as popular as search engines, social networks or video streaming services.

Even though one-click file hosting sites generate huge volumes of data, they cannot compete in popularity with the mentioned domains, as can be observed in Figure B.5. In particular, Figure B.5 (top) shows the top 15 domains ranked by number of hits. This rank is populated by domains that tend to attract a larger number of clients, but also favors sites that serve the content over multiple connections (e.g., using AJAX to incrementally update page contents). Figure B.5 (bottom) contains the top 15 domains in number of unique IP client addresses. This list introduces sites that serve content to automatic software update agents as well as domains that are linked to by a large number of websites, such as statistics collection or advertising services.

Table B.2 shows the ranks of the domains found in Figure B.4 (top) according to several

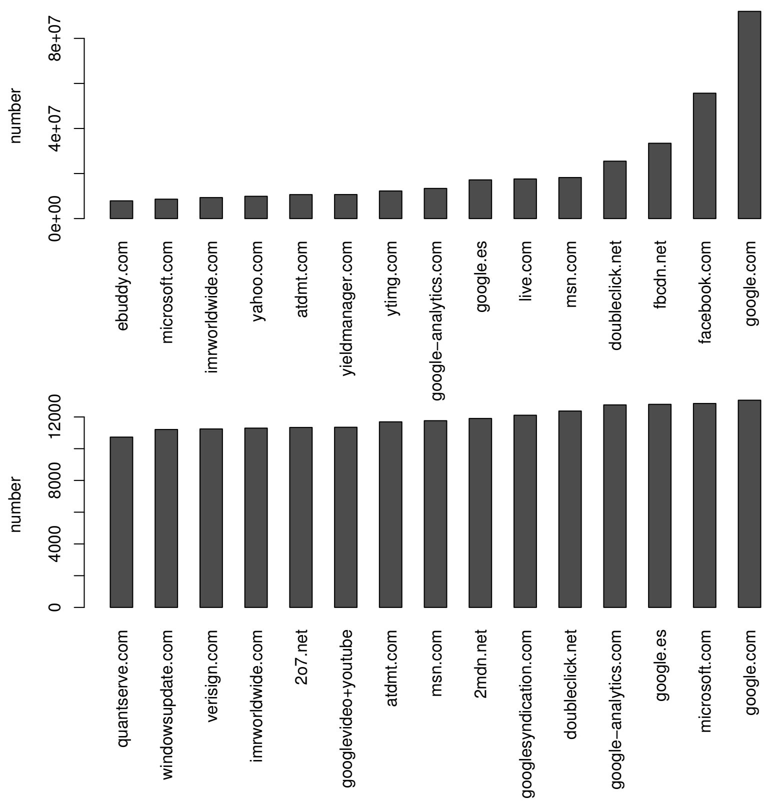


Figure B.5: Top 15 sites by number of hits (top) and number of clients (bottom)

criteria. We have also included, as a reference, the Alexa "traffic ranks" [2] of each of the domains for the country where the network is located. Alexa is a company that collects information about the popularity of web sites. They offer a toolbar that users can install as an add-on to their web browser, which collects information about their web browsing activities. This service constitutes the best available source of information on website popularities to the best of our knowledge.

Figure B.4 (bottom) focuses only on one-click file hosting services. While the amount of companies that offer file hosting services is rather large, the market appears to be dominated by the already mentioned Megaupload and RapidShare by an overwhelming margin. Together, they account for more than 90% of the one-click file hosting traffic volume and over 20% of all incoming HTTP traffic.

site	DD	Ranks			
		volume	#hits	#clients	Alexa
megaupload.com	✓	1	31	262	30
googlevideo+youtube	-	2	21	10	6
rapidshare.com	✓	3	49	304	25
nai.com	-	4	146	50	48104
adobe.com	-	5	83	17	89
google.com	-	6	1	1	4
windowsupdate.com	-	7	100	14	28
megavideo.com	-	8	239	467	15
microsoft.com	-	9	17	2	28
ubuntu.com	-	10	278	290	1897
facebook.com	-	11	2	31	5
fbcdn.net	-	12	3	33	3620
apple.com	-	13	44	148	108
debian.org	-	14	472	754	4328
llnwd.net	-	15	118	47	7108

Table B.2: Ranks of the top content serving domains

B.3.4 User Behavior

So far, HTTP data have been presented aggregated or on a per-domain basis. In this section, we focus only on direct download (DD) traffic and, in particular, on the behavior of the users of these services. Although a large number of sites exist that offer DD services, we restrict our analysis to Megaupload and RapidShare, since as shown in Section B.3.3, they account for more than 90% of all DD traffic in our network.

We extended our traffic analysis software to include HTTP session tracking capabilities. In particular, we instrumented our code to detect logins to Megaupload and RapidShare, and to detect whether the users have signed up for premium membership accounts. In order to protect the privacy of the users, while still enabling us to track user behavior, we anonymize the user names prior to the analysis, and we avoid storing any information linking connections to individuals. Ref. [26] instead performs heuristic account detection based on the download speeds attained by clients; we therefore expect our measurements to be more accurate.

We did not enable our login tracking feature until May 28. Since then, we observed around 0.88 million accesses to Megaupload, which resulted in 2.79 TB of downstream traffic and

0.30 TB of upstream traffic, and 0.74 million accesses to RapidShare that served 1.50 TB and received 0.12 TB.

Paid Membership. File hosting services generally offer downloads for free. However, they artificially limit the performance of their service and, in order to generate revenue, offer paid *premium* accounts that bypass such restrictions. These limitations include bandwidth caps, restrictions on the data volumes that can be downloaded per day, as well as delays in the order of 30 seconds before users can start the downloads. While this limited version of the service can still appeal to a casual user, heavy users are compelled to obtain a premium account for several reasons:

1. they can download an unlimited or very large number of files per day (e.g., up to 5GB per day in the case of RapidShare);
2. they are able to simultaneously download several files;
3. they are allowed to pause and resume downloads using HTTP file range requests;
4. downloads can be accelerated by fetching a single file using multiple locations, using software download managers;
5. they can upload larger files (e.g., the limit in RapidShare increases from 200MB to 2GB and, in the case of Megaupload, from 500MB to an unlimited file size);
6. they can automate downloads, e.g, users are not required to solve CAPTCHAs [129] prior to each download;
7. premium accounts bypass download delays.

RapidShare premium accounts can be easily detected by checking for the presence of a cookie in the HTTP request.[1] It is slightly more complex in the case of Megaupload, since they offer two types of accounts: “premium” and “member” accounts (that have some over advantages over unauthenticated access). In their case, our analyzer tries to match two patterns characteristic to each of the account types to the first bytes of each HTTP response from Megaupload servers.

[1]According to our data, RapidShare enhanced their login procedure on June 8th. As of this writing, sessions cannot be tracked with this simple approach.

According to our data, 470 (around 10%) users of DD services in our network have paid for premium accounts, an observation that is consistent with [26]. Although premium users are a minority, they contribute to a large portion of the download traffic on both Megaupload and RapidShare. In our data, premium users account for 29.6% of the total download traffic.

B.3.4.1 Bandwidth

The primary reason why DD links are popular is that downloads start immediately upon request (especially for premium users) and at a sustained high speed, compared to peer-to-peer systems, where download speeds are highly influenced by the number of peers that have pieces of the content and their available uplink bandwidth. Figure B.6 plots the Cumulative Distribution Function (CDF) of the download speeds that unregistered users reach when downloading from Megaupload (top) and RapidShare (bottom). It is apparent that RapidShare throttles connections. Two bandwidth limitations can be clearly appreciated in the figure: one around 125KB/s and another around 225KB/s, due to RapidShare changing their policies on the constraints applied to non-premium users during this study [15]. Ref. [26] does not identify the lower bandwidth cap, since it was introduced during May 2009, after their study had finished. No such throttling appears to be done by Megaupload, where download speeds average around 200KB/s. Usually, peer-to-peer networks can only achieve such download speeds on extremely popular content. However, the popularity of a file does not directly affect the rate at which it can be downloaded from DD sites. An interesting comparison of the data rates obtained using RapidShare versus BitTorrent can be found in [26].

As expected, premium users obtain considerably higher download rates, as can be seen in the figures. Premium downloads average around 600KB/s from Megaupload and around 2200KB/s from RapidShare. These higher data rates are achieved for two main reasons. First, in the case of RapidShare, premium downloads are not artificially throttled and are served at the highest possible rate. Second, in both services, premium users are allowed to open multiple connections to the servers, thus increasing their chances to get a larger share of network resources.

The gain obtained by opening multiple, simultaneous connections is caused by TCP congestion control mechanisms [23], which are designed to allocate an equal share of network resources to each connection. In other words, TCP is fair to connections, but not directly to users. As a result, users can unfairly increase their overall bandwidth share by parallelizing

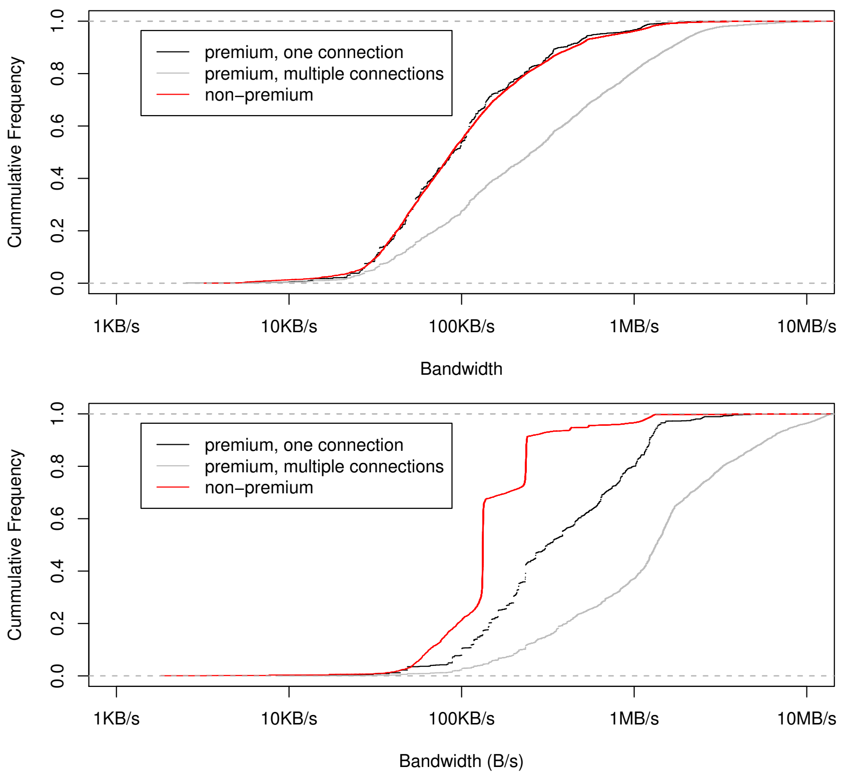


Figure B.6: Bandwidth obtained by Megaupload users (top) and RapidShare users (bottom)

downloads, at the expense of users who do not resort to this practice. This is a known problem to network managers [39] and a widespread practice among DD users, as will be seen in section B.3.4.2.

Figure B.6 also shows the CDF of the data rates achieved by premium users when opening a single connection. This way, we can isolate the throttling mechanisms used by each site from the effects of TCP congestion control algorithms. The figure confirms that Megaupload does not give special treatment to connections from premium users, since, when opening a single connection, their data rates are equal to non-premium users (who are restricted to one connection). One can therefore conclude that, at Megaupload, the performance gains that premium users obtain over non-premium ones are exclusively a consequence of their ability to parallelize downloads. On the contrary, RapidShare treats connections differently, serving premium connections more than twice as fast. However, RapidShare premium users can still benefit from better data rates by opening several concurrent connections, as can be observed in the figure. Thus, in DD services, users have a clear incentive to open multiple connections, often with the aid of software download managers, which we discuss in the next section.

B.3.4.2 Use of Downloaders

Mass file downloading from DD services is time-consuming and ineffective. Unlike P2P downloads, where the software usually handles download queues, in the case of DD the client is a regular web browser, which does not have such capabilities by default. This inconvenience is aggravated because of the fact that, very often, large files are broken into several pieces by the uploader before reaching the DD servers, in an attempt to overcome the upload file size limitations. Such files must be independently downloaded by the user. Thus, even if a user is interested in one particular content, she is often forced to download several files.

For this reason, software download managers exist that are specialized on DD services. For example, RapidShare and Megaupload not only do not discourage their use but, instead, they offer their own download managers to their user base for free. Third party download managers that are compatible with several DD sites are also available (e.g., JDownloader [8]). The principal reason to use them is that they automate download queues. However, another important reason to use a download manager is they often provide download acceleration by simultaneously retrieving several files, or even opening several connections in parallel to download a single file, achieving noticeably higher download speeds by abusing TCP congestion control mechanisms to gain an unfair share of network resources, as discussed in section B.3.4.1.

Figure B.7 presents the distribution of the number of simultaneous connections for premium users. While in more than 50% of the cases only one download was active per user, there is a non negligible amount of parallel downloads.

B.3.5 Downloaded Content

This section is devoted to the analysis of the contents that are downloaded from direct download (DD) services. In order to protect the privacy of the users, while we analyze file names, their extension, and size, we discard any information on which user has downloaded each file.

B.3.5.1 File Types

We obtained a list of file names being downloaded, which is trivially found in each HTTP request. We also recorded their associated download volumes and number of requests. In

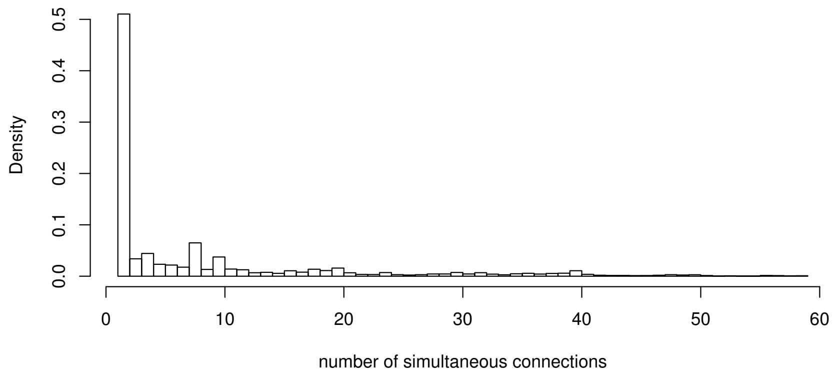


Figure B.7: Number of simultaneous connections for premium users

total, our traffic analysis module has captured 181249 unique file names, all of them hosted at Megaupload or RapidShare.

Table B.3 presents a list of the most prevalent file extensions. Notably, it can be observed that RAR archives are overwhelmingly the most prevalent (almost 75% of files), followed by the video extension AVI (around 9%).

We classified files into the following categories: $video$, $music$, $software$, $document$, and $image$. An additional $unknown$ category was introduced for the cases where we were unable to classify a file.

In an effort to identify the kind of contents that are being distributed within RAR archives, we stripped the ‘partX’ sub-string of each RAR file name (if present) and removed duplicates. Then, we randomly drew 3772 samples (around 3% of all RAR archives) out of the resulting list and manually classified their content type. Still, we were unable to classify around 43% of the RAR archives of our sample, due to file names being too cryptic or generic to determine the category of the content.

For the rest of the files (i.e., all but RAR archives), we classified their content type according to their extensions. In this case, we easily identified the content type of 80% of the files, since well known file extensions are prevalent.

Figure B.8 (left) shows the content breakdown in terms of number of files, assuming that the sample of RAR files we classified is representative. We classified around 40% of the files as video, 9% as software, 7% as music and 4.5% as documents. Figure B.8 (right) presents the

extension	#files	%	extension	#files	%
pps	66	0.04%	rev	358	0.20%
iso	74	0.04%	rmvb	534	0.29%
divx	79	0.04%	exe	540	0.30%
djvu	112	0.06%	mp3	667	0.37%
doc	121	0.07%	wmv	835	0.46%
cab	125	0.07%	cbr	1474	0.81%
mkv	128	0.07%	mp4	2201	1.21%
dlc	131	0.07%	pdf	2220	1.22%
par2	135	0.07%	r(number)	3664	2.02%
flv	148	0.08%	zip	4237	2.34%
7z	149	0.08%	(number)	11025	6.08%
jdu	154	0.08%	avi	16596	9.16%
srt	222	0.12%	rar	133263	73.52%
mpg	324	0.18%	(other exts.)	1667	<1%

Table B.3: Most prevalent file extensions

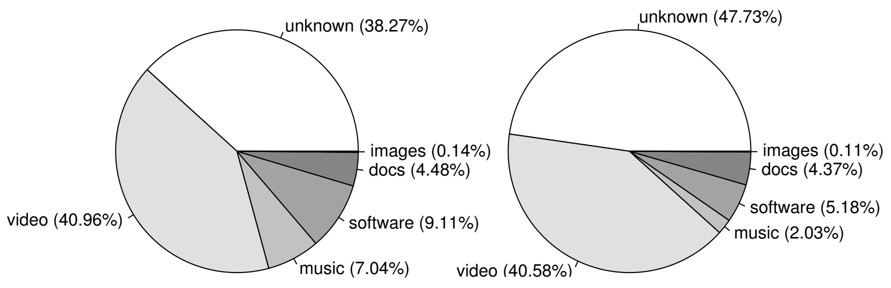


Figure B.8: Content by number of files (left) and by number of hits (right)

percentage of hits per file type, which provides a measure of the popularity of each category.

Our results differ from those of [26] since, in our case, we collect file names from the traffic generated by the users of the network and, therefore, our results reflect the popularity of the users of this network. Instead, ref. [26] gathers a list of files from content indexing sites, which is less prone to geographical bias, but omits part of the traffic of files interchanged directly by users (hence the large percentage of unknown files in our study). We find a greater amount of video and, most notably, a much smaller proportion of software compared to [26].

B.3.5.2 Fragmented Content

It is a very common practice for users to manually split a large file in several parts prior to sharing them. This is a practice that is not found in P2P file sharing, since it does not bring any benefits. However, in this environment it does provide several advantages.

First, it circumvents upload and download file size restrictions. Second, it allows unpaid users, who are usually not allowed to pause and resume downloads, to download the file in smaller pieces. Third, the content provider gathers more *reward points*. In order to gain market share, DD sites usually reward the uploaders of popular content; for example, as of this writing, one of the major sites rewards any user with $10,000 every 5 million downloads of the content he has uploaded.

This practice explains the popularity of RAR archives, which can contain any kind of content, conveniently partitioned in smaller fragments. This is an extremely common practice, to the point that 85.8% of the RAR files we have observed (and therefore almost two thirds of *all* the files) were of the form *filename.partX.rar*, where X denotes a number. For example, WinRAR [19] uses this exact file naming scheme when fragmenting an archive.

B.3.5.3 File Sizes

We obtained the file sizes of the downloaded files by extending our measurement software to parse additional fields of the HTTP request. We enabled the analysis of the downloaded files in our measurement software on May 28, so the results presented in this section correspond to a period of three weeks. Overall, our traffic analysis module has collected 66,330 unique file names and sizes, all of them hosted at Megaupload or RapidShare. Note that in [26], file sizes are not directly analyzed. Instead, their analysis is centered around flow sizes, which tend to be smaller.

Figure B.9 (top) plots the CDF of the observed file sizes. The most notable observation from this graph is that, according to our sample, around 60% of the files have a size of around 100MB. Figures B.9 (middle and bottom) show the same CDF only for Megaupload and RapidShare files. Interestingly, while 60% of the files hosted at RapidShare are also around 100MB, this service hosts a smaller number of larger files compared to Megaupload. This difference is a result of the different upload file size restriction policies of each site.

In the case of Megaupload, the limit is 500MB, while at RapidShare, the limit is currently set at 200MB. This raises the question of why the 100MB file size is the most prevalent. The ex-

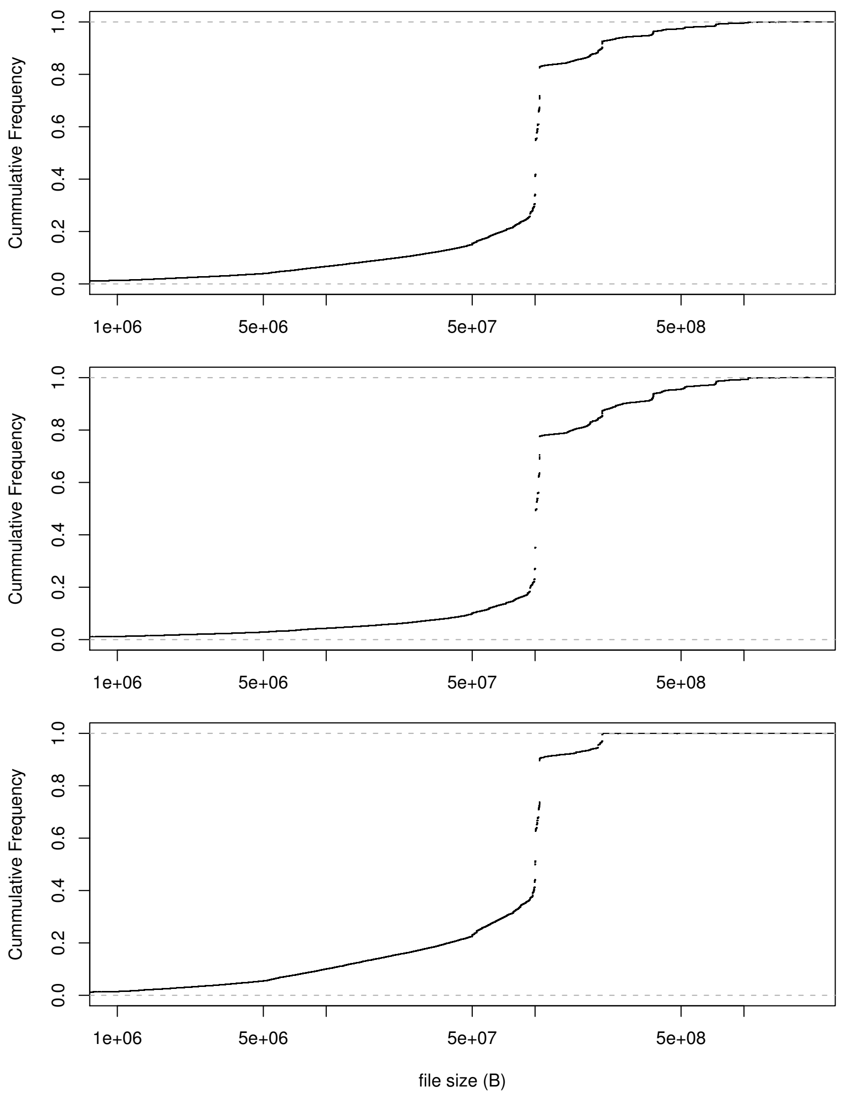


Figure B.9: Observed file sizes: overall (top), Megaupload (middle), RapidShare (bottom)

planation we have found for this fact is two-fold. First, until July 2008, the limit for RapidShare was 100MB. Often, content uploaders publish the content in more than one direct download service for unpaid users to be able to parallelize downloads, and in order to provide a backup plan in the case a file is removed (e.g., in the event of a copyright claim over the content). Therefore, files uploaded to Megaupload where also of the same size. Second, as explained, WinRAR [19] appears to be the software being used to partition files in multiple fragments. WinRAR permits partitioning in any file size, but offers several presets that match popular storage media: floppy disks, Zip100 disks, CD-ROM and DVD. In particular, the Zip100 preset is exactly 98,078KB, and that is the most prevalent file size across RAR files (12%), closely followed by exactly 100MB (around 9%) and 100 million bytes (around 6%).

It is apparent that, while the 100MB file limit was initially enforced by the sites, is has become the *de facto* standard file fragment size.

B.3.6 Server Infrastructure

As shown in previous sections, one-click file hosting domains serve extremely large data volumes compared to other web-based services. Therefore, it is interesting to analyze the server infrastructure that supports such services and compare it to that of other popular domains.

We collected a list of web servers that served HTTP traffic for each domain. For each HTTP connection, we also analyzed the Host header field of the requests [68]. We obtained a lower bound on the number of mirrors of each domain by calculating the number of unique IP addresses that served content for each particular domain. The distance of this lower bound to the actual number of mirrors that serve a particular domain will vary depending on several factors, such as the amount of client traffic (the more popular, the larger the part of its servers that will surface) or whether sites dedicate part of the mirrors to serve requests from particular geographical areas.

The results we have obtained are summarized in Figure B.10. Surprisingly, rapidshare.com is the Internet domain for which the largest number of mirrors has been observed (almost 9000), even above domains as popular as facebook.com (around 5500 servers) and google.com (around 4000). Next direct download services in number of observed mirrors are Megaupload and Mediafire, with around 800 each.

Alexa.com traffic ranks [2] indicate that, worldwide, RapidShare is the most popular file hosting service. This explains why their server infrastructure is larger compared to Megau-

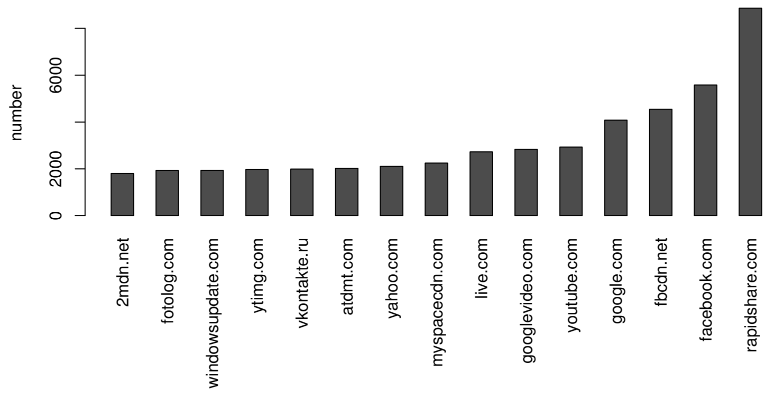


Figure B.10: Internet domains with the highest observed number of servers

pload's. However, according to Alexa, Megaupload and RapidShare rank in very close positions in the country where the analyzed network is located.

In Figure B.11 we present the cumulative number of servers as a time series (top) and as a function of the number of accesses (bottom). During weeks 13 and week 23, we observe the addition to Megaupload of new IP address blocks belonging to four different /24 subnets. In the case of RapidShare, we did not observe new IP address blocks during the analysis, but their infrastructure appears to be considerably larger. We find a larger number of IP addresses compared to [26], which suggests that RapidShare have kept upgrading their infrastructure.

Table B.4 exposes the DNS naming scheme of both RapidShare and Megaupload's server infrastructure. We used the IP address to Autonomous System (AS) mappings provided by Team Cymru [17] and the MaxMind GeoLite Country IP geolocation database [11] to approximate the geographical location of the mirrors. Both services have server infrastructure hosted in several ASs. While [11] reports IP addresses to belong to several different countries, ref. [26] features a more accurate geolocation study based on packet round-trip times from a number of Planetlab nodes that pinpoints RapidShare servers to a single location in Germany.

In both the cases of Megaupload and RapidShare, as can be observed in Table B.4, the content servers are easy to identify from their DNS name. This result suggests that, even though new servers are being deployed by both, traffic shaping of these services (to either reduce their bandwidth consumption or even block them) is relatively easy. In the case of P2P, traffic identification (and hence, policing) is more complex given its de-centralized architecture and the use of obfuscated protocols.

pattern	AS	AS name	#IPs	location (#IPs)
wwwqX.megaupload.com	29748	CARPATHIA	33	US (33)
wwwX.megaupload.com	38930	FIBERRING	39	NL (39)
	46742	CARPATHIA-LAX	133	US (133)
	35974	CARPATHIA-YYZ	135	US (135)
	16265	LEASEWEB	259	UNK (198) NL (61)
	29748	CARPATHIA	265	US (265)
rsXcg2.rapidshare.com	174	COGENT	182	DE (182)
rsXtl4.rapidshare.com	1299	TELIANET	193	EU (193)
rsXtl3.rapidshare.com	1299	TELIANET	383	DE (191) EU (192)
rsXl34.rapidshare.com	3356	LEVEL3	559	DE (182) GB (377)
rsXcg.rapidshare.com	174	COGENT	557	DE (557)
rsXgc2.rapidshare.com	3549	GBLX	557	US (557)
rsXtg2.rapidshare.com	6453	GLOBEINET	567	DE (191) UNK (184) EU (192)
rsXdt.rapidshare.com	3320	DTAG	618	DE (618)
rsXgc.rapidshare.com	3549	GBLX	748	US (748)
rsX.rapidshare.com	3356	LEVEL3	749	DE (557) GB (192)
rsXl3.rapidshare.com	3356	LEVEL3	749	DE (557) GB (192)
rsXl32.rapidshare.com	3356	LEVEL3	749	DE (373) GB (376)
rsXtg.rapidshare.com	6453	GLOBEINET	749	DE (375) GB (182) EU (192)
rsXl33.rapidshare.com	3356	LEVEL3	750	DE (374) GB (376)
rsXtl.rapidshare.com	1299	TELIANET	750	DE (558) EU (192)
rsXtl2.rapidshare.com	1299	TELIANET	752	DE (559) EU (193)

Table B.4: DNS naming patterns, associated Autonomous Systems, and approximate geographical location of Megaupload and RapidShare mirrors

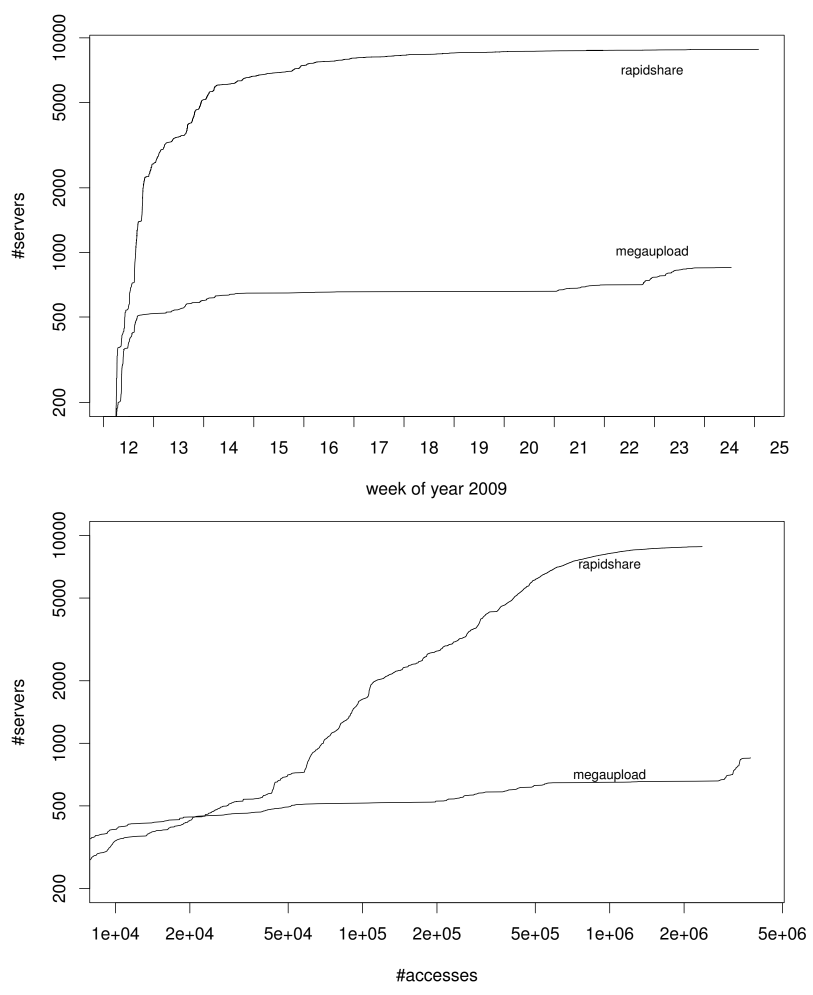


Figure B.11: Number of servers and accesses observed for Megaupload and RapidShare

DD server infrastructures appear to run in a few specific data centers. In contrast, P2P architectures are distributed around the globe. As a consequence, P2P tends to use fewer backbone Internet link bandwidth compared to DD, primarily as an effect of the locality of user interest on files (additionally, [85] finds that P2P could exploit locality to achieve greater bandwidth savings in transit links). Thus, if HTTP-based file sharing were to continue growing, traffic load in transit links should be expected to rise noticeably in the future.

To confirm this, we have also geolocalized the P2P traffic we have observed during the month of September 2009. The results of this experiment are summarized in Table B.5. Most notably, we find that around 46% of the P2P traffic was exchanged within the country of the network under study (ES), and more than 72% within the same continent. In contrast, all DD

country code	traffic	%	country code	traffic	%
ES	38.15 TB	46.02%	PL	778.28 GB	0.94%
(unknown)	7.20 TB	8.69%	NO	702.11 GB	0.85%
US	5.29 TB	6.38%	CN	679.21 GB	0.82%
FR	4.70 TB	5.67%	PT	638.57 GB	0.77%
IT	2.93 TB	3.54%	MX	552.43 GB	0.67%
GB	2.56 TB	3.10%	GR	534.08 GB	0.64%
SE	2.31 TB	2.79%	HU	520.60 GB	0.63%
DE	1.67 TB	2.02%	AR	513.09 GB	0.62%
NL	1.49 TB	1.80%	RO	497.28 GB	0.60%
CA	1.36 TB	1.65%	RU	445.78 GB	0.54%
JP	900.72 GB	1.09%	CL	426.71 GB	0.51%
BR	853.57 GB	1.03%	CH	399.04 GB	0.48%
AU	850.68 GB	1.03%	(others)	5.91 TB	7.12%

Table B.5: Exchanged P2P traffic per country

traffic comes from abroad.

B.4 Concluding Remarks

We analyzed the HTTP traffic of a large research and education network during three months (from March to June 2009). Our measurement-based study, which includes data from over 1.6 billion HTTP connections, reveals that the increase of HTTP traffic on the Internet reported in recent studies can be mostly attributed to two single Internet domains: RapidShare and Megaupload. The popularity of these sites is surprising considering that, for file sharing purposes, P2P based architectures are considered technologically superior. We performed an exhaustive packet-level analysis of the traffic of these two popular one-click file hosting services (a.k.a. direct download (DD) services) at four different levels: traffic properties, user behavior, content distribution and server infrastructure.

We can summarize the main results of this study in the following findings: (1) DD services generate a large portion of HTTP traffic, (2) DD services are among the Internet domains that generate the largest amount of HTTP traffic, (3) a significant fraction of file sharing is in the form of HTTP traffic, (4) DD services rely on a huge server infrastructure even larger than other extremely popular Internet domains, (5) a non-negligible percentage of DD users paid for premium accounts, (6) premium users proportionally generate a larger amount of DD traffic than unregistered users, (7) restriction polices applied to non-premium users differ significantly between different DD sites, (8) DD users can highly benefit from TCP congestion control mechanisms by parallelizing their downloads using specialized software download managers, (9) the prototypical download from a DD site is a 100MB RAR archive fragment.

The aim of this study was to investigate the characteristics of a service that is responsible for a large percentage of the Internet traffic volume. We also discussed practical implications for network management, and found that this kind of traffic is easy to identify and hence police, especially relative to P2P. Additionally, we found that direct download services utilize more transit link bandwidth compared to P2P, which is able to leverage locality.

Appendix C

Publications

Publications directly related to this Thesis (journals):

- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Josep Solé-Pareta: “Measurement Based Analysis of One-Click File Hosting Services”, Journal of Network and Systems Management, 2011.

Publications directly related to this Thesis (conferences):

- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Josep Solé-Pareta, Gabriella Andriuzzi: “A Lightweight Algorithm for Traffic Filtering over Sliding Windows”, IEEE International Conference on Communications (ICC) 2012.
- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Nick Duffield, Ramana Kompella: “Cuckoo Sampling: Robust Collection of Flow Aggregates under a Fixed Memory Budget”, IEEE INFOCOM Mini-Conference 2012.
- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Nick Duffield, Ramana Kompella: “Sketching the Delay: Tracking Temporally Uncorrelated Flow-Level Latencies”, ACM/USENIX Internet Measurement Conference (IMC) 2011.
- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Josep Solé-Pareta: “Validation and Improvement of the Lossy Difference Aggregator to Measure Packet Delays”, Traffic Monitoring and Analysis (TMA) Workshop 2010.
- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Josep Solé-Pareta: “Counting Flows over Sliding Windows in High Speed Networks”, IFIP Networking 2009.

Publications directly related to this Thesis (to be submitted):

- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Ramana Kompella, Josep Solé-Pareta: “Validation and Improvement of the Lossy Difference Aggregator to Measure Packet Delays”, to be submitted to a journal (TBD).
- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Nick Duffield, Ramana Kompella: “Cuckoo Sampling: Robust Collection of Flow Aggregates under a Fixed Memory Budget”, to be submitted to a journal (TBD).

Other related publications:

- Jakub Mikians, Pere Barlet-Ros, Josep Sanjuàs-Cuxart, and Josep Solé-Pareta: ”A practical approach to portscan detection in very high-speed links”. In Passive and Active Measurement Conf. (PAM) 2011.
- Pere Barlet-Ros, Gianluca Iannaccone, Josep Sanjuàs-Cuxart, Josep Solé-Pareta: Predictive Resource Management of Multiple Monitoring Applications, IEEE/ACM Transactions on Networking, 2010.
- Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Gianluca Iannaccone, Josep Solé-Pareta: ”Distributed Scheduling in Large Scale Monitoring Infrastructures”, CoNext Student Workshop 2008.
- Pere Barlet-Ros, Gianluca Iannaccone, Josep Sanjuàs-Cuxart, Josep Solé-Pareta: ”Robust Network Monitoring in the presence of Non-Cooperative Traffic Queries”. Computer Networks, 2008.
- Pere Barlet-Ros, Josep Sanjuàs-Cuxart, Josep Solé-Pareta, Gianluca Iannaccone: ”Robust Resource Allocation for Online Network Monitoring”, 4th International Workshop on QoS in Multiservice IP Networks (QoSIP) 2008.
- Pere Barlet-Ros, Gianluca Iannaccone, Josep Sanjuàs-Cuxart, Diego Amores-Lpez, Josep Solé-Pareta: ”Load Shedding in Network Monitoring Applications”, USENIX Annual Technical Conference 2007.
- Pere Barlet-Ros, Diego Amores-Lpez, Gianluca Iannaccone, Josep Sanjuàs-Cuxart, Josep Solé-Pareta: ”On-line Predictive Load Shedding for Network Monitoring”, IFIP Networking 2007.