Acknowledgements

First of all, I would like to thank my family, my gilfriend Maria and my friends for all of their love and support during these years. I would also like to thank all of the people in my lab for their contribution to my professional growth and for the very good time we spent together. A special acknowledgment to the guys of the NP team: Fabio, Gianni and Domenico. It has been great to work together. Thanks to my tutors prof. Franco Russo, prof. Stefano Giordano and Dr. Gregorio Procissi for their help and support. A special thought for Gregorio, for all of the good time we spent together while travelling abroad and working in the lab. A special acknowledgement also to Nicola: working with him was a great opportunity, as well as a pleasure. I would also like to thank all of people I worked with at the NEC Laboratories of Heidelberg: Felipe, Saverio and many others. They played a big role in my professional and human growth. Finally, I want to thank all of the colleagues I worked with (and had fun with) during the European projects I participated to. It has been a wonderful experience that I am sure I will miss.

Sommario

Con la tendenza di Internet a diventare sempre di più un'infrastruttura critica e con l'aumento continuo del volume di traffico che essa è chiamata a trasportare (accompagnato da una crescita parallela delle minacce alla sicurezza informatica), l'attività di monitoraggio sta assumendo sempre di più un ruolo cruciale per garantire il funzionamento della rete e dei servizi che su di essa si basano. D'altra parte, la quantità di dati da analizzare e l'estrema varietà delle analisi da svolgere, unite alla necessità di correlare informazioni derivate da sorgenti differenti e alle limitazioni imposte dalla legislazione per la protezione della privacy, rendono il monitoraggio un'attività estremamente difficoltosa dal punto di vista tecnico. In questa tesi esploreremo diversi filoni di ricerca, accomunati dalla loro importanza per le attività di network monitoring. Per prima cosa, presenteremo tecniche di network tomography, che permettono di ottenere una stima dello stato interno della rete applicando tecniche di inferenza statistica a partire da osservazioni ricavate dai nodi periferici, senza alcuna cooperazione da parte degli apparati interni. Successivamente, presenteremo algoritmi e strutture dati originali per velocizzare alcune funzioni di packet processing particolarmente impegnative dal punto di vista computazionale, come l'ispezione di tutto il contenuto informativo del pacchetto. Ci sposteremo poi su tematiche più architetturali e mostreremo come processori general purpose e dispositivi dedicati possano essere usati in maniera complementari per progettare sistemi di monitoring e testing che offrono un compromesso ottimale tra flessibilità d'uso e prestazioni. Inoltre, mostreremo come il potenziale dell'hardware moderno (che è caratterizzato da un alto grado di parallelismo) possa essere sfruttato per migliorare le prestazioni del software di monitoraggio della rete. Infine, affronteremo alcuni argomenti relativi ai sistemi distribuiti di monitoraggio e proporremo delle soluzioni originali per costruire degli overlay di sonde che possano correlare i dati osservati in maniera efficiente, evitando così il collo di bottiglia in cui si incorre nelle architetture caratterizzate da un singolo collettore.

Abstract

As internet is becoming a critical infrastructure and the amount of traffic carried on it is rapidly growing, along with the potential security threats, monitoring is becoming more and more a crucial activity to the correct operations of networks and network based services. However, the amount of data to be analyzed, the extreme variety of the analysis to be supported, along with the need to correlate data from different sources and the limitations imposed by the privacy legislation make network monitoring a difficult and challenging task. In this work we explore several research fields, all of them related to network monitoring and testing. First of all, we propose tomographic techniques, that allow to infer the internal state of the network by applying statistical analysis to measurements carried out by the end–hosts, with no cooperation from the internal nodes. We then illustrate novel algorithms and data structures for speeding up expensive packet processing tasks, such as deep packet inspection. Subsequently, we move on to architectural topics and show how general purpose processors and special purpose devices can complement each other in order to build monitoring and testing systems offering an optimal trade–off between flexibility and performance. Moreover, we also investigate on the potential that the modern commodity hardware (which is highly parallel) provides and on how this can be leveraged for the benefit of the network monitoring applications. Finally, we delve into the topic of distributed monitoring and propose novel solutions for building an overlay of monitoring probes which can efficiently correlate the observed data, thus avoiding the scalability bottleneck of an architecture based on a single collection point.

Contents

List of Figures

List of Tables

Introduction

In the last years internet is evolving more and more towards a critical infrastructure, whose availability is crucial to a number of economic activities. At the same time, the amount of data moving through the network is rapidly increasing and so are the capacities of the internal links. Legitimate traffic, however, is not the only evolving player in the last years: cyberthreats and malicious activities are also exploiting the pervasive and open nature of the network. Armies of infected hosts controlled by technically skilled players can be used in order to throw large-scale attacks against which can cripple important activities and cause huge money loss. Moreover, as more and more heterogeneous devices are being connected to the internet, new kind of attacks and exploits are surfacing: skilled hackers are able to re–route and high-jack phone calls performed by VoIP–enabled devices, while more and more attacks are targeting mobile terminals connected to the network through mobile broadband access networks. In this evolving scenario, the ability of constantly monitoring the traffic in order to promptly detect and face security threats and malfunctions in crucial to the correct operation of networks and network–based services. This purpose clashes with a number of both technological and operational issues: on one hand, the growing amount of data makes traffic analysis challenging while, on the other, the amount of available information is often limited. In particular, administrative limitations can restrict the position and the scope of the links to be monitored, while legislation imposes limitations on the information that can be collected and exported for monitoring purposes. In this work, we contribute to this crucial technological challenge by exploring several topics in the vast field of network monitoring, ranging from inference algorithms to software architectures for packet processing.

The first topic to be addressed is network tomography: such generic definition covers in fact a large variety of techniques which try to obtain information about the state of the network (be it either its topology or the state of congestion of its internal links) by applying statistical inference to measurements which are performed by the end nodes. The main advantage of such an approach is that it does not require cooperation from internal nodes, thus being able to cross the administrative borders of the networks; furthermore it does not require any high performance probes to be deployed at the core links. In this field, our main contribution is a set of techniques which try to infer the network internal topology from edge measurements, by leveraging the a–priori knowledge of the possible link capacities (since the link speeds usually belong to a well known standard set, this is a well–grounded assumption). In addition, novel solutions to infer the link congestion state over an end–to–end path are proposed.

The second topic that we address in this work is the development of novel algorithms and data structures for efficient traffic analysis. Due to the very strict timing requirements which characterize packet by packet computations, proper solutions are crucial to the performance of network monitoring devices. Indeed, the key factor for obtaining good results turns out to be the efficient exploitation of the memory hierarchy that characterizes most of the network processing devices, be them special–purpose embedded systems or general purpose processors: layered data structures which allow to split information on different levels (with different access frequency) can leverage at best such architectures, bringing to huge performance gains. In particular, our main contributions concern deep packet inspection algorithms and Bloom filters (a compressed data–set representation which is used in a number of networking applications). As for the former, we design a novel compression technique for deterministic finite automata which achieves an excellent trade–off between memory footprint and speed, while, in the latter case, we propose more efficient versions of the well-known data structure based on a layered implementation.

Subsequently, we shift from algorithmic research to architectural issues. In particular, we investigate how special purpose architectures for network processing can be effectively leveraged in order to compose hybrid systems, where the flexibility of software can be complemented by the performance of dedicated devices. In our work we use both network processors and the recently released Netfpga platform. We propose two extensible and modular architectures built around a network processor and its host PC: the first one allows high performance and precise traffic generation while the second one uses a network processor to act as a packet demultiplexer towards a set of software–based sensors. The latter, in particular, is then used in the framework of a more involved ”smart probe” architecture whose purpose is to perform complex application–aware traffic analysis already at the capturing stage. In addition, we also propose a ”smart–switch” architecture, based on a Netfpga platform, which adopts and extends the well–known Openflow switching paradigm.

We then focus on packet processing and network monitoring architectures wholly based on commodity hardware. The leading principle of our research in this field is to investigate how the potential of modern platforms (whose distinctive feature is an ever increasing degree if parallelism) can be leveraged by networking software. Our contributions encompass a novel packet capturing engine (named PFQ) and a synthetic traffic generator; both of them achieve higher performance than their competitors and good scalability as the number of available cores increases. On top of our packet–capturing engine we also develop a full–fledged middleware for supporting modular monitoring applications, which can be easily composed as a set of interconnected and reusable blocks.

Finally, we delve into the topic of distributed monitoring. With the emergence of botnets and distributed cyberhtreats, this research field has become of paramount importance information security. Indeed, the traditional approach to such problem (a set of distributed probes exporting data to a central collector) does not offer the required scalability and flexibility when it comes to correlating the huge amount of information stemming from the traffic monitoring activities; moreover, data export is strictly limited by privacy–related regulations. In order to address such issue, we propose a distributed protocol for inter–protocol correlation and a scalable infrastructure for coordinating an overlay of probes, so as to get rid of duplicated measurements and to optimize resource utilization and illustrate the functioning principle of a duplicate–avoiding aggregation scheme.

Chapter 1

Network tomography and active measurements

The term network tomography is in fact used as a large umbrella covering a number of techniques whose common goal is to infer information about the internal state of the network with no support and cooperation from the internal nodes. These techniques usually entail collecting a large set of samples from active measurements (be them delay values or inter-arrival values) and applying statistical inference techniques. The advantages of such an approach are many–fold:

- it is possible to gather information about a whole network path, whatever the administrative domains the path crosses.
- there is no need to deploy special–purpose probes to monitor the traffic.
- the performance requirements are much looser with respect to passive network monitoring.

Tomographic techniques have been proposed to infer several details about the internals of the network: network topology, packet loss rate, delay distribution, capacity of the bottleneck link on a path or available bandwidth. Overall, the techniques that we propose in this chapter target most of such variables. In particular, in section 1.1 we address the topology discovery problem and we enhance the so–called packet sandwich technique by proposing a selective filtering algorithm that allows to discard the samples whose information content has been corrupted by the interaction between probing traffic and regular traffic. In section 1.2 we leverage a similar technique in order to build a reliable tool that estimates the capacity of the bottleneck link between two network hosts. In section 1.3 we introduce a novel and different approach for the problem of topology discovery. In particular, we leverage the additional information coming from the a–priori knowledge of the possible link capacity values in order to transform the inference problem into a decision problem with a well defined set of possible hypotheses. In section 1.4 we develop further such a novel approach by observing that the requirement to keep different decision consistent among themselves can be used in order to reduce the hypotheses space, thus improving the reliability of the subsequent decisions. In section 1.5 we introduce another improvement of the proposed technique, by specifying an algorithm that allows to merge the topologies inferred by multiple vantage points in order to obtain a global view of the network. Subsequently we shift from the topology discovery problem to the inference of delay statistics and loss rates on the links of an end–to–end network path. In particular, in section 1.6 we show how most of the traditional tomographic inference techniques, which had been designed to be used in a point–to–multipoint layout, can be adapted to work on a single network path. Finally, in section 1.7 we face again the problem of inferring the delay on the links of a network path by adopting a different perspective and working in the frequency domain.

1.1 Noise Reduction Techniques for Network Topology Discovery

The concept of network topology discovery groups all the techniques that allow, by any means, to obtain the knowledge of the internal topology of a communication network. This kind of knowledge can be useful in several fields, such as troubleshooting, SLA verification, topology-aware (multicast) applications, network management, routing decisions and so forth. The most widespread techniques for topology discovery are those based on the *Traceroute* tool, on OSPF messages listening and on SNMP queries and need a certain degree of cooperation from the interior routers which is not always available. This limitation can be overcome through network tomography techniques that extract the information on the internal state of a network from end-to-end measurements, with no need for cooperation of internal routers and, therefore, can be applied in nearly every scenario. Due to the interference of regular traffic with probe packets, the measurements obtained by active probing are generally affected by noise that can lead to a wrong topology reconstruction. In this section, we present a novel approach to analyze typical noise patterns based on the extension of the model of [1, 2] originally proposed for packet trains. Moreover, we address the issue of noise reduction by developing two original model-free noise reduction algorithms whose performance is evaluated through ns2 simulations.

1.1.1 Topology Reconstruction by Hierarchical Clustering

In recent years, several tomographic techniques for network topology discovery have been proposed, though, in most cases, they are based on a common theoretical framework which has been formalized in [3]. Generally speaking, a host of the network sends probing packets to a large set of hosts located across the network to perform some kind of end-to-end measurement. By off-line processing of such data, a tree topology which represents the logical structure of the network is built. The root of the tree represents the sender of the probe packets, the leaves are the receivers, while the internal nodes of the tree are the so called *branching points*, as

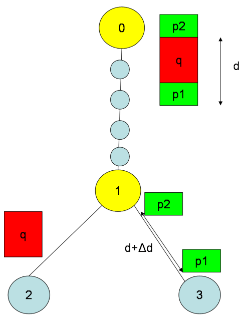


Figure 1.1: Working principle of a packet sandwich probe

they represent the locations where the paths from the sender to the receivers split. The inference of the tree can be accomplished by processing end-to-end measures through specific algorithms, like those proposed in [4, 5]. In addition, the solution proposed in [6] can be used to merge trees with different roots in order to achieve a more complete view of the network topology.

Most of the proposed algorithms are based on the so called *similarity metrics*, that is the end-to-end measurable variables that indicate the length of the shared path between the sender and a pair of receivers (i.e. the path between sender and their associated branching point). They therefore reveal the proximity of each pair of leaves in the sender tree.

1.1.2 The Packet Sandwich Similarity Metric

Despite several variables (end-to-end delay covariance, shared packet loss..) have been proposed to be used as similarity metrics, this section will focus on the metric related to packet sandwich probes.Such probes, thoroughly described in [4], are a particular kind of packet trains whose packets are directed to different receivers. A packet sandwich consists of three packets (in the following referred to as p_1, q and p_2) which are sent back to back by the probing host. Packets p_1 and p_2 are sent to the same receiver R_1 and have the same length L_1, while q is sent to receiver R_2 and has length L_2, that is usually much bigger than L_1. In particular, L_1 is typically 56 bytes, while the value of L_2 generally equals the value of the MTU of the network under test.

The basic idea underlying packet sandwich is quite simple and is explained in figure 1.1: at every hop of the shared path between the sender and the two receivers, the third packet gets queued behind the second much bigger packet and the delay difference between p_1 and p_2 increases. The interarrival time $d_{(R_1,R_2)}$ of packets p_1 and p_2 is an end-to-end measurable metric which increases with the extent of the path shared by receivers R_1 and R_2 in that it increases at every hop of that path. It is possible to derive a simple expression for the interarrival time of the packets p_1 and p_2 as a function of the capacity of the N_{shared} links composing the path shared by receivers R_1 and R_2. Although the derivation of such a formula is quite simple, we have not found it so far in the technical literature concerning packet sandwich probes.

Let us first consider the delay increment that packet p_2 experiences on a single link C_i of the shared path, assuming no cross traffic. On link C_i, packet p_1 will only experience the transmission delay, while packet p_2 will also experience a given amount of queuing delay in that it must wait for the transmission of packet q; the interarrival time will then be exactly increased by such a queuing delay. Let us now consider the time instant when q starts its transmission on link C_i: at the same time, packet p_2 will start its transmission on link C_{i-1}. Whenever p_2 reaches the queue of link C_i, namely after L_1/C_{i-1} seconds, it will be queued waiting for packet q to finish its transmission: it will therefore wait in the queue for a time:

$$\Delta_{d_i} = \frac{L_2}{C_i} - \frac{L_1}{C_{i-1}} \tag{1.1}$$

Summing up all the delay increments together with the packets intedeparture time we obtain:

$$d_{(R_1,R_2)} = \frac{L_2}{C_1} + \sum_{i=2}^{N-1} \frac{L_2 - L_1}{C_i} + \frac{L_2}{C_N} \tag{1.2}$$

1.1.3 Noise analysis in packet sandwich measurements

In the literature, the interarrival time of packets p_1 and p_2 of the packet sandwich is typically modeled as a constant value (the similarity metric itself) afflicted by an additive zero mean noise. In particular, in [4] the authors invoke the Central Limit Theorem to model the sample mean of a sequence of independent packet sandwich probe measurements as a Gaussian random variable, although they do not propose a statistical model for the interarrival time itself.

1.1.3.1 Simulation layout for the characterization of the interarrival time

The effect of the interaction between sandwich probe packets and cross traffic is evaluated through simulations by using the ns2 simulator [7].

Figure 1.2 shows a typical simulation scenario: in order to evaluate the impact of the length of the shared path on the interarrival time distribution, the number of shared links has been varied throughout the simulations. Each of the shared links is loaded with multiple TCP flows; to reproduce the multi-modal distribution of Internet packet length, TCP segments have different sizes. In order to accurately tune the load of cross traffic interacting with packet probes on each link, TCP connections are established as shown in figure 1.2. For example, the cross traffic loading the link which connects nodes 5 and 6 is generated by TCP connections established between nodes tcp_5 and the downstream node 6. The link that connects the out–of–path node tcp_5 and the node 5 is the bottleneck link of the TCP connections; by simply varying its capacity, it is possible to accurately tune the cross traffic load on each link of the shared path. The physical speed of the links of the path has been chosen according to widespread standards in use, such as Ethernet links or Optical Carrier.

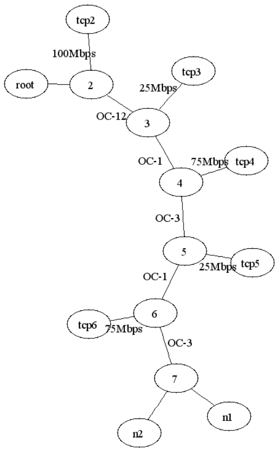


Figure 1.2: Simulation scenario for the characterization of packet sandwich interarrival times.

1.1.3.2 Models of Noise Patterns

The idea of modelling the measurements noise due to regular traffic by means of simple random variables turns out to be a very complex task, due to the extreme variability of statistical patterns that emerge from simulations.

In order to derive effective models of noise patterns, we extend the approach of [1, 2] that was originally proposed in the framework of bottleneck capacity estimation to model the empirical distribution of interarrival times of packet train probes. Therefore, let us consider a packet pair and let δ be the delay difference experienced by the two consecutive packets. Two possible cases may occur and are described in the following.

- The two consecutive packets, while crossing the network, are never queued together at the same node: under this hypothesis, if we denote with l_1 and l_2 the packet lengths of the two packets, and with W_i^k the queuing delay experienced by the k-th packet on link i, the expression of the delay difference becomes

$$\delta = \sum_{i=1}^{Nlink} \left(\frac{l_2 - l_1}{C_i} + W_i^2 - W_i^1 \right) \tag{1.3}$$

- The two consecutive packets are queued together for the last time at the h-th link of the network. In this case the expression of the delay difference is:

$$\delta = -t_h + x_h + \frac{l_1}{C_h} + \sum_{i=h+1}^{Nlink} \left(\frac{l_2 - l_1}{C_i} + W_i^2 - W_i^1 \right) \tag{1.4}$$

where the term x_h indicates the transmission time of the cross traffic packets queued between the two probe packets, while t_h is the interarrival time of the packets.

Notice that in the second case the information about the capacity of links which are upstream to the h-th is overwritten.

Equations (1.3) and (1.4) can be successfully extended to packet sandwiches: the interarrival time of packets p_1 and p_2, namely their interarrival time at the ending node of the shared path, can be expressed as a sum of two terms: the first term is the interarrival time of packets p_1 and q, while the second term is the interarrival time of packets q and p_2.

Under the hypothesis of uncongested network, packets q and p_1 will normally never queue at the same internal node and their path delay difference can therefore be expressed by (1.3). Thus, the interarrival time of packets q and p_1 is the sum of their interdeparture time and their delay difference:

$$\begin{aligned} t^{interarr}_{p_1,q} &= t^{interdep}_{p_1,q} + \delta_{p_1,q} \\ &= \frac{L_2}{C_1} + \sum_{i=2}^{N} \left(\frac{L_2 - L_1}{C_i} + W_i^q - W_i^{p_1} \right) \end{aligned} \tag{1.5}$$

Moreover, since packets q and p_2 always travel back to back, their interarrival time is simply given by the transmission time of packet p_2 on the last shared link.

Therefore, by summing up the two contributions of the interarrival time, we obtain the following expression for the packet sandwich metric:

$$\begin{aligned} d &= t^{interarr.}_{p_1,q} + t^{interarr.}_{q,p_2} \\ &= \frac{L_2}{C_1} + \sum_{i=2}^{N} \left(\frac{L_2 - L_1}{C_i} + W_i^q - W_i^{p_1} \right) + \frac{L_1}{C_N} \\ &= \frac{L_2}{C_1} + \sum_{i=2}^{N-1} \left(\frac{L_2 - L_1}{C_i} + W_i^q - W_i^{p_1} \right) + \frac{L_2}{C_N} \end{aligned} \tag{1.6}$$

which, with no cross traffic (W_i equal to zero), corresponds to the interarrival time equation (1.2).

In the case of heavily loaded networks, the packet q could be queued behind p_1 and the term δ assumes the form (1.4); in this case, the measured sample could significantly differ from its theoretical value.

1.1.3.3 Analysis of collected sample patterns

The model described in the previous subsection can be used to explain the reason of some of the delay distribution patterns exhibited by simulation results. In [1] and [2], the authors define several types of cross traffic related noise patterns that can affect packet train probes; some of those patterns can be found in the analysis of packet sandwich samples distributions.

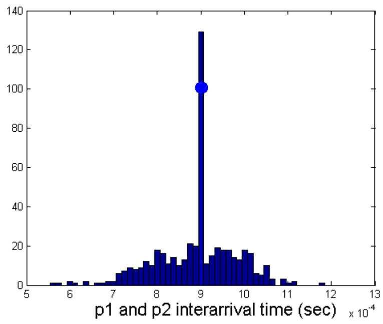


Figure 1.3: Interarrival time histogram with independence signature.

A measurement pattern which is often identifiable in simulation data is the so called *independence signature*, which is typical of lightly loaded networks: in this case, packets p_1 and q are never queued at the same node and their interarrival time is actually given by (1.6). The only random element in this expression is the queuing delay difference, that may be considered as a continuous zero mean random variable; there is a significant probability that a probe experiences no queuing delay at all along the whole path. For these reasons, the interarrival time distribution shows a mode corresponding to the theoretical value of the metric and a continuously distributed noise. A typical distribution representing the case of *independence signature* is given in figure 1.3 where the filled dot corresponds to the actual value of the measured metric.

Another pattern that can be detected in the interarrival time distribution is the so called *quantification signature*, which is typical of highly loaded networks where cross traffic packet lengths assume values in a small set. In such scenarios, packets q and p_1 are likely queued together and the end–to–end observed delay difference depends on the transmission time of cross traffic packets queued between them (represented by the x_h term of (1.4)). In presence of highly quantized packet lengths the values of x_h have a multimodal distribution which, in turn, causes the appearance of well defined secondary modes, as shown in figure 1.4. Again, the filled dot corresponds to the actual value of the measured metric; the empty dot is obtained by adding to the actual metric the transmission time of a cross traffic packet on one of the congested links. The secondary mode corresponding to this value is evident in the distribution.

Finally, in a network where congestion takes place on a particular link, the samples of interarrival time may show the so called *rate signature*: probe packets arriving at a certain node get likely queued together and the delay difference information they carried will be likely overwritten. The delay difference increases again when crossing downstream less congested links; however a secondary mode corresponding to an interarrival time value smaller than the theoretical one will be found in the samples distribution.

Unfortunately, the three scenarios above depicted rarely occur alone: in many real cases, the interpretation of results is not straightforward because of the combina-

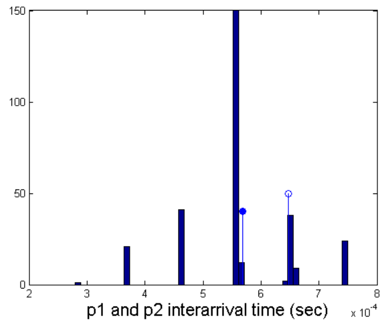


Figure 1.4: Interarrival time histogram with quantification signature.

tion of the effects of different signature phenomena. This, as stated in [1, 2], prevent from deriving tools for automatic pattern signature recognition. The large variety of noise patterns which emerges from simulation results makes the goal of deriving a single statistical model capable of capturing the whole presented phenomena far from being achieved.

1.1.4 A noise reduction algorithm for packet sandwich probes

The whole discussion presented in the previous subsection led us to address the problem of noise reduction by using an approach which does not rely on any a priori statistical model, but that only assumes the knowledge of packet sandwich interdeparture times. Out of all the available measurements, the algorithm that will be next illustrated selects the ones that are minimally affected by the interaction with cross traffic packets. Since cross traffic interaction results in increased queuing times, the smaller the queuing delay experienced by probe packets, the better the measurements. Let us therefore assume that N_{probe} packet sandwiches directed to the same pair of receivers are sent with a D time spacing (actually, our algorithm still works with any pattern of interdeparture times, provided that it is deterministically known). Let us focus on the n–th pair of small packets p_1^n and p_2^n, whose interarrival time is the measured variable: the end-to-end delay experienced by the k-th small packet ($k = 1, 2$) of the n-th sandwich is given by the sum of a deterministic time R^k (the one experienced by packets that do not interact with cross traffic) and a random, non-negative queuing delay Q_n^k, induced by the interaction of cross traffic . The packet arrival time a_n^k is therefore:

$$a_n^k = d_n^k + R^k + Q_n^k \tag{1.7}$$

where d_n^k denotes the deterministic departure time.

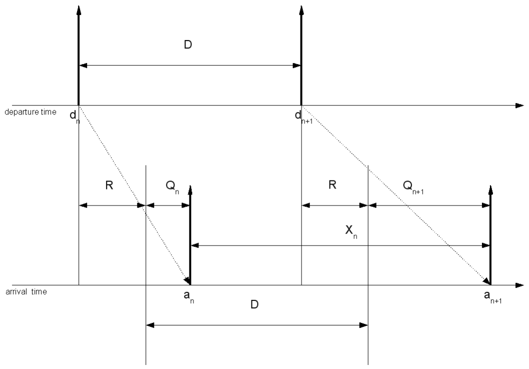


Figure 1.5: Time intervals involved in the derivation of equation (1.8).

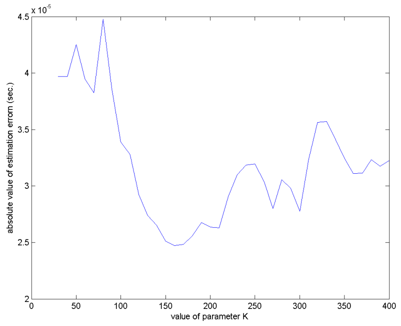


Figure 1.6: Absolute estimation error vs. K

The interarrival time X_n^k of packets p_k^n and p_k^{n+1} can therefore be expressed as:

$$X_n^k = a_{n+1}^k - a_n^k = Q_{n+1}^k - Q_n^k + (d_{n+1}^k - d_n^k) = Q_{n+1}^k - Q_n^k + D \tag{1.8}$$

The relationships among the different time intervals involved in the derivation of the previous formula are represented in figure 1.5. Equation (1.8) can be seen as a recursion between the queuing delay of sandwich packets belonging to consecutive probes:

$$Q_{n+1}^k = Q_n^k + X_n^k - D \tag{1.9}$$

The initial condition Q_0^k is unfortunately unknown; nevertheless, by choosing any arbitrary initial condition, queuing delay estimates calculated by (1.8) are biased by the same, constant, initial error ϵ_k. By indicating with $\widehat{Q_n^k}$ the series of queuing delay estimates, the following relation holds:

$$\widehat{Q_n^k} = Q_n^k + \epsilon_k \tag{1.10}$$

Therefore, it is possible to reorder the set of sandwich probe packets in terms of their estimated queuing delay $\widehat{Q_n^k}$ in that, as previously mentioned, the constant estimation bias does not change the sequence ordering.

Equation (1.9) holds for both packets p_1 and p_2 of the sandwich: the sample selection is therefore based on the two estimates sequences $\widehat{Q_n^1}$ and $\widehat{Q_n^2}$. The noise reduction algorithm evolves according to the following steps:

Algorithm 1

```
• Calculate the sequences $\widehat{Q_n^1}$ and $\widehat{Q_n^2}$ of queuing delays.
• Sort the two sequences in ascending order.
• Create the sets $G_1$ and $G_2$ that include the first $K$ samples of $\widehat{Q_n^1}$ and $\widehat{Q_n^2}$, respectively.
• Let $d_i$ be the interarrival time of the $i$-th packet sandwich:
    – If $\widehat{Q_i^1} \in G_1$ and $\widehat{Q_i^2} \in G_2$ then accept the sample $d_i$;
    – otherwise discard the sample.
• Calculate an estimation of the packet sandwich metric by averaging the accepted samples.
```

The behavior of the algorithm 1 depends on the value of the parameter K. If K is too large, the algorithm will probably accept samples significantly affected by noise. If K is too small, the algorithm may not converge: indeed, the selection of the best samples is based on the sequences $\widehat{Q_n^1}$ and $\widehat{Q_n^2}$ and the sets G_1 and G_2, if too small, may exhibit empty intersubsection.

We have not found a theoretically optimal choice for the K parameter; however, the heuristic choice $K = 0.3 \times N_{probe}$ has proved to be effective and generally leads to satisfactory simulation results. Figure 1.6 shows the estimation error as a function of the parameter K in a simulated shared path composed by 7 links; the value returned by our empirical formula (about 200 samples over the total number of 600) actually approaches the absolute minimum error.

Table 1.1: Metric estimation error (μsec). Metric value: $1ms$ (10 links), $0.54ms$ (5 links)

% load	M. (10 L.)	Alg. (10 L.)	M. (5 L.)	Alg. (5 L.)
20	22	2	0.23	0.12
40	40	6	0.56	0.18
50	30	13	0.57	0.42
75	15	5.6	5.5	2
90-100%	69	22	47	18
20 to 65%	61.5	18	8.7	6.9
40 to 80%	28	10	10	6.12

1.1.5 Performance Evaluation

To evaluate the effectiveness of the noise reduction algorithm presented in the previous subsection we performed a set of simulations by using ns2. The impact of our algorithm on the estimation error reduction has been evaluated in simulation scenarios analogous to those presented in subsection 1.1.3.1. The performance achieved by the noise reduction algorithm is compared to that achieved by averaging all available interarrival time measurements. ns2 simulations are carried out under different network load conditions and shared path lengths; results are reported in table 1.1.

1.2 PingPair: a Lightweight Tool for Measurement Noise Free Path Capacity Estimation

The knowledge of the capacity of a network path can be useful for many purposes: multimedia applications, service level agreement verification, network monitoring and management. Several solutions have been proposed for this purpose and many tools are available. Out of the many proposals, the most effective techniques are those which infer the path capacity from the packet pair dispersion, i.e. the interarrival time of two packets of the same length sent back to back by the source. The concept of packet pair dispersion was first introduced in [8] but its application for bottleneck link speed estimation was first proposed in [9]. Later on, the idea of measuring packet dispersion by averaging the interarrival times of a train of N packets was proposed, as shown, for example, in [10]. Unfortunately, this kind of measurements can be seriously distorted by the queueing delay due to cross traffic: its effect on the distribution of the packet pair dispersion measurement has been studied in several papers [1, 2, 11]. It is generally established that, due to the influence of interfering packets, some evident modes can emerge in the sample distribution, corresponding to dispersion values which can be significantly different than the actual one. The occurrence of such values is determined by many causes, including the length of the probe packets as well as that of the cross traffic packets. In order to discard the distorted measurements, several techniques have been proposed and many tools have been implemented: among the most well-known are Pathrate [11], Capprobe [12], Nettimer [13]. However, the queueing delay is not the only cause of capacity measurement errors: the interarrival time measurements obtained by a user level application are affected by a measurement noise that is mainly due to the variable latencies related to the transfer of a packet between the wire and the application socket. In this section we assume a Gaussian model for such noise and verify this hypothesis through experimental analysis; also, we develop a novel selective filtering algorithm in order to deal with this kind of disturb as well. Unlike most of the available tools, PingPair is based on one-point measurements: the packet pair sent by the probing host is composed by two *ICMP echo request* packets and the dispersion of the couple of the corresponding *ICMP echo reply* packets is measured by the same host. For this reason, our tool can be deployed in almost all network scenarios, since no special cooperation from the destination host is required (many other tools, such as Pathrate [11], must run on both ends of the network path). The solution proposed in this section is based on the queueing delay estimation method described in the previous section and selects, in the set of packet pair dispersion measurements, the ones which have experienced the minimum queueing delay throughout the network: the values of their dispersion is then used to derive a reliable estimate of the whole path capacity. The performance of the tool is compared to that of Capprobe [12], since both techniques are based on one point measurements (even if a client-server implementation of Capprobe is also available).

1.2.1 Queueing delay estimation

While Capprobe selects the most reliable dispersion samples based on the probe packets' round-trip times (or one-way delays) our algorithm selects the best measurements on the basis of an estimate of the probe packets' queueing delay. The queueing delay estimation technque that we describe in this subsection is largely based around the same principles that have been illustrated in subsection 1.1.4.These ideas can be applied without significant modifications to packet pairs, since it is based on the knowledge of the packets'interdeparture and interarrival times only. In this section, though, we also add an evaluation of the effect of the measurement noise. Let us therefore indicate with $p_k^{(n)}$ the k-th (with $k \in \{1,2\}$) packet of the n-th packet pair and let us assume that both its departure time $\pi_k^{(n)}$ and its arrival time $a_k^{(n)}$ are deterministically known; this hypothesis is easily satisfied since both the arrival times and the departure times are measured by the probing host. The end-to-end delay that $p_k^{(n)}$ experiences is given by the sum of a deterministic time R_k (the one experienced by packets that do not interact with cross traffic) and a random, non-negative queueing delay $Q_k^{(n)}$, induced by the interaction of cross traffic. The packet arrival time $a_k^{(n)}$ is therefore:

$$a_k^{(n)} = \pi_k^{(n)} + R_k + Q_k^{(n)} \tag{1.11}$$

The interarrival time $X_k^{(n)}$ of packets $p_k^{(n)}$ and $p_k^{(n+1)}$ can therefore be expressed as:

$$\begin{aligned} X_k^{(n)} &= a_k^{(n+1)} - a_k^{(n)} \\ &= Q_k^{(n+1)} - Q_k^{(n)} + \left(\pi_k^{(n+1)} - \pi_k^{(n)}\right) \\ &= Q_k^{(n+1)} - Q_k^{(n)} + \tau_{n+1} \end{aligned} \tag{1.12}$$

where τ_{n+1} is the interdeparture time of packets $p_k^{(n)}$ and $p_k^{(n+1)}$. Equation (1.12) can be seen as a recursive relation between the queueing delay of the corresponding packets belonging to consecutive packet pair probes:

$$Q_k^{(n+1)} = Q_k^{(n)} + X_k^{(n)} - \tau_{n+1} \tag{1.13}$$

Unfortunately, the initial condition $Q_k^{(0)}$ is unknown; nevertheless, by choosing any arbitrary initial condition, the queueing delay estimates given by (1.12) are biased by the same, constant, initial error ϵ_k. By indicating with $\widehat{Q}_k^{(n)}$ the sequence of queueing delay estimates, the following relation holds:

$$\widehat{Q}_k^{(n)} = Q_k^{(n)} + \epsilon_k \tag{1.14}$$

Therefore, it is possible to reorder the set of packet pairs in terms of their estimated queueing delay $\widehat{Q}_k^{(n)}$ in that, as previously mentioned, the constant estimation bias does not modify the sequence order.

The packet probes experiencing the smallest queueing delay are those which most likely give the best (i.e. less corrupted by cross traffic interference) interarrival time measurements.

Equation (1.13) holds for both packets p_1 and p_2 of a packet pair: the sample selection is therefore based on the two estimates sequences $\widehat{Q}_1^{(n)}$ and $\widehat{Q}_2^{(n)}$.

Unfortunately, given the particular structure of (1.13), the sequences of estimates can be seriously affected by the propagation of measurement errors. Let us therefore consider the measured interarrival time of packets $p_k^{(n)}$ and $p_k^{(n+1)}$; the measured time interval $X_k^{(n)}$ can be expressed as

$$X_k^{(n)} = Q_k^{(n+1)} - Q_k^{(n)} + \tau_{n+1} + \sigma_k^{(n)} \tag{1.15}$$

where $\sigma_k^{(n)}$ is a random measurement noise term. In subsection 1.2.2 we will further investigate the causes and the characteristics of the measurement noise, showing that each sample $\sigma_k^{(n)}$ can be modeled as a zero mean Gaussian random variable. If compared to the other time intervals involved in the estimation process, each sample of such a noise can be considered negligible; nevertheless, as it is easy to infer from (1.13), all the noisy terms are summed up together by the recursive estimator. As a consequence, after some dozens of iterations, the estimated queueing delay $Q_k^{(n)}$ can be affected by a considerable amount of measurement noise. Even though the measurement noise will be shown to have zero mean, its variance grows at each iteration of the recursive estimation. The selection of the best packet pair dispersion sample would then be based on an unreliable information.

It is then necessary to devise an effective sample selection algorithm to take advantage of the information provided by (1.13) and to cope with the effects of measurement noise.

1.2.2 Analysis of the measurement noise

The noisy term $\sigma_k^{(n)}$ takes into account all the phenomena that influence the interarrival time measurements and that cannot be included in the queueing delay term; a wide variety of elements can contribute to the measurement error but the most relevant source of noise is introduced by the measurement process at the user level application. Such uncertainty is mainly due to the variable latencies associated with the interrupt based mechanism which is in charge of transferring a packet between the Network Interface Card and the application socket. The extent of such a delay is related to the features of the computer and the operating system on which the tool runs; moreover, it heavily depends on the workload of the whole system. With reference to (1.13), each $\sigma_k^{(n)}$ term can be expressed as the difference of the noise affecting the interarrival time measurement and the one affecting the interdeparture time measurement. Since both the noisy terms are due to the same causes and are related to a wide variety of independent phenomena, we assume that the overall measurement noise $\sigma_k^{(n)}$ may be safely modeled as a zero mean Gaussian random variable. We will assess the validity of such a hypothesis by analyzing the results of the experiments we performed on an ad-hoc experimental testbed.

Experimental testbed layout Figure 1.7 depicts the testbed used to characterize the measurement noise. The probing PC, which runs the PingPair application and probes a remote server (*probed server* in the picture), is connected to a 1Gbps Ethernet switch featured with the *port mirroring* functionality; this switch is in turn connected to the Internet via a 10Mbps Ethernet switch (this is the bottleneck link). The two ingress interfaces involved in the probing process are mirrored to an optical port (*mirrored port* in the picture) of the GigaEthernet switch that is connected to a PC (*DAG-equipped PC* in the picture). The PC is equipped with a DAG card in order to take on-wire hardware packet timestamps. The interarrival and interdeparture times captured by the probing PC and those captured by the DAG-equipped PC are compared to characterize the measurement noise.

Analysis of the experimental data After measuring the actual values of the interarrival and interdeparture times, we extracted the values of the noisy terms $\sigma_k^{(n)} \forall n$ and performed some statistical analysis on the experimental data, in order to verify our hypotheses.

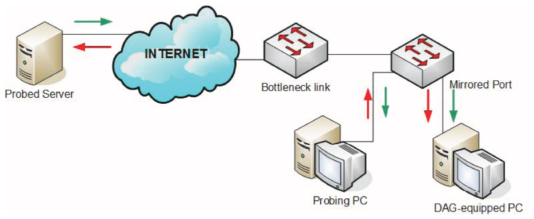


Figure 1.7: Experimental testbed for measurement noise characterization

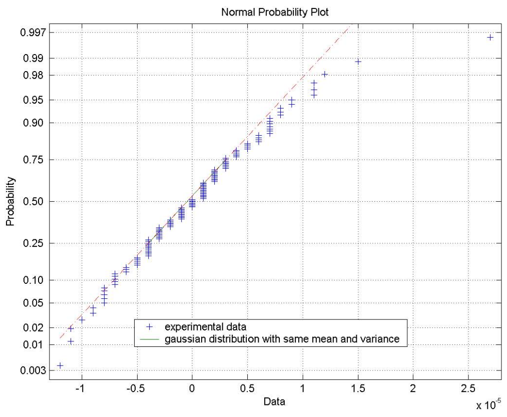


Figure 1.8: Normal probability associated to 600 experimental samples of $\sigma_k^{(n)}$

By averaging the whole set of noise samples, we found that the zero mean hypothesis is well grounded: the sample mean was always several orders of magnitude smaller than the samples themselves.

In order to verify the Gaussian distribution hypothesis, we further calculated the *normal probability plot* of each set of data. In all cases, the normal probability plots associated with the experimental data approximated quite well the linear plot, which is typical of a Gaussian distribution. An example of such a plot is shown in figure 1.8. As a more formal verification of our hypothesis, the *Smirnoff-Kolmogorov test* (with $\alpha = 0.05$) on the same set of experimental data yielded a positive result, thus confirming our qualitative conclusion.

The standard deviation of the measurement noise we observed in our experiments turned out to be of the order of about $10^{-5}sec$. Even if the noisy terms themselves are negligible (as compared to the measured dispersion values with a bottleneck of $10Mbps$), after a few dozens of iteration of (1.13) the overall noise can determine significantly wrong estimations of the queueing delay.

1.2.3 Measurements Selective Filtering

The simplest approach to cope with the accumulation of the measurement noise is to split the whole sequence $Q_k^{(n)}$ (which includes all the queueing delay estimates) into several subsequences, each of them evaluated from null initial conditions. The queueing delay estimates can then be computed over disjoint sets of consecutive samples and the measurement error propagation is limited to each set.

The experimental results that have been previously shown suggest that about a dozen of samples for each subsequence turns out to be a reasonable choice in order to keep the overall measurement error at negligible values.

Once the subsequences have been calculated, the best (i.e. less influenced by the queueing delay) sample out of each of them is selected. Notice that, if the subsequences are composed by a few packet pairs, it is possible that none of the corresponding packets has crossed the network without being significantly affected by queueing delay; a sample is accepted only if both the first and the second packets of the corresponding packet pair are the best (i.e. they have the smallest estimated queueing time) of their subsequences. After selecting the best packet pair dispersion measurements, the dispersion is calculated as the *statistical mode* of their distribution. More formally, the algorithm steps are described in Algorithm 1.

Algorithm 2 Algorithm for the selection of the most reliable dispersion measurements.

- Let N_{sub} be the number of samples composing a subset and N be the total number of available samples.
- Let B be the (initially empty) set of the reliable dispersion measurements.
- Partition the whole set of packet pair samples in a $\frac{N}{N_{sub}}$ non-overlapping subsets composed by N_{sub} consecutive probes. Let S_l be the $l-th$ subset: it will be then composed by packet pairs $\{(l-1) \times N_{sub} \dots l \times N_{sub} - 1\}$.
- For each subset S_l and for $k \in \{1,2\}$:
  - set $Q_k^{(l-1)\times N_{sub}} = 0$;
  - compute $Q_k^m \quad \forall m \in \{(l-1) \times N_{sub} + 1 \dots l \times N_{sub} - 1\}$ using equation (1.13);
  - if $\exists h | Q_k^h = \min_{j \in S_l} Q_k^j \quad \forall k \in \{1,2\}$ add the dispersion measured by packet pair h to the set B.
- Compute the dispersion as the mode of the distribution of the dispersion values in B.

The rationale of such an approach is intuitive: despite the acceptance criterion previously described, a "bad" sample can be selected out of a subsequence if no

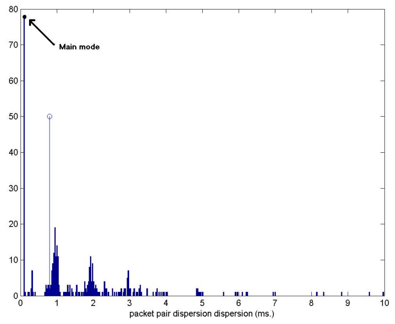


Figure 1.9: Histogram of the dispersion values of 500 probes sent to public address 66.249.93.147

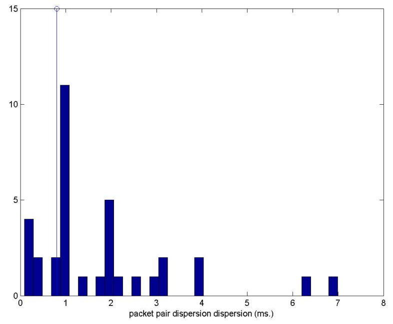


Figure 1.10: Histogram of the dispersion selected by algorithm 1 among the samples whose distribution is shown in figure 1.9

"good" (i.e. not significantly afflicted by queueing delay) samples belong to that subsequence. Although the selected "good" samples will be concentrated in the neighborhood of a mode centered in the nominal dispersion value, "bad" samples will only result as outliers. The effects of the application of this selection criterion can be easily noticed by comparing figure 1.9 and figure 1.10: the first figure shows the distribution of the dispersion of 500 packet pairs sent to Google (www.google.it) over the Internet, while the second figure shows the distribution of the "good" samples selected by algorithm 1 out of all the measurements. It is evident that, while in figure 1.9 the main mode corresponds to a dispersion value which is significantly lower than the actual one (which is indicated in both figures by the empty sample and corresponds to the actual 10 Mbps bottleneck link capacity), the dispersion of the selected samples is concentrated nearby the theoretical dispersion value.

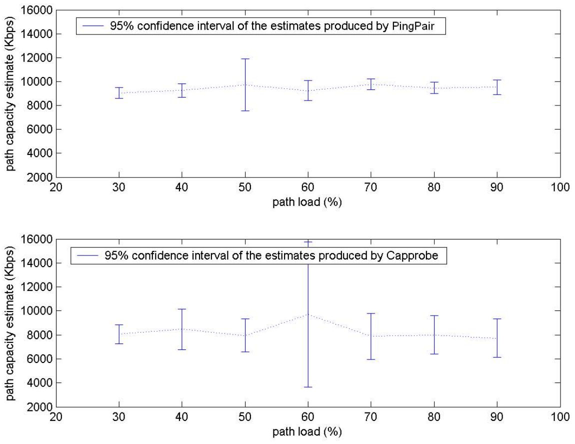


Figure 1.11: Estimates produced by Capprobe and PingPair in several network scenarios.

1.2.4 Performance evaluation through NS2 simulations

In order to assess the performance of our tool in a totally controlled network scenario, we ran several simulations by using NS2 [7]. We compared the performance to that of Capprobe, which can be simulated by using the NS2 module available at [14]. The simulation scenario consists of a 6-links path with capacities of 51.84 Mbps, 155.52 Mbps (typical of the OC links) and 10 Mbps (the bottleneck link). The cross traffic is generated by one–hop persistent TCP connections and the traffic load is the same on each link of the path.

Both capacity estimation tools have been tested in a wide range of path load conditions; in each scenario 30 independent simulations have been performed to properly estimate a confidence interval for the performance of both tools. In order to perform a fair comparison, both estimates were based on a set of 200 probes; such an amount of probes is fairly small if compared to the that needed by other tools (e.g.: Pathrate sends at least 1500 probes to estimate the capacity of the path). The results of these tests are summarized in fig. 1.11.

The results show that the estimates provided by PingPair are generally more accurate than those provided by Capprobe, since the central values of the corresponding confidence intervals is, in most of the cases, closer to the actual capacity value of 10Mbps. In addition, the estimates provided by Capprobe generally exhibit larger variability, as it can be inferred from the wider extension of the corresponding confidence intervals.

Table 1.2: Path capacity estimates provided by Capprobe and PingPair during experimental tests over the Internet. The estimates are expressed in Mbps. The actual path capacity is 10 Mbps (except for www.ecole-francaise.it, whose capacity is 2 Mbps)

path dest.	dest. addr.	probes	PingPair	Capprobe
google.it	66.249.93.147	101	9.630	15.059
youtube.com	208.65.153.251	151	8.680	11.907
myspace.com	216.178.39.13	101	9.867	3.413
ucla.edu	169.232.33.135	151	8.204	3.436
ecole-francaise.it	193.204.146.3	101	1.986	1.590
gmail.com	66.249.91.18	101	9.414	10.449
hotmail.com	213.19.160.188	101	8.467	15.515
berkely.edu	169.229.131.92	101	8.641	16.516
mit.edu	18.7.22.83	101	9.491	3.507
Tin DNS	212.216.112.112	101	9.830	0.342

1.2.5 Internet Measurements

In order to test PingPair in a real network scenario, we implemented it by writing a user level application which, at present, is still at its beta version, but will be available as soon as possible at the website of our research goup (*http://netgroup.iet.unipi.it*).

From a host located in the Network Laboratory of the Dept. of Information engineering in Pisa, we sent packet pair probes to several hosts over the global Internet and estimated the capacity of the corresponding paths; the bottleneck was always the 10Mbps Ethernet link connecting the sending host to the network. The location of the bottleneck link in the first hop of a path is quite common in real scenarios, since the links at the edges of a network are often slower than those located in the core. Again, we compared the performance of our tool to that achieved by the Linux implementation of Capprobe [14]. We always ran Capprobe immediately after PingPair, so as to test both tools in the same network conditions. The two tools used the same number of packet pairs.

The results of several experiments are reported in table 1.2 and confirm the capability of PingPair of providing fairly good estimates by using an extremely limited number of samples.

1.2.6 Field Trial Measurements

Finally, we tested PingPair in an experimental testbed, in order to assess its performance in a scenario which is both realistic (a true path probing process is performed) and controlled (both the amount and the characteristics of the interfering traffic are perfectly known).

The logical topology of the testbed is shown in figure 1.12: the network path crossed by probes consists of four Linux Boxes equipped with several Ethernet NICs

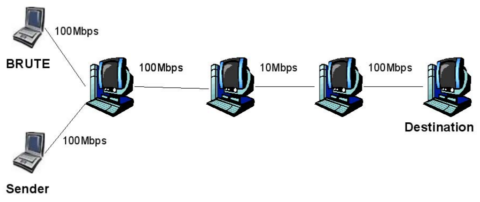


Figure 1.12: Topology of our experimental testbed.

Table 1.3: Path capacity estimates provided by Capprobe in field-trial experiments. The estimates are expressed in Mbps while the actual value of the path capacity is 10 Mbps

tr. type	tr. rate (Mbps)	tr. pkt size	Capacity estimate
CBR	9.8	800	8.9
CBR	9.5	1000	9.55
CBR	9.5	1400	10.833
Poisson	8.5	800	9.451

and statically configured for routing. Packet pairs and cross traffic are originated by two laptops connected to the first PC of the path and are directed to the last PC; in particular, interfering traffic was generated by using the open source BRUTE [15] traffic generator. This kind of layout allowed to test our tool with highly persistent cross traffic, since the interfering traffic shared the whole network path with the probe packets. As in the previous experiments, the bottleneck link corresponds to a 10 Mbps Ethernet link.

As for most of the Internet experiments, the path capacity was estimated by sending 100 probes only. The size of the interfering packets was varied through the experiments; as claimed in [11], the size of cross traffic packets may have a strong influence on the location of the modes of the distribution of the packet pair dispersion.

The results of the experiments are reported in table 1.3, together with the characteristics of the interfering traffic. Once more, they show that, even in a heavily loaded network, PingPair provides fairly good estimates by taking advantage of a very few packet pairs.

1.3 A Decision Theoretic Approach to Network Topology Discovery

In section 1.1 we discussed how hierarchical clustering techniques, when applied to the packet sandwich probes, can be used to infer the network topology. In this section we will propose a completely novel approach to such problem, which radically changes the perspective by turning the topology inference problem into a decision within a discrete hypotheses set. Indeed, under the hypothesis that network link capacities belong to standard well known sets (Ethernet, OC), we define a finite space of possible topological hypotheses; within such a domain, the most likely hypothesis is chosen. This assumption appears to be realistic, at least in local area networks, where almost all links use the Ethernet (or 802.11 in the wireless scenario) MAC protocol, which defines a restricted set of link bandwidths. As it will be shown, with respect to traditional tomographic approaches, our technique allows to reveal a larger number of internal nodes; in addition, the capacity of the network links can be effectively inferred.

1.3.1 The decision theoretic approach

Let us first recall briefly some details about the packet sandwich probe, that has already been introduced in subsection 1.1.2. A packet sandwich (as illustrated in figure 1.1)is a particular packet train composed by three packets, which are sent back to back by the sender and that will be indicated as p_1, q and p_2. Packets p_1 and p_2 are L_1 Bytes long and are directed to the same receiver, while q is L_2 Bytes long and directed to another receiver (usually, L_1 is 56 Byte while L_2 equals the network MTU). At every hop of the shared path between the sender and the two receivers, the third packet gets backlogged behind the second (much bigger) packet and the delay difference between p_1 and p_2 increases.The interarrival time d of packets p_1 and p_2 can therefore be used as a similarity metric, since it increases with the length of the path shared by the pair of receivers.

Equation (1.16) shows the analytical expression of the interarrival time of packets p_1 and p_2 as a function of the capacities of the N links composing the shared path (the complete derivation of the formula below can be found in [16]:

$$d_{i,j} = \frac{L_2}{C_1} + \sum_{k=2}^{N-1} \frac{L_2 - L_1}{C_k} + \frac{L_2}{C_N} \tag{1.16}$$

Equation (1.16) shows that (as stated also in [4]) the packet sandwich dispersion is incremented at each link k of the shared path, provided that the following relation,involving the capacity C_k of link k and the capacity C_{k+1} of the following link, is satisfied:

$$\frac{L_2}{L_1} > \frac{C_{k+1}}{C_k}$$

With a MTU of 1500 bytes and $L_1 = 56B$ such a condition turns out to be roughly $\frac{C_{k+1}}{C_k} < 25$, which, as also stated in [4], is typically satisfied in actual networks. Equation (1.16) shows that the packet sandwich based metric is basically given by the sum of the inverse of the shared path link capacities. In real networks, link capacities usually belong to a restricted set of standard values (Ethernet, Optical Carrier ...); the set of capacities $\{C_1, C_2, \ldots\}$ is then discrete. In addition, it is generally possible to obtain a reliable estimate of the depth of the spanning tree, i.e. the maximum number of links composing the path between the probe sender and each receiver by taking advantage of the TTL field of IP packets. Therefore, given the set of all possible link capacities, if an upper bound of the number of hops composing a path is available, equation (1.16) can be used to off-line pre-evaluate a table with all the possible values of the similarity metric d. More formally, let us define:

$$\mathcal{C} = \{c^1, c^2 \dots c^{k-1}, \infty\}$$

as the set of all the possible values of link capacities. Notice that $\mathcal{C}$ includes the symbol ∞ in order to represent sequences of different length with the same number of elements (i.e., including zero delay links). Any possible combination of D elements of $\mathcal{C}$ is an element of the Cartesian product $\mathcal{C}^D$ and will be referred to as *Link Capacities Combination (LCC)*.

Let us define the function:

$$\gamma : \mathcal{C}^D \to H$$

as:

$$\gamma(\mathbf{C}) = \frac{L_2}{C_1} + \sum_{k=2}^{D-1} \frac{L_2 - L_1}{C_k} + \frac{L_2}{C_D} \tag{1.17}$$

where $\mathbf{C} = (C_1, C_2, \dots, C_D)$ is an arbitrary D-links capacity combination and $H = \{d_1, d_2, \dots, \}$ is the finite discrete subset of $\mathbb{R}$ which contains all the possible values assumed by (1.17). As previously mentioned, the vector $\mathbf{C}$ may include ∞-valued components. In order to list the actual finite-capacity elements of $\mathbf{C}$, we define the set:

$$\mathcal{B} = \{C_i \,:\, C_i < \infty\}$$

Given a specific pair of destinations (i, j), an LCC associated to their shared path is indicated as $\mathbf{C}_{i,j}$, while the set of actual link capacities is referred to as $\mathcal{B}_{i,j}$. Notice that, while vectors $\mathbf{C}_{i,j}$ have all the same length D, the cardinality of $\mathcal{B}_{i,j}$ depends on the actual number of links of the shared path. Given estimates $\hat{d_i}$ of the metric (1.17) obtained by averaging a given number of interarrival time samples, the fundamental decision problem is to select the correct LCC of length D that originated it. Although the space $\mathcal{C}^D$ is both discrete and finite, equation (1.17) is not invertible, since different link sequences yield the same value in H; as an example, two LCCs that only differ for the elements order produce the same value of the metric. Topology reconstruction based on LCCs decisions can therefore be affected by a certain degree of ambiguity due to link ordering; nevertheless, the incidence of such ambiguity, for realistic network sizes and capacity sets, is limited, as it can be noticed from table 1.4, where the percentage of ambiguous LCCs over the number of all the possible LCCs is evaluated for different network scenarios. In addition, by sending packets p_1, p_2 and q of a sandwich to the same destination, it is possible to find out the whole combination of end-to-end capacities. In those cases, to make the notation consistent to (1.16), the metric measured when all packets of the sandwich are sent to the same receiver i will be denoted as $d_{i,i}$.

The use of the decision theory is motivated by the evidence that packet sandwich measurements are disturbed by the interaction with regular traffic, that acts as noise. It will be then necessary to decide which LCC most likely generated the observed interarrival time measurements: effective strategies should then be devised in order to select the most likely hypothesis in the space H. The main issue of decision is the lack of a statistical model of the cross traffic interference on packet sandwich interarrival times. We can nonetheless model the sample mean of a set of samples of interarrival time as a Gaussian random variable [4] by invoking the Central Limit Theorem. Under this hypothesis, the optimal decision can be made according to the Maximum Likelihood criterion that, in the space H, collapses into the minimum distance criterion. Let $\widehat{d}$ be the sample mean of the available measured interarrival times; the selected link combination $\overline{\mathbf{C}}$ will be:

$$\overline{\mathbf{C}} = \arg\min_{\mathbf{C}\in\mathcal{C}^{\mathcal{D}}} \left|\widehat{d} - \gamma(\mathbf{C})\right| \tag{1.18}$$

An upper bound on the probability of taking a wrong decision can be obtained through the union bound:

$$P_e \leq \sum_i \sum_{\substack{m \\ m\neq i}} Q\left(\frac{d_m - d_i}{2\sigma}\right) Pr(d_m) \tag{1.19}$$

where $d_m, d_i \in H$, $Q(x) = \int_x^{+\infty} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$ is the error function, σ is the variance of the sample mean and $Pr(d_m)$ represents the probability of d_m. Clearly, equation (1.19) shows that the reliability of decisions improves as the distance of the metrics associated to the different link combinations in H increases. Once the decisions are taken for each pair of receivers, they can be elaborated to infer the network topology. Traditional hierarchical clustering algorithms cannot be adapted for this purpose, in that they require the knowledge of a continuous-valued similarity metric $\gamma_{i,j}$ for each pair (i, j) of receivers. On the contrary, in this case, for each pair of receivers an LCC $\mathbf{C}_{i,j}$ is available upon decision. Such a task is accomplished by the next algorithm 3, which reconstructs the complete spanning tree of the probe sender based on the decided $\mathbf{C}_{i,j}$ LCCs. In order to reconstruct the complete topology of the tree, it is also necessary to estimate the $d_{i,i}$ metric for each receiver i, so as to obtain, by decision, the $\mathbf{C}_{i,i}$ combination. The underlying idea of the proposed algorithm is analogous to that of the binary tree algorithm [5], but it does not include a tree pruning phase. Indeed, with discrete-valued similarity metrics, if three nodes (m, n, l) have the same parent, the following relations will be exactly verified:

$$\mathbf{C}_{m,n} = \mathbf{C}_{m,l} = \mathbf{C}_{n,l} \tag{1.20}$$

Therefore the tree reconstructed by algorithm 3 will not be in general binary.

Let us consider now two arbitrary internal nodes k and h; the set of link capacities composing the path between the internal nodes k and h will be denoted as $\Lambda_{k\rightarrow h}$. Once all the combinations $\mathbf{C}_{i,j}$ have been decided for each pair (i, j) of receivers, the reconstruction algorithm 3 operates as follows:

If the capacity combinations $\mathbf{C}_{i,j}$ are correctly decided, algorithm 3 reconstructs the complete topology of the network as it is visible by the probe sender through its

Table 1.4: Ambiguous LCCs in different network scenarios

Capacities	% ambiguous	tree depth
10Mb-100Mb-1Gb and OC (max OC-48)	4.5%	6
Ethernet (10Mb-100Mb-1Gb-10Gb)	0%	6
OC (OC-1 to OC-765)	5.4%	4
10Mb-100Mb-1Gb and OC (max OC-48)	0.6%	4
Ethernet (10Mb-100Mb-1Gb-10Gb)	0%	4

Algorithm 3 Tree Reconstruction

```
• Initialize the set $L$ containing the leaves of the spanning tree.
• For each leaf $i$, define $\Lambda_{root \to i} = \mathcal{B}_{i,i}$
• Until the set $L$ is not empty
    1. Choose the set $\mathcal{B}_{i,j}$, with $(i, j) \in L$, composed by the maximum number of links
    2. Find the set $M \subset L$ such that $\mathcal{B}_{m,n} = \mathcal{B}_{i,j} \forall n, m \in M$
    3. Remove from $L$ all the nodes that belong to $M$
    4. Add to $L$ a new node $k$ ($k$ will be the nearest common ancestor of the $M$ nodes)
    5. For each node $h \in L$ such that $h \neq k$, define $\mathcal{B}_{k,h} = \mathcal{B}_{m,h}$ with $m \in M$
    6. Define $\Lambda_{root \to k} = \mathcal{B}_{i,j}$
    7. For each $m \in M$, define $\Lambda_{k \to m} = \Lambda_{root \to m} \setminus \Lambda_{root \to k}$
```

spanning tree. However, the topology so inferred still suffers from a certain degree of ambiguity: in general, the algorithm cannot reveal the ordering of the link capacities that compose the path between two adjacent branching points. In the case of wrong prior decisions, the combinations $\mathbf{C}_{i,j}$ will likely be incoherent and the algorithm will probably not converge: in particular, the operation $\Lambda_{root \to m} \setminus \Lambda_{root \to k}$ will be mathematically impossible. This feature, actually, turns out to be an advantage, since the presence of decision errors can often be revealed.

1.3.2 Noise reduction

The noise affecting the packet sandwich measurements can lead to a completely wrong topology reconstruction, especially in the cases where the possible metrics in H are very close to each other. For this reason we take advantage of a noise reduction algorithm in order to "denoise" the dispersion measurements obtained from the packet sandwich probes, before taking any decision about the most likely LCCs and using the tree reconstruction algorithm. Such a noise reduction technique, whose detailed discussion can be found in [16], does not rely on any statistical delay model and can be applied to any kind of dispersion measurements. Paper [17] proves it to be effective in enhancing dispersion based capacity estimation and that it can also cope with the measurement noise caused by the variable latencies associated with the interrupt based packet reception in ordinary PCs.

The rationale of the algorithm is to choose the most reliable samples of interarrival times out of all the available observations. Since the interaction of cross traffic with packet sandwich probes results in an increased queuing time, the smaller the queuing delay experienced by probe packets, the better the measurements. Therefore the noise reduction algorithm we take advantage of first estimates the queuing delays experienced by the probe packet through a recursive equation and, after that, based on the estimated issued by such equation, selects the best dispersion samples.

1.3.3 Computational complexity

In order to evaluate the scalability of our technique, we discuss here the computational complexity of the approach described in subsection 1.3.1. Such approach basically consists of two phases: the decision, for each pair of receivers, of the most likely LCC and the actual topology reconstruction, performed by applying algorithm 3.

In order to keep the problem analytically tractable, we assume to apply our algorithm to a binary tree topology where the sender is located in the root. Since the number of iterations of the tree reconstruction algorithm grows linearly with the number of branching points, such a topology represents, from that point of view, a worst case scenario. In addition, some properties of the binary tree topology are still valid also for most random graph models which are commonly used to describe the Internet Topology: in Scale Free (SF) graphs [18] the overall number of nodes grows as fast as the number of leaves (the two quantities are asymptotically equivalent). Moreover the network diameter (that we use here to upper bound the depth of the spanning tree) is proportional to the logarithm of the number of nodes (because of the small-world properties of SF graphs). Let then n be the number of probe receivers, N_{branch} be the number of branching points (which, in the binary tree case, turns out to be also the number of internal nodes) and d be the number of hops between the probe sender and any receiver. The following relations are easily verified:

$$N_{branch} = n - 2 \qquad (1.21)$$

$$d = \log_2 n \qquad (1.22)$$

We evaluate the computational cost as a function of n since the number of leaves is the only topological characteristic which is known a priori. Let us first consider the decision phase: for each receiver pair the minimum distance LCC must be determined, and, therefore, a list of all the possible metrics must be searched. By assuming a set of K possible link capacities, since the order of the capacities for each LCC is irrelevant, the number of elements composing such a list will be:

$$\frac{(K+d-1)!}{(K-1)!\cdot d!}=\frac{(K+\log_2 n-1)!}{(K-1)!\cdot \log_2 n!}$$

Since, as already pointed out, such a list can be built off-line, it is possible to speed up the algorithm by building it as an ordered (with reference to the dispersion metric) list, so as to use a binary search algorithm which allows to find an element in a logarithmic (instead of linear) time. Under such a hypothesis, and by noticing that a decision has to be made for each pair of receivers i, j (even when $i = j$), the cost of the whole decision phase equals:

$$\omega_{dec}=\frac{n^2}{2}\cdot\log_2\left(\frac{(K+\log_2 n-1)!}{(K-1)!\cdot \log_2 n!}\right)$$

In order to evaluate the computational cost of alg. 3, we observe that such an algorithm basically consists of finding, for each of the N_{branch} branching points, the intersubsection and the difference of two decided LCCs; by adopting a trivial technique (for each element in a LCC, a match in the other one is searched for) such an operation is performed in a time proportional to l^2, where l is the number of elements composing the LCCs. Under the assumption of a binary tree, and by considering $l = d = \log_2 n$, the overall cost of alg. 3 equals:

$$\omega_{alg}=N_{branch}\cdot d^2=(n-2)\cdot\log_2^2 n$$

The overall computational cost of our technique $\omega = \omega_{dec} + \omega_{alg}$ is therefore:

$$\omega(K,n)=\frac{n^2}{2}\cdot\log_2\left(\frac{(K+\log_2 n-1)!}{(K-1)!\cdot \log_2 n!}\right)+(n-2)\log_2^2 n \tag{1.23}$$

For high values of n, the largest contribution in (1.23) is given by the cost of the decision phase and it turns out to be of the order of the thousands of operations even for very large networks, with up to $n = 50$ probe receivers.

Finally, for more complex networks, the value of $\omega(K,n)$ should be multiplied by the number of sender nodes required. However, this is not a big issue since the number of required sender nodes (*beacons*) to discover a complex SF topology can be limited to the order of $\log(n)$ [19].

1.3.4 Performance evaluation

We evaluate our topology discovery technique by performing a set of simulations based on ns2 [7]. We simulate the probing process over a wide series of topologies. However, due to space limitations, we can report only a few examples. The first case study we describe is based on the SF topology reported in fig. 1.13 which has been generated by BRITE [20] according to the Barabasi-Albert model; such a topology is composed by 35 nodes and 34 links whose capacities are randomly selected within

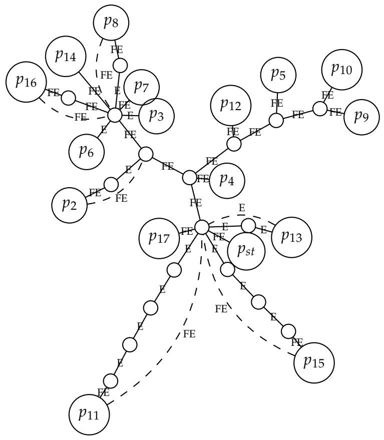


Figure 1.13: 35 nodes topology generated by BRITE and its reconstruction by an ideal hierarchical clustering algorithm (dashed links)).

the standard Ethernet set and described as *E*, *FE* and *GE* respectively for 10 Mbps, 100 Mbps and 1Gbps. In order to reproduce the interaction between the probing traffic and the regular traffic over an actual network, we flooded the internal links with packets generated by variable bit rate traffic sources; the interfering traffic rate is, for each link, about 50% of the capacity. We simulate other probing process by sending, from the node p_{st}, 60 packet sandwiches for each pair of leaves of the topology tree. The topology reconstructed by our algorithm differs from the one shown in fig. 1.13 only for 4 mistaken link capacities and one additional inexistent $100Mbps$ link attached to receiver p_{15} (differences are dashed in the figure);

In order to provide a benchmark, we compared the output of our algorithm withthe topology which would be yield, in case of a completely correct reconstruction, by a traditional clustering based technique. Out of the 35 nodes of the original topology, only 26 are detected, and the capacities of the links are not revealed; the overall picture turns out to be consistently less detailed than the one produced by our technique.

By using the same simulation methodology (in terms of interfering traffic, topology generation and probing scheme) we then apply our technique to a larger topology, composed by 45 nodes and shown in figure 1.14. In this case, the inferred topology differs from the real one for 6 mistaken link capacities only.

Another case study is based on a non-randomly generated topology: it is inspired to an actual network, where the peripheral nodes are connected to core nodes which have a higher degree and are attached to faster network links; again, the capacities are chosen within the Ethernet set (10,100,1000 $Mbps$). In this case, we generate the interferent traffic by establishing TCP connections between the leaves and some of the core nodes, thus reproducing realistic traffic patterns. The simulated topology is shown in figure 1.15 (along with the reconstruction errors). A hundred packet sandwiches for each pair of leaves are sent from node p_{1a} and, in the reconstruction

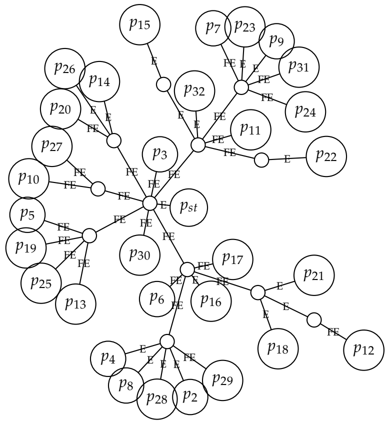


Figure 1.14: 45 nodes topology generated by BRITE.

given by our algorithm (that we do not report here for lack of space)only two little mistakes (dashed links) are present: an additional node, connected to node labeled p_{1d}, is present and a link capacity is wrongly revealed. However, the overall topology is quite close to the original one.

Those simulative results show that our of technique is capable of inferring a detailed and quite reliable picture of the network topology and, besides, that its effectiveness is only marginally influenced by the size of the network under probing.

1.4 Network Topology Discovery through Self-Constrained Decisions

In the previous subsection we introduced a decision theoretic approach for network topology discovery. The cornerstone of such approach is the correspondence between a measurement based similarity metric and a discrete sequence of possible link capacities $\mathbf{LCC}_{i,j}$ associated to the receiver pair i, j.

If the capacity combinations $\mathbf{LCC}_{i,j}$ are correctly decided, the proposed algorithm reconstructs the complete topology of the network as it is visible by the probe sender through its spanning tree. However, the topology so inferred still suffers from a certain degree of ambiguity: in general, the algorithm cannot reveal the ordering of the link capacities that compose the path between two adjacent branching points. Besides, in the case of wrong prior decisions, the combinations $\mathbf{LCC}_{i,j}$ will likely be incoherent and the algorithm will probably not converge.In this section, we extend the previoulsy illustrated technique by making measurements quality (practically their variance) to influence the decision process so that the decisions on the most reliable (i.e., lowest variance) measurements are made at first and they help the decision process on less reliable ones. This is motivated by the large degree of correlation among measurements: when the links of a subpath of a larger path are decided, the

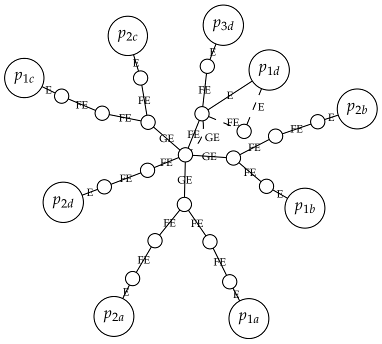


Figure 1.15: Realistic topology (continuous lines) composed by 25 nodes and its reconstruction by our technique (dashed links represents the differences).

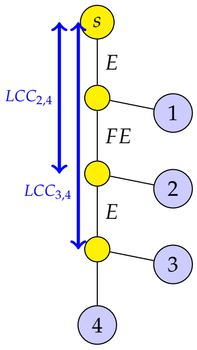


Figure 1.16: Topological dependencies.

hypotheses domain for the links of the larger path can be effectively reduced, thus improving the decision and the overall process. As we show, our technique allows to reveal a larger number of internal nodes than traditional tomographic approaches and our previous proposal. In addition, the capacity of the network links is effectively inferred.

1.4.1 Self-constrained topology reconstruction

A minimum distance based decision criterion seems to be the best choice for coping with several independent measures. However, as LCCs have, in general, topological relations with each other, in the following we propose a decision scheme which takes advantage of these dependencies in order to provide more reliable decisions. Such a scheme represents the main novel contribution of this section. In order to explain how the topological dependencies among LCCs can be exploited, let us consider the example in figure 1.16 and let us assume that the decision $LCC_{2,4}$ has been correctly made and that the maximum tree depth has been estimated to be 4 hops. If the measurements are not completely wrong, it will appear that $\gamma_{2,4} \leq \gamma_{3,4}$ and, therefore, $LCC_{2,4} \subseteq LCC_{3,4}$. Since, as assumed, $LCC_{3,4}$ can encompass 4 link capacities at most and, since it has to admit $LCC_{2,4}$ as a subset, the value of three capacities out of four is already constrained. Therefore the reference decision space of the topological hypotheses is quite reduced (it actually collapses into the space of the possible link capacities) and, as a consequence, the error probability is remarkably reduced. This simple example provides the motivation to our work and its generalization defines an useful idea: the results of prior decisions can be exploited, in terms of topological constraints, in order to improve the reliability of the following decisions.

In details, the topological constraints which can be issued by the decision of a generic $LCC_{i,j}$ can be formalized as follows: $\forall k \neq i, j \in \mathcal{R}$, where $\mathcal{R}$ is the set of receivers

$$\begin{aligned} \gamma_{k,i} \leq \gamma_{i,j} &\Rightarrow LCC_{k,i} \subseteq LCC_{i,j} \\ \gamma_{k,i} \geq \gamma_{i,j} &\Rightarrow LCC_{k,i} \supseteq LCC_{i,j} \end{aligned} \tag{1.24}$$

These two cases are depicted respectively in fig. 1.17(a) and fig. 1.17(b). We associate to each $LCC_{k,i}$ two constraints in the form of a pair of set $SB_{k,i}$ and $SP_{k,i}$, so that :

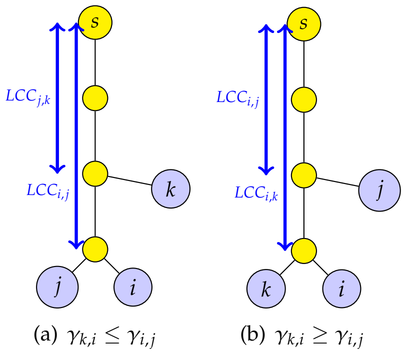
$$\begin{aligned} LCC_{k,i} &\subseteq SP_{k,i} \\ LCC_{k,i} &\supseteq SB_{k,i} \end{aligned} \tag{1.25}$$

After each decision, these two sets are updated as follows:

$$\begin{aligned} SP_{k,i} &= SP_{k,i} \cap LCC_{i,j} \\ SB_{k,i} &= SB_{k,i} \cup LCC_{i,j} \end{aligned} \tag{1.26}$$

where the last decision has been made on $LCC_{i,j}$. It is trivial to verify that the sets of equations (1.25) and (1.26) simply enforce the conditions in (1.24).

As more and more decisions are taken, the remaining constraints for the LCCs become tighter; since the tighter the constraints, the smaller the hypotheses space, the probability of decision errors for heavily constrained LCCs decreases (indeed, (1.19) states that the error probability increases as the hypotheses space grows crowded). Furthermore, contraints from previous decisions can prevent errors that may occur due to the presence of ambiguous LCCs. However, the order in which the decision are taken appears to be critical for the efficiency of this algorithm: if an error is made during the very first iterations, it will propagate to the rest of the inferred topology. In order to schedule the decisions properly we sort the available similarity metrics according to the module of their difference from the associated minimum distance LCC. The rationale of such approach is intuitive: measurements which were not distorted by the interfering traffic will be very close to their theoretical value. More formally, by taking advantage of well know results in the field of decision theory, it is straightforward to prove that the distance of the noisy observation from the nearest



$LCC_{j,k}$
$LCC_{i,j}$
s
k
j
i
(a) $\gamma_{k,i} \leq \gamma_{i,j}$
$LCC_{i,j}$
$LCC_{i,k}$
s
j
k
i
(b) $\gamma_{k,i} \geq \gamma_{i,j}$

Figure 1.17: Superset and subset constraints.

theoretical value grows monotonically with the decision error probability.

1.4.2 Theoretical Limitations

In this subsection we give a bound on the performances of our approach which also provides a guideline for its application. We aim to show that the self-constrained technique is stochastically less error-prone than a regular minimum-distance decision approach, more formally:

$$P_i^{sc}(e) \leq P_i^{md}(e) \tag{1.27}$$

where i is the decision step (we have a decision per link) and the exponents of $P_i(e)$ denote the approach (*sc*: self-constrained, *md*: minimum-distance). The self-constrained approach heavily depends on the goodness of early decisions (that is why the less-noisy decisions are made first), therefore the expansion of $P_i^{sc}(e)$ can be divided into two cases according to the occurrence of the event of "no error so far" (i.e. before step i). Indeed, if an error occurred somewhere between step 0 and $i-1$, then the constrains we apply might be mistaken and they might affect the decision on step i producing another error. On the other hand, if all the previous decisions are correct, also the constrains are correct and the decision can only benefit from them. However, in both cases, once the constraints are applied, the self-constrained approach requires a minimum-distance decision to be performed, therefore we can write:

$$\begin{aligned} P_i^{sc}(e) = \quad & P_i^{md}[e|\text{NO ERROR}]P_i[\text{NO ERROR}] + \\ + \quad & P_i^{md}[e|\text{SOME ERRORS}]\,(1 - P_i[\text{NO ERROR}]) \end{aligned}$$

Now, the conditioned probabilities $P_i^{md}[e|\text{NO ERROR}]$ and $P_i^{md}[e|\text{SOME ERROR}]$ are respectively lower and higher than the unconditioned $P_i^{md}(e)$:

$$P_i^{md}[e|\text{NO ERROR}] = P_i^{md}(e) - \pi_g$$
$$P_i^{md}[e|\text{SOME ERRORS}] = P_i^{md}(e) + \pi_l$$

where $\pi_g, \pi_l \geq 0$ are factors that capture the penalty and advantages of bad and good decisions in the previous steps. Then, through simple substitutions on eq.(1.27) we obtain:

$$P_i[\text{NO ERROR}] = \prod_{j=1}^{i-1} \left(1 - P_j^{sc}(e)\right) \geq \frac{\pi_l}{\pi_g + \pi_l} \tag{1.28}$$

In the following tractation, we impose $\pi_l = \pi_g$. Here we provide a simplicistic argument for this choice: whenever we apply constraints (either they are correct or mistaken) we ultimately eliminate a number of possible solutions in the solution domain; without additional knowledge of the solutions, the best model is the uniform one, hence the imposition of equality. Under this conditions, the probability of having no errors up to step i must be simply greater than $1/2$. Now, a couple of considerations are possible. First: eq. (1.28) sooner or later will go unsatisfied because of the large productory that is monotonically decreasing with i. This means that the self-constrained approach is stochastically better than the minimum-distance one up to a certain number of decisions (there is a critical i_{th} such that any $i \geq i_{th}$ in eq. (1.28) makes the disequation unsatisfied). This suggests we should divide the network graph under observation in different subgraphs that can be separately explored by means of the self-constrained technique. A second consideration actually helps relaxing this directive: in real graphs, we observe trees which, intrinsically, are divided in subtrees with very few (if any) links in common. This means that the separation mentioned above is naturally provided by the topology (i.e., there is a sort of *limited memory* in the exploration of a tree) and eq. (1.28) is always satisfied, which implies that the self-constrained approach performs always stochastically better than the minimum distance technique.

1.4.3 Algorithms evaluation

In the following a number of tests results are shown which aim to show the effectiveness of the algorithms we proposed in this section. We evaluate our topology discovery technique by performing a set of simulations based on ns2 [7]. We simulate the probing process over a wide series of topologies generated by BRITE [20]. Graphs have been generated according to well-known models: Barabasi-Albert [18] and Waxman [21], which have proved to well-describe real topologies and are widely adopted in literature.

1.4.3.1 Measuring Graph similarity

The main concern of this work is to improve the reconstruction of the network graph from a certain set of measurements, therefore the problem of defining the quality of the reconstruction arises. In details, we want to measure how similar the reconstructed topology and the original topology are. This is a widely investigated problem in the field of computer vision [22] and pattern recognition. Unfortunately the problem of graph isomorphism (recognizing if two graphs are topologically equivalent) is known to be NP-hard [23], hence also the computation of the *edit distance* of two graphs (i.e., the number of editing operations to perform on a graph in order to obtain the second) is unacceptably expensive.

However, it has been shown [24] that the *edit distance* of two graphs $G1$ and $G2$ is linearly dependent with the euclidean distance of their spectra (i.e., the ranked set of their eigenvalues). Therefore, in this section we adopt this metric as a "graph similarity score". In particular, we compute the spectrum of the *signless* Laplacian matrix L because it best reproduces the linear dependence with the edit distance as shown in [24].

By defining a graph as $G{=}(V,E)$ with V as vertices and E as edges, the Laplacian matrix L of G can be computed as:

$$L(u,v) = \begin{cases} d_u & \text{if} \quad u = v \\ 1 & \text{if} \quad (u,v) \in E \\ 0 & \text{otherwise} \end{cases} \tag{1.29}$$

where d_u represents the degree of node u.

It is worth reminding that this definition of the matrix $L(u,v)$ does not capture any information on the bandwidth of the edges in the network, hence a spectra comparison based on such a matrix does not allow for the recognition of errors in the bandwidth reconstruction, because links are simply defined with a 1 in the row and column corresponding to the ends of the edge. However, as stated in [24], this spectral approach can be easily extended to weighted graphs. In this case, we re-define the signless Laplacian matrix of G as:

$$L_w(u,v) = \begin{cases} \sum\limits_{j=1 \dots d_u, j \neq u} w(u,j) & \text{if} \quad u = v \\ w(u,v) & \text{if} \quad (u,v) \in E \\ 0 & \text{otherwise} \end{cases} \tag{1.30}$$

where $w(u,v)$ defines the weight of the link (u,v). In the following we impose the weights as $1,2,3$ respectively for Ethernet (10Mbps), Fast Ethernet (100Mbps) and Gigabit Ethernet (1Gbps). With this linear choice of weights, a mistaken decision for a 1Gbps link in place of a 10Mbps counts as twice as an error between a 100Mbps and a 10Mbps.

1.4.3.2 Results

The results of the experimental evaluation of the self-constrained technique are shown in tab. 1.5. The table presents the euclidean distance between the spectra of the original topology and the reconstructed one obtained through a minimum distance approach (Δ_{md}) and through the self-constrained technique (Δ_{sc}). The results are reported both for the spectra of the simple signless Laplacian matrix and for the weighted signless Laplacian matrix of the graphs (denoted by the w exponent in the

Nodes	Noise	Δ_{md}	Δ_{sc}	Δ^w_{md}	Δ^w_{sc}
30	Moderate	1.86	1.4	3.12	2.23
35	Moderate	2.94	1.77	4.39	2.31
35	Heavy	3.63	2.75	4.84	3.22
40	Moderate	3.33	2.72	11.57	8.63
40	Heavy	4.21	3.75	11.43	8.76
45	Moderate	7.52	2.92	11.67	4.34

Table 1.5: Euclidean distance of spectra between reconstructed graph and original topology for a regular technique (Δ_{md}) and the self-constrained approach (Δ_{sc}). The exponent w indicates the similarity score of weighted laplacian matrices.

Δ symbol).

The values in the table show that the reconstruction of the self-constrained technique is always more accurate ($\Delta_{sc} \leq \Delta_{md}$) than the minimum distance approach for all the topologies under examination. Therefore this means that, referring to the results of sec. 1.4.2, the approach is effective even for quite large topologies, confirming the intuition of a *limited memory* in tree topologies explorations. Moreover, it is noteworthy that even if the amount of noise in the experiments varies from moderate to heavy, the results do not change: the self-constrained approach gives again the best reconstructions for both similarity scores. This confirms the effectiveness of the technique for the detection of real link capacities.

1.5 Merging Spanning Trees in Tomographic Network Topology Discovery

Analogously to all standard topology discovery techniques, the algorithm described in the previous subsection is intended to reconstruct the spanning tree of the probe sender. In order to achieve a complete representation of the network, it is therefore necessary to merge the spanning trees associated with all the different roots. Merging the trees from different beacons can be a hard task [25], especially because any solution is affected by the aliasing problem. Such a phenomenon occurs whenever the same node is observed within several measurements that cannot be correlated: as a result, two or more nodes are redundantly revealed and associated with different labels. Most of the existing tools for alias resolution use an active probing approach, which induces a significant traffic overhead into the network and critically depends on the participation of the routers. More recent solutions try to avoid such an overhead. For instance, Alias Resolver [26] utilizes the common IP address assignment scheme to infer IP aliases from collected path traces: it is able to significantly detect several IP aliases. Moreover, the difficulty of the tree-merging process also depends on the amount of information provided together with the trees. Indeed, traditional topology discovery techniques usually need a certain degree of cooperation from internal nodes (such as *traceroute* [27], SNMP querying [28], OSPF listening [29]) and produce address-labeled graphs. Therefore, DNS and IP address correlation may help the tree-merging process [26]. However, in most cases, those techniques cannot be applied in practice for many reasons such as the presence of ICMP rate limiters as well as query blocking firewalls, etc. However, this is not the focus of network tomography techniques, which intend to reveal network internals by considering the network itself as a black box and by using active measurements obtained from probe packets exchanged within a set of network hosts (usually at the border of the graph). In [30] a procedure for combining end-to-end multicast measurements made independently from multiple sources is described, but the authors assume that the topology is known. Instead, the approach in [25] does not require the prior knowledge of topology and presents a multiple source active measurement procedure using a semi-randomized probing scheme and packet arrival order measurements. However, such an approach requires, in most cases, additional probe packets apart from those needed to reconstruct the single trees. In this section we propose a solution to the tree-merging problem that can be applied to trees with very limited information, such as the ones obtained by means of tomographic topology discovery techniques. In particular, such an algorithm is specifically designed to merge the trees produced by the topology discovery method we have presented in the previous sections. Many standard methods of tree merging are not able to handle trees which present also internal nodes which are not branching points, or they need further information by the probes. Therefore, an enhanced and more efficient tree merging algorithm is needed. The algorithm that we propose yields, in case of no errors of the underlying decision based technique and no ambiguous nodes, a perfect picture of the network without need for further probing.

1.5.1 The merging algorithm

A novel tree merging algorithm is presented in this subsection. It does not require further probing traffic and is able to handle spanning trees where all the nodes of the network (i.e. not only the branching points) are revealed. In addition, this algorithm is specifically designed to be applied in network scenarios where each cooperating host is both a probe sender and a probe receiver. It is assumed that the network under test implements symmetric routing, that is the route which connects any arbitrary pair of nodes A and B in the forward direction crosses the same links as the reverse route.

The algorithm works in two phases: at first, it scans the path connecting each sender-receiver pair and assigns the same label to the nodes representing the same routers on different trees. After that, a tree merging operation based on the value of the labels is performed. More formally, let $I_n^{i \to j}$ be the n-th node on the network path connecting node i and node j, and let $N_{i,j}$ the total number of nodes composing such a path; the tree merging algorithm works as follows:

By combining the description of the algorithm and the hypothesis of symmetric routing, it is easy to prove the correctness of the described procedure: two nodes are assigned the same label if and only if they correspond to the same hop of the same end-to-end path and, as a consequence, two nodes which are assigned the same labels always stand for the same physical node. However, it is unfortunately not

Algorithm 4 Tree merging algorithm

```
• For each pair $(i, j)$ of hosts
    – For $n = 1, 2 \dots N_{i,j}$
        * if neither node $I_n^{i\to j}$ nor node $I_{N_{i,j}+1-n}^{j\to i}$ are labeled yet, then both
          nodes are assigned the same label
        * if one of the two nodes $I_n^{i\to j}$ and $I_{N_{i,j}+1-n}^{j\to i}$ has already been labeled,
          the other node is assigned the same label
• Merge labeled trees as follows:
    – Map all nodes with the same label into the same node of the output graph
    – Map all the edges whose ends are assigned the same pair of labels into the
      same edge of the output graph
```

always true that all the nodes corresponding to the same physical device are assigned the same label. In some specific topologies, such as the one shown in figure 1.18, the label assignment algorithm may fail as it may assign different labels to nodes which actually correspond to the same router. In this case, multiple instances of these nodes, which will be referred to as *multiple label nodes*, will be present in the reconstructed global topology. However, we will show in subsection 1.5.3 that the actual impact of such nodes is quite low also in the case of very meshed topologies.

1.5.2 Computational Complexity

In order to give an evaluation of the amount of computation required by our algorithm, we will make some assumptions. In particular, by indicating with N_l the number of leaves in a tree graph, with N_t the total number of nodes in the network, with D the network diameter (which we will assume to be equal to the depth of the reconstructed trees) and with d_m the mean connectivity degree of the trees, the following relations will be assumed:

- $N_l = \log\left(\frac{N_t d_m}{2}\right)$
- $D = \log\left(N_t\right)$

As for the first assumption, it has been proved in [31] that a network can be effectively monitored by using a number of beacons that grows as the logarithm of the total number of edges. The second assumption comes from the small-world properties of the scale-free graphs, which are commonly used to model network topologies [31]. Furthermore, the mean connectivity degree can be considered independent

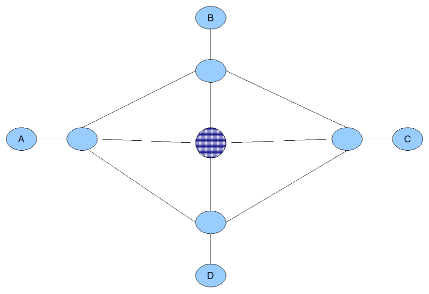


Figure 1.18: Topology which causes the failure of the tree merging algorithm: the dark node is assigned multiple labels.

from the total number of nodes, as a directed consequence of the scale-free property (the degree distribution does not change with the size of the network).

Since the label assignment step of the algorithm consists of inspecting, on each tree, all the paths leading to the roots of the other trees (which, in turn, correspond to the leaves of the root), $N_l(N_l - 1)$ end-to-end paths have to be inspected; we will assume the length of each of these paths to be equal to the network diameter, thus carrying on a worst case analysis.

The second step of the algorithm consists of joining the nodes with the same labels belonging to different trees; we will assume that each node of the network is represented on each tree. This is not true in real cases, since each tree spans a different subset of the network nodes, but, again, we perform a worst case analysis. As a consequence we will assume that $N_t \times N_l$ nodes have to be joined. Since the join operation basically consists in redirecting or deleting every link of a node, we will assume d_m basic steps for such a task.

By summing up all these remarks and by making use of the assumed relations, we can express the number of basic steps required by the algorithm as a function of the total number of the trees and of their mean connectivity degree:

$$\log(N_t) \times \log\left(\frac{N_t d_m}{2}\right) \times \log\left(\frac{N_t d_m}{2}\right) - 1) +$$
$$N_t \times \log\left(\frac{N_t d_m}{2}\right) \times d_m$$

Therefore, since the order of complexity of our algorithm is roughly $N_t * log(N_t)$, it can scale up to very large networks without raising serious computational issues.

1.5.3 Performance Evaluation

All our experiments proved that, under ideal conditions (trees correctly reconstructed by the discovery techniques, no *multiple label nodes*, probe packets crossing every link of the network), our algorithm perfectly reconstructs the network topology. However, in practical cases, an incorrect or incomplete picture of the network can be caused by either the presence of *multiple label nodes* or by the fact that, because of the routing scheme, some links or nodes are not crossed by the active probes and, as a consequence, are not revealed in the trees. As for the second phenomenon, this issue is common to all the active probing based technique: links and nodes which, due to the routing scheme, are not crossed by the probing traffic (the so called *side-links*) cannot be revealed. Such a problem has been analyzed within many probing based topology discovery projects, for example Mercator [32] and Skitter [33]. However, [31] refers that, if the number of beacons is large enough, a good coverage of the network topology can be achieved (over 90% of the links can be discovered).

Instead, in order to evaluate the incidence of *multiple label nodes*, we applied our tree merging algorithm to several random topologies generated by the BRITE generator [20] according to the Barabasi-Albert and Waxman models for router-level topologies. Even if the first model is quite simplistic with respect to actual network topologies, such a generator is a broadly adopted tool, thus providing a good benchmark to test our algorithm. Furthermore, the generated topologies turn out to be very meshed, thus representing a challenging scenario for our solution. For the topologies generated according to the Waxman model, they turn out to be quite tree-like, thus providing an ideal scenario for our algorithm, which always issues a perfect reconstruction. Two visual examples of the topologies which are issued by the two different models are reported in figure 1.19 (Waxman) and 1.20 (Barabasi-Albert).

Hence, we generated topologies through BRITE with different number of nodes and different mean connectivity degrees (m). The results are reported in table 1.5.3 and show that the number of multiple label nodes is low (always less than 10% of the total number of nodes) in ordinary network scenarios.

In order to test the algorithm in a more realistic scenario, we also developed our own topology generator, based on a structural approach described in [34]. Such a tool, which will be soon available online, generates a random topology characterized by three levels of node aggregation; in particular, low connectivity peripheral nodes are connected to intermediate nodes, which, in turn, are attached to an internal mesh of core nodes. Such a structure represents quite well the architecture of common ISP networks, which are made up of Access, Distribution and Core nodes.

We generated several topologies according to this model with sizes spanning from 30 up to 80 nodes. Even in these cases the topology reconstructed by our algorithm showed no *multiple label nodes*.

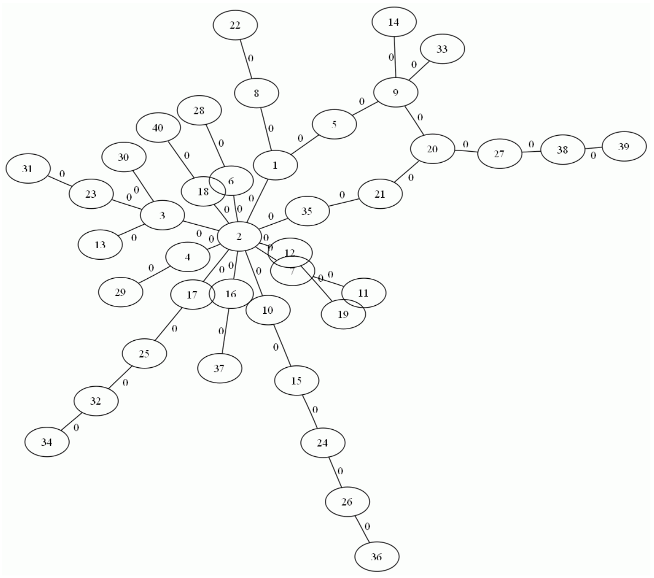


Figure 1.19: A sample topology generated by BRITE (Waxman model).

Barabasi-Albert				Waxman
nodes	m=1.2	m=1.4	m=1.7	m=1.4
60	0	0	7	0
70	1	1	2	0
80	6	1	4	0
90	3	2	4	0
100	3	7	4	0
110	8	8	13	0
120	8	4	8	0

Table 1.6: Number of multiple label nodes generated by our algorithm.

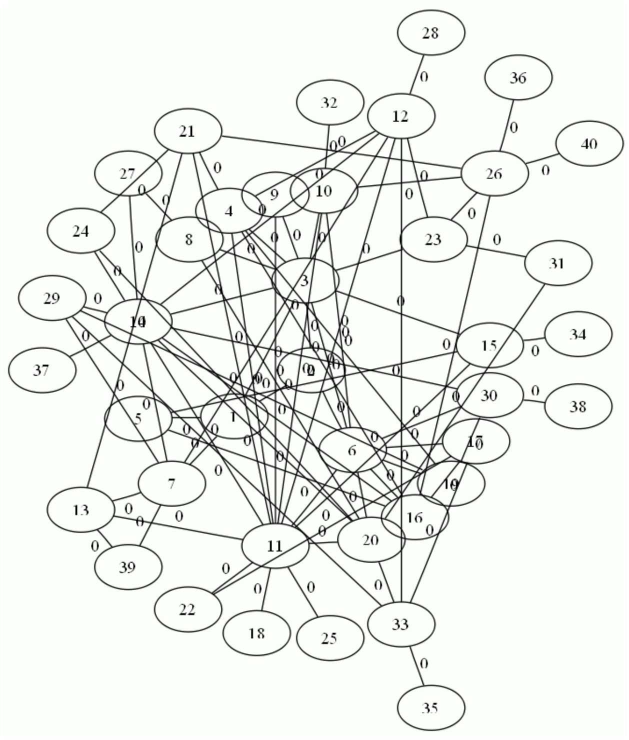


Figure 1.20: A sample topology generated by BRITE (Barabasi-Albert model).

1.6 End–to–End Inference of Link Level Queueing Delay Distribution and Variance

After investigating the topic of tomographic topology discovery, we will address here another application of tomographic techniques: the inference of the distribution of the queueing delay across the network links. Such a knowledge is very useful for delay sensitive applications (such as multimedia services), which can choose a particular server on the basis of the delay associated with the corresponding network path, as well as for network management and traffic engineering issues: a link introducing a heavy queueing delay is likely to be congested, and the network administrator, either manually or by using automated load balancing algorithms, can route incoming traffic flows away from that link. However, even if the distribution of the queueing delay provides the maximum first order statistical information, the simple knowledge of the delay variance itself can be an indicator of the congestion state of a link. In addition, high delay variance itself can be the cause of performance degradation: TCP connections can consider a packet lost even if actually it is only experiencing higher delay and, as a consequence, they reduce their windows and slow down unnecessarily, while multimedia flow quality can be consistently reduced when the associated packets arrive possibly much later than they are supposed to. The focus of this section is to use some of the tomographic tools, taking advantage of the techniques which have been developed to be used in a single-sender/multiple-receivers scenario, to reveal some statistical characteristics (namely distribution and variance) of the queueing delay associated with each link of an end–to–end path. In fact, almost all the proposed tomographic techniques we refer to are intended to infer the delay statistics over a whole network, by making use of active measurements performed by a probe sender node and a set of probe receivers. The network topology is thus modelled as a tree, whose root represents the probe sender and whose leaves stand for the receivers: for each arc of such, a tree a delay distribution is inferred. Unfortunately, none of the proposed techniques allows, in general, the inference of the statistics of the queueing delay on each link of a network path, since an arc of the topology tree is a logical link that can be associated with a multitude of physical links. The delay distribution referring to a logical link can thus be considered as the convolution of the delay distributions referring to different links. In addition, the cooperation of several receiver nodes is not always available, thus reducing the extent of application of such algorithms. On the contrary, the solution that we propose is based on two–points measurements only, thus guaranteeing the maximum flexibility of use.

1.6.1 Tomographic techniques for queueing delay distribution estimation

Most of the algorithms which have been proposed in the literature are based on a discretized delay model: the domain of the possible delay values for each link is partitioned into a finite set of bins (which can be of either variable or fixed length) and the probability of the queueing delay falling within each bin is inferred. The inference of a continuous distribution is therefore transformed into a parameter es-

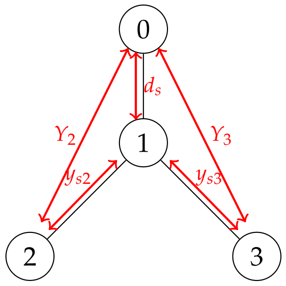


Figure 1.21: End–to–end and logical link level delays experienced by a packet pair probe.

timation problem. However, different kinds of approaches can be also used: paper [35] proposes to model the queueing delay as a linear combination of different continuous distributions and to estimate the coefficients of such a mixture, while paper [36] proposes to estimate the cumulant generating function of the delay distribution. Such an approach takes advantage of the cumulant generating function of the end–to–end delay evaluated as the sum of the cumulant generating functions of the delay distributions over each link: a linear system involving end–to–end measurements and the unknown link level distributions can therefore be written. However, in this section we use a parametric approach. The probes on which the active measurements are based can be multicast packets directed to the whole set of receivers (as proposed in [37]) or couples of unicast packets directed to a receivers pair. However, our interest will be focused on unicast based techniques only, which can be adopted in every network scenario. A unicast probe is composed by a pair of back–to–back packets directed to different receivers; once the packets reach their destination, their one way delay is measured. Such a delay is modelled as the sum of two random variables: the first one (referred to as d_s) represents the delay experienced on the shared path, and is assumed to have the same value for both packets (since the two packets are expected to cross the shared path back–to–back, perfect correlation is hypothesized), while the second one (referred to as y_s) represents the delay experienced on the path between the branching point and the receiver. For each packet, such a delay is assumed to be an independent random variable. More formally, let us consider a pair of probe packets directed to receivers i and j and let us indicate with Y_i and Y_j the corresponding end–to–end delays: the following relation holds

$$\begin{aligned} Y_i &= d_s + y_{si} \\ Y_j &= d_s + y_{sj} \end{aligned} \tag{1.31}$$

The relationship between the quantities involved is graphically represented in figure 1.21.

The vector of samples $\mathbf{Y}_{i,j}$, composed by the measurements pair $\{Y_i, Y_j\}$, is the basic data sample which is used by all unicast based estimation algorithms. The delay distribution inference is based on the knowledge of a certain number (N) of delay measurements $\mathbf{Y}_{i,j}^{(n)}$ for different pairs i, j of receivers. Since the information provided by those samples is insufficient for the purpose of a deterministic inference of the actual experienced delay values (the corresponding linear system would be widely underdetermined), the most common approach to the inference of the delay distribution is based on the Expectation Maximization (EM) algorithm [38].

1.6.2 Tomographic techniques for delay variance estimation

Most of the tomographic solutions concerned with delay variance estimation ([39],[40]) are based on the same probing scheme which has been illustrated in subsection 1.6.1, i.e. on packet pair (or multicast packet) probes which are sent from a single host to several pairs (or sets) of receivers. In particular, referring to (1.31), it can be proven that, for a probe directed to receivers i and j, the following holds:

$$cov(Y_i, Y_j) = \sigma_{d_s}^2 \tag{1.32}$$

As a consequence, the overall delay variance over the shared links crossed by the probe packets can be trivially estimated by calculating the covariance of the corresponding measured end–to–end delays. After calculating an estimate for each couple of receivers, the delay variance associated with each logical link of the sender–based tree can be inferred. Let us consider, as an example, the tree represented in figure 1.22 and apply equation (1.32). The following relations are easily verified:

$$cov(Y_a, Y_b) = \sigma_{l_1}^2 + \sigma_{l_2}^2$$

$$cov(Y_a, Y_c) = \sigma_{l_1}^2$$

The delay variances over the internal logical links can then be computed, while those referring to the logical links which are directly connected to each receiver can be obtained as the difference between the end–to–end delay variance and the delay variance associated with the shared path.

1.6.3 Delay distribution inference over tree–like topologies

Let us first describe the parameter estimation problem which has to be solved by means of the EM algorithm. As previously described, the span of the possible values which can be taken by the queueing delay will be partitioned into N_{bin} bins; the probability of the queuing delay X_k associated with each link k falling within each bin b_i has then to be inferred. More formally, the parameters to be estimated will be $\alpha_i^{(k)} = \mathbb{P}\{X_k \in b_i\} \quad \forall i, k$. The actual queueing delays $\{X_k\}$ are unfortunately impossible to determine through end–to–end measurements: they will then constitute the unobserved data in the EM algorithm. Notice that the measured delay is actually given by the sum of a set of variable queuing delays and of a set of constant delay terms (physical signal propagation delay, packet transmission delay, packet processing delay). Such quantities are of no interest for queuing delay estimation and can usually be compensated for: under the hypothesis that the minimum delay packet for a given path experiences no queueing delay at all, it is possible to remove the fixed delay terms by subtracting it from the other available measurements. Many network tomography algorithms, such as the one used by the widely known tool PathChar [41], rely on such hypothesis. For this reason, all the fixed delays, including any synchronization offset, can be considered irrelevant with respect to the delay estimation algorithm; there is then no need for precise synchronization among sender and receivers in order to measure one-way-delays, provided that the effect of the clock frequency drift among the corresponding hosts can be considered negligible over the time interval employed to perform all the measurements. Several applications of the EM algorithm have been proposed to solve the delay distribution inference problem; in this section we will refer to the recent work of [42], but we point out that all the other existing solutions can be used in our scheme. The algorithm follows the canonical two steps of the EM algorithm. The E-step consists of estimating, under a given value of $\alpha_i^{(k)}$, the unknown data $n_i^{(k)}$, that is the expected number of packets experiencing a queueing delay falling within the $i-th$ bin on the $k-th$ link. After the E-step is performed, the M-step is quite trivial: $\alpha_i^{(k)} = n_i^{(k)}/n^{(k)}$ where $n^{(k)}$ is the total number of packets crossing link k. The estimation of the $n_i^{(k)}$ parameters is the most complex part of the algorithm. Let us suppose a link is crossed by N packet pairs directed to nodes m and l; then

$$n_i^{(k)} = \sum_{n=0}^{N} \mathbb{P}\{X_k \in b_i | \mathbf{Y}_{l,m}^{(n)}\}.$$

In [43], it has been proposed to estimate $\mathbb{P}\{X_k \in b_i|\mathbf{Y}_{l,m}^{(n)}\}$ through an upward–downward algorithm, but the solution we are referring to is based on another approach. Let us consider a two leaves–tree as illustrated in figure 1.21; the following expressions hold:

$$\begin{aligned}\mathbb{P}\{D_s = k|\mathbf{Y}_{2,3}^{(n)}\} &= \frac{\mathbb{P}\{D_s = k, \mathbf{Y}_{2,3}^{(n)}\}}{\mathbb{P}\{\mathbf{Y}_{2,3}^{(n)}\}} \\ &= \frac{\mathbb{P}\{D_s = k, Y_{s2} = Y_2^{(n)} - k, Y_{s3} = Y_3^{(n)} - k\}}{\mathbb{P}\{\mathbf{Y}_{2,3}^{(n)}\}} \\ &= \frac{\mathbb{P}\{D_s = k\} \cdot \mathbb{P}\{Y_{s2} = Y_2^{(n)} - k\} \cdot \mathbb{P}\{Y_{s3} = Y_3^{(n)} - k\}}{\mathbb{P}\{\mathbf{Y}_{2,3}^{(n)}\}}\end{aligned} \tag{1.33}$$

Since the delay distribution on each link on the path is supposed to be known (it has been calculated by the E-step), all of the terms in the previous equation can be directly computed. It is then simple to extend such an approach to any tree topology; let us take as an example the topology shown in figure 1.22: by considering the two–receivers tree ending in A and B, it is possible to infer the distribution of the cumulative queueing delay introduced by links l_1 and l_2, while, with reference to

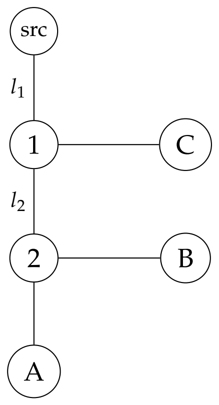


Figure 1.22: Generic single-source multiple-receivers logical tree.

the receivers couple $\{B, C\}$ it is possible to estimate the queueing delay distribution for link l_1. Finally, by deconvolving the two estimated distributions it is then possible to also obtain the distribution of the delay associated with link l_2.

1.6.4 Link level delay distribution inference

The solutions described in the previous subsections are intended to provide estimates of the queueing delay distributions in a single–sender/multiple–receivers measurement scenario; however, in many cases, only two-points measurements are available. An end–to–end path can be modelled as a sequence of links, each of them consisting of two independent queues, one belonging to the forward path and associated to the queueing delay $X_f^{(k)}$ and one to the reverse path and corresponding to the queueing delay $X_r^{(k)}$; such a reference scenario is shown in figure 1.23. In order to extend the existing algorithms to such a scenario, we propose to adopt another kind of probe: such a probe is still a packet pair, but, while the first packet is an unicast packet directed to the host at the other end of the path, the second is a *ping-like* packet (i.e. a packet that forces the receiver to send an immediate response to the sender, such as an *ICMP echo request*) directed to one of the intermediate nodes of the path, as shown in figure 1.23. The two packets are sent back–to–back by the probe sender and, as in the multiple receivers scenario, perfect correlation on the shared links is assumed; from such a probe, a pair of samples $\mathbf{Y}^{(l)}(n) = \{y_o(n), y_p^{(l)}(n)\}$ (with $n \in \{1, N\}$, indicates the $n-th$ pair out of N measurements) is originated, where y_o is the end–to–end delay experienced by the unicast packet directed to the other end of the path and y_p^l is the round trip delay experienced by the $ping-like$ packet directed to the $l-th$ node of the path (which globally consists of N_h links). By considering the path model previously described, the two quantities can be ex-

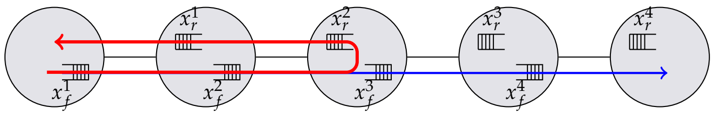


Figure 1.23: Working principle of the packet pair probes used for link-level delay estimation.

pressed as:

$$y_o(n) = \sum_{k=1}^{N_h} x_f^{(k)}(n) \tag{1.34}$$

$$y_p^{(l)}(n) = \sum_{k=1}^{l} x_f^{(k)}(n) + \sum_{k=1}^{l} x_r^{(k)}(n) \tag{1.35}$$

Such expression can be verified by examining figure 1.23, where the behavior of one of the probes is illustrated; the two arrows indicate the path that is crossed by both the $ping - like$ packet and the one way packet, respectively.

It can be therefore noticed that such a delay model is analogous to that of the packet pair case: in particular, referring to (1.31), it can be verified that:

$$d_s^{(l)} = \sum_{k=1}^{l} x_f^{(k)}(n)$$

$$y_{si}^{(l)} = \sum_{k=1}^{l} x_r^{(k)}(n)$$

$$y_{sj}^{(l)} = \sum_{k=l+1}^{N_h} x_f^{(k)}(n)$$

It is thus possible to map the whole end–to–end path into a virtual logical tree, as represented in figure 1.24. The right-side branches of such a tree, which connect one of the intermediate nodes to one of the out–of–path nodes are logical links that correspond to the whole reverse path from the intermediate node to the receiver. The out–of–path nodes, in turn, do not correspond to different network devices, but can be seen as multiple instances of the probe sender. Notice that, in our assumption, the stochastic queueing delay terms $y_{si}^{(l)} \quad \forall l < N_h$ are considered to be statistically independent even if the sets of links they correspond to are partially overlapping; such an assumption is justified by the fact that the measurements referring to different probes are taken at sufficiently spaced time instants. More formally, for sufficiently spaced probes, it can be assumed that $x^{(k)}(n)$ and $x^{(k)}(m)$ are realizations of independent identically distributed random variables for each $k < N_h$ and for each $n \neq m$. Under such an assumption, the mathematical model underlying the problem is for-

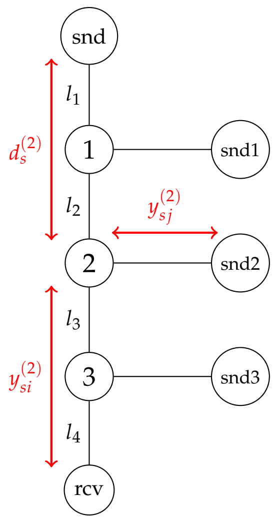


Figure 1.24: Virtual logic tree corresponding to an end–to–end path.

mally equivalent to that of the single–source/multiple–receivers case. Such a result is the main result of our work since, by introducing the concept of virtual logical tree, it allows to infer link level statistics by using the same tomographic techniques proposed for a whole network. In particular for each value of l, the distributions of the three cumulative random variables $D_s^{(l)}, Y_{si}^{(l)}, Y_{sj}^{(l)}$ can be estimated by applying the proceedure described in subsection 1.6.3 and based on the measured delay pairs $\mathbf{Y}^{(l)}(n)$. In order to obtain an estimate of the queueing delay distribution over link k, it is then sufficient to deconvolve the estimated distributions of $D_s^{(k)}$ and $D_s^{(k-1)}$.

1.6.5 Numerical issues about deconvolution of probability distributions

The recursive nature of EM method requires the calculation of several deconvolutions involving estimated delay distributions. Since, especially during the first iterations of the EM algorithm, the estimated distributions can be significantly different from the actual ones, such deconvolutions are not guaranteed to produce a sequence of positive values. In particular, numerical deconvolution of two estimated queueing delay distributions could even result in a sequence with non negligible negative values that, in turn, could cause convergence problems during the following iteration of the EM algorithm. The classical recursive deconvolution method can lead to huge error propagation and is therefore not suitable for our application. In order to reduce error propagation, an effective way of performing deconvolution is to operate in the frequency domain, by calculating the ratio of the DFTs of both operands. Nevertheless, this approach may not produce a non–negative real sequence. Since the numerical deconvolution of two sequences can be performed by solving an associated linear system, the use of one of the existing algorithm for linear system solution with positivity constraints seems to be a suitable solution. Paper [42] suggests the use of the Non Negative Least Squares (NNLS) algorithm. Such an iterative algorithm yields a positive coefficients vector which has the least Euclidean distance from the exact solution of the system. However, the NNLS algorithm, in some cases, yields a vector filled with very small values, which can result in huge convergence problems for the whole algorithm (even a forced normalization of the probability distribution can be problematic, since it leads to divisions by a very small number). For this reason, we prefer the adoption of the Fully Constrained Least Squares (FCLS) algorithm in order to force the output vector to be normalized and positive. Although computationally more expensive, such an algorithm generally guarantees a smooth convergence of the whole delay estimation method, as we verified through simulation campaigns.

1.6.6 Link level delay variance estimation

The variance estimation techniques described in subsection 1.6.4 can be applied to the virtual logical tree associated with the end–to–end path, since their probing scheme, as already pointed out, is the same of that of the delay distribution estimation algorithms. In particular, by combining equations (1.34),(1.35),(1.32), the following relation holds for $l \in \{1, N_h - 1\}$:

$$cov(Y_o, Y_p^{(l)}) = \sum_{i=1}^{l} \sigma^2_{X_f^{(i)}} \tag{1.36}$$

In addition, it is straightforward to note that:

$$\sigma^2_{Y_o} = \sum_{i=1}^{N_h} \sigma^2_{X_f^{(i)}} \tag{1.37}$$

Since the left side term of both equations can be estimated from the available measurements, a triangular and always determined system in the unknowns $\sigma^2_{X_f^{(1)}} \ldots \sigma^2_{X_f^{(N_h)}}$ can be written down. The solution of such a system is shown below:

$$\sigma^2_{X_f^{(1)}} = cov(Y_o, Y_p^{(1)})$$

$$\sigma^2_{X_f^{(l)}} = cov(Y_o, Y_p^{(l)}) - cov(Y_o, Y_p^{(l-1)}) \quad \forall l \in \{2, N_h - 1\}$$

$$\sigma^2_{X_f^{(N_h)}} = \sigma^2_{Y_o} - cov(Y_o, Y_p^{(l)})$$

It is worth noticing that, in the expression of each unknown term, only two measured quantities are involved; as a consequence, the variance of each estimate is, in any case, equal to the sum of the two variances associated with the measured quantities. Therefore, no error accumulation issues arise and, furthermore, a wrong measurement can only affect two variance estimates. In addition, for both delay variance and delay covariance, unbiased and efficient estimators are available: as a consequence, the variance of the quantities to be estimated can arbitrarily reduced by using more samples.

1.6.7 Simulation results: delay distribution estimation

We assess the performance of our algorithm by performing two series of simulations; the first one is model–based, and uses random variables generators available under MATLAB, while in the second one we simulate the realistic scenario of a network path by using the ns2 simulator [7]. It is worth noticing that the performance of our algorithm is related to that of the EM algorithm which is adopted; in all cases we use the algorithm described in 1.6.3, but, as already mentioned, several others variations are available. However, the focus of this section is not the EM algorithm itself, but rather its application to link level delay estimation. In figure 1.25 we show the histograms of the delay distributions as inferred by our algorithm, plotted against the real distributions which are Exponential. In both cases, our network scenario is composed by a path made up of four links, each of them associated with a forward and a reverse queueing delay with different distributions.

The performance of the algorithm appears to be quite good, and it is possible to detect the most congested links of the network. In the ns2 simulations, we set up a scenario composed by 4 links, each of them loaded with TCP cross traffic at a different rate; the TCP cross traffic crossing each link is generated by several connections, each of them characterized by a different segment size, in order to reproduce the multimodal distribution of packet lengths over the real Internet. Such a cross traffic is generated by out–of–path nodes and directed to different intermediate nodes of the path; the link connecting each out–of–path node with a node belonging to the path represents the bottleneck link of the TCP connections: by varying its capacity it is then possible to accurately tune the amount of TCP cross traffic loading each link of the path. In order to verify the correctness of our estimates, the actual distributions of the queueing delays is measured by analyzing the ns2 traces referring to each queue of the network. At first, we assume a path composed by links characterized by the same capacity (10 Mbps) and loaded with different amounts of cross traffic. Again, the results shown in figure 1.26 report the estimated queueing delay distribution plotted against the actual one; in spite of a few detection failures, it can be noticed that our algorithm correctly locates the most congested links, and even reveals the multimodal nature of the delay distribution on the last queue. In a second scenario, we assume links of different capacities, (10, 4, 4 and 3 Mbps, respectively) and, again, different amounts of cross traffic. The results are presented in figure 1.27 and show that the estimates provided by our algorithm allow to correctly reveal the last link of the path as the most congested link.

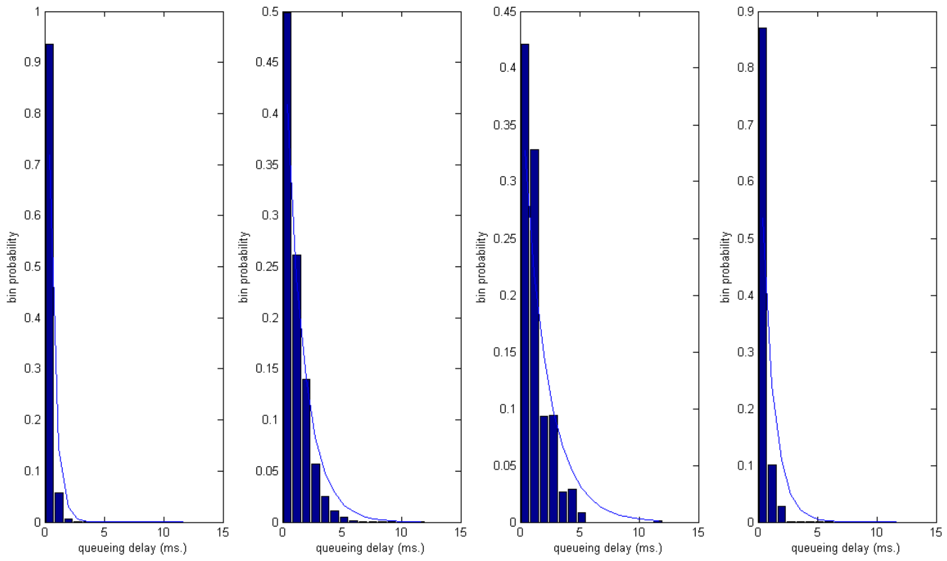


Figure 1.25: Estimated and real queueing delay distributions on a four links network path with exponentially distributed simulated delay.

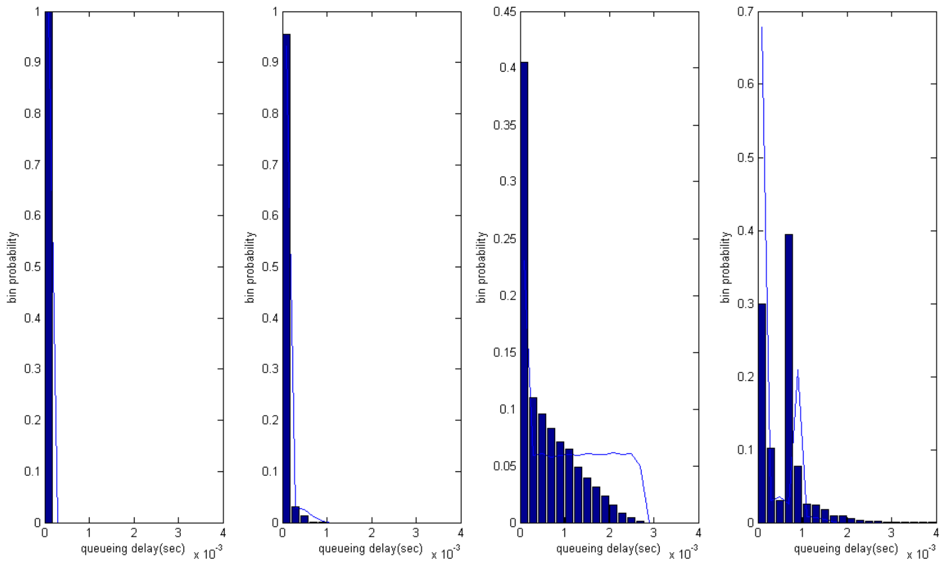


Figure 1.26: Estimated and real queueing delay distributions on a four links network path simulated using ns2.

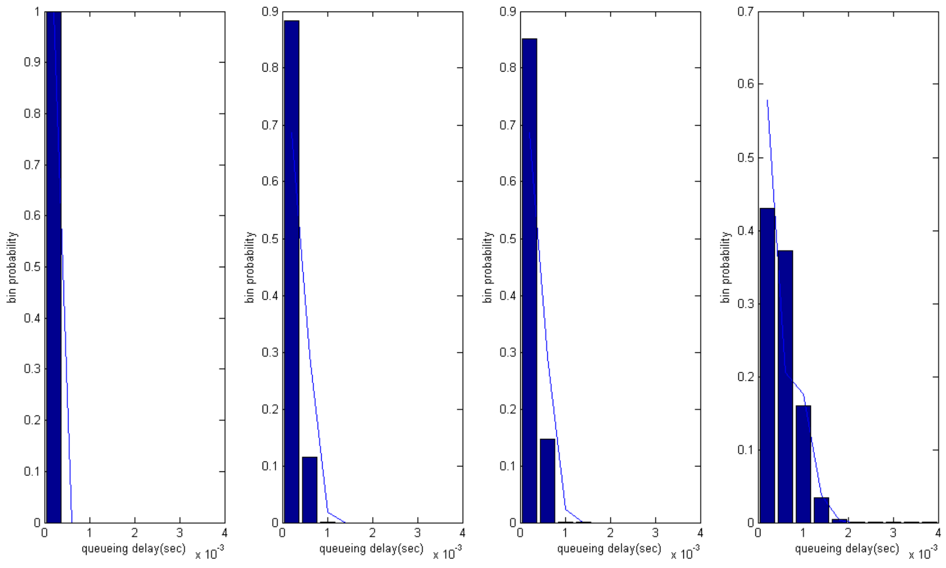


Figure 1.27: Estimated and real queueing delay distributions on a four links network path simulated using ns2.

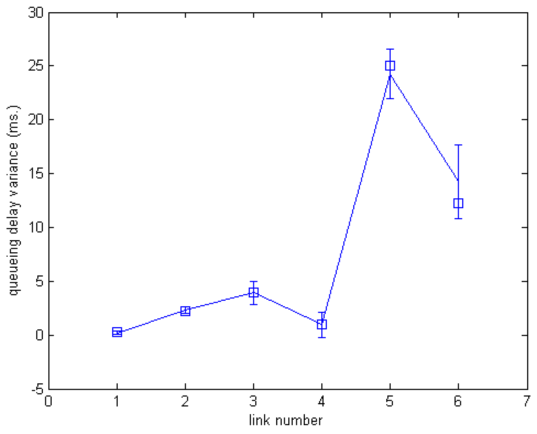


Figure 1.28: Confidence intervals of the estimated delay variances corresponding to each link of a 6 hop network path in the case of exponentially distributed delay. The squares correspond with the actual variance value.

1.6.8 Simulation results: delay variance estimation

We evaluate the performance of our delay variance estimation method by using the same approach of the previous subsection: again, we perform both model-based simulations and ns2 based simulations. In the first case, we take advantage of the high scalability of the variance estimation algorithm and we assess its performance in a larger network scenario, composed by 6 links. We test our algorithm both for Exponential and Erlang distributed delay and, in each case, we perform 20 simulations and evaluate the 95% confidence interval for the estimated variance corresponding to each link. The results are shown in figures 1.28 and 1.29. Simulations show our algorithm to provide quite reliable estimates of the delay variance.

We also perform a set of ns2 based simulations in order to test our algorithm in more realistic conditions. The simulated network scenario consists of a network path composed by 5 hops and crossed by TCP cross traffic; the generation mode of such traffic is the same which has been described in the previous subsection. The 95% confidence intervals of the corresponding estimates are plotted in figure 1.30. Even in this case, it is worth noticing that our algorithm provides fairly good estimates, with a generally very small variation interval. By examining figure 1.30, it would be straightforward, in an hypothetical troubleshooting application, to locate the link(s) that is(are) causing performance degradation.

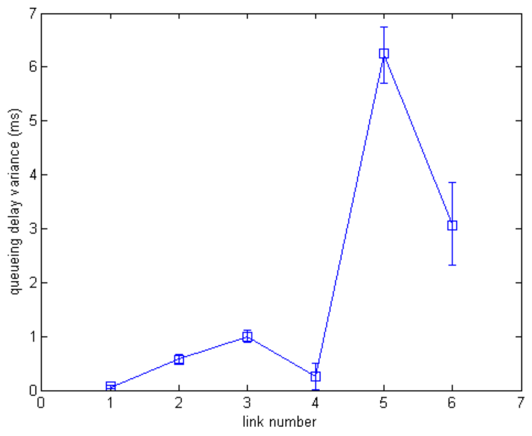


Figure 1.29: Confidence intervals of the estimated delay variances corresponding to each link of a 6 hop network path in the case of Erlang distributed delay. The squares correspond to the actual variance value.

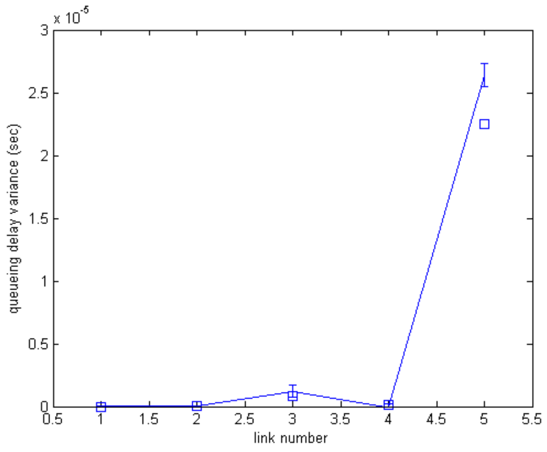


Figure 1.30: Confidence intervals of the estimated delay variances corresponding to each link of a 5 hop network path crossed by TCP traffic and simulated over ns2. The squares correspond to the actual variance values.

1.7 End–to–End Inference of Link Level Queueing Delay Statistic through cumulant estimation

The algorithm that we proposed in the previous section allows, in general, the inference of the statistics of the queueing delay on each physical link of a network path, since arcs of the topology tree are logical links that can be associated with a multitude of physical links. The delay distribution referring to a logical link can thus be considered as the convolution of the delay distributions referring to different physical links. In this section, we address the same problem by taking a completely different approach. In particular, we estimate the cumulants of the queueing delay distribution as a simple linear combination of the cumulants of the available data set (which, in turn, can be estimated without bias by means of the well known *k-statistics*). The estimator is then unbiased and of low complexity (the coefficients of the linear combination are fixed and can be easily pre-computed offline). The only limitation of our approach is the impossibility of calculating the first order cumulants of the queueing delays, since, in that case, the problem is intrinsically undetermined. However, we propose here an approach that allows to estimate such a statistic for the most congested links, which are the most relevant for traffic engineering and troubleshooting purposes. Indeed, the most common approach to the inference of the delay distribution is based on the Expectation Maximization (EM) algorithm [38]. This way of modeling is somehow similar to that proposed in the previous section. However, our preceeding techniques showed several limitations, both from the point of view of numerical stability and in terms of processing complexity; therefore, the approache that we will present is both more reliable and less complex.

1.7.1 Link level delay distribution inference

In this section, we will hypothesize to use a packet pair probe and rely on the assumptions that we already illustrated in subsection 1.6.1.

In general, an end–to–end path can be modelled as a sequence of links, each of them consisting of two independent queues, the one belonging to the forward path and associated with the queueing delay $X_f^{(k)}$ (k is the index of the node) and the one associated with the reverse path and corresponding to the queueing delay $X_r^{(k)}$; such a reference scenario is shown in figure 1.23. In this scenario, our main goal is estimating the cumulants of the delay distributions associated with each link of the path.

The $r-th$ cumulant of random variable X is defined as:

$$K_X^r = \left.\frac{\partial^r G_X(t)}{\partial t^r}\right|_{t=0} \tag{1.38}$$

where $G_X(t)$ is the cumulant generating function associated with X, defined as

$$G_X(t) = \log(\mathbb{E}(e^{tx})) \tag{1.39}$$

The reasons for the choice of such statistics are manyfold:

- the cumulants are linear with respect to the sum: the k-th cumulant of the sum of independent random variables is equal to the sum of the k-th cumulants of each random variable;
- unbiased estimators of the cumulants of a distribution are available through the use of *k*-statistics;
- the moments of a random variable can be exactly calculated based on the knowledge of its cumulants;
- the distribution of a random variable can be approximated based on the knowledge of the cumulants by means of either the saddlepoint method or the Edgeworth series.

Another interesting property of the cumulants, which will be crucial to the development of our technique, is:

$$K^r_{\alpha X} = \alpha^r K^r_X \tag{1.40}$$

which can be trivially derivated from (1.38).

We probe the end-to-end path by using a proper packet pair, whose property is already described in the previous section. Such a probe is a packet pair, but, while the first packet is a unicast packet directed to the host at the other end of the path, the second is a *ping-like* packet (i.e. a packet that forces the receiver to send an immediate response to the sender, such as an *ICMP echo request*) directed to one of the intermediate nodes of the path, as shown in figure 1.23. The two packets are sent back–to–back by the probe sender and, as in the multiple receivers scenario, perfect correlation on the shared links is assumed; the generic n-th probe ($1 \le n \le N$) originates a pair of samples $\mathbf{Y}^{(l)}(n) = \{y_o(n), y_p^{(l)}(n)\}$, where y_o is the end–to–end delay experienced by the unicast packet directed to the other end of the path and $y_p^{(l)}$ is the round trip delay experienced by the $ping - like$ packet directed to the $l - th$ node of the path. By considering the path model previously described and by assuming the whole path consists of N_h links, the two quantities can be expressed as:

$$y_o(n) = \sum_{k=1}^{N_h} x_f^{(k)}(n) \tag{1.41}$$

$$y_p^{(l)}(n) = \sum_{k=1}^{l} x_f^{(k)}(n) + \sum_{k=1}^{l} x_r^{(k)}(n) \tag{1.42}$$

Such expressions can be verified by examining figure 1.23, where the behavior of one of the probes is illustrated; the two arrows indicate the path that is traversed by both the $ping - like$ packet and the one way packet, respectively. As we are intersted in the queueing delay only, a minimum filtering over the observed data is preliminarly performed in order to compensate for constant delay terms (e.g. transmission and propagation latencies).

Since the goal of our technique is to estimate the $r-th$ cumulants $K^r_{X^{(k)}_f}, K^r_{X^{(k)}_r}\,\forall\, k \in [0, N_{hop}]$ for any arbitrary order r, we can now write down a linear system which allows to compute such cumulants from the cumulants of the measurements which can be obtained through packet pair probing. A first set of equations can be easily obtained by re-writing (1.41) and (1.35) in terms of cumulants and by taking advantage of their linear property. Thus, the following equations hold:

$$\sum_{k=1}^{N_h} K^r_{x_f^{(k)}} = K^r_{y_o} \tag{1.43}$$

$$\sum_{k=1}^{l} K^r_{x_f^{(k)}} + \sum_{k=1}^{l} K^r_{x_r^{(k)}} = K^r_{y_p^{(l)}}\,\forall\, l \in [1, N_h] \tag{1.44}$$

The two relations above provide N_h+1 linearly independent equations; therefore, for the system to be solved, N_h-1 additional independent relations are needed. In order to obtain them, let us consider the sum $S^(l)$ of the measured delays experienced by the two packets of each packet pair (i.e. the sum of the round–trip–time experienced by the ping-like packet directed to node l and the end–to–end delay experienced by the one-way packet). By combining (1.41) and (1.42), the following relation can be obtained:

$$S^{(l)}(n) = 2\sum_{k=1}^{l} x_f^{(k)}(n) + \sum_{k=1}^{l} x_r^{(k)}(n) + \sum_{k=l+1}^{N_h} x_f^{(k)}(n) \tag{1.45}$$

By expressing the relation above in terms of cumulants (to this end, let us take advantage of (1.40)), we can obtain N_h-1 equations of the form ($\forall\, l \in [1, N_h-1]$):

$$2^r \sum_{k=1}^{l} K^r_{x_f^{(k)}} + \sum_{k=1}^{l} K^r_{x_r^{(k)}} + \sum_{k=l+1}^{N} K^r_{x_f^{(k)}} = K^r_{S^{(l)}} \tag{1.46}$$

The overall linear system, for each cumulant order r, can then be written as:

$$\mathbf{H}^{(r)}\,\mathbf{X}^{(r)} = \mathbf{Y}^{(r)} \tag{1.47}$$

where $\mathbf{X}^{(r)}$ is the unknowns' vector:

$$\mathbf{X}^{(r)} = \left(X_f^{(1)}, \ldots, X_f^{(N_h)}, X_r^{(1)}, \ldots, X_r^{(N_h)}\right)^T$$

and the matrix $H^{(r)}$ is of the form:

$$\mathbf{H}^{(r)} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 1 & 0 & 0 & 0 & \dots \\ 1 & 1 & 0 & 0 & \dots & 1 & 1 & 0 & 0 & \dots \\ 1 & 1 & 1 & 0 & \dots & 1 & 1 & 1 & 0 & \dots \\ 1 & 1 & 1 & 1 & \dots & 1 & 1 & 1 & 1 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \dots \\ 2^r & 1 & 1 & 1 & \dots & 1 & 1 & 0 & 0 & \dots \\ 2^r & 2^r & 1 & 1 & \dots & 1 & 1 & 0 & 0 & \dots \\ 2^r & 2^r & 2^r & 1 & \dots & 1 & 1 & 1 & 0 & \dots \\ 2^r & 2^r & 2^r & 2^r & \dots & 1 & 1 & 1 & 1 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

The vector of the known terms, in turn, is of the form:

$$\mathbf{Y}^{(r)} = \left(K^r_{Y_p^{(1)}}, \dots, K^r_{Y_p^{(N_h)}}, K^r_{Y_o}, K^r_{S_p^{(1)}}, \dots, K^r_{S_p^{(N_h-1)}} \right)^T$$

All of the terms in such a vector can be estimated from the data-set obtained by packet-pair probing, by means of non-biased estimators. It is easy to prove that $H^{(r)}$ is non-singular for each cumulant order $r \geq 2$: as a consequence, our technique allows to obtain an unbiased estimator for each of the cumulants of the link-level delay distributions except for the first order cumulant. Since the first order cumulant corresponds to the mean of a random variable, this conclusion is not surprising. Indeed, due to the linearity of the expectation operator, the problem of evaluating the average delay associated with the internal links based on the end–to–end measurements is formally equivalent to computing the value of each delay realization $X^{(l)}_{r,f}(m)$ which is, obviously, an undetermined problem. In the following subsection we will illustrate a heuristic method that allows to estimate the first order cumulant at least for the most congested links. A final remark on the estimators' robustness is necessary: in the present work we do not actually provide theoretical confidence bounds for our estimates. However, we point out that the estimates yelded by our method are computed as linear combinations of the cumulants of the measured delays; the coefficients of such combinations only depend on the number of links and are therefore known a priori and the confidence ranges of the cumulants of experimental data-sets can be expressed analytically. Though fairly complex, theoretical confidence bounds can therefore be derived, but, for lack of space, we decide to omit such a discussion here and to postpone it to a follow-up work.

1.7.2 Heuristic solution

As proved in the previous subsection, calculating the mean value of the link level delays based on end-to-end measurements only is an underdetermined problem. However, it is possible to extract from the available data-set some useful information concerning, at least, the most congested links. To this end, we will rely on the following hypothesis: in real networks there are few links where a packet experiences a heavy queueing delay, while the queueing delay on most links is often negligible; we will assume the mean delay associated to such links to be zero. If the links with negligible queueing delay could be located, the number of unknowns of the linear system could be reduced and hopefully, it would be possible to calculate the mean delay associated with the most congested links. Those lightly loaded links can be revealed based on the other cumulants: since their associated delay distributions can be approximated as a Dirac delta function, all of their estimated cumulants will be negligible. Once a lightly loaded link has been located, its associated column can be set to zero within the $\mathbf{H}^{(r)}$ matrix. Such a procedure can be iterated until the rank of the system reaches the number of non-negligible delay terms; when such condition is satisfied, the resulting overdetermined system (the number of equations is the same as that of the orginal system, but the number of unknowns has been reduced) can be solved by using, for example, a least squares approach. Such a technique would not work in the case of a link associated with a nearly constant but non-zero queueing delay (the distribution would in that case be modeled as a translated delta, whose translation factor could not be estimated throughthe cumulants). However, in this case, such a delay would be included in the constant delay slack which is compensated for during the preliminary minimum filtering phase and it would not be possible, in any case, to recover it from the available data-set. Of course, the number of delay terms that can be estimated depends on the specific delay distribution: deletion of a column in the system matrix can cause the overall rank either to be decremented or not, depending on the values in the specific column.

1.7.3 Experimental results

In order to assess the accuracy of our estimation algorithm, we performed two separate evaluation trials, one focused on the heuristic we described in the previous subsection and one focused on the estimation of higher order cumulants by means of the system described in subsection 1.7.1. As for the assessment of the cumulant estimation accuracy, we ran simulations by using both Matlab generated random variables and Network Simulator ns2 testbed scenario. In the first case, we simulated a network path composed of 8 hops, each one associated with an exponentially distributed queueing delay. For the sake of clarity, we show the results associated only with the forward links (the estimator of the delay associated with the reverse links has the same mathematical form). The estimated and actual values of the cumulants up to the 6-th order are shown in figure 1.31.

As it appears from the graph, our estimates are in the large majority of cases very close to the actual value of the cumulants. Of course, the estimation error grows with the order of the cumulant; this is due to the fact that the variance of the k-statistics (the unbiased estimators used to retrieve the cumulants of the end-to-end measurement from the data-set) increases with the cumulant order. Therefore, since our estimator is a linear combination of such k-statistics, its variance grows as well, thus originating higher estimation errors. However, because of the consistency property of the k-statistics, their variance can be arbitrarily reduced by using a larger data-set. As for the ns2 simulations, we set up a scenario composed by 8 links, each of them

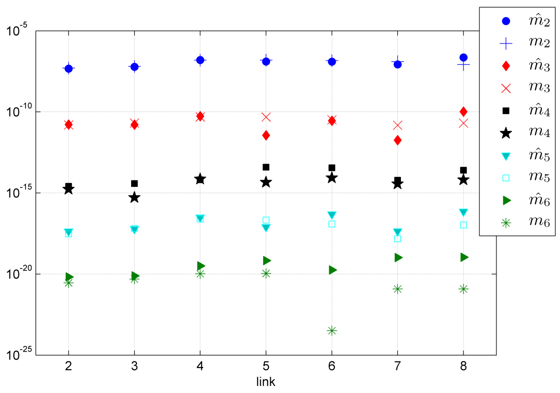


Figure 1.31: Estimated and actual cumulants of link delay distribution in a Matlab simulated model based scenario.

loaded with TCP cross traffic at a different rate; the TCP cross traffic crossing each link is generated by several connections, each of them characterized by a different segment size, in order to reproduce the multimodal distribution of packet lengths over the real Internet. Such a cross traffic is generated by out–of–path nodes and directed to different intermediate nodes of the path; the link connecting each out–of–path node with a node belonging to the path represents the bottleneck link of the TCP connections: by varying its capacity it is then possible to accurately tune the amount of TCP cross traffic loading each link of the path. Notice, however, that no assumption on assumption on the traffic distribution is made.

Again, we show in figure 1.32 the results for the forward links only; the same increase in the estimation error with the order of the cumulants emerges also in this simulation run.

In order to test our technique in a more congested scenario, we increased the traffic load on each queue and repeated the simulation run; the results are shown in figure 1.33 and the estimation accuracy does not appear to be significantly affected by the increased link load.

In order to evaluate the heuristic described in subsection 1.7.2, we relied on model based Matlab simulations. The motivation for this choice is two-fold: first, model based simulation allows for a more strict control of the scenario, second, the calculation of the mean is not affected by the correlation among delays, which, on the contrary, influences the estimation of higher order cumulants. A first run of simulations has been performed by assuming again exponentially distributed delays and by hypothesizing the presence of only 4 congested links (the other links are assumed to be lightly loaded, i.e. associated with a mean queueing delay which is by at least an order of magnitude lower than that of the highly loaded links). The results are shown in table 1.7 and it clearly appears that, even if some errors affect the estimates associated with the less loaded links, the mean delay associated with congested links is generally well estimated.

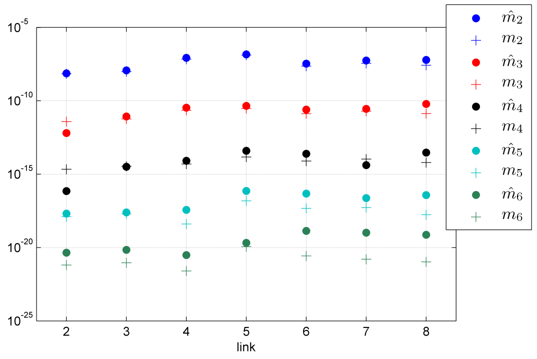


Figure 1.32: Estimated and actual cumulants of link delay distribution in a light-load scenario.

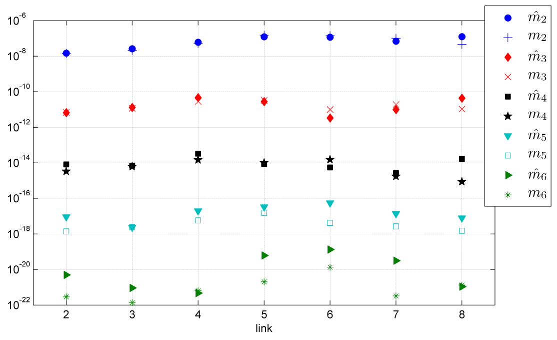


Figure 1.33: Estimated and actual cumulants of link delay distribution in a heavy-load scenario.

Table 1.7: Mean queueing delay estimates in a scenario with exponential queueing delay and 4 congested links

forw. est. μ	forw. actual μ	rev. est. μ	rev. actual μ
0.0036	0.0010	0	0.0020
0	0.0010	0.1009	0.1000
0	0.0010	0.0403	0.0400
0.0421	0.0400	0	0.0010
0	0.0010	0.0197	0.0200
0	0.0030	0.0048	0.0010
0	0.0010	0.0601	0.0600
0.0117	0.0100	0	0.0010

Table 1.8: Mean queueing delay estimates in a scenario with exponential queueing delay and 8 congested links

forw. est. μ	forw. actual μ	rev. est. μ	rev. actual μ
0	0.0010	0.0194	0.0200
0.0341	0.0300	0	0.0010
0	0.0010	0.0380	0.0400
0.0444	0.0400	0	0.0010
0	0.0010	0.0573	0.0600
0.0841	0.0800	0	0.0010
0	0.0010	0.0653	0.0600
0	0.0100	0	0.0010

In a second simulation run we alternated an equal number of heavily loaded links to almost idle links; the results are illustrated in table 1.8 and show our heuristic to correctly locate the congested links and to give a good approximation of their associated mean delay.

Again, the mean delay associated with the most congested links is generally well approximated, while that of the less congested ones is reduced to zero, in order to make the linear problem solvable.

Chapter 2

Algorithms and data structures for high performance network processing

Packet processing on high–speed links is a very challenging task, especially if non–trivial computations have to be performed and a relevant amount of state information is involved. A 10Gb link filled with minimum–sized packets can carry as many as 14 million messages per seconds and, therefore, the time budget for a network monitoring device to process a packet is in the order of tens of clock cycles, thus requiring thorough processing optimization in order to keep up with the data rate. However, most of the time, the bottleneck for this kind of processing is not the processing power itself, but rather the latency involved in fetching data from external memory blocks. Indeed, both general purpose processors and dedicated ones come with a hierarchy of memory blocks, each of them characterized by a different capacity and access latency (usually being inversely proportional to each other). Besides, packet processing applications are usually associated with a huge amount of state which is characterized by a random access pattern: therefore traditional cache managing policies are not effective in avoiding the bottleneck. For this reason, the aim of our work is to design cache friendly algorithms, that allow to segment the data into different structures, the most likely accessed among them hopefully small enough to fit the lower layer caches. In particular, one of the most challenging packet–by–packet operations is deep packet inspection, which is usually carried out by using finite state automata. Such state machines provide an effective way of searching for a set of regular expressions into a byte stream, but, on the other hand, they can require a huge amount of state for storage. Even if techniques for compressing a state machines do exist, they involve an increased number of access to the memory where the machine state is stored. In section 2.1 we propose a compression scheme that, thanks to a local transition cache which can fit into a small memory block, achieves an optimal trade–off between memory consumption and number of memory accesses. In section 2.2 we optimize further such solution by considering two–step transitions. In section 2.3 we propose to adopt homomorphic transformations for compressing the transition table when several input bytes at a time are processed. In section 2.4, instead, we propose a completely different and novel approach to speed up pattern matching: we propose to sample a subset of bytes within the traffic stream to be fed into the finite automaton, while using a traditional state machine for confirmation of an actual matching. Another important data structure for fast packet processing is the Counting Bloom Filter, which can efficiently represent a set of items while allowing both element insertions and removals. Bloom Filters and similar structures are used in a number of network monitoring applications but, despite their good compression level, they can grow too large to fit into small memories. In order to address such an issue, in section 2.5 we propose a further optimization, that partitions a Bloom Filter into several layers, the first of them usually small enoughto fit into small caches. Finally, in section 2.6 we propose a modified version of the Bloom Filter that can be used as a hash table. Notice that some real–world applications of the techniques that we illustrate in this chapter will be shown in the following one (in particular, refer to sections 3.3 and 3.4).

2.1 An Improved DFA construction for fast and efficient regular expression matching

Many important services in current networks are based on payload inspection, in addition to headers processing. Intrusion Detection/Prevention Systems as well as traffic monitoring and layer-7 filtering require an accurate analysis of packet content in search of matching with a predefined data set of patterns. Such patterns characterize specific classes of applications, viruses or protocol definitions, and are continuously updated. Traditionally, the data sets were constituted of a number of signatures to be searched with string matching algorithms, but nowadays regular expressions are used, due to their increased expressiveness and ability to describe a wide variety of payload signatures [44]. They are adopted by well known tools, such as Snort [45] and Bro [46], and in firewalls and devices by different vendors such as Cisco[47]. Typically, finite automata are employed to implement regular expression matching. Nondeterministic FAs (NFAs) are representations which require more state transitions per character, thus having a time complexity for lookup of $O(m)$, where m is the number of states in the NFA; on the other hand, they are very space-efficient structures. Instead, Deterministic FAs (DFAs) require only one state traversal per character, but for the current regular expression sets they need an excessive amount of memory. For these reasons, such solutions do not seem to be proper for implementation in real deep packet inspection devices, which require to perform on line packet processing at high speeds. Therefore, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets [48][49][50][51].

This work focuses in memory savings for DFAs, by introducing a novel compact representation scheme (named δFA) which is based on the observation that, since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only. The δ in δFA just emphasizes that it focuses on the differences between adjacent states. Reducing the redundancy of transitions appears to be very appealing, since the recent general trend in the proposals for compact and fast DFAs construction (see sec.2.1.1) suggests that the information should be moved towards edges rather than states. Our idea comes from D^{2}FA [48], which introduces default transitions (and a "path delay") for this purpose.

Unlike the other proposed algorithms, this scheme examines one state per character only, thus reducing the number of memory accesses and speeding up the overall lookup process. Moreover, it is ortoghonal to several previous algorithms (even the most recent XFAs [51][52][53] and H-cFA [49]), thus allowing for higher compression rates. Finally, a new encoding scheme for states is proposed (which we will refer to as *Char-State compression*), which exploits the association of many states with a few input characters. Such a compression scheme can be efficiently integrated into the δFA algorithm, allowing a further memory reduction with a negligible increase in the state lookup time.

In summary, the main contributions of this work are:

- a novel compact representation of DFA states (δFA) which allows for iterative reduction of the number of states and for faster string matching;

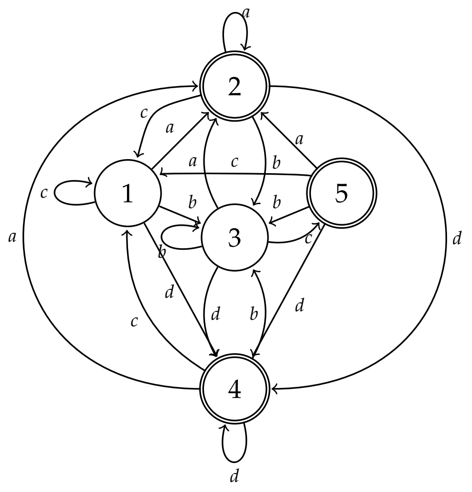
- a new state encoding scheme (*Char-State compression*) based on input characters;

2.1.1 Related Work

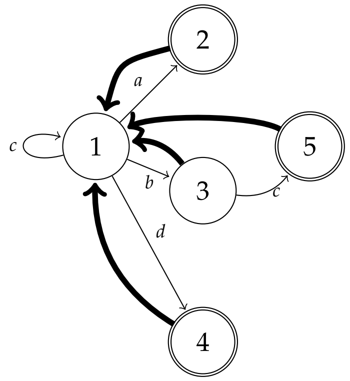
Deep packet inspection consists of processing the entire packet payload and identifying a set of predefined patterns. Many algorithms of standard pattern matching have been proposed [54][55][56], and also several improvements to them. In [57] the authors apply two techniques to Aho-Corasick algorithm to reduce its memory consumption. In details, by borrowing an idea from Eatherton's Tree Bitmap [58], they use a bitmap to compress the space near the root of the state machine, where the nodes are very dense, while path compressed nodes and failure pointers are exploited for the remaining space, where the nodes become long sequential strings with only one next state each. Nowadays, state-of-the-art systems replace string sets with regular expressions, due to their superior expressive power and flexibility, as first shown in [44]. Typically, regular expressions are searched through DFAs, which have appealing features, such as one transition for each character, which means a fixed number of memory accesses. However, it has been proved that DFAs corresponding to a large set of regular expressions can blow up in space, and many recent works have been presented with the aim of reducing their memory footprint. In [59] the authors develop a grouping scheme that can strategically compile a set of regular expressions into several DFAs evaluated by different engines, resulting in a space decrease, while the required memory bandwidth linearly increases with the number of active engines. In [48], Kumar et al. introduce the Delayed Input DFA (D^2FA), a new representation which reduces space requirements, by retrieving an idea illustrated in [60]. Since many states have similar sets of outgoing transitions, redundant transitions can be replaced with a single default one, this way obtaining a reduction of more than 95%. The drawback of this approach is the traversal of multiple states when processing a single input character, which entails a memory bandwidth increase to evaluate regular expressions. To address this issue, Becchi and Crowley [61] introduce an improved yet simplified algorithm (we will call it BEC-CRO) which results in at most $2N$ state traversals when processing a string of length N. This work is based on the observation that all regular expression evaluations begin at a single starting state, and the vast majority of transitions among states lead back either to the starting state or its near neighbors. From this consideration and by leveraging, during automaton construction, the concept of state distance from the starting state, the algorithm achieves comparable levels of compression with respect to D^2FA, with lower provable bounds on memory bandwidth and greater simplicity. Also, the work presented in [62] focuses on the memory problem of DFAs, by proposing a technique that allows non-equivalent states to be merged, thanks to a scheme where the transitions in the DFA are labeled. In particular, the authors merge states with common destinations regardless of the characters which lead those transitions (unlike D^2FA), creating opportunities for more merging and thus achieving higher memory reduction. Moreover the authors regain the idea of bitmaps for compression purposes. Run-Length-Encoding is used in [63] to compress the transition table of DFAs. The authors show how to increase the characters processed per state traversal and present heuristics to reduce the number of memory accesses. Their work is specifically focused on an FPGA implementation. The work in [50] is based on the usual observation that DFAs are infeasible with large sets of regular expressions (especially for those which present wildcards) and that, as an alternative, NFAs alleviate the memory storage problem but lead to a potentially large memory bandwidth requirement. The reason is that multiple NFA states can be active in parallel and each input character can trigger multiple transitions. Therefore the authors propose a hybrid DFA-NFA solution bringing together the strengths of both automata: when constructing the automaton, any nodes that would contribute to state explosion retain an NFA encoding, while the others are transformed into DFA nodes. As shown by the experimental evaluation, the data structure presents a size nearly that of an NFA, but with the predictable and small memory bandwidth requirements of a DFA. Kumar et al. [64] also showed how to increase the speed of D^2FAs by storing more information on the edges. This appears to be a general trend in the literature even if it has been proposed in different ways: in [64] transitions carry data on the next reachable nodes, in [62] edges have different labels, and even in [49] and [51][52] transitions are no more simple pointers but a sort of “instructions”. In a further comprehensive work [49], Kumar et al. analyze three main limitations of the traditional DFAs. First, DFAs do not take advantage of the fact that normal data streams rarely match more than a few initial symbols of any signature; the authors propose to split signatures such that only one portion needs to remain active, while the remaining portions can be “put to sleep” (in an external memory) under normal conditions. Second, the DFAs are extremely inefficient in following multiple partially matching signatures and this yields the so-called *state blow-up*: a new improved Finite State Machine is proposed by the authors in order to solve this problem. The idea is to construct a machine which remembers more information, such as encountering a closure, by storing them in a small and fast cache which represents a sort of history buffer. This class of machines is called History-based Finite Automaton (H-FA) and shows a space reduction close to 95%. Third, DFAs are incapable of keeping track of the occurrencies of certain sub-expressions, thus resulting in a blow-up in the number of state: the authors introduce some extensions to address this issue in the History-based counting Finite Automata (H-cFA). The idea of adding some information to keep the transition history and, consequently, reduced the number of states, has been retrieved also in [51][52], where another scheme, named extended FA (XFA), is proposed. In more details, XFA augments traditional finite automata with a finite scratch memory used to remember various types of information relevant to the progress of signature matching (e.g., counters of characters and other instructions attached to edges and states). The experimental tests performed with a large class of NIDS signatures showed time complexity similar to DFAs and space complexity similar to or better than NFAs.

2.1.2 Delta Finite Automaton

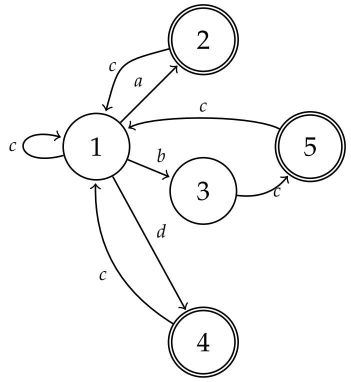
In this section we introduce δFA, a D^2FA-inspired automaton that preserves the advantages of D^2FA and requires a single memory access per input char.



(a) The DFA



(b) The D^{2}FA



(c) The δFA

Figure 2.1: Automata recognizing (a^+),(b^+c) and (c^*d^+).

2.1.2.1 Motivation through an example

In order to make clearer the rationale behind δFA construction and the differences with D^2FA, we start by analyzing the same example brought by Kumar et al. in [48]: the figure 2.1(a) represents a DFA on the alphabet $\{a, b, c, d\}$ that recognizes the regular expressions (a^+),(b^+c) and (c^*d^+).

In figure 2.1(b) the D^2FA for the same set of regular expressions is shown. The main idea is to reduce the memory footprint of states by storing only a limited number of transitions for each state and a default transition to be taken for all input char for which a transition is not defined. When, for example, in figure 2.1(b) the state machine is in state 3 and the input is d, the default transition to state 1 is taken. State 1 "knows" which state to go to upon input d, therefore we jump to state 4. In this example, taking a default transition costs 1 more hop (1 more memory access) for a single input char. However, it may happen that also after taking a default transition, the destination state for the input char is not specified and another default transition must be taken, and so on. The works in [48] and [61] show how we can limit the number of hops in default paths and propose refined algorithms to define the best choice for default paths. In the example, the total number of transitions was reduced to 9 in the D^2FA (less than half of the equivalent DFA which has 20 edges), thus achieving a remarkable compression.

However, observing the graph in fig.2.1(a), it is evident that most transitions for a given input lead to the same state, regardless of the starting state; in particular, adjacent states share the majority of the next-hop states associated with the same input chars. Then if we jump from state 1 to state 2 and we "remember" (in a local memory) the entire transition set of 1, we will already know all the transitions defined in 2 (because for each character they lead to the same set of states as 1). This means that state 2 can be described with a very small amount of bits. Instead, if we jump from state 1 to 3, and the next input char is c, the transition will not be the same as the one that c produces starting from 1; then state 3 will have to specify its transition for c. The result of what we have just described is depicted in fig.2.1(c) (except for the local transition set), which is the δFA equivalent to the DFA in fig.2.1(a). We have 8 edges in the graph (as opposed to the 20 of a full DFA) and every input char requires *a single state traversal* (unlike D^2FA).

2.1.2.2 Definition of our automaton

As shown above, the target of δFA is to obtain a similar compression as D^2FA without giving up the *single state traversal per character* of DFA. The idea of δFA comes from the following observations:

- as shown in [61], most default transitions are directed to states closer to the initial state;
- a state is defined by its transition set and by a small value that represents the accepted rule (if it is an accepting state);
- in a DFA, most transitions for a given input char are directed to the same state.

By elaborating on the last observation, it becomes evident that most adjacent states share a large part of the same transitions. Therefore we can store only the differences between adjacent (or, better, "parent-child"[1]) states.

This requires, however, the introduction of a supplementary structure that locally stores the transition set of the current state. The main idea is to let this local transition set evolve as a new state is reached: if there is no difference with the previous state for a given character, then the corresponding transition defined in the local memory is taken. Otherwise, the transition stored in the state is chosen. In all cases, as a new state is read, the local transition set is updated with all the stored transitions of the state. The δFA shown in fig.2.1(c) only stores the transitions that *must* be defined for each state in the original DFA.

2.1.2.3 Construction

In alg.5 the pseudo-code for creating a δFA from a N-states DFA (for a character set of C elements) is shown. The algorithm works with the *transition table* $t[s, c]$ of the input DFA (i.e.: a $N \times C$ matrix that has a row per state and where the i-th item in a given row stores the state number to reach upon the reading of input char i). The final result is a "compressible" transition table $t_c[s, c]$ that stores, for each state, the transitions required by the δFA only. All the other cells of the $t_c[s, c]$ matrix are filled with the special LOCAL_TX symbol and can be simply eliminated by using a bitmap, as suggested in [57] and [62]. The details of our suggested implementation can be found in section 2.1.6.

The construction requires a step for each transition (C) of each pair of adjacent states ($N \times C$) in the input DFA, thus it costs $O(N \times C^2)$ in terms of time complexity. The space complexity is $O(N \times C)$ because the structure upon which the algorithm works is another $N \times C$ matrix. In details, the construction algorithms first initializes the t_c matrix with EMPTY symbols and then copies the first (root) state of the original DFA in the t_c. It acts as base for subsequently storing the differences between consecutive states.

Then, the algorithm observes the states in the original DFA one at a time. It refers to the observed state as *parent*. Then it checks the *child* states (i.e.: the states reached in 1 transition from parent state). If, for an input char c, the child state stores a different transition than the one associated with any of its parent nodes, we cannot exploit the knowledge we have from the previous state and this transition must be stored in the t_c table. On the other hand, when all of the states that lead to the child state for a given character share the same transition, then we can omit to store that transition. In alg.5 this is done by using the special symbol LOCAL_TX.

Equivalent states After the construction procedure shown in alg.5, since the number of transitions per state is significantly reduced, it may happen that some of the states have the same identical transition set. If we find j identical states, we can simply store one of them, delete the other $j - 1$ and substitute all the references to those with the single state we left. Notice that this operation creates again the opportunity

[1]here the terms *parent* and *child* refer to the depth of adjacent states

Algorithm 5 Pseudo-code for the creation of the transition table t_c of a δFA from the transition table t of a DFA.

```
1:  for $c \leftarrow 1, C$ do
2:      $t_c[1, c] \leftarrow t[1, c]$
3:  end for
4:  for $s \leftarrow 2, N$ do
5:      for $c \leftarrow 1, C$ do
6:          $t_c[s, c] \leftarrow$ EMPTY
7:      end for
8:  end for
9:  for $s_{parent} \leftarrow 1, N$ do
10:     for $c \leftarrow 1, C$ do
11:         $s_{child} \leftarrow t[s_{parent}, c]$
12:         for $y \leftarrow 1, C$ do
13:             if $t[s_{parent}, y] \neq t[s_{child}, y]$ then
14:                 $t_c[s_{child}, y] \leftarrow t[s_{child}, y])$
15:             else
16:                 if $t_c[s_{child}, y] ==$ EMPTY then
17:                     $t_c[s_{child}, y] \leftarrow$ LOCAL_TX
18:                 end if
19:             end if
20:         end for
21:     end for
22: end for
```

for a new state-number reduction, because the substitution of state references makes it more probable for two or more states to share the same transition set. Hence we iterate the process until the number of duplicate states found is 0.

2.1.2.4 Lookup

Algorithm 6 Pseudo-code for the lookup in a δFA. The current state is s and the input char is c.

```
procedure $Lookup(s, c)$
1: read($s$)
2: for $i \leftarrow 1, C$ do
3:     if $t_c[s, i] \neq$ LOCAL_TX then
4:         $t_{loc}[i] \leftarrow t_c[s, i]$
5:     end if
6: end for
7: $s_{next} \leftarrow t_{loc}[c]$
8: return $s_{next}$
```

The lookup in a δFA is computed as shown in alg.6. First, the current state must be read with its whole transition set (step 1). Then it is used to update the local transition set t_{loc}: for each transition defined in the set read from the state, we update the corresponding entry in the local storage. Finally the next state s_{next} is computed by simply observing the proper entry in the local storage t_{loc}. While the need to read the whole transition set may imply more than 1 memory access, we show in sec.2.1.5 how to solve this issue by means of a compression technique we propose. The lookup algorithm requires a maximum of C elementary operations (such as shifts and logic AND or popcounts), one for each entry to update. However, in our experiments, the number of updates per state is around 10. Even if the actual processing delay strictly depends on many factors (such as clock speed and instruction set), in most cases, the computational delay is negligible with respect to the memory access latency.

In fig.2.2 we show the transitions taken by the δFA in fig.2.1(c) on the input string *abc*: a circle represents a state and its internals include a bitmap (as in [57] to indicate which transitions are specified) and the transition set. The bitmap and the transition set have been defined during construction. It is worth noticing that the "duplicate" definition of transitions for character c. We have to specify the c-transition for state 2 even if it is the same as the one defined in state 1, because state 2 can be reached also from state 3 which has a different next state for c. We start ($t = 0$) in state 1 that has a fully-specified transition set. This is copied into the local transition set (below). Then we read the input char a and move ($t = 1$) to state 2 that specifies a single transition toward state 1 on input char c. This is also an accepting state (underlined in figure). Then we read b and move to state 3. Note that the transition to be taken now is not specified within state 2 but it is in our local transition set. Again state 3 has a single transition specified, that this time changes the corresponding one in the local transition set. As we read c we move to state 5 which is again accepting.

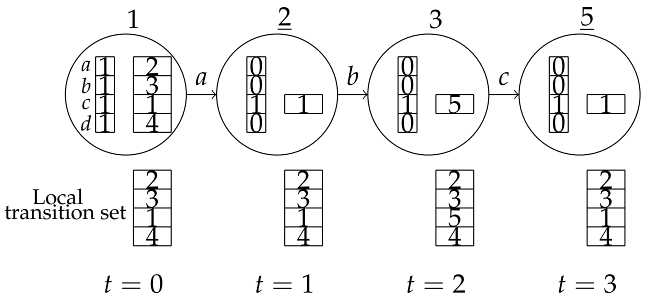
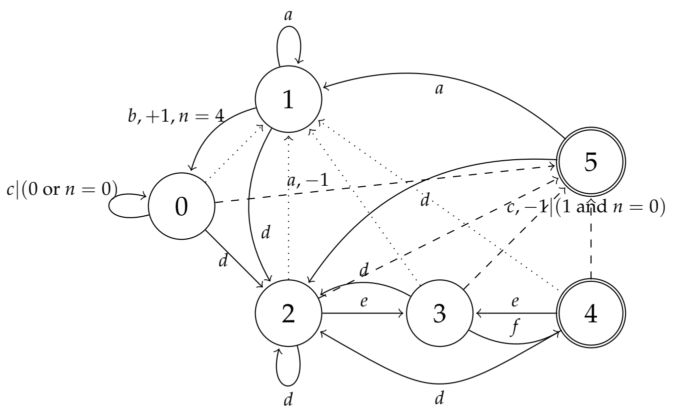


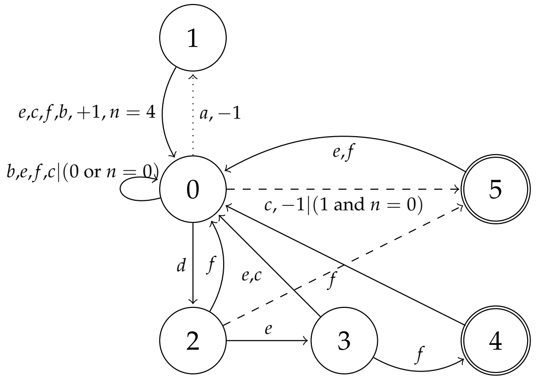
Figure 2.2: δFA internals: a lookup example.

2.1.3 Application to H-cFA and XFA

One of the main advantage of our δFA is that it is orthogonal to many other schemes. Indeed, very recently, two major DFA compressed techniques have been proposed, namely H-cFA [49] and XFA [51][52]. Both these schemes address, in a very similar way, the issue of state blow-up in DFA for multiple regular expressions, thus candidating to be adopted in platforms which provide a limited amount of memory, as network processors, FPGAs or ASICs. The idea behind XFAs and H-cFA is to trace



(a) The H-cFA. Dashed and dotted edges have same labels, respectively $c, -1|(1 \text{ and } n = 0)$ and $a, -1$. Not all edges are shown to keep the figure readable. The real number of transitions is 38.



(b) The δH-cFA. Here all the 18 transitions are shown.

Figure 2.3: Automata recognizing *.*ab [^a]*c* and *.*def*

the traversal of some certain states that corresponds to closures by means of a small scratch-memory. Normally those states would lead to state blow-up; in XFAs and H-cFA flags and counters are shown to significantly reduce the number of states.

The application of δFA to H-cFA and XFA (which is tested in sec.2.1.7) is obtained by storing the "instructions" specified in the edge labels only once per state. Moreover edges are considered different also when their specified "instructions" are different. To better clarify the idea, an example of the application to H-cFA (again taken from a previous paper [49]) is reported in fig.2.3(a). The aim is to recognize the regular expressions *.*ab [^a]*c* and *.*def*, and labels include also conditions and operations that operate on a flag (set/reset with +/-1) and a counter n (for more details refer to [49]). A DFA would need 20 states and a total of 120 transitions, the corresponding H-cFA (fig.2.3(a)) uses 6 states and 38 transitions, while the δFA representation of the H-cFA (fig.2.3(b)) requires only 18 transitions.

2.1.4 Compressing char-state pairs

In a δFA, the size of each state is not fixed because an arbitrary number of transitions can be present, and therefore state pointers are required, which generally are standard memory addresses. They constitute a significant part of the memory occupation associated with the DFA data structure, so we propose here a compression technique which remarkably reduces the number of bits required for each pointer. Such an algorithm is fully compatible with δFA and most of the other solutions for DFA compression already shown in section 2.1.1. Our algorithm (hereafter referred to as *char-state compression* or simply *C-S*) is based on a heuristic which is verified by several standard rule sets: in most cases, the edges reaching a given state are labelled with the same character. Table 2.1 shows, for different available data sets (see section 2.1.7 for more details on sets) the percentage of nodes which are reached only by transitions corresponding to a single character over the total number of nodes.

Data set	p_{1char} (%)	r_{comp} (%)	η_{acc}	T_S (KB)
Snort34	96	59	1.52	27
Cisco30	89	67	1.62	7
Cisco50	83	61	1.52	13
Cisco100	78	59	1.58	36
Bro217	96	80	1.13	11

Table 2.1: Percentage of states reached by edges with the same one label (p_{1char}), *C-S* compression (r_{comp}), average number of scratchpad accesses per lookup (η_{acc}) and indirection-table size (T_S).

As a consequence, a consistent number of states in the DFA can be associated with a single character and can be referred to by using a "relative" address. More precisely, all the states reached by a transition labelled with character c will be given a "relative" identifier (hereafter simply *relative-id*); since the number of such states will be smaller than the number of total states, a relative-id will require a lower

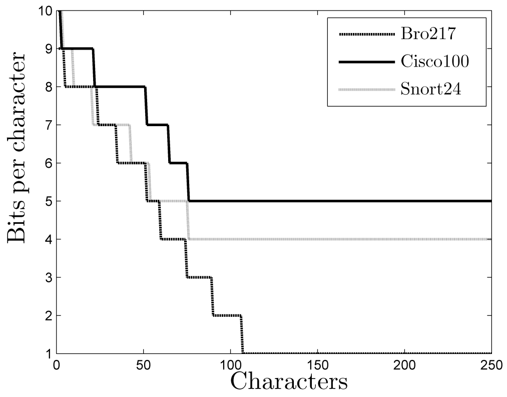


Figure 2.4: Distribution of the number of bits used for a relative identifier with our compression scheme for standard rule sets.

number of bits than an absolute address. In addition, as the next state is selected on the basis of the next input char, only its relative-id has to be included in the state transition set, thus requiring less memory occupation. In a D^{2}FA, where a default transition accounts for several characters, we can simply store it as a relative-id with respect to the first character associated with it. The absolute address of the next state will be retrieved by using a small indirection table, which, as far as our experimental results show, will be small enough to be kept in local (or in a scratchpad) memory, thus allowing for fast lookup. It is clear that such a table will suffer from a certain degree of redundancy: some states will be associated with several relative-ids and their absolute address will be reported more than once. In the next subsection we then propose a method to cope with such a redundancy, in the case it leads to an excessive memory occupation. Figure 2.4 shows the distribution of the number of bits that may be used for a relative-id when applying our compression scheme to standard rule sets. As it can be noticed, next state pointers are represented in most cases with very few bits (less than five); even in the worst case, the number of bits is always below ten. In the second column of table 2.1, we show the compression rate achieved by *C-S* with respect to a naive implementation of DFA for the available data sets. As it appears from the table, the average compression is between 60% and 80%.

2.1.4.1 Indirection Table Compression

As claimed above, the implementation of *Char-State compression* requires a lookup in an indirection table which should be small enough to be kept in local memory. If several states with multiple relative-ids are present in such a table, this might be an issue. For this reason we present a lookup scheme which offers an adaptive trade-off between the average number of memory accesses and the overall memory occupation of the table. The table that we use in our scheme encompasses two kinds of pointers: absolute pointers and local ones. When a state has a unique relative-id, its absolute address is written in the table; otherwise, if it has multiple relative-ids, for each one of them the table reports a pointer to a list of absolute addresses; such a pointer will require a consistently smaller number of bytes than the address itself. An absolute address is then never repeated in the table, thus preventing from excessive memory occupation. Such a scheme is somewhat self-adapting since, if few states have multiple identifiers, most of the translations will require only one memory access, while, if a consistent amount of redundancy is present, the translation will likely require a double indirection, but the memory occupation will be consistently reduced. Notice that the presence of different length elements in the table poses no severe issues: since the relative address is arbitrary, it is sufficient to assign lower addresses to nodes which are accessible with only one lookup and higher addresses to nodes requiring double indirection, and to keep a threshold value in the local memory. The results in terms of memory accesses and size of such a scheme applied to the available data sets are reported in tab.2.1.

2.1.5 Applying C-S to our automaton

The *C-S* can be easily integrated within the δFA scheme and both algorithms can be cross-optimized. Indeed, *C-S* helps δFA by reducing the state size thus allowing the read of a whole transition set in a single memory access on average. On the other hand, *C-S* can take advantage of the same heuristic of δFA: successive states often present the same set of transitions. As a consequence, it is possible to parallelize the retrieval of the data structure corresponding to the next state and the translation of the relative address of the corresponding next-state in a sort of "speculative" approach. More precisely, let s and $s+1$ be two consecutive states and let us define A_s^c as the relative address of the next hop of the transition departing from state s and associated with the character c. According to the previously mentioned heuristic it is likely that $A_s^c = A_{s+1}^c$; since, according to our experimental data (see sec.2.1.7), 90% of the transitions do not change between two consecutive states, we can consider such an assumption to be verified with a probability of roughly 0.9. As a consequence, when character c is processed, it is possible to parallelize two memory accesses:

- retrieve the data structure corresponding to state $s+1$;
- retrieve the absolute address corresponding to A_{s+1}^c in the local indirection table.

In order to roughly evaluate the efficiency of our implementation in terms of the state lookup time, we refer to a common underlying hardware architecture (described in section 2.1.6). It is pretty common [65] that the access to a local memory block to be than twice as faster than that of to an off-chip memory bank: as a consequence, even if a double indirection is required, the address translation will be ready when the data associated with the next state will be available. If, as it is likely, $A_s^c = A_{s+1}^c$, it will be possible to directly access the next state (say $s+2$) through the absolute pointer that has just been retrieved. Otherwise, a further lookup to the local indirection table will be necessary.

Dataset	# of regex	ASCII length range	% Regex w/ wildcards (*,+,?)	Original DFA	
				# of states	# of transitions
Snort24	24	6-70	83.33	13886	3554816
Cisco30	30	4-37	10	1574	402944
Cisco50	50	2-60	10	2828	723968
Cisco100	100	2-60	7	11040	2826240
Bro217	217	5-76	3.08	6533	1672448

Table 2.2: Characteristics of the rule sets used for evaluation.

Such a parallelization can remarkably reduce the mean time needed to examine a new character. As an approximate estimation of the performance improvement, let us suppose that our assumption (i.e. $A_s^c = A_{s+1}^c$) is verified with probability $p = 0.9$, that one access to on-chip memory takes $t_{on} = 4T$ and to an external memory $t_{off} = 10T$ [65], and that an address translations requires $n_{trans} = 1.5$ memory accesses (which is reasonable according to the fourth column of table 2.1). The mean delay will be then:

$$\overline{t_{par}} = (1-p)(t_{off} + n_{trans} \times t_{on}) + p \times t_{off} = 10.6T$$

This means that even with respect to the implementation of δFA the *C-S* scheme increases the lookup time by a limited 6%. On the contrary, the execution of the two tasks serially would required:

$$\overline{t_{ser}} = (t_{off} + n_{trans} \times t_{on}) = 16T$$

The parallelization of tasks results then in a rough 50% speed up gain.

2.1.6 Implementation

The implementation of δFA and *C-S* should be adapted to the particular architecture of the hardware platform. However, some general guidelines for an optimal deployment can be outlined. In the following we will make some general assumptions on the system architecture; such assumptions are satisfied by many network processing devices (e.g. the Intel IXP Network Processors [66]). In particular, we assume our system to be composed by:

- a standard 32 bit processor provided with a fairly small local memory (let us suppose a few KBs); we consider the access time to such a memory block to be of the same order of the execution time of an assembly level instruction (less than ten clock cycles);
- an on-chip fast access memory block (which we will refer to as scratchpad) with higher storage capacity (in the order of 100 KB) and with an access time

of a few dozens of clock cycles;

- an off-chip large memory bank (which we will refer to as external memory) with a storage capacity of dozens of MBs and with an access time in the order of hundreds of clock cycles.

We consider both δFA and *Char-State compression* algorithms. As for the former, two main kinds of data structures are needed: a unique local transition set and a set of data structures representing each state (kept in the external memory). The local transition set is an array of 256 pointers (one per character) which refer to the external memory location of the data structure associated with the next state for that input char; since, as reported in table 2.3(b) , the memory occupation of a δFA is generally smaller than 1 MB, it is possible to use a 20 bit-long offset with respect to a given memory address instead of an actual pointer, thus achieving a consistent compression. A δFA state is, on the contrary, stored as a variable-length structure. In its most general form, it is composed by a 256 bit-long bitmap (specifying which valid transition are already stored in the local transition set and which ones are instead stored within the state) and a list of the pointers for the specified transitions, which, again, can be considered as 20 bit offset values. If the number of specified transitions within a state is small enough, the use of a fixed size bitmap is not optimal: in these cases, it is possible to use a more compact structure, composed by a plain list of character-pointer couples. Note that this solution allows for memory saving when less than 32 transitions have to be updated in the local table. Since in a state data structure a pointer is associated with a unique character, in order to integrate *Char-State compression* in this scheme it is sufficient to substitute each absolute pointer with a relative-id. The only additional structure consists of a character-length correspondence list, where the length of the relative-ids associated with each character is stored; such an information is necessary to parse the pointer lists in the node and in the local transition set. However, since the maximum length for the identifiers is generally lower than 16 bits (as it is evident from figure 2.4), 4 bits for each character are sufficient. The memory footprint of the character-length table is well compensated by the corresponding compression of the local transition set, composed by short relative identifiers (our experimental results show a compression of more than 50%). Furthermore, if a double indirection scheme for the translation of relative-ids is adopted, a table indicating the number of unique identifiers for each character (the threshold value we mentioned in section 2.1.4.1) will be necessary, in order to parse the indirection table. This last table (that will be at most as big as the compressed local transition table) can be kept in local memory, thus not affecting the performance of the algorithm.

2.1.7 Experimental Results

This subsection shows a performance comparison among our algorithm and the original DFA, D^{2}FA and BEC-CRO. The experimental evaluation has been performed on some data sets of the Snort and Bro intrusion detection systems and Cisco security appliances [47]. In details, such data sets, presenting up to hundreds of regular expressions, have been randomly reduced in order to obtain a reasonable amount of

(a) Transitions reduction (%). For δFA also the percentage of duplicate states is reported.

Dataset	D^2FA					BEC-CRO	δFA	
	$DB = \infty$	$DB = 14$	$DB = 10$	$DB = 6$	$DB = 2$		trans.	dup.states
Snort24	98.92	98.92	98.91	98.48	89.59	98.71	96.33	0
Cisco30	98.84	98.84	98.83	97.81	79.35	98.79	90.84	7.12
Cisco50	98.76	98.76	98.76	97.39	76.26	98.67	84.11	1.1
Cisco100	99.11	99.11	98.93	97.67	74.65	98.96	85.66	11.75
Bro217	99.41	99.40	99.07	97.90	76.49	99.33	93.82	11.99

(b) Memory compression (%).

Dataset	D^2FA					BEC-CRO	δFA + C-S
	$DB = \infty$	$DB = 14$	$DB = 10$	$DB = 6$	$DB = 2$		
Snort24	95.97	95.97	95.94	94.70	67.17	95.36	95.02
Cisco30	97.20	97.20	97.18	95.21	55.50	97.11	91.07
Cisco50	97.18	97.18	97.18	94.23	51.06	97.01	87.23
Cisco100	97.93	97.93	97.63	95.46	51.38	97.58	89.05
Bro217	98.37	98.34	95.88	95.69	53	98.23	92.79

Table 2.3: Compression of the different algorithms in terms of transitions and memory.

memory for DFAs and to observe different statistical properties. Such characteristics are summarized in table 2.2, where we list, for each data set, the number of rules, the ascii length range and the percentage of rules including "wildcards symbols" (i.e. *, +, ?). Moreover, the table shows the number of states and transitions and the amount of memory for a standard DFA which recognizes such data sets, as well as the percentage of duplicated states. The choice of such data sets aims to mimic the size (in terms of DFA states and regular expressions) of other sets used in literature [62][48][50],[61] in order to obtain fair comparisons.

Tables 2.3 illustrate the memory compression achieved by the different algorithms. We have implemented the code for our algorithm, while the code for D^2FA and BEC-CRO is the *regex-tool* [67] from Michela Becchi (for the D^2FA the code runs with different values of the diameter bound, namely the diameter of the equivalent maximum weight spanning tree found in the space reduction graph [48]; this parameter affects the structure size and the average number of state-traversals per character). By means of these tools, we build a standard DFA and then reduce states and transitions through the different algorithms. The compression in tab. 2.3(a) is simply expressed as the ratio between the number of deleted transitions and the original ones (previously reported in tab.2.2) , while in 2.3(b) it is expressed considering the overall memory saving, therefore taking into account the different state sizes and the additional structures as well. Note also, in the last column of tab.2.3(a) , the limited but effective state-reduction due to the increased similarity of states obtained by the δFA (as described in sec.2.1.2.3). Although the main purpose of our work is to reduce the time complexity of regular expression matching, our algorithm achieves also a degree of compression comparable to that of D^2FA and BEC-CRO, as shown by tab.2.3. Moreover, we remark that our solution is orthogonal to these algorithms (see sec.2.1.3), thus allowing further reduction by combining them.

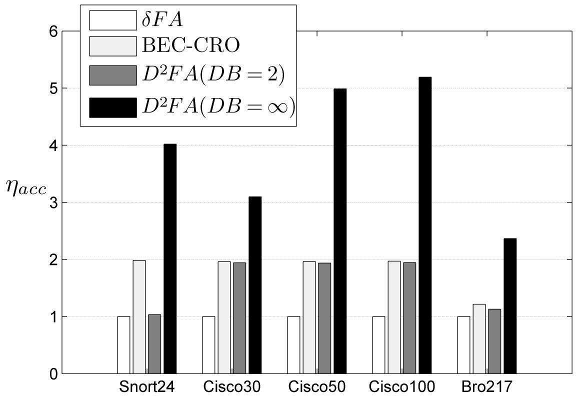


Figure 2.5: Mean number of memory accesses for δFA, BEC-CRO and D^{2}FA for different datasets.

Figure 2.5 shows the average number of memory accesses (η_{acc}) required to perform pattern matching through the compared algorithms. It is worth noticing that, while the integration of *C-S* into δFA (as described in sec.2.1.5) reduces the average state size, thus allowing for reading a whole state in slightly more than $1 (< 1.05)$ memory accesses, the other algorithms require more accesses, thus increasing the lookup time. We point out that the mean number of accesses for the integration of δFA and *C-S* is not included in the graph in that *C-S* requires accesses to a local scratchpad memory, while the accesses the figure refers to are generally directed to an external, slower memory block; therefore it is difficult to quantify the additional delay introduced by *C-S*. However, as already underlined in section 2.1.5, if an appropriate parallelization scheme is adopted, the mean delay contribution of *C-S* can be considered nearly negligible on most architectures.

Dataset	# of states	# of trans. XFA	# of trans. δXFA	Compr. %
c2663-2	14	3584	318	92
s2442-6	12	3061	345	74.5
s820-10	23	5888	344	94.88
s9620-1	19	4869	366	92.70

Table 2.4: Number of transitions and memory compression by applying δFA+*C-S* to XFA.

Finally, table 2.4 reports the results we obtained by applying δFA and *C-S* to one

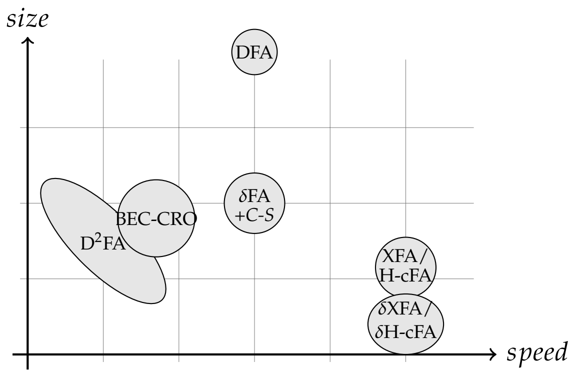


Figure 2.6: Comparison of speed performance and space requirements for the different algorithms.

of the most promising approach for regular expression matching: XFAs [51][52] (thus obtaining a δXFA). The data set (courtesy of Randy Smith) is composed of single regular expressions with a number of closures that would lead to a state blow-up. The XFA representation limits the number of states (as shown in the table). By adopting δFA and *C-S* we can also reduce the number of transitions with respect to XFAs and hence achieve a further size reduction. In details, the reduction achieved is more than 90% (except for a single case) in terms of number of transitions, that corresponds to a rough 90% memory compression (last column in the table). The memory requirements, both for XFAs and δXFAs, are obtained by storing the "instructions" specified in the edge labels only once per state. Figure 2.6 resumes all the evaluations by mixing speed performance (in terms of memory accesses) and space requirements in a qualitative graph (proportions are not to be considered real). It is evident that our solution almost achieves the compression of D^{2}FA and BEC-CRO, while it proves higher speed (as that of DFA). Moreover, by combining our scheme with other ones, a general performance increase is obtained, as shown by the integration with XFA or H-cFA.

2.2 Second order delta enconding to improve DFA efficiency

In the previous section, we have introduced a compact representation scheme (named δFA) which is based on the observation that, since most adjacent states share several common transitions, it is possible to delete most of them by taking into account the different ones only. In this section, we present a novel automaton which takes advantage of the ideas of δFA and adds the concept of "temporary transition". It extends the δFA main assumption some step further: while δFA specifies the transition set of a state with respect to its direct parents, the adoption of 2-step "ancestors" (in this definition a direct parent is a 1-step ancestor) increases the chances of compression. As we will show in the following, the best approach to exploit this second order dependence is to define the transitions of the states between the ancestors and the child as "temporary". This, however, introduces a new problem during the construction process: the optimal construction (in terms of memory or transition reduction) appears to be an NP-complete problem. Therefore, a direct and oblivious approach is chosen for simplicity. Results (on real rule-sets from Snort, Bro and Cisco devices) show that our simple approach do not differ significantly from the optimal (if ever reachable) construction. Since the technique we propose is an extension to δFA that exploits second order dependence, we name this scheme δ^2FA.

2.2.1 The Main idea

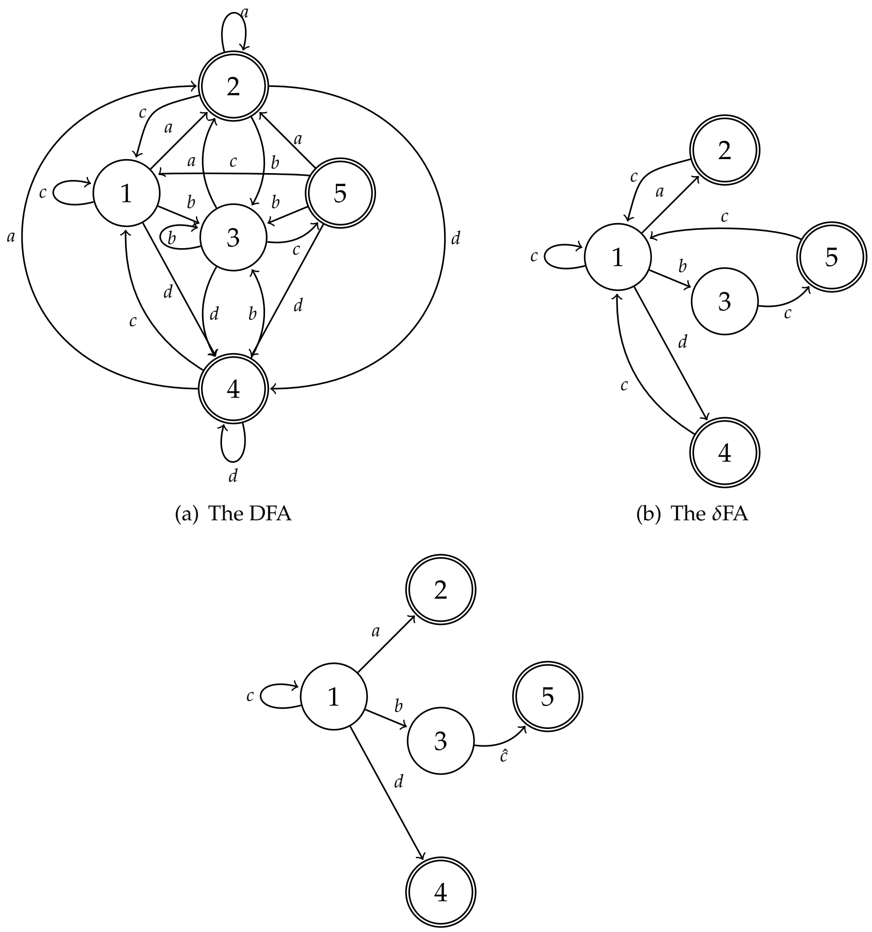
Consider again the DFA in 2.7(a). Although the δFA in fig. 2.7(b) shows a remarkable saving in terms of transitions with respect to the standard DFA, its main assumption (all parents must share the same transition for a given character) somewhat limits the effectiveness of the compression. In the example, all the transitions for character c are specified (and hence stored) for all the 5 states, because of a single state 3 that defines a different transition (the transition for c is directed to state 1 for states $1, 2, 4$ and 5, while 3 defines an edge to 5). Notice that this is due to the strict definition of δFA rules that do not "see" further than a single-hop: the transition set of a state is stored as the difference with respect to all its direct parents.

Intuitively, just as a D^{2}FA with long default-transitions paths compresses better than a bounded D^{2}FA with B=2 [48], by relaxing the definition of "parents" to "grandparents" (i.e., 2-step neighbor nodes) the effectiveness of the δFA approach increases because of the larger number of possibilities.

However, a blind adoption of this concept does not provide better results in δFA: for instance, in fig. 2.7(b) defining the transitions for c as difference with respect to all the "grandparents" still would not allow to eliminate any new transition. Moreover this scheme would require to store 2 local transition sets (doubling the local memory needed).

A better approach is, instead, to define the transition for c in state 3 as "temporary", in the sense that it does not get stored in the local transition set. In this way, we force the transition to be defined uniquely within state 3 and not to affect its children. This means that, whenever we read state 3, the transition for c in the local transition set is not updated, but it remains as it was in its parents. Then, we can avoid storing the transitions for c in states $2, 4$ and 5, as shown in fig. 2.7(c) where the temporary transition is signaled with $\hat{c}$.

By defining temporary transitions, we effectively exploit 2-nd order relationships among states in a simple way, without incurring in the need for 2-times larger local memories.


(a) The DFA

(b) The δFA

(c) The δ^2FA

Figure 2.7: Automata recognizing (a^+), (b^+c) and (c^*d^+).

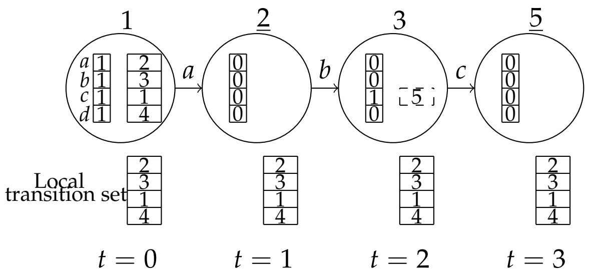


Figure 2.8: δ^2FA internals: a lookup example.

2.2.1.1 Lookup

The lookup in a δ^2FA differs very slightly from that of δFA. The only difference concerns the way we handle temporary transitions: temporary transitions are valid within their state but they are not stored in the local transition set. Fig. 2.8 shows also an example of the lookup process for a δ^2FA: the whole transition set of state 1 (where we start at time $t = 0$) is copied into the local transition set. Then by char a, we move ($t = 1$) to state 2 which does not specify any transition. When we read b ($t = 2$), we move to state 3, where a temporary transition (dashed box) is specified: this transition is valid only within state 3. Finally ($t = 3$) we read c, take the temporary transition, and end up in state 5.

2.2.1.2 Construction

The construction process of the δ^2FA requires the corresponding δFA to be constructed beforehand and used as input. Then, the process works by recognizing subsets of nodes where a transition for a given character can be defined as temporary. In fig. 2.9, nodes are shown as divided into sets according to their parent-child relationships (highlighted by the bold arrows) and their transitions (for a given character). In particular, all nodes with the same transition for a given char x share the same color: sets S_1, S_2 and S_4 all provide the same transition for char x, while S_3 defines a different next state for x. If we set all the transitions for x in S_3 as temporary, we can avoid storing the transition for x in S_1.

In a real implementation, in order to recognize the nodes where a transition for a given character can be defined as temporary, for each char x of each state s, if the corresponding transition $t[s, x]$ in the δFA is stored (i.e., it is different from that $t[p, x]$ of all its parents) the following steps are required:

- a search is performed in all the children of s: whenever at least a child has the same transition $t[p, x]$ of its "grandparents", the second step follows;
- check all the other parents (except for s) of such a subset of children in order to check if they have the same transition $t[p, x]$;

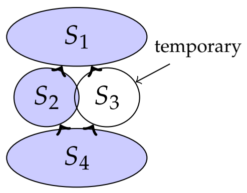


Figure 2.9: Schematic view of the problem. Same color means same properties. If the properties of S_3 are set temporary, the ones in S_1 can be avoided.

- in this case, the transition $t[s, x]$ in s can be set as temporary and the process ends.

The process is also described in alg. 7 where, for the sake of readability, we adopt the same notation of fig. 2.9.

Algorithm 7 Pseudo-code for the creation of the transition table t_2 of a δ^2FA from the transition table t of a δFA.

```
$t_2 \leftarrow t$
for all state $s$ in $\delta$FA do
    for all char $c$ do
        if $t[s,c] \neq$ LOCAL_TX then
            $S_4 \leftarrow \{$ parents of $s\}$
            if $t[s_j,c]$ $\forall s_j \in S_4$ are equal and specified then
                $S_1 \leftarrow \{$ children of $s\}$
                if $\exists s_j \in S_1$ s.t $t[s_j,c] ==$ LOCAL_TX then
                    break
                end if
                if $\exists s_j \in S_1$ s.t $t[s_j,c] == t[S_4,c]$ then
                    $S_2 \leftarrow \{$ parents of $s_j\} \setminus s$
                    if $t[S_2,c] == t[s_j,c] == t[S_4,c] \neq t[s,c]$ then
                        $t_2[s,c] \leftarrow TEMP\_TX$
                        delete $t_2[s_j,c]$
                    end if
                end if
            end if
        end if
    end for
end for
```

A few remarks (which ultimately result in constraints in the construction process) can be explained by referring to fig. 2.9 (where the transitions for x in S_3 are set temporary):

1. no state in S_4 can have a temporary transition for x. The reason is simple: a temporary transition for x in the parents S_4 means that such a transition does not modify the local transition table and therefore we have no way to “remember” the next-state when (after some hops) we reach the children S_1;

Dataset	Cisco30	Cisco50	Snort24	Snort31	Bro217
Del. ratio	97%	89%	100%	99%	99%
Temp. ratio	84%	76%	100%	98%	99%

Table 2.5: Simple vs. Optimal approach: ratio of deleted and temporary transitions.

2. all children states in S_1 must have specified transitions for x, because if the transitions in S_3 are temporary and an un-specified transition exists in a state $s_j \in S_1$, the ultimate result is that $t[s_j, x] = t[S_4, x]$ while s_j was meant to inherit $t[S_3, x]$.

Hence, this process introduces some constraints and, as usual when dealing with constraints on graphs, this creates new problems: as described above, when setting a subset y of transitions as temporary, we must rely on some other transitions (the granparents of y) to be non-temporary. This can be classified as a graph-coloring problem which is known to be NP-hard.

Because of this severe problem, we adopt a straight and oblivious construction: we construct the δ^2FA in a single run by observing all the transitions and setting all the transitions that satisfy the above-mentioned constraints as temporary. This solution is very fast because it does not explore the whole solution domain and simply gives up the idea of optimality. While this may appear unusual and is certainly non-optimal, it is however motivated by a number of experimental results (reported in the following section), where this approach does not differ significantly from the optimal setting (if ever reachable) in terms of transitions reduction. Moreover, notice that the optimal construction would require an exhaustive search of all the solution domain, thus questioning the advantages of the optimal setting.

2.2.2 Experimental Results

In this subsection we report the experimental results of our proposed technique (δ^2FA) applied to real-world regular expression-set from IDS/IPSs such as Snort and BRO and from Cisco security devices [47]. As a first set of results, in order to motivate the simplistic approach to the construction of δ^2FA, we compare the best (if ever reachable) construction and the simple approach we adopt. Since the ultimate goal of this work is to come up with an efficient way to further reduce the number of transitions to store in a δFA, the comparison is expressed in terms of deleted transitions. The results in tab. 2.5 show the ratio between the number of deleted (and temporary) transitions of our simple approach and the maximum number of deleted (and temporary) transitions we may have in the optimal setting. The latter is computed by accepting the violation of the two constraints described in the previous section. Hence, in this sense, this optimal value is actually a bound. The values in the table suggest the simple approach is effective and provides very good results, reaching the maximum number of deleted transitions in almost all the cases.

Tab. 2.6 shows a performance comparison among δFA and δ^2FA (which include

(a) Transitions reduction (%).

Dataset	D^{2}FA		BEC-CRO	δFA	δ^2FA
	$DB = \infty$	$DB = 2$			
Snort24	98.92	89.59	98.71	96.33	96.82
Cisco30	98.84	79.35	98.79	90.84	92.01
Cisco50	98.76	76.26	98.67	84.11	86.11
Cisco100	99.11	74.65	98.96	85.66	86.90
Bro217	99.41	76.49	99.33	93.82	94.30

(b) Memory compression (%).

Dataset	D^{2}FA		BEC-CRO	δFA	δ^2FA
	$DB = \infty$	$DB = 2$			
Snort24	95.97	67.17	95.36	95.02	95.90
Cisco30	97.20	55.50	97.11	91.07	92.65
Cisco50	97.18	51.06	97.01	87.23	89.03
Cisco100	97.93	51.38	97.58	89.05	90.3
Bro217	98.37	53	98.23	92.79	93.4

Table 2.6: Compression of the different algorithms. In (b) the results for δFA and δ^2FA include char-state compression.

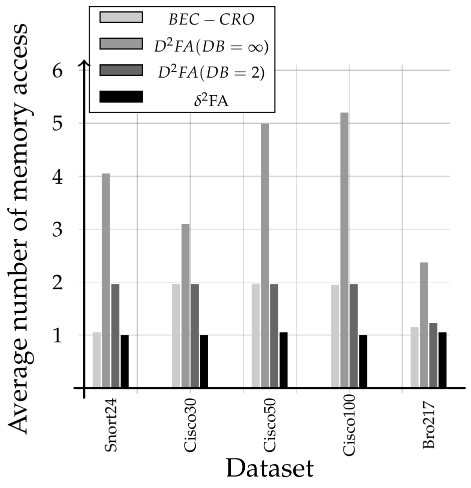


Figure 2.10: Mean number of memory accesses.

also the *Char-State* encoding scheme for further memory compression, as explained in 2.1.4) and the most efficient previous solutions. For D^{2}FA and BEC-CRO, we use the code of *regex-tool* [67], which builds a standard DFA and then reduces states and transitions through the different algorithms. In particular, for the D^{2}FA the code runs with two different values of the bound B (i.e., 2 and ∞), which is a parameter that affects the structure size and the average number of state-traversals per character [48]. The compression in tab. II(a) is simply expressed as the ratio between the number of deleted transitions and the original ones, while in tab. II(b) it is expressed by considering the overall memory consumption, therefore taking into account the different state sizes and the additional structures as well. Our algorithms achieve a degree of compression comparable to that of D^{2}FA and BEC-CRO, while allowing for a higher lookup speed by preserving one transition per character. This is the main strength of our scheme, which allows for reducing lookup time by exploiting the adoption of wide memory accesses which are very common in DRAMs. As shown by results, δ^2FA provides an improvement with respect to δFA at practically no cose, since it requires a minimal change in the lookup algorithm. Finally since our solutions are orthogonal to previous algorithms, a further reduction is possible by combining them. Fig. 2.10 shows the average number of memory accesses required to perform pattern matching through the compared algorithms. It is worth noticing that, while δ^2FA (just as δFA) needs about < 1.05 accesses (more than 1 because of the integration with the *Char-State* scheme), the other algorithms require more accesses, thus increasing the lookup time.

2.3 Homomorphic encoding of DFAs

In this section we propose a solution to increase the speed of regular expression searching techniques by multiplying the amount of bytes processed per cycle while also reducing memory requirements. Indeed, very few works have explored this possibility. The main reason is that, when processing k bytes per step, 256^k transitions per state are needed, so even observing only 2 bytes per cycle would require each DFA state to define 65536 transitions. Of course, the amount of states reachable in one-hop from a given state is not that large, on the contrary it is limited and concentrated on its average. Such fact is exploited in this work in order to define a simple and effective way to build small-sized and fast DFAs that process k bytes per step. This involves the definition and application of an homomorphism [68], hence we name our DFA representation Homomorphic-DFA (h-DFA).

2.3.1 Related works

The current trend in research and industry is to use DFAs to represent regular expressions, in order to obtain higher performance, while trying to solve their problems in terms of memory requirements. Many recent works have been presented with the aim of reducing their memory footprint. For a complete survey of these works, please refer to section 2.1.1. This work focuses on speed as main issue for current regular expression searching techniques. Very few works have explored the possibility to increase searching–speed in DFAs. Basically, the idea of these previous works is to multiply the amount of bytes processed per cycle, thus working with 2, 3 or 4-byte strides. However, even observing only 2 bytes per cycle would require each DFA state to include 2^{16} transitions. To solve this problem, the authors of [63] suggest a solution by observing that in actual FAs the number of different transitions (even when k bytes are processed) is more limited. In particular, they propose the use of Equivalent Character Identifiers defining the set of input words (strides of k bytes) which produce transitions to the same next state. Moreover, Run Length Encoding is used to encode the transition table. Such an approach is not general and presents some limitations, as highlighted by [69]. Indeed, it is not feasible in contexts where big DFAs (more than 100 states) and/or large compressed alphabets are involved. Therefore, the authors of [69] try to make a k-DFA feasible by taking advantage of alphabet-reduction and default transition compression. The use of alphabet-reduction, as well as in [63], is justified by the fact that, when the number of processed bytes increases, the automaton actually uses only a small subset of the entire alphabet. Instead, the default transition compression acts by removing the transitions redundancy present in a DFA. Indeed, if the stride doubles, the number of transitions in the DFA increases quadratically, but the number of states does not; therefore, intuitively, the fraction of distinct transitions decreases and the transition redundancy tends to increase.

2.3.2 An efficient representation for DFAs

In the following we introduce the basics of our scheme. We want to succintly describe the outgoing transitions for each state, so that, when computing the corresponding k-step DFA, we have to combine a small amount of one-step transitions.

Our main idea is to group all the symbols that produce a transition to a given node into a subset and find a series of functions that, only when applied to such a subset, provides a specific result or a set of results. When applied to all the other symbols, the result must be different. More formally, in each state, for each subset of symbols S_j that produces a transition to a node n_j, we look for a function $h_j(c)$ such that

$$h_j(c) = x_j \in \begin{cases} X_j & \forall c \in S_j \\ U \setminus X_j & \forall c \notin S_j \end{cases} \tag{2.1}$$

where U is the image of $h_j(c)$ and X_j is the subset of the image of $h_j(c)$ for $c \in S_j$.

By means of this series of functions h_j, we can describe the transition set of each node as an array of tuples:

$$(h_1 : x_{1,1}, \ldots, x_{1,N_1} : n_1) \ldots (h_d : x_{d,1}, \ldots, x_{d,N_d} : n_d) \tag{2.2}$$

where d is the state outdegree, $n_1 \ldots n_d$ are the reachable states, $x_{k,1} \ldots x_{k,N_k}$ are the different values that h_k takes in S_k or, in other words, a representation of X_k and N_k is the cardinality of X_k.

Such a representation helps reducing the redundancy of DFAs as regular Alpha-

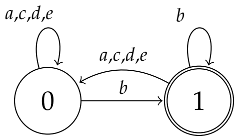


Figure 2.11: A very simple DFA

bet Compression Tables: it requires to store $\sum_k N_k$ values, d functions and d pointers to next states. As an example of the compactness of such representation, let us observe the transition set of the DFA state 1 in fig.2.11. In this case, the characters a,c,d and e all belong to a subset S_0 that produces a transition to state 0. Therefore we can describe the transition set with two tuples only:

$$\{h_0{:}X_0{:}0\}, \quad \{h_1{:}X_1{:}1\}$$

where h_0 and h_1 are defined as in (2.1): when h_0 is applied to a,c,d and e, the result is in X_0, while $h_1(b) \in X_1$.

By defining a set of functions to a DFA, we exploit the properties of inverse homomorphisms applied to DFAs. An homomorphism [68] is an application that maps symbols to strings belonging to a language L. An inverse homomorphisms translate strings of a language L into symbols belonging to a given alphabet. From our point of view, by grouping all our functions h_j into a function H^{-1} such that

$$H^{-1}(c) = h_j(c) \quad \forall c \in X_j$$

we define an inverse homomorphism (the exponent emphasizes it is an *inverse* homomorphism). On the other hand, by means of the representation in tuples we apply an homomorphism H to a DFA. The composition of the two is, of course, the original DFA.

2.3.3 The look for an effective Homomorphism

In order to find a description for $H^{-1}(c)$, we test the following possible "bit-friendly" definition for $h_j(x)$:

1. $h_j(x) = (p_j \times x + q_j) \bmod m_j$
2. $h_j(x) = (p_j \text{ AND } x) \bmod m_j$
3. $h_j(x) = \text{popcount}(x \bmod m_j)$

These possible definitions are applied (with parameters p_j, q_j, m_j varying from 1 to 256 because x itself is a byte) to DFAs that recognize real data-sets (the ones shown in sec.2.3.6).

In each test, we start by looking for a function that provides a single result in a given subdomain S_j; if none is found, we look for 2 results and so on. Once we find such a function, we follow the "definition" of $h_j(x)$: we check if it outputs the same value inside and outside a subset S_j, that is we check for the following condition:

$$\{x_i = h_j(c_i) : \forall c_i \notin S_j\} \bigcap \{x_j = h_j(c_j) : \forall c_j \in S_j\} \overset{?}{=} \emptyset \tag{2.3}$$

If the condition is not verified (the intersection is not empty), we drop the function and change the parameter again, as described by the pseudocode in algorithm 8. The algorithm can either finish the computation because it finds a good function (i.e.: the return value is FOUND) or fail. The failure happens if no combination of the parameters $\{p_j, q_j, m_j\}$ produces a function h_j whose image set X_j has less than C_{max} elements.

Algorithm 8 Pseudocode for the search of function $h_j(x)$

```
1:  for $\{p_j, q_j, m_j\} \leftarrow \{0,0,0\}, \{255,255,255\}$ do
2:      for $a \leftarrow 1, C_{max}$ do
3:          for all $c_j \in S_j$ do
4:              Compute the set $X_j = \{x_j = h_j(c_j)\}$.
5:          end for
6:          if $Card(X_j) > a$ then
7:              Try with a larger Cardinality $a$, goto 2
8:          end if
9:          for all $c_i \notin S_j$ do
10:             if $h_j(c_i) \in X_j$ then
11:                 Try another function, goto 2
12:             end if
13:         end for
14:         return FOUND with parameters $\{p_j, q_j, m_j\}$.
15:     end for
16: end for
17: return FAIL.
```

The results show that, for practical values of the parameter C_{max} (i.e.: $_{max} \leq 64$), only $h_j(x) = (p_j \text{ AND } x) \bmod m_j$ does not cause the algorithm to fail. Moreover, it turns out that $m_j = 255$ in all the tests. Therefore we can define $h_j(x)$ as a simple AND operation with a bitmask:

$$h_j(x) = x \text{ AND } p_j \tag{2.4}$$

Such an outcome has a number of advantages: the number of parameters is limited to 1 (i.e.: small memory footprint), the operation is one of the most basic logic operation (i.e.: it costs a simple logic gate in an hardware implementation and it is very fast and parallelizable if our aim is a software engine) and the definition of $h_j(x)$ is amenable to be described by means of a tree, which means we can redefine each state transition set in a Longest-Prefix-Matching (LPM) description. Finally such a description is always achievable: even in the worst case (all characters produce a different transition and have a different tuple) the correctness of the scheme is not affected. In the following sections , we walk through the properties of such a representation and provide optimizations for our scheme. However, the main advantage is the possibility to concatenate two or more $h_j(x)$, such that we easily obtain a k-step DFA. As an example, if $\{h_0 : X_0 : n_1\}$ describes a transition from state n_0 to n_1 and $\{h_1 : X_1 : n_2\}$ is a transition from n_1 to n_2, then it it straightforward to verify that we the transition from n_0 to n_2 of the corresponding 2-step DFA is as simply as : $\{h_0||h_1 : X_0||X_1 : n_2\}$, where $h_0||h_1$ indicates the concatenation of h_0 and h_1 and $X_1||X_2$ is defined as:

$$X_1||X_2 = \{a_1||a_2 : \forall a_1 \in X_1, \forall a_2 \in X_2\} \tag{2.5}$$

Therefore all we have to take care of is the cardinality of the image sets X_j, that determines the memory requirements for this representation.

2.3.4 Optimizations

In the following we describe the advantages and the properties of the bitmask definition for $h_j(x)$ and elaborate upon the problem of minimizing the cardinalities of the image sets.

2.3.4.1 Permutation for LPM

A first observation on subsets S_j is that, in some cases, they may not be contiguous, i.e.: they may be the union of two or more non-contiguous subsets of symbols. Of course this is detrimental to our mission to minimize the cardinality of X_j. We solve this problem by introducing a *permutation* of symbols: we define a translation table (since we are dealing with bytes, it is a small 256 bytes table that does not increase the cost in terms of external memory accesses) that moves symbols in order to make non-contiguous subsets as contiguous as possible. Finding the optimal translation table is a complex issue since we can define a single translation table for the whole DFA, while subsets may vary from state to state. The good news is that in practical DFAs the number of different subsets is very limited. The bad news is that, as subsets vary from state to state, it may happen that a certain symbol occurs in different subsets. Therefore finding the optimal translation table is an *NP-complete* problem as it is equivalent to the *weighted maximum set packing* problem[70]: we want to find a set-packing (a collection of disjoint subsets) that maximize the total weight of its subsets.

Such weight (w) must take into account the memory impact of the subsets (a large and frequent subset has high utility because, once we put it in the translation table, it is likely to be described by a single bitmask). Therefore w is defined as the product of the cardinalities of subsets and the number of times those subsets appear in the whole DFA.

We attack the problem with the *Co-occurrence Permutation* algorithm, which is based on the co-occurrence of symbols in subsets. First, it computes the character co-occurrence matrix $A^{(0)}$, where an element $a_{i,j}^{(0)}$ represent the number of times characters i and j appears in the same subset multiplied by the cardinalities of the subsets they appear into (such that we replicate our weight metric). Then, the algo-

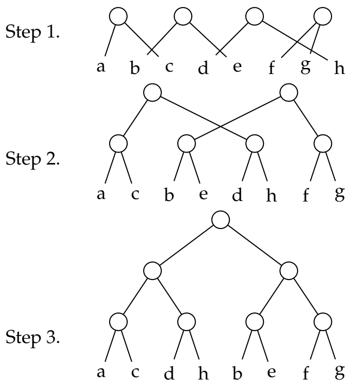


Figure 2.12: An example of *Co-occurrence Permutation* for 3-bit characters

rithm aggregates all 256 characters in 128 pairs, by grouping characters that present the largest co-occurrence, as depicted in the example of fig.2.12. After that, a new co-occurrence matrix $A^{(1)}$ is computed for all the 128 pairs. Again, pairs are aggregated thus forming 4-characters groups, another $A^{(2)}$ matrix is computed and so on. Therefore the algorithm recursively aggregates characters in a tree and the last matrix $A^{(8)}$ actually collapse into a scalar. Of course we have to define the co-occurrence of groups: for instance, given two symbols pairs i, j and l, m, we can define the pairs co-occurrence as:

- $\frac{1}{4}(a_{i,m} + a_{i,l} + a_{j,l} + a_{j,m})$
- $max(a_{i,m}, a_{i,l}, a_{j,l}, a_{j,m})$
- $min(a_{i,m}, a_{i,l}, a_{j,l}, a_{j,m})$

In our tests, we used the last option ($min(.)$) as it showed best results on all datasets. Finally we put the symbols in the leaves of the tree into a table and the translation table is simply the inverse permutation of such a table. Now, by means of *Co-occurrence Permutation*, subsets S_j can be described with single bitmasks. Then we can use a Longest-Prefix-Matching description of state transition-set and enlarge the number of ideas we can exploit for efficient implementation of DFAs, either taken from the widely studied field of IP lookup or newly proposed.

The effectiveness of the proposed schemes is measured by the total number of $h_j(x)$ we find for all states, once we permute the characters, as such a value represents the "cost" of our h-DFAin terms of transitions. The closer this number gets to the volume of the DFA graph (the cardinality of the edge set), the better the permutation scheme works. The results are shown in fig.2.13: *Co-occurrence Permutation* always gave good results, reaching, for many datasets, the minimum number of transitions or a value very close to it.

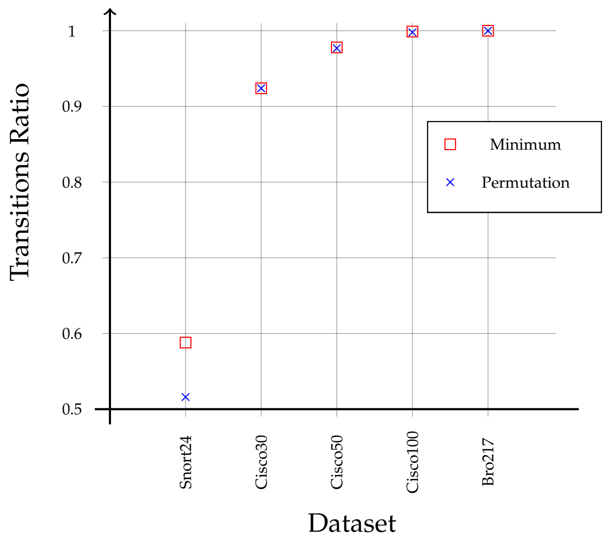
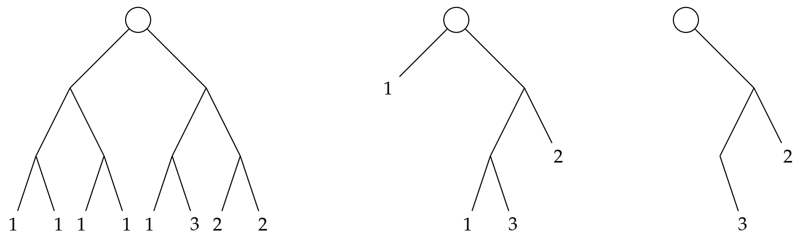


Figure 2.13: Ratio of transitions stored when *Co-occurrence Permutation* is used compared with the minimum number of transitions. The ratio is computed with respect to the case when no permutation is adopted.

2.3.4.2 Bitmap trees

As described above, thanks to the *permutation*, we can define each state by means of LPM structures, such as trees. The adoption of trees is twofold useful: it reduces the memory footprint of the bitmask description and it provides us with another faster way to compute the bitmask parameter in (2.4). As for the latter issue, it is straightforward to see that computing p_j and X_j (the result of h_j on subset S_j) can be now simply demanded to the creation of a tree of all the 256 possible values of the symbol (where last level leaves point to next state) and its subsequent pruning. Therefore, to store our tuples representation (2.2), we can simply use a bitmap tree. However, this does not preclude *permutation*; on the contrary, it takes advantages from the use of a permutation algorithm, because if the characters of same subsets are close to each other, they most likely produce short branches in the tree.

In order to construct a bitmap tree representation of a state, for each character c we get the next state s_{next} and add c in a tree, such that the leaf points to s_{next} as shown in fig.2.14 (where next states are 1, 2 and 3). Once we observed all the symbols, we prune the tree: if both children of a node x point to the same next state, x inherits children's pointer and children are removed. Finally, we can also remove from the tree the subset described with the largest number of leaves, as it can be stored as a "default transition" to be taken when no match is obtained. In the example of fig.2.14, we remove the leaves pointing to state 1 as they are the most frequent.



(a) Filling (b) Pruning (c) Largest subset removed

Figure 2.14: An example of state construction in h-DFA for 3-bit characters. The numbers on the leaves are pointers to next states

2.3.4.3 The overall algorithm

Here we retrieve the pieces decribed in the previous paragraphs and finally compose our algorithm for the creation of bitmask-based (or LPM) DFAs. The first step of the algorithm is the computation of subsets and of a series of functions $h_j(x)$ that can define an inverse homomorphism. Then we add a translation table by adopting *Co-occurrence Permutation*, and then we can simply compute an LPM description of each state of our DFA. Notice that an LPM description requires rules be stored in order in (2.2) or in a bitmap tree.

2.3.5 The k-step DFA

As described earlier in sec.2.3.3, the homomorphic (or LPM) description allows for a simple yet memory-efficient computation of k-step DFAs. The algorithm for the creation of a k-step h-DFA is shown in alg.9: it is based on a recursive procedure *compute_1-step* that takes a k-step h-DFA D' and a 1-step h-DFA D and computes the $(k+1)$-step h-DFA D''. As shown in the pseudocode, we add transitions defined by the concatenation of functions $h_1||h_2$ as defined in 2.3.3. Such a concatenation of functions may as well be seen as a concatenation of trees.

2.3.6 Results

The experimental runs have been performed on data sets of the Snort and Bro intrusion detection systems and Cisco security applications [47]. Such data sets, presenting up to hundreds of regular expressions, have been randomly reduced in order to mimic the size (in terms of DFA states and regular expressions) of other sets used in literature [62][48][50][61], as a fair comparison. The characteristics of the data sets are summarized in table 2.2, since we adopt the same datasets of previous sections. As for a construction timing evaluation, our preliminary code always required less than 2 minutes for each DFA to compute the corresponding h-DFA on a Pentium 4

Algorithm 9 Pseudocode for the creation of a k-step DFA

```
procedure compute_1-step($D, D'$)
1: for all state $s \in$ DFA $D$ do
2:     for all next state $s_1$ of $s$ do
3:         for all next state $s_2$ of $s_1$ in $D'$ do
4:             Add_transition($D''$,$s$,$s_2$, $h_1||h_2$);
5:         end for
6:     end for
7: end for
8: return $D''$
procedure compute_k-step($D$)
1: for $i \leftarrow 1, k$ do
2:     $D' \leftarrow$ compute_1-step($D, D'$)
3: end for
4: return $D'$
```

machine and always achieved a successful construction. The regular expessions in the data sets are given as input to the *regex-tool* [67], that produces the corresponding standard DFAs. Such DFAs are, in turn, used as start-point for our algorithms.

Tables 2.7 display the percentage of transitions (and memory) reduction for the different 1-step algorithms with respect to data sets representations through a standard DFA. h-DFA achieves a compression degree which is comparable to the other algorithms while requiring a single memory access per state. Because of its orthogonality with other schemes, this is a very appealing result. By using *Co-occurrence Permutation*, we are able to obtain good transitions and memory savings.

Fig.2.15 completes the comparison among 1-step algorithms by showing the mean number of memory accesses per character. D^2FA always requires the largest amount of memory accesses, while h-DFA requires always a single access (as D^2FA).

However, the main results are shown in tab.2.8, where the memory reduction obtained by computing 2 and 3-step h-DFA according to sec.2.3.5 are reported. In those results, h-DFAis combined with *Char-State compression* 2.1.4 to reduce the number of bits required to store transitions (notice that those techniques 2.1.4 do not add memory accesses). The memory reduction is computed with respect to a standard representation of 2 and 3-step DFA (respectively with 256^2 and 256^3 transitions per state). The compression percentages are very high. Certainly, 3-step h-DFAs still require too large amounts of memory (in the order of tens or even hundreds of megabytes). However, considering the consumptions of standard 2 and 3-step DFAs, which reach even hundreds of gigabytes, our solution is still appealing for DRAM storage. In fact, h-DFArequires at most 10 megabytes to represent 2-step DFA and (in some cases) less than 100MB for a 3-step, thus offering a great speed-up without the unfeasible memory requirements of standard DFAs. Moreover, as our technique is orthogonal to other schemes, we believe that a combination of different compression schemes can reach higher speed-ups requiring less amount of memory.

(a) Transitions reduction (%). For δFA also the percentage of duplicate states is reported.

Dataset	D^2FA		BEC-CRO	δFA		h-DFA	
	$DB = \infty$	$DB = 2$		trans.	dup. states	No Perm.	P_{Co-Occ}
Snort24	98.92	89.59	98.71	96.33	0	94.05	96.52
Cisco30	98.84	79.35	98.79	90.84	7.12	91.28	91.96
Cisco50	98.76	76.26	98.67	84.11	1.1	90.47	90.75
Cisco100	99.11	74.65	98.96	85.66	11.75	87.81	87.93
Bro217	99.41	76.49	99.33	93.82	11.99	78.48	78.48

(b) Memory compression (%).

Dataset	D^2FA		BEC-CRO	δFA + C-S	h-DFA + C-S	
	$DB = \infty$	$DB = 2$			No Perm.	P_{Co-Occ}
Snort24	95.97	67.17	95.36	95.02	84.16	90.73
Cisco30	97.20	55.50	97.11	91.07	87.91	88.87
Cisco50	97.18	51.06	97.01	87.23	83.82	84.31
Cisco100	97.93	51.38	97.58	89.05	81.63	81.82
Bro217	98.37	53.00	98.23	92.79	69.02	69.02

Table 2.7: Compression of the different 1-step algorithms in terms of transitions and memory.

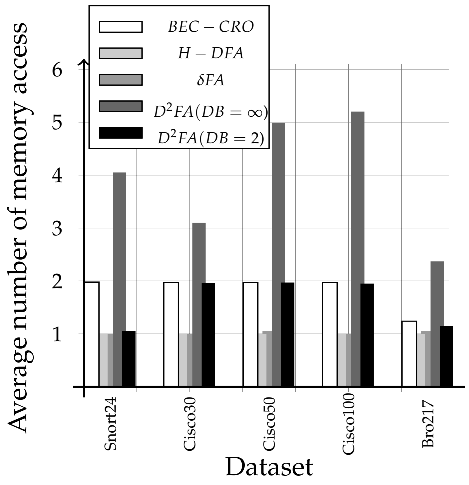


Figure 2.15: Mean number of memory accesses per character.

Dataset	2-step h-DFA + C-S		3-step h-DFA + C-S	
	mem.	trans.	mem.	trans.
Snort24	88.48	98.35	99.34	99.69
Cisco30	98.73	99.50	99.87	99.99
Cisco50	97.84	99.07	99.81	99.92
Cisco100	95.67	98.06	99.42	99.56
Bro217	96.97	95.57	97.98	99.08

Table 2.8: Memory and transition compression (%) for 2 and 3-step h-DFA + *Char-State compression*

2.4 Sampling techniques to accelerate regular expression matching

The previous works which propose acceleration techniques (as we discussed in subsection 2.3.1) multiply the amount of bytes (strides) processed per cycle. The obvious problems which arise is memory blow-up (essentially due to the exponential growth of edge numbers with the stride size) and can be partially mitigated through smart coding for the transition table, alphabet-reduction and default transition compression. Our approach to the finite automata speed-up is completely innovative: sampling the text, thus having less symbols to be processed. Clearly, sampling introduces some issues; in details, particular automata for the processing are required and a certain probability of false alarms is introduced. We address these issues by the combination of a "sampled" and a "reverse" DFA, two different versions of the original automaton. We perform a first fast search on the traffic through the sampled DFA, which is able to exclude most part of non–malicious traffic, and, if necessary, a more accurate processing through the reverse DFA is triggered, in order to confirm a match. While other works [71] in the area of intrusion detection already show how to sample *messages* to reduce the amount of messages to be processed in a distributed system, the application of sampling to regex-matching is a novelty and one of the main contributions of this work.

2.4.1 Sampling DFAs

In this section we introduce the motivation for DFA sampling and describe the main concepts by means of an example. Moreover, we provide a taxonomy to distinguish the different ways we can sample a regular-expression set or the corresponding DFA.

2.4.1.1 Motivation

The motivation for this work relies on the following assumptions:

- IDS regex data sets are well-written;
- Regular internet traffic does not match properly written IDS regexes.

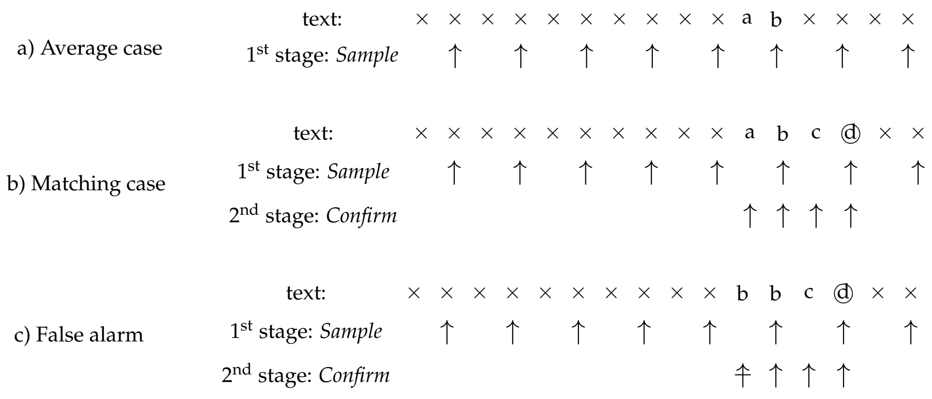


Figure 2.16: Examples of sampling with $\theta = 2$. The regex to match is $ab.*cd$, the sampled one is $[ab].*[cd]$ and the text consists of 16 bytes. Arrows point to observed chars. Sampling performs 12 memory accesses in case of a real match(b) or false alarm(c) or even 8 in the average non-matching case(a). In (c) the striked arrow point to the non-matching char.

Indeed, if regex sets are not poorly written (in our tests we use real and effective IDS signatures from Bro and Cisco security applications) a signature match will occur with malicious traffic only. The main assumption is that the majority of traffic is not malicious. Therefore we can take advantage of the fact that a match is a rare event and speed up the average case (regular traffic).

The idea of DFA sampling is to speed up the regex matching by simply "sampling" the traffic stream: we extract a byte every θ bytes from the stream, where θ is the *sampling period*. The sampled bytes are then used as input to a proper sampled DFA. The outcome is that all regular traffic is processed θ times faster. The price to pay is that this process may introduce false alarms: strings that would not match the original non-sampled regex could match the sampled one. Therefore whenever we have a match in the sampled DFA, we have to process the suspect packet through a regular non-sampled DFA.

It is worth noticing that we aim at reducing the number of memory accesses to the main memory that stores the state-machine, while we cannot reduce the number of accesses to the memory where the packet is stored. The reason is that even if we were interested in, say, a byte every two, memories would allow accesses in minimum sizes of k bytes long words. However, in cached-systems, this is an advantage when performing the second stage to check for false alarms: all the memory accesses for this second stage will result in cache-hits, thus reducing the cost of false alarms.

2.4.1.2 A Motivating Example

Fig. 2.16 shows the principles of our scheme, with the example regex $ab.*cd$. We use a sampled DFA (that matches $[ab].*[cd]$) and a regular non-sampled one. We perform a first check on the text by using the sampled DFA; if we find a match, we move to the second stage.

Fig. 2.16(a) represents the common scenario with traffic that does not match signatures. It is evident, in this case, that the number of memory accesses and operations to be performed is divided by the sampling period.

Whenever the sampled regex is matched (lower two cases in figure 2.16, where the circled letter indicates the sample where we find the match), the non-sampled text has to be checked to confirm the match. To address this issue, the simplest and fastest way (see section 2.4.3.2) is to adopt DFAs that match reversed signatures (in our example, $dc.*ba$). Any regular language is closed with respect to reversing operations [68], therefore we can always reverse a regular expression and match it inside a text by observing the text backwards from the end to the beginning. Then, if a match occurs in the second stage, because of the equivalence of reversed and forward DFAs, we have a confirmed match (fig. 2.16(b)). Otherwise, we can claim that a false alarm occurred (fig. 2.16(c)).

2.4.1.3 Taxonomy of DFA Sampling

Sampling can be performed in a number of different ways. In the following we give a brief description of these techniques and of the models that we use. From the point of view of the sampling period θ, we can have:

1. Fixed Period Sampling (CPS) : θ is constant;

2. Variable Period Sampling (VPS) : $\theta = \theta(s, c)$ is a function of the DFA state s and/or the input character c.

The construction of a sampled DFA can be classified as:

1. Static : $\theta(s, c)$ is decided during construction;

2. Adapting/Evolving : $\theta(s, c)$ evolves adapting to traffic features, reducing false alarms and maximizing speed-up.

However, in this first work on sampling we focus on static constant period sampling.

2.4.2 Regex sampling rules

Here we introduce basic theoretical results on DFA sampling. As in signal sampling theory Nyquist condition is the only one rule to satisfy, also when dealing with regular expressions matching a simple unique condition has to be satisfied to perform a correct sampling:

Lemma 1. *Let DFA A describe a single regular expression $\boldsymbol{R}$[1] and let a text T match $\boldsymbol{R}$. The corresponding sampled DFA $A^{\mathbb{S}}$ will match the sampled text $\mathbb{S}T$ if the sampling period θ satisfies the following:*

$$\theta \leq \min |r| \qquad \forall r \in \boldsymbol{R}$$

Proof. The proof is straightforward. In order to match the sampled text, we have to extract (by sampling) at least a character from the substring of T that matched A. Thus the condition follows. $\square$

Lemma 1 limits the sampling period that can be used when matching a single regular expression. However when working with DFAs that match a set of regular expressions, it still applies, as long as the limit is moved to the minimum length of any string that match any regular expression in the set. Moreover the lemma states that, if the condition is satisfied, we may have false positives but we cannot have false negatives. This important result is the basis of the research presented in the rest of this research.

2.4.2.1 Regex rewriting

The application of sampling can be performed by rewriting regular expressions according to few simple rules. In the following we use the notation:

$$\mathbb{S}_{\{\theta\}}^{X_0} \boldsymbol{a}$$

to refer to the application of sampling to the regular language $\boldsymbol{a}$. In particular, the symbol $\mathbb{S}$ represents the *sampling* operator, $\{\theta\}$ is the series of sampling periods $\theta_0, \theta_1, \ldots, \theta_N$, and X_0 is the position of the first sampled character. In the rest of this section, the sampling operator $\mathbb{S}$ will be adopted also as an exponent (i.e.: $A^{\mathbb{S}}$) to denote the sampled version of a DFA.

Basically, we show the application of the $\mathbb{S}$ operator to four main cases:

1. simple string str
2. concatenation of regular expressions $\boldsymbol{a}$ and $\boldsymbol{b}$: $\boldsymbol{ab}$
3. union of regular expressions $\boldsymbol{a}$ and $\boldsymbol{b}$: $\boldsymbol{a}|\boldsymbol{b}$
4. star closure of a character a followed by a regular expression $\boldsymbol{b}$: $a{*}\boldsymbol{b}$

The sampling of a string is straightforward, and it simply consists of extracting characters at the positions defined by $\{\theta\}$ with offset X_0:

$$\mathbb{S}_{\{\theta\}}^{X_0} str = \{str(X_0 + \{\theta\})\}$$

[1]Throughout the whole section, bold letters represent regular expression, while non-bold stand for single letters.

The offset X_0 is critical also when sampling the concatenation and union of regular expressions, it is immediate to show that:

$$\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{ab} = (\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{a})(\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{b})$$

and

$$\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{a}|\boldsymbol{b} = (\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{a})|(\mathbb{S}^{X_0}_{\{\theta\}} \boldsymbol{b})$$

Finally, a star closure of a character a followed by a regular expression is simply :

$$\mathbb{S}^{X_0}_{\{\theta\}} a * \boldsymbol{b} = a * \left|_{i=0}^{\theta-1} \mathbb{S}^{i}_{\{\theta\}} \boldsymbol{b} = a * \mathbb{S}_{\{\theta\}} \boldsymbol{b}\right.$$

We can easily verify the sampling of $a*$ is again $a*$. Then, since $a*$ consists of all the possible n-repetitions of a (where $n \in \mathbb{N}$), the sampling offset we apply to $\boldsymbol{b}$ can be any, hence the big **OR** operator $|$.

Now, although the last case follows from the first three (concatenation and union), it is worth describing it because of its frequent occurrence in real regular expression sets. Indeed, many regular expressions in real IDS/IPS data-sets adopt star closures and most of the times they are unanchored rules (because security signatures may occur everywhere in the text) of the form: $\boldsymbol{.*a}$. Therefore sampling produces a union of the regex $\boldsymbol{a}$ sampled with all different possible offsets. As an example, let us suppose we have $. * abcde * fgh$ and we are sampling with fixed period $\theta = 2$. By applying the previous rules, it follows that:

$$\mathbb{S}_2[. * abcde * fgh] = . * (ac|bd)e * (fh|g)$$

2.4.3 Constant Period Sampling

2.4.3.1 First stage: Sampled DFA

As above mentioned, the idea of DFA sampling is to speed up the string matching by extracting a byte every θ bytes from the stream and giving such characters as input to a "sampled DFA" for a first approximate search (to be subsequently confirmed). Regarding the Constant Period Sampling (CPS) case, such sampled DFA can be simply obtained by properly rewriting the regexes and building the DFA according to the new rule-set.

In details, for the process of regex rewriting, we can apply the results of section 2.4.2.1 by selecting a single value θ for all the sampling periods θ_i. Instead, concerning the offset X_0, which is the position of the first character to be sampled in the regex, we have to take into account all the possible starting values. This way, the resulting complete automaton can be used for string searching regardless of the point in which we start to sample the traffic.

The pseudocode 10 just shows the overall procedure for rewriting a regex by using a constant period θ and by adopting all the possible values for the starting offset: we split the regular expressions into sub-elements that can be processed directly by adopting the rules in sec. 2.4.2.1. In order to simplify the code, a pre-processing (not shown here) is adopted to convert "+" closures in "*" (i.e., $a+$ becomes $aa*$) and to take care of the cases when the sampling period is higher than the length of the minimum string between two closures or the presence of unions ("|"). By repeating such a process for all the regexes belonging to the set, we obtain the "sampled" rules on which the "sampled DFA" has to be built. Such a resulting automaton is a simple DFA and does not require additional information on the states or on the transitions. From this observation and as suggested by the results of section 2.4.2, we can claim that a regular language is closed with respect to the fixed sampling operator.

However, even after pre-processing, some regular expressions may still be so short to make sampling unconvenient. For instance $\mathbb{S}_3 abc = [abc]$: although the sampled regular expression is valid, it is only 1-character long, thus potentially yielding a large number of false alarms. The good news is that these extremely short regular expressions are not very frequent. Therefore a good and effective solution is to hardcode them, moving the matching problem from data to code and adopting a function *regex_match(c)* which is basically composed of *switch() – case* and *if – then* statements. This is a well-investigated idea [72][73][74] that is shown to be very useful with a small number of regular expressions. It is also compatible to our sampling approach: the *regex_match(c)* function can still access all the bytes of the un-sampled text (which are available to the code, as discussed in section 2.4.3.3) thus keeping the processing engine busy between two successive memory accesses to the sampled DFA. Since the number of regular expressions to be matched by such a code is small (as already pointed out, short regular expressions are fairly rare), the whole data set which is necessary for such a code can be kept in the local cache, thus requiring no further accesses to the external memory blocks.

Algorithm 10 FPS of a regex **a** with period θ.

```
procedure sample_regex($\mathbf{a}, \theta$)
 1: $p_{next} \leftarrow$ first_pos($\mathbf{a}$,"*") $-1$
 2: $l \leftarrow \mathbf{a}[p_{next}]$
 3: while $p_{next} \neq$ NULL do
 4:     $\mathbf{b} \leftarrow \varepsilon$
 5:     for all offset $x$ do
 6:         $\mathbf{b} \leftarrow \mathbf{b}|$ sample_str($\mathbf{a}(p_{prev} + x \ldots p_{next})$)
 7:     end for
 8:     $\mathbf{a}' \leftarrow \mathbf{b}l * \mathbf{a}'$
 9:     $\mathbf{a} \leftarrow \mathbf{a} << p_{next}$
10:     $p_{next} \leftarrow$ first_pos($\mathbf{a}$,"*") $-1$
11: end while
```

2.4.3.2 Second stage: Reverse DFA

Whenever in the "sampled DFA" a matching happens (i.e., an accepting state is reached), we have to process the text again, by means of the original non-sampled DFA, in order to obtain a confirmation of the match. As already mentioned, the reason is that sampling a DFA introduces a false alarm probability, since we check only a subset of the characters of the string.

Let us suppose the matching has been detected at the $k-th$ sample in the text. By using the original DFA for such a further search, a problem arises: which byte of the packet has to be the starting point for the matching confirmation processing?

The simplest solution could be processing again the overall packet (i.e., starting from the first byte of packet), but this heightens the processing and yields an excessive delay. Therefore, a more efficient technique could be to "remember" the last time the process has been in the root state (hereafter simply called state 0) and start from the corresponding character. However, even this solution could be too expensive, requiring the processing of a big number of characters, as shown in the following example.

Let us assume our DFA (shown in figure 2.17(a)) matches the two regular expressions:

$$.*ab.*fgh$$

$$.*cded$$

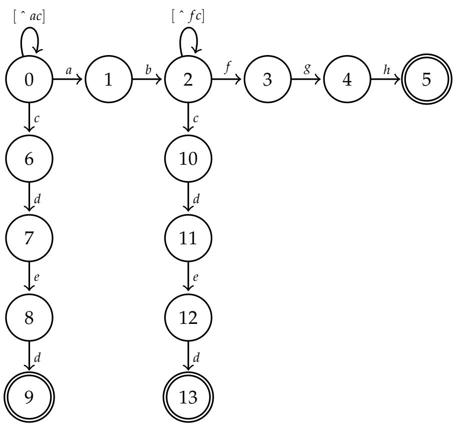
The sampled DFA (fig. 2.17(b)) is built on the sample regexes:

$$.*(a|b).*(fh|g)$$

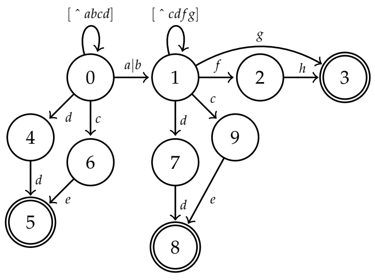
$$.*(ce|dd)$$

Suppose we read the text: $T = xxabxxxxxxxcdedxxx$ and we sample the text with $\theta = 2$, obtaining the sequence: $T' = xbxxxcexx$. By triggering the sampled DFA with such a sequence, the processing of the character e reports a match in state 8, which has to be confirmed with the standard 1-step DFA (fig. 2.17(a)). The last sampled character for which the sampled DFA was in state 0 is the second x of T. Therefore, if we use a forward DFA to confirm the match, we need to process all the characters from a (position 3) to the last d (position 16): we read 13 characters to confirm a match of a 4-bytes string ($cded$). This is due to the presence of a closure ($.*$) within one of the regex matched by the DFA. Indeed, for each closure, the DFA replicates the states corresponding to some regular expressions (as happens in state 2 in fig. 2.17(a)). This means that we can start matching a whole signature starting from a state which is not the root state 0. This requires the processing of a much larger number of characters than strictly needed.

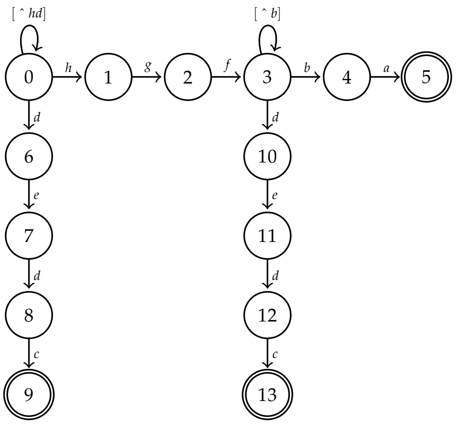
For these reasons, in order to improve the performance of the second stage, we propose a novel scheme where a *reverse DFA* has to be built. Such a technique requires a slightly larger amount of off-line processing: all the regexes have to be independently reversed and a new DFA has to be built according to such new rules. However, this approach has the advantages that we can start the second stage reverse-matching from the last sample. More precisely, in order to take into account all the characters belonging to the string, the correct starting point for the reverse DFA is the $(k+1)$-th sampled char in the text. The reason is that the sampled DFA may report a match for a signature that ends between the actual and the next sample. Therefore, we process some useless characters too (the first ones in the text after the matched string), but this does not affect the detection of the string by the reverse DFA. On the contrary, since the match in the sampled DFA may occur some character before the



(a) The DFA



(b) The Sampled DFA



(c) The Reverse DFA

Figure 2.17: Example of the finite automata needed for sampling (only the forward transitions are shown for readability): (a) is the standard DFA, (b) is the sampled one (with $\theta = 2$) and (c) is the reverse DFA.

real match of the non-sampled string, by moving one sample further we ensure the correctness of the scheme at the cost of processing a few more (less than θ) characters than the strictly needed.

Thus, if we adopt a reverse DFA (fig. 2.17(c)) in the previous example, and we have a match in the k-th sample, we start the reverse DFA from the next sample (the $(k+1)$-th) and go backwards, processing the substring $T'' = xdedc$ (i.e., the reverse of the substring $cdedx$ from T). As easily verifiable, the reverse DFA correctly confirms the match by processing 5 characters only, while the forward DFA needed 13 bytes. Notice that, in this example, if we started the reverse DFA from the matching sample (character e) we would wrongly miss the match.

In the confirmation stage performed through a reverse DFA, we confirm a match whenever an accepting state is reached (notice that an accepting state in such a DFA represents the beginning of an original non-reversed regex). Instead, we can immediately detect a false alarm and stop our search whenever we return to the state 0 with any character belonging to the subset of positions $0 \ldots k \times \theta$. Indeed, the characters not belonging to the string, which are between the $(k \times \theta + 1) - th$ and the $((k+1) \times \theta) - th$ char, could trigger a return to the state 0, but this does not imply a false alarm. Instead, if the return to the state 0 is forced by any character from the $(k \times \theta) - th$ to the first one (which certainly belong to the probable matching string), then a false alarm is detected.

Further performance refinement However, in some cases, locating the $(k+1)$-th sampled char in the text is not a good choice for the performance of the correct matching search. An example is shown in fig. 3 where the sampled DFA (with $\theta = 2$) for the regular expression $abc * d$ is depicted. With such a DFA, when we process the text $T = xxxabcccccccccdx$ we trigger a reverse DFA confirmation for each of the sampled character c inside the text, always reporting a false alarm. When, at last, we read the character d, we restart the final and conclusive confirmation stage with the reverse DFA (with a right positive result). Notice that this problem occurs because of the closure on the accepting state of the DFA and affects the performance of the technique solely, while the correctness is not invalidated. A simple solution to this problem is to start the confirmation process when leaving a matching state only (i.e., state 2 in the example). In this way, we make sure that the reverse DFA has really started matching a regular expression.

The pseudocode for the lookup is shown in alg.11. In the listing, s represents the actual state, s_{next} is the next state and i and j are the text position we currently read respectively for the sampled DFA $A^{\mathbb{S}}$ and the reverse DFA A^R (this convenient exponent–notation will be adopted hereafter). In the pseudocode, lines 1-3 and 22-24 are part of the regular sampled DFA walk. Line 4 represents the condition we discussed above: we start a confirmation match only when we leave an accepting state (s.acc is the accepted rule) and move to a state that does not match the previous rule. This takes care of the cases where the accepting state has a loop. In lines 5-8 we initialize and start the first part of the reverse DFA walk and do not care about the occurrence of state 0. Then the first while loop (lines 9-12) is in charge of cases where the first state of the reverse DFA has a closure (for instance: $(abcde*)^R = e * dcb$). Finally the next *while* loop performs the reverse DFA walk.

Algorithm 11 Pseudo-code for the lookup procedure.

```
procedure lookup ($T$,$A^{\mathbb{S}}$,$A^R$,$\theta$)
 1: $s \leftarrow 0$
 2: while $i < length(T)$ do
 3:     $s_{next} \leftarrow A^{\mathbb{S}}[s, T[i]]$
 4:     if $s$.acc $> 0$ AND $s_{next}$.acc $\neq s$.acc then
 5:         $s' \leftarrow 0$
 6:         for $j \leftarrow i, i - \theta(s)$ do
 7:             $s' \leftarrow A^R[s', T[j]]$
 8:         end for
 9:         while $s' == 0$ do
10:             $s' \leftarrow A^R[s', T[j]]$
11:             $j \leftarrow j - 1$
12:         end while
13:         while $s' \neq 0$ do
14:             $s' \leftarrow A^R[s', T[j]]$
15:             if $s'.acc > 0$ then
16:                 Confirm Match of $s'.acc$
17:                 return to outer loop
18:             end if
19:             $j \leftarrow j - 1$
20:         end while
21:         claim False Alarm
22:     end if
23:     $i \leftarrow i + \theta(s_{next})$
24:     $s_{next} \leftarrow s$
25: end while
```

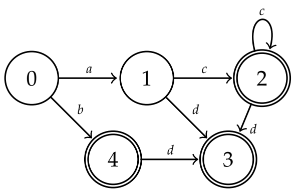


Figure 2.18: Example of a sampled DFA for regular expression: $abc * d$. Only some edges are shown.

Splitting the reverse DFA As a final comment, we consider splitting the reverse DFA into several smaller DFAs, one for each subset of regular expressions corresponding to a matching state in the sampled DFA. In details, for each accepting state $\underline{s_j}$ in the sampled DFA $A^{\mathbb{S}}$, we observe the subset of regular expressions $X_j = (\mathbf{r_1}, \mathbf{r_2}, \ldots \mathbf{r_k})$ that $\underline{s_j}$ accepts. For each different subset X_j we create a reverse DFA A_j^R. This way, whenever the sampled DFA reaches a matching state $\underline{s_j}$, we start the reverse match with the corresponding A_j^R. This approach reduces the number of steps to perform in the reverse match when a false alarm occurs. Indeed, on a large DFA A^R a piece of text that does not match a given regular expression may match a part of any other regular expression in the set hence keeping the walk away from state 0 and preventing us to claim the false alarm. This does not happen when adopting several small DFAs A_j^R. Moreover this scheme remarkably reduces memory wastage, since, as shown in [59], n DFAs are less expensive than a single DFA for n signatures in terms of number of states and size. However, using n reverse DFAs requires in the sampled DFA additional information which link each accepting state to its own reverse DFA. In the final experiments we will discuss about the performance of both schemes.

2.4.3.3 Possible implementations

The discussion above shows that the basic idea of our approach is to divide the problem into common cases (no matches) and "exceptional" events (a match). To deal with these two cases, we perform two different processing stages. It is worth mentioning that the second stage does not have to be performed necessarily by the same processing engine that executes the first stage. For instance, k_2 processing engines can be allocated for this job, dealing with all the alarms produced by $k_1 > k_2$ first-stage engines. However, while this possible implementation can increase the speed of our solution, in this work we describe our approach as performed by single entities (first and second stages in the same engine), as we are interested in showing a proof of concept rather than the best possible implementation.

2.4.3.4 Dealing with DoS attacks

Generally, when dealing with security applications, every approach that tries to optimize a frequent case by relying on the assumptions that certain events (for us, a signature match) are relatively uncommon, is subject to be affected by aimed attacks that try to increase the probability of the rare events, thus invalidating the purpose of the method. However, as proposed in [49], such Denial Of Service (DoS) attacks can be taken care of by observing the "behavior" of the incoming flows and distributing them into different queues (with different service rates) accordingly. In our scheme, the "good" or "bad" behavior of a flow is measured by the number of false alarms it generates within a time frame, since false alarms represent the largest portion of the processing cost. Therefore, according to this mechanism, flows that generate a large number of false alarms are sent to the queue with lowest service rate, while "good" flows (i.e.: with few false alarms) are queued and serviced with high rates. However, in this work we do not deal with the details of such an approach, inviting the interested readers to find more details in [49].

2.4.4 Experimental Results

In order to understand the advantages and costs of our approach, in the following we present the results of a number of experimental tests on real traffic. In details, the purpose of these tests is to show the behavior of the sampling approach in real cases and as the parameters of the problem vary.

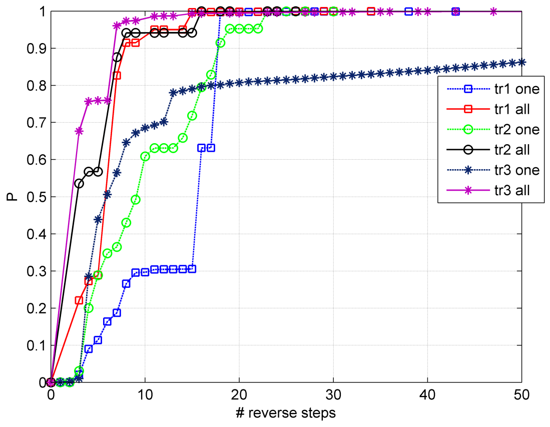
To propose verifiable and valid tests, we use:

- the datasets of regexes of real Bro and Snort intrusion detection systems and Cisco security appliances [47];
- the Michela Becchi's regex tool (which is freely available [67] and proven very stable and powerful) to create the DFAs from the "sampled" regexes.

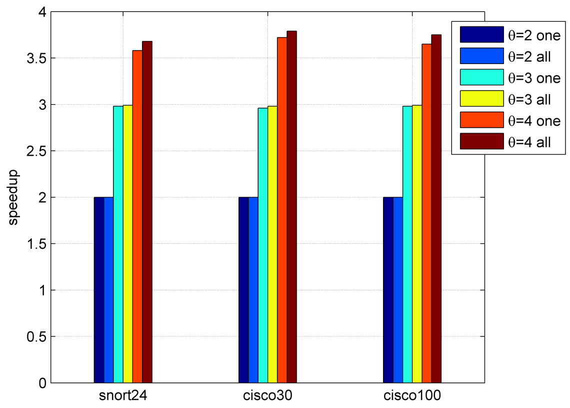
More precisely, we processed the real regex datasets (where the number in the name indicates the number of regexes) with our tools for creating the new regexes (i.e. sampled and reverse), which are then parsed by the Michela Becchi's regex tool to create the DFAs (which therefore result to be the sampled and reverse DFAs).

We dumped several traffic traces from our department network. Such traces were composed by several flows, associated with different kinds of applications (peer-to-peer, web browsing, multimedia, ftp), therefore they encompass a realistic mix of both mainly textual streams and binary streams. The different TCP connections have been reassembled by using TCPflows [75] and the resulting streams have been concatenated in order to obtain the overall traces. The first runs aim at comparing the efficiency of using an overall reverse DFA for all the regexes (*one* in figure 2.19) or a single reverse DFA for each subset of regular expressions corresponding to a matching state in the sampled DFA (*all* in figure 2.19). The graph in figure 2.19(a) shows the number of steps required in the reverse walk by using the two different techniques when processing three real traces (of length 52MB, 48MB and 66MB respectively) with Cisco100 as regex databases. Instead, figure 2.19(b) illustrates the speedup (computed as ratio between the trace length and number of accesses required) when processing the firs trace (52MB) for different regexes datasets. As foreseen in section 2.4.3.2, using one reverse DFA for each subset of regex reduces the number of steps to perform in the reverse walk when a false alarm occurs and allows higher speedups, along with a memory saving.

To compare our sampling scheme with a classical DFA which processes all bytes, we took as reference hardware platform the Network Processor Intel IXP2800 [76]. Network Processors offer very high packet processing capabilities (e.g. for gigabit networks) and combine the programmability of general-purpose processors with the high performance typical of hardware-based solutions. In particular the IXP2800 is designed to perform a wide range of functionalities, including multi-service switches, routers, and broadband access devices. It is a fully programmable network processor, characterized by a hierarchy of processing units (a XScale core and 16 32-bit microengines MEv2, running at 1.4GHz) and memory devices (4KB of local memory,



(a) Required steps in the reverse walk.



(b) Overall speedup.

Figure 2.19: Using an overall reverse DFA (*one*) or one DFA per regex subset (*all*).

16KB of scratchpad memory, 256MB of external SRAM and 2GB of external DRAM). The bigger the memory, the slower the access to it.

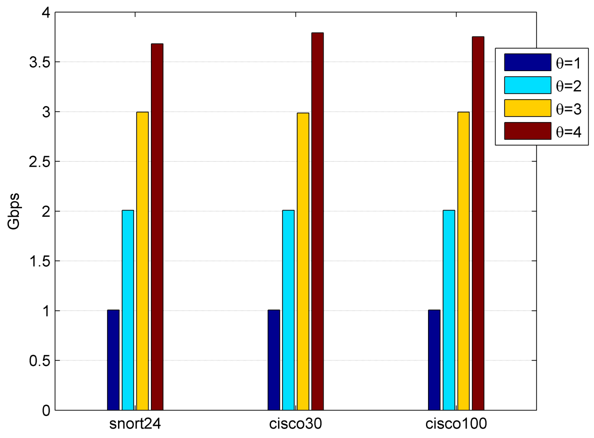


Figure 2.20: Bit rate with a standard DFA ($\theta = 1$) and sampled DFAs ($\theta = 2, 3, 4$).

We simulated the functioning of our algorithm by putting the automata in DRAM, given the large memory required by standard DFAs, and reserving for pattern matching 4 microengines with zero-overhead full threading support (i.e., 8 threads per microengine with no penalty for context switch). Consider that each state traversal requires a DRAM access, as well as the readings of packets, and that, in turn, each DRAM access costs on the average 1270 clock cycles. Fig. 2.20 reports the results in terms of bit rate when processing a real trace of 20MB with a common DFA (i.e. $\theta = 1$) or with our sampling scheme (with $\theta = 2, 3, 4$). It is evident the speedup of sampling DFA, which allows to multiply the bit rate. The payment due to the check in the reverse DFA is very slight because of the low occurrencies of false positives in real traffic. Therefore, the sampling DFA enables a big saving of processing, according to the sampling period, which results in a higher sustainable bit rate.

In order to perform comparisons between our solution and the more efficient schemes for speeding up the matching search in DFAs, we implemented the techniques proposed in [69]. In details, we apply alphabet-reduction and default transition compression to k-DFAs introduced in [69], which in turns are based on D^2FAs.

The graph in figure 2.21 shows the data rates achieved when processing real traces of 52MB (*trace*1) and 48MB (*trace*2) and by adopting Cisco100 as dataset. In particular, we compared our scheme and the one implemented according to the directions in [69] by setting $\theta = k$, where the former represents the sampling period and the latter the stride length (i.e., the amount of bytes which are processed at each step in [69]). Notice that the runs pointed out that both the schemes detect the overall number of attacks in each case (i.e. for each mix of traces and databases). The 1st and 3rd bars of the histogram represent the bit rates for our sampling scheme, while the other two bars report the values for the multi-stride scheme. The advantages of our schemes are clear.

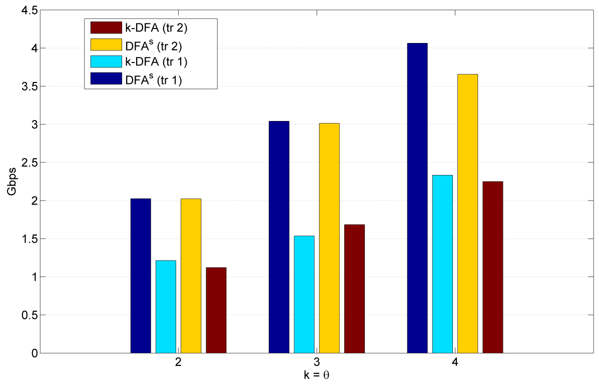


Figure 2.21: Bit rate with k-DFA and our DFA^s.

As for the memory size requirements, we do not report the results because, since our technique produces regular DFAs, it can actually be coupled with many of the compression schemes proposed in literature (and cited in the previous introductory sections) thus avoiding memory blowup. However, it is important to point out that the sampling operation does not overly increase the size of the DFAs. Another experiment aims at describing the effect of the features of regular expressions on the number of false alarms (and hence speed) in the sampling approach.

In fig. 2.22 we report three aligned graphs that correlate the measured probability of false alarms (P_{FA}, top graph) generated by each signature to the signature length (graph in the center) and the signature range (bottom graph). The signatures are labeled by the numbers on the x-axis and the graphs represents the results of a sampled match of the Cisco200 regular expressions with $\theta = 4$. The length of a signature is defined as the length of the shortest string that matches the signature, while we define the range as the cardinality of the set of all strings matching that signature. Of course, closures (* or +) would cause the range of a signature to be infinite (remember closures represent the unlimited repetition of a character). For this reason, in order to properly represent ranges in the graph, we set the range of a closure to a large number (10000). The figure shows that a few short signatures are responsible for the majority of the false alarms. In the following, from the inspection of a few distinct cases, we show how to extrapolate the general behaviour. Signature 99 contributes to more than 25% of all false alarms, this is mainly due to its short length and to its fairly regular range. On the other hand, signature 110 does not produce any false alarm because, even if it has a large range, it has a remarkable length. Signature 115, instead, is quite short (the shortest length bar in the middle graph), but it does not contribute to false alarms because of its very limited range. These examples justify the intuitive idea that short signatures with large range are the most likely to provoke false alarms. Another comment is that the length of a

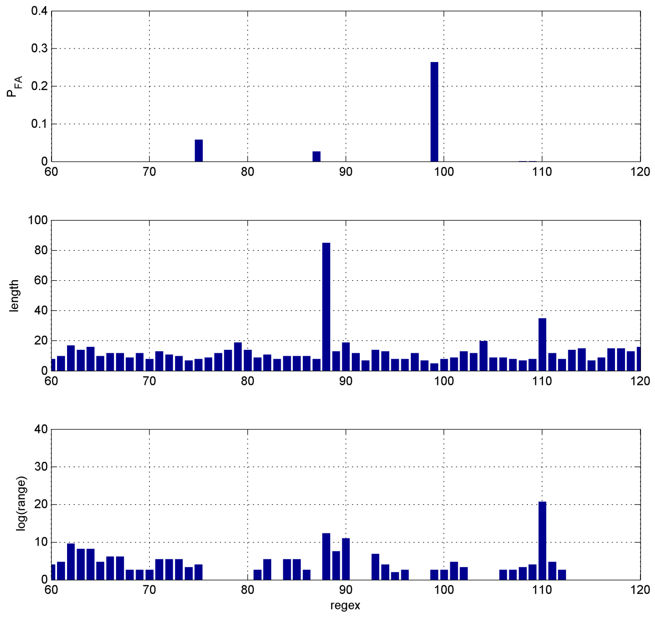


Figure 2.22: False alarms, length and range for each signature.

regular expression has a larger effect on false alarms than its range.

2.5 Enhancing Counting Bloom Filters through Huffman-Coded MultiLayer Structures

Nowadays streamed data processing is a basic problem in many areas related to computer applications. In particular, detecting whether an item belongs to a set is one of the most challenging tasks, especially when the amount of data to be processed per unit of time is very large and rapidly changes.

A Bloom Filter (BF) is a simple data structure for information representation and query processing. It is a randomized method based on hash functions; thus, it allows for false positives, but the space savings often outweigh this drawback. BFs were introduced by Burton Bloom [77] in the 1970s for database applications, but recently they have received a great attention also in the networking area [78], for collaborating in overlay and peer-to-peer networks, packet routing, and measurements. BFs are also proposed for many distributed networking protocols: for example, in order to share web cache, a proxy periodically broadcasts BFs that represent the contents of their cache. In this situation, BFs are not only data structures but also messages being transmitted in a network. Thus, several performance parameters have to be taken into account: the probability of false positives, memory size, number of items to be managed and transmission size.

BFs do not address the issues of inserting and deleting items in the set. For example, a set may change over time, with elements being inserted and deleted. Deletion cannot be done by simply reversing the insertion operation, because of the collisions created by the hash functions. In order to allow these operations, Counting Bloom Filters have been designed [79]. They are based on the same idea of BFs, but they use fixed size counters (also called bins) instead of single bits of presence. When an item is inserted, the corresponding counters are incremented; deletions can then be safely done by decrementing the counters. CBFs present the problem of counters overflow, which has to be considered in the design.

This section adopts a simple upper bound for the CBF overflow probability which is functional to the design of new efficient solutions.

The central idea of the section is that, by leveraging on the bound, a novel paradigm in CBF design can be adopted; such a paradigm involves *compression* – to improve CBFs in terms of fast access and limited memory consumption (up to 50% of memory saving in comparison with the standard solutions) – and the introduction of layer *hierarchy* in the CBF data structure.

The target could be to take advantage of the built-in memory hierarchy of many systems (such as Network Processors – NPs) to implement compressed data structures in the small but fast local memory or "on-chip SRAM" of such devices. As an example of the advantages of our compressed CBFs, we propose a compact solution to the detection of evasion attacks to Intrusion Prevention Systems (IPSs).

In details, the main contributions of this section (which is an extended version of the work in [80]) are:

- the use of Huffman code in CBF, which is optimal for independent symbols (such as the bins of a CBF);
- the idea of a hierarchical multilayer structure;
- the proposal of an efficient CBF for systems with limited memory such as NPs and programmable routers;
- the adoption of these efficient structures in the solution of a difficult task such as recognizing evasion attacks.

This sectionr is organized in two main parts: first we describe the proposed algorithms and then we show a brief example of their application, which is shown in more details in [81]. A comparison among our algorithms and the algorithms defined in literature is performed in subsection 2.5.5, by adopting NP Intel IXP2800 as a referential hardware platform.

2.5.1 Background on Bloom Filters

A Bloom Filter represents a set S of n elements from a universe U by using an array of m bits, denoted by $B[1], ..., B[m]$, initially all set to 0. The filter uses k independent hash functions $h_1, ..., h_k$ with $\log_2(m)$ bits long output, that independently map each element in the universe to a random number uniformly distributed over the range. For each element x in S, the bits $B[h_i(x)]$ are set to 1, for $1 \leq i \leq k$ (a bit can be set to 1 multiple times).

To answer a query of the form *"Is y in S?"*, we check whether all $B[h_i(y)]$ are set to 1. If not, y is not a member of S, by construction. If all $B[h_i(y)]$ are set to 1, it is assumed that y is in S, hence a BF may yield a false positive. The probability of a false positive f can be tuned by choosing the proper values for m and k. It is a well-known result [79] that the minimum f is obtained for $k = (m/n) \ln 2$. In this configuration, all bits $B[1], ..., B[m]$ are set or cleared with probability $p = 1/2$ (thus, roughly, the same number of ones and zeros are present in the BF).

Many works about BFs have been presented, and the major improvements are compressed BFs [82], distance-sensitive BFs [83], dynamic BFs [84], space-code BFs [85].

As previously stated, BFs do not allow insertion and deletion of an item in the set. Therefore, CBFs have been introduced, which use m fixed size bins instead of m single bits of presence. When an item is inserted (or deleted), the corresponding counters are incremented (or decremented).

However, CBFs present the problem of counters overflow, which has to be considered in the design. Although for most network applications four bits long counters are sufficient [78], the distribution of counters load across bins changes dramatically (according to Poisson arrivals [79]), suggesting that four bits per bin is a safe choice and that a certain amount of compression is achievable. Moreover, by using a fixed number of bits, the problem of counters overflow in CBFs is not completely solved. It results in a lack of adaptiveness and inaccuracy of stored information.

In order to waive these limitations and achieve better performance, many improvements to CBFs have been done. Mitzenmacher [82] shows that unbalancing the number of ones and zeros in a standard BF can help achieving a good compression ratio before transmission (e.g. for web-caching application). This way, by keeping the same amount of bits of the uncompressed case, it is possible to either reduce the false positive probability or use a lower number of hash functions.

Spectral Bloom Filters (SBFs) [86] are an extension of standard BFs to multi-sets, allowing estimates of the multiplicities of individual items with a small error probabilities. The word "Spectral" means that SBFs allow only filtering of elements whose multiplicities are within a requested spectrum (therefore they do not preserve bins from overflow in a conclusive way). The main goal of SBFs is the optimal counter space allocation, so they dynamically vary the size of their counters in order to minimize the number of necessary bits. To achieve this flexibility, SBFs include additional slack bits among the counters and complex index structures, that increase both memory needs and access time as compared to standard CBFs. Finally, SBFs introduce techniques for filter compression based on Elias code, that reduce the transmission size of data structures but increase again the processing load.

Dynamic Count Filters (DCFs) [87] are data structures designed for speed and adaptiveness in a very simple way. They do not require the use of indexes, thus obtaining a fast access time, and avoid permanently counters overflow. DCFs consist of two different vectors: the first one is a basic CBF with counters of fixed size, the second one is the Overflow Counter Vector, which has a counter for each element of first vector that keeps track of the number of overflow events. The size of counters in Overflow Counter Vector changes dynamically to avoid saturation; this implies that, for each update, a structure rebuilding is required. Moreover, the decision of having the same size for all these counters (for direct access entails that many bits are not used. Therefore, this solution can be improved, especially in terms of memory consumption.

The d-left CBFs (dlCBFs) [88] are simple alternatives based on d-left hashing and fingerprints of bins. They do not rely on the principles of Bloom Filters, but they offer the same functionalities. The dlCBFs use less space, generally saving a factor of two or more for the same fraction of false positives, and the construction is very simple and practical, much like the original Bloom Filter construction. Indeed the simplicity in constructing and maintaining data structures is maybe the greatest contribution of [88] as compared to previous works. Moreover, even dlCBFs have the limitation of potential counters overflow and the need for an additional fingerprint for each bin in the data structure.

A successive proposal [89] advocates the use of rank indexing to achieve compact representations of BFs and CBFs through a hierarchical construction. The main idea of this proposal is to implement a CBF as a hash table where a fingerprint (hash value) for each key is stored. Even if the data structure does not actually perform any counting operation, dynamic insertions and deletions from the set are supported. The authors use several layers of bitmaps to avoid the overhead associated to the canonical pointer based implementation. While the use of multilayer bitmaps suggests a similarity with our work, its focus is significantly different, as it does not really provide "counting" functionalities and it cannot support multi-sets.

The memory utilization is the parameter that is better taken into account in this work. As previously mentioned, there are several cases where network bandwidth is still expensive and transmission size becomes a fundamental parameter (e.g., Web cache sharing or P2P applications). Moreover, although memory appears plentiful today, there are many hardware architectures used in network devices (e.g. Network Processors) that may take advantage of using very space-efficient data structures, in terms of both performance and costs. Indeed, memory saving can greatly speedup a device by requiring rare access to slower off-chip memory; further, while ordinarily DRAM memory is cheap, fast SRAM memory and especially on chip SRAM continue to be comparatively scarce. All these issues have led our research, which had the target of an efficient and practical data structure for CBF.

2.5.2 Theoretical Results

In this subsection we present the main theoretical results on the CBF counter overflow probability and on Huffman coding of bin counters that will be the basis of the data structures proposed in the rest of the section.

The following classical result [78] on CBF gives a bound on the overflow probability $P(\varphi \geq j)$ that is widely adopted to design the bin size:

$$P(\varphi \geq j) \leq \left(\frac{enk}{jm}\right)^j \tag{2.6}$$

However, (2.6) is pretty loose; the next theorem 1 presents a tighter bound for $P(\varphi \geq j)$.

Lemma 1. *Let φ be a CBF counter value and $\alpha = \frac{nk}{m-1}$. If $\alpha < 1$, the function $\chi(j) = P(\varphi = j)$ is a monotonically decreasing function.*

Proof. The probability of the event $\{\varphi = j\}$, for $j \geq 1$, is given [78] by:

$$\chi(j) = \binom{nk}{j}\left(\frac{1}{m}\right)^j\left(1-\frac{1}{m}\right)^{nk-j}$$

The ratio between two consecutive values is:

$$\frac{\chi(j+1)}{\chi(j)} = \frac{nk-j}{j+1}\frac{1}{m-1} < \frac{\alpha}{j+1} < 1 \tag{2.7}$$

which gives the proof. $\square$

For $k = (m/n)\ln 2$, $\alpha = (m \times \ln 2)/(m-1)$. α is less than 1 for $m > (1-\ln 2)^{-1} \approx 3.26$. In the CBFs, the previous condition is always satisfied, since $m \gg 1$.

Theorem 1. *Let φ be a CBF counter value and $\alpha = \frac{nk}{m-1}$. If the number of hash functions is chosen so as to minimize the probability f of false positive (i.e., $k = (m/n)\ln 2$), then:*

$$P(\varphi \geq j) < \frac{\alpha(j+1)}{j(j+1-\alpha)}P(\varphi = j-1)$$

Proof. By repeatedly applying eq. 2.7:

$$P(\varphi \geq j) = \sum_{i=j}^{+\infty} P(\varphi = i) < P(\varphi = j) \sum_{i=0}^{+\infty} \frac{j!\alpha^i}{(j+i)!} \tag{2.8}$$

The right hand sum of (2.8) can be bounded as:

$$\sum_{i=0}^{+\infty} \frac{j!\alpha^i}{(j+i)!} < \sum_{i=0}^{+\infty} \left(\frac{\alpha}{j+1}\right)^i = \frac{j+1}{j+1-\alpha}$$

to finally obtain:

$$P(\varphi \geq j) < \frac{\alpha(j+1)}{j(j+1-\alpha)} P(\varphi = j-1) \tag{2.9}$$

□

Corollary 1. *Under the previous results, if* $\alpha < 1$*:*

$$P(\varphi > j) < P(\varphi = j-1) \tag{2.10}$$

Proof. From (2.8), by changing the lower limit of the series from 0 to 1, we obtain:

$$P(\varphi > j) < \frac{\alpha^2}{j(j+1-\alpha)} P(\varphi = j-1) \tag{2.11}$$

Then, considering that $j \geq 1$, $\alpha^2/(j(j+1-\alpha)) < 1$. □

Lemma 1 allows to approximate $P(\varphi = 0)$ and $\mathbb{E}[\varphi]$:

$$1 = \sum_{j=0}^{+\infty} P(\varphi = j) \simeq P(\varphi = 0) \sum_{j=0}^{\infty} \frac{\alpha^j}{j!} = P(\varphi = 0)e^{\alpha} \tag{2.12}$$

Then $P(\varphi = 0) \simeq e^{-\alpha}$. As for the expectation of φ we get:

$$\mathbb{E}[\varphi] = \sum_{j=0}^{+\infty} jP(\varphi = j) \simeq P(\varphi = 0) \sum_{j=0}^{\infty} j\frac{\alpha^j}{j!} = \alpha \tag{2.13}$$

If the CBF minimizes f, $\mathbb{E}[\varphi] \simeq \ln 2 = 0.693$, which is a a very tight approximation in several cases.

It is interesting to see that, as shown in fig. 2.23, the previous bound can be much tighter than the widely used (2.6). For instance, if $n = 1000$, $k = 10$ and $m = nk/\ln 2$, eq. (2.6) yields $P(j > 15) \leq 1.37 \times 10^{-15}$ while our bound produces

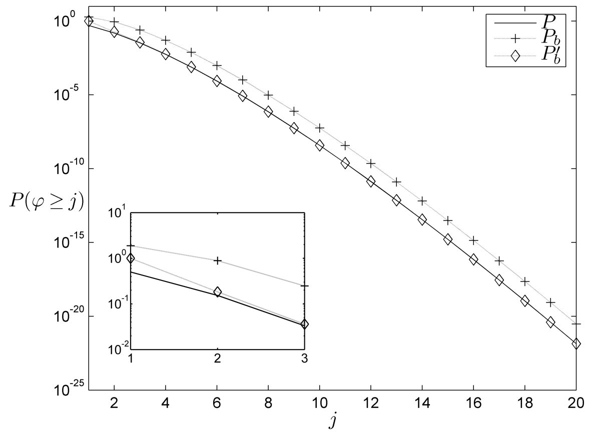


Figure 2.23: Bounds comparison for $n = 1000$, $k = 10$ and $m = nk/\ln 2$. P is the actual $P(\varphi \geq j)$, P_b is the well known (2.6) while P'_b is that provided by the theorem 1. In the smaller graph, a zoom on the contour of $j = 2$. P'_b is always tighter than P_b.

$P(j > 15) < 1.51 \times 10^{-16}$, with a gain of an order of magnitude. Moreover, the results of this first theorem are the basis for the following one.

Observation 1. *Let $H(\sigma)$ be the Huffman coding of σ, $len(\cdot)$ the "bit-length" operator, φ a CBF counter value; then:*

$$len(H(\varphi)) = \varphi + 1$$

Indeed, Huffman codes can be obtained by using a binary tree. The tree is constructed from a list of N nodes (symbols) whose weights correspond to the symbol probabilities.

The whole procedure is the following:

- let x and y be the two nodes with the lowest weight;
- x and y are aggregated into a parent node whose weight is set to the sum of the two nodes;
- the parent node replaces x and y in the list.

These steps are repeated until the list contains one node only.

To perform Huffman coding of CBF bin counters, we first construct a tree whose nodes $X_0, ..., X_N$ correspond to the possible values of the counters $j = 0, ..., N$; the weight of the j-th node is set to $P(\varphi = j)$. Let L_τ be the list of nodes at step τ and let X_τ be the parent node to be created at this step. Suppose we have $L_\tau = \{X_0, X_1, \ldots, X_{N-\tau-1}, X_{\tau-1}\}$; the weight of the parent node $X_{\tau-1}$ created at the previous step is $P(X_{\tau-1}) = P(j > N - \tau - 1)$.

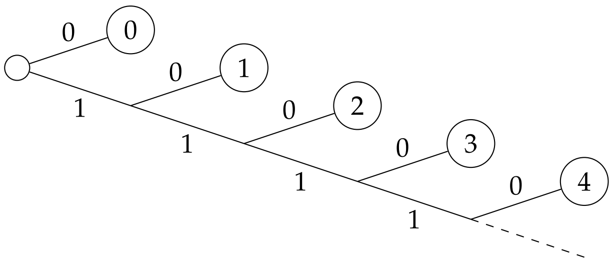


Figure 2.24: A Huffman tree for the CBF bin counters.

By using the result of corollary 1, we obtain:

$$P(X_{\tau-1}) < P(j = N - \tau - 2)$$

Moreover, the previous inequality also implies that $P(X_{\tau-1})$ is smaller than any of the values in the set $\{P(X_0), \ldots, P(X_{N-\tau-2})\}$. Then, at step τ, the nodes with the smallest weights are $X_{\tau-1}$ and $X_{N-\tau-1}$ and they shall be aggregated into the parent node X_τ. Thus:

$$L_\tau = \{X_0, ..., X_{N-\tau-2}, X_{N-\tau-1}, X_{\tau-1}\} \Rightarrow$$
$$\Rightarrow L_{\tau+1} = \{X_0, ..., X_{N-\tau-2}, X_\tau\}$$

The resulting tree turns out to be completely unbalanced (i.e., the depth of all N nodes is given by the sequence of the first N naturals) such as the one of fig. 2.24. Therefore, the depth of node φ is $\varphi + 1$, i.e. the encoding of the value φ of a CBF counter is $\varphi + 1$ bit-long. This result, which comes in turn from the results of theorem I, is one of the basic principles of our structures.

2.5.3 Huffman Counting Bloom Filters

The target of our data structures is an improvement of CBFs by avoiding counters overflow and reducing memory needs. The drawback, as we will see below, is a very slight increase of complexity for the insertion/deletion of an element.

The first step towards the above mentioned target (that will be fully accomplished through the layered structure presented in the next subsection) involves the use of Huffman coding in CBF. The result is Huffman Counting Bloom Filter (HCBF); in order to introduce this data structure, we begin by recalling Spectral Bloom Filters [86].

They use a memory-efficient structure that encodes any bin with Elias coding. This way, bins do not have a fixed position and, for all k hash functions, we have to find the right bin it points to by looking up a certain amount of words. Lookup implies to decode a number of bins until the right one is found. Moreover each insertion and deletion imply a potential shift of the whole structure.

To simplify these operations, SBFs divide the entire structure in subsegments and

$$\underbrace{\overset{w_i}{\boxed{0111011010011011}}}_{popcount=10}\overset{w_{i+1}}{\boxed{1101001011001\cdots}}$$

$$\Downarrow$$

$$16-10=6 \text{ symbols in } w_i$$

Figure 2.25: Example of fast lookup through popcount.

use a set of tables in aid to the lookup. In addition, a certain number ε of empty bits (called slack bits) are inserted to reduce shifts operations for insertions and deletions. Elias compression scheme is a perfect choice when dealing with large numbers, such as those of multiset membership query applications. However, for smaller values (recall that in a regular CBF, 16 is widely considered as a high loose bound), other codings can perform better. By leveraging on Observation 1 of the previous subsection, our proposal is to encode a number σ with σ consecutive ones and a trailing zero (fig. 2.24). This way, the encoding produces $\sigma + 1$ bits for symbol σ: it is a Huffman coding, as shown in subsection 2.5.2. This is a major advantage since Huffman is the minimum redundancy coding for independent symbols such as the bins of a CBF.

Moreover, our coding scheme allows an easy lookup since most processors provide an instruction that counts the number of bits set to one in a word (*popcount*). By taking advantage of such an instruction, we do not have to decode each value we find during lookup, but simply count the number of cleared bits in a word. The number of cleared bits is the number of symbols encoded in that word (see example in fig. 2.25). Clearly, we still have to perform a shift for each insertion or deletion and we need a table to speed up lookup but the total size of the structure is very close to the minimum (given by the entropy of all symbols).

2.5.3.1 Size

In order to simplify the operations and reduce the cost of lookups and insertions/deletions, we group the bins in B blocks of D bins (with few slack bits) and we address the blocks with the table. The average size of the HCBF is:

$$\mathbb{E}[S] = m(1 + \mathbb{E}[\varphi]) + B\left(\varepsilon + \log_2\left[(m - D)\left(\varphi_{max} + 1\right)\right]\right)$$

where ε is the number of slack bits kept at the end of each block. The last part of the above formula takes into account the table size. The table is addressed by the first $\log_2 B$ bits of the hash, the remaining bits represent the bin index. Each entry of the table represents the starting address of the corresponding block thus requiring less than $(m - D)(\varphi_{max} + 1)$ bit.

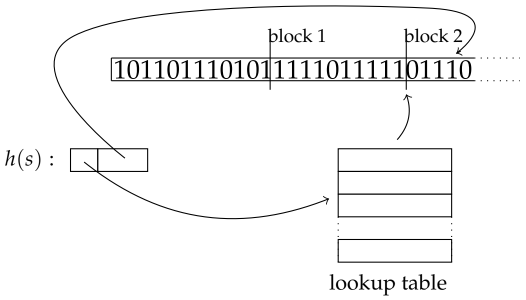


Figure 2.26: An example of HCBF.

2.5.3.2 Lookup

As for operation complexity, a lookup requires k hash functions and, for each of them, a check in the table and a search in the corresponding block for the bin we need (see fig. 2.26). Thus, on average, $D/2$ bins have to be looked and $W/(\mathbb{E}[\varphi]+1)$ bins will be found in a word of W bits. The overall average number of operations for a lookup is then:

$$\omega = k\left(\frac{D(\mathbb{E}[\varphi]+1)}{2W}\right)$$

As shown in subsection 2.5.2, $\mathbb{E}[\varphi] \simeq \ln 2$. Therefore, the average number of operations for a lookup is constant and its complexity is $O(1)$.

2.5.3.3 Insertion/Deletion

In order to insert a new element, we need to perform a lookup and to add a “1” digit for each bin in the code. This corresponds to shifting all the bits at the bin’s right by one position and a table update. Thus, for all insertions, the number of operations is:

$$\omega = k\left(\frac{D(\mathbb{E}[\varphi]+1)}{W}\right)$$

It is straightforward to see that, since even deletion requires a lookup and a shift, the overall cost is the same as insertion. The complexity of these operations is $O(1)$, as for lookup.

2.5.4 MultiLayer Compressed CBF

The drawbacks of the algorithm described in sec. 2.5.3, as well as SBF, are related to the memory wastage due to slack bits and to the complexity of a searching based lookup (even if aided by index tables).

In the following, the MultiLayer Compressed Counting Bloom Filter (ML-CCBF) is presented, which is a CBF that reduces the memory requirements and the complexity of lookup. The idea is to explode the CBF along another dimension, hence creating a *multilayer* structure, where, for each encoded symbol, a bit per layer is stored. This construction, in conjunction with the Huffman coding defined in sec. 2.5.3, provides a stack of bitmaps $(L_0, ..., L_N)$, where the first layer L_0 is a standard BF. The other layers are built and modified dynamically when needed. The relationship between ML-CCBF and the previously described HCBF can be expressed in a few words by saying that ML-CCBF is, somewhat, the rotated version of HCBF, with all bits representing the Huffman coded values of counters in HCBF placed in ascending layers.

To the best of our knowledge, although a limited degree of hierarchy is sometimes obtained by adding a CAM [90] or another counter [87], this is the first attempt to introduce the idea of a hierarchy of arrays in CBFs, which results in a *multilayer* structure where counters may span over different levels.

Let *popcount(u)* be the number of 1s in the bitmap $(0, ..., u-1)$; the construction is as follows:

- L_i keeps all the i-th binary digits of our Huffman encoded counters;
- on L_i, the j-th bit belongs to the counter whose *popcount* on L_{i-1} is j.

Figure 2.27 shows an example of a ML-CCBF. In the example, we are counting a bin φ for symbol σ. The bin at layer 0 is pointed by the hash function $h(\sigma)$. The number of ones before $h(\sigma)$ is computed (i.e. $popcount(h(\sigma)) = 5$) and used as index for layer 1. The procedure is repeated until we find a "0" digit (that is the end of the code). Therefore the resulting Huffman code for the counter is 1110, which corresponds to value 3.

2.5.4.1 Complexity and properties

One of the most significant advantage of our algorithm is that it is an extension of a standard BF. Thus, the lookup is as simple and fast as in a standard BF as we need to check only bits at layer 0. Therefore the lookup complexity is $O(1)$.

Instead, for insertions and deletions, we need to explore different layers in the structure. We refer to m_i as the number of bits in layer i. The size of layer i can be obtained as:

$$m_i = m_0 P(\varphi \geq i)$$

The above formula provides a useful mean for dimensioning the overall data structure. As the (binomial) distribution of counters is known, the maximum length of each layer can be estimated and the corresponding memory allocated accordingly. Also, the formula allows to allocate the number of levels as well, by selecting the number for which the probability of overflow is negligible. In addition, when multiset has to be supported, the maximum cardinality for a key has to be taken into account.

Since jumping one layer up requires a *popcount* on a potentially large number of bits, we divide all layers in blocks of the same bit-size D and add a table for

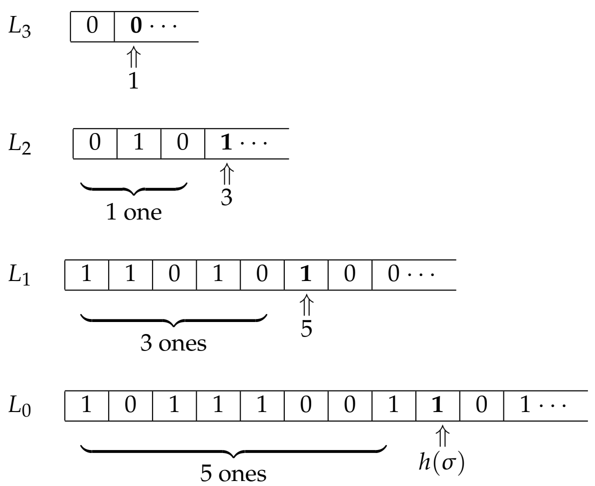


Figure 2.27: ML-CCBF example. The resulting Huffman code for φ is 1110.

each level. When computing *popcount*(u_j) at layer j, the first $\log_2(m_j/D)$ bits of u_j are used as index to table j. Each entry of the table represents the number of ones preceding the start of the block. Thus, if W is the number of bits in a word, the actual *popcount* operation works only on less than D/W words. Therefore, the average cost of a *popcount* is $1 + \frac{D}{2W}$.

Algorithms 12 and 13 show the pseudocode for insertion and deletion procedures in a ML-CCBF. Both operations require, for all k bins, the complete lookup of multiplicity (by exploring a certain amount of layers), a shift by one position and the update of the last explored table. Such an update simply consists of an increment or a decrement on a limited number of entries. Therefore, the average number of operations for insertion and deletion is given by:

$$\omega = k\left[\mathbb{E}[\varphi]\left(1 + \frac{D}{2W}\right) + 2\right]$$

Once again, $\mathbb{E}[\varphi] \simeq \ln 2$, thus the average amount of operations is fixed and the complexity for insertion/deletion is $O(1)$.

A major advantage of ML-CCBF over HCBF and SBF comes from having update and lookup operations decoupled: all insertions/deletions work with higher layers or may flip some bits in the bottom layer 0, requiring no shift nor enlargement of layer 0. This means that we can still perform lookups during dataset-updates if we just take precautions (by means of mutexes) when dealing with changes in the bottom layer (the BF).

Algorithm 12 The insertion of an element in a ML-CCBF

```
1:  for $i \leftarrow 1, k$ do
2:      $j \leftarrow 0$
3:      $u_0 \leftarrow h_i(s)$
4:      while $(L_j(u_j) = 1)$ do
5:          $u_{j+1} \leftarrow popcount(u_j)$
6:          $j \leftarrow j + 1$
7:      end while
8:      $L_j(u_j) \leftarrow 1$
9:      $u_{j+1} \leftarrow popcount(u_j)$
10:     $j \leftarrow j + 1$
11:     $L_j(u_j + 1, \ldots, m_j + 1) \leftarrow L_j(u_j, \ldots, m_j)$
12:     $m_j \leftarrow m_j + 1$
13:     $L_j(u_j) \leftarrow 0$
14:     $UpdateTable(L_j)$
15: end for
```

Algorithm 13 The deletion of an element in a ML-CCBF

```
1:  for $i \leftarrow 1, k$ do
2:      $j \leftarrow 0$
3:      $u_0 \leftarrow h_i(s)$
4:      while do$(L_j(u_j) = 1)$
5:          $u_{j+1} \leftarrow popcount(u_j)$
6:          $j \leftarrow j + 1$
7:      end while
8:      $L_j(u_j, \ldots, m_j) \leftarrow L_j(u_j + 1, \ldots, m_j + 1)$
9:      $m_j \leftarrow m_j - 1$
10:     $L_{j-1}(u_{j-1}) \leftarrow 0$
11:     $UpdateTable(L_j)$
12: end for
```

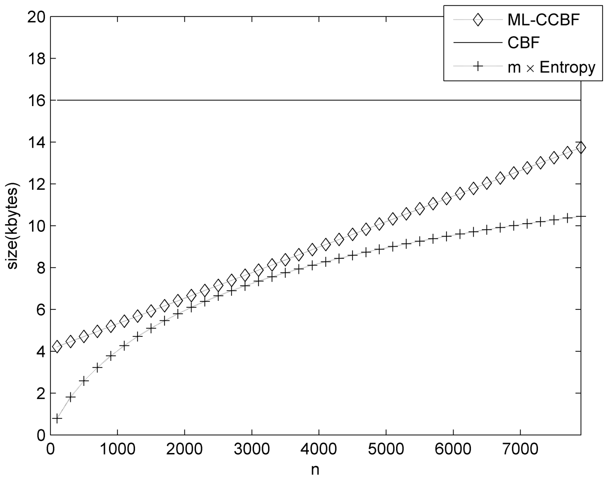


Figure 2.28: Size comparison among ML-CCBF, CBF and $m \times$ Entropy.

2.5.4.2 Size

ML-CCBF is a multilayer transposition of the algorithm shown in sec.2.5.3, with no need for slack bits. Hence, it results in a lower memory requirement:

$$S = m_0 + \sum_{i=1}^{m_0} \varphi_i + \sum_{i=1}^{n_{tab}} TS_i$$

TS_i is the size of the table required for layer i, which needs $n_i = \lceil m_i/D \rceil$ entries of size $\log_2(m_i)$, thus resulting in:

$$TS_i = n_i \log_2(m_i) = \left\lceil \frac{m_0}{D} \right\rceil P(\varphi \geq i) \log_2 \left[m_0 P(\varphi \geq i) \right]$$

The average amount of required memory is then:

$$\mathbb{E}[S] = m_0(1 + \mathbb{E}[\varphi]) + TS$$

A closed form expression for $TS = \sum_{i=1}^{n_{tab}} TS_i$ is not simple to obtain in a general case. However, we use the results of theorem 1 to compute a bound for TS.

If $\alpha = \ln 2$ to minimize the false positive probability, then:

$$TS \leq \left\lceil \frac{m_0}{D} \right\rceil (2 \log_2(m_0) - 1.85)$$

Clearly, as updates occur, upper layers may change in size, thus requiring some

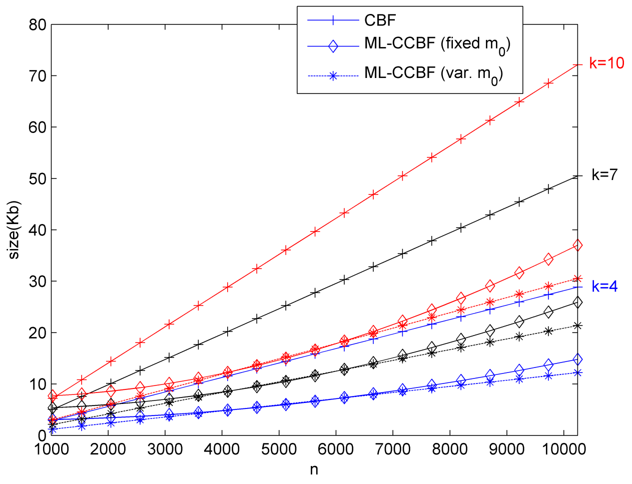


Figure 2.29: Size comparison between CBF and ML-CCBF (for fixed and variable number of bits m_0 for layer 0)

extent of overprovisioning or memory dynamic allocation schemese. However, those layers are designed to be placed in memories whose size is not critical (as opposed to layer 0) and this does not affect the overall scheme properties.

Figure 2.28 shows the comparison among the sizes of ML-CCBF, standard CBF and the minimum amount of bits for independent symbols (BF entropy $= m \times$ entropy), for $k = 10$ and $m = 32768$ (notice that m is fixed regardless of n, therefore the probability of false positives f is not minimized). The memory saving of our method is clear as it approaches the minimum value. Note that the optimal number of elements $n = 2270$ (i.e., the value that minimizes f) minimizes the distance from the BF entropy as well.

Figure 2.29 instead reports, for $k = 4, 7, 10$, the curve of the structure size (in Kbytes) for various number of elements ($n = 1024, \ldots, 10240$) between a standard CBF (constructed so as to minimize false positives) and a ML-CCBF constructed with a fixed layer 0 or with a variable layer 0. For the latter, m_0 has been set as minimizing false positives (i.e., $m_0 = n \times k / \ln 2$), while for ML-CCBFs with a fixed layer 0, m_0 has been set to $n'k/\ln 2$ (with $n' = 4096$). Setting the size m_0 of the bottom layer of ML-CCBF (basically the corresponding BF) as fixed is an easy and fast way to construct the structure but it does not provide the best results in terms of false positives and memory efficiency. However, the figure shows that, even for $n \simeq 2n'$ (i.e., for twice the optimal number of elements), the size penalty for a ML-CCBF with fixed m_0 is limited to less than 20% with respect to an optimal (i.e. with variable m_0) construction. Moreover, it is noteworthy that the difference between the two type of constructions (in terms of size) is minimal, thus showing that the choice of n' (and hence of m_0) in the scenario with a fixed layer 0 is not critical.

Table 2.9: Number of Clock Cycles for Operations in the IXP2800

Operations	Number of cycles
hash	10
popcount	1
shift	1
read/write in local memory	2
read/write in scratchpad memory	60

2.5.5 Comparative Analysis

For the evaluation of the algorithms proposed in this section and the comparison with other known in literature, the Network Processor Intel IXP2800 has been taken as referential hardware architecture. NPs are platforms that offer very high packet processing capabilities (e.g. for gigabit networks) and combine the programmability of general-purpose processors with the high performance typical of hardware-based solutions. The IXP2800 is designed to perform a wide range of functionalities, including multi-service switches, routers, and broadband access devices. It is a fully programmable network processor, characterized by a hierarchy of processing units (a XScale core and 16 32-bit microengines MEv2) and memory devices (4KB of local memory, 16KB of scratchpad memory, besides external memories of the host card). The bigger the memory, the slower the access to it. For more details about Intel IXP2800 we refer to [91].

The hierarchy of memory devices in the IXP2800 reflects the memory architecture of many systems, which present small fast memories and slower big ones. Therefore, although referred to a certain hardware platform, the results of our research are very general.

As shown in tab. 2.9, we have weighted, according to the IXP2800 Hardware Reference Manual [92], the operations of the algorithms in terms of clock cycles for microengines (which are the processors designed to handle fast data path).

In the analysis, we always considered a few clock cycles for emptying the pipeline from all operations. Indeed, all costs reported in the tables are *average costs*. They are not the minimal costs (those than can be achieved when the pipeline is full and no additional costs are payed for switching among operations). For example, to access local memory, if pointers are all set, only 1 clock cycle is needed while the cost of setting pointers is 3 clock cycles. We always conservatively estimated 2 clock cycles penalty; in fact, generally, once the pointer is set, one can access long words (LWs) around with no extra costs (in terms of clock cycles). This is actually a very common case, particularly when working with ML-CCBF and HCBF where typically LWs to be processed are next to each other.

Each algorithm has been simulated and its performance has been measured in terms of memory consumption and processing load for lookup and insertion/deletion. In simulation runs, the total number of data elements is $n = 2000$, $k = 10$, and the number of bins for the main vector is 2.8×10^4, thus minimizing the probability

Table 2.10: Performance Algorithms Comparison

	ML-CCBF	ML-CCBF	CBF	DCF	HCBF	SBF	dlCBF
Size (KB)	6.13	6.13	14.1	14.1	6.42	12.12	5.2
Main structure (KB)	3.52 (local)	6.13 (scratch.)	14.1 (scratch.)	14.1 (scratch.)	5.92 (scratch.)	8.12 (scratch.)	5.2 (scratch.)
Secondary structures (KB)	2.4 (scratch.)	-	-	-	-	-	-
Index tables (KB)	0.21 (local)	-	-	-	0.5 (local)	4 (local)	-
Probability of false positives	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	10^{-3}	1.5×10^{-3}
Lookup (clock cycles)	120	700	700	700	780	801	800
Insertion/Deletion (clock cycles)	1064	1893	710	710	1058	1217	810

of false positives. For the algorithms which divide data structure in subsegments, the number of blocks is $B = 64$. All other parameters are set to obtain about the same probability of false positives among the different algorithms and to be able to manage the same number n of elements. Moreover, for the algorithms which present a hierarchical structure, we have located each substructure in the fastest memory as possible (see tab. 2.10).

Concerning ML-CCBF, the main BF vector L_0 and index tables are stored in local memory, while the remaining vectors in scratchpad. A lookup only requires checking the first vector, therefore only local memory is accessed. For insertion and deletion we still need to explore different layers in the structure, thus both memories are accessed.

For a standard CBF, built with four bits for bin, the overall structure has been located in scratchpad. Therefore lookup, insertion and deletion require accesses to this memory.

With the data of our simulation, DCF (see subsection 2.5.1) does not experiment any overflow of counters in CBF vector. Therefore, Overflow Counter Vector are not necessary and DCF exhibits exactly the same behavior of CBF, in terms of both size and complexity.

Regarding HCBF, we have stored in scratchpad the main structure and in local memory the index tables. As said in subsection 2.5.3, a lookup requires, for each hash function, to check the table in local memory, to search for the corresponding block in scratchpad for the bin we need and to compute a popcount. The same number of operations are required for inserting/deleting an element, with the addition of shifting by one position the bits in the bin, to increment/decrement a counter. Remember that HCBF is a simple alternative version of SBF, which is a structure optimized for multi-set. SBFs use, for values greater than 2, Elias code instead of Huffman code and several more index tables, thus resulting in higher memory consumption and operational complexity.

Finally, the overall unique structure of dlCBF has been located in scratchpad. A lookup requires k hashes, k permutations and k accesses to scratchpad, while an insertion or a deletion needs the same operations for locating the candidate bins, k accesses to scratchpad to find the right bin and, finally, depending on the counter value, either one incrementing (or decrementing) operation or the insertion (removal) of a new fingerprint and its associated counter.

As for Rank-Indexed Hashing, the functionality it provides is somehow different from those offered by the other CBFs. Although it does support dynamic insertion or deletion of elements and it certainly provides good compression, it does not really support counting functionalities and it cannot keep track of multi-sets. Therefore it is not suitable for all of the applications that use CBFs. Moreover, as the lookup operation in this data structure is equivalent of walking through a list, it cannot be parallelized as in the case of the other solutions, where the locations specified by multiple hash functions can be accessed independently by different cores at the same time. For this reason, we believe that a direct comparison (in terms of plain clock cycles) is not fair to the other algorithms, which can be simply sped up through parallelization.

From results in tab. 2.10, it is clear that the solutions proposed in this section show a significant memory saving in comparison with standard CBF and DCF (saving of 56% for ML-CCBF and 54% for HCBF), and also compared to SBF. Instead, there is a memory consumption increase in comparison with dlCBF (from 0.93 KB up to 1.22 KB). Hovewer, our methods, inspired by dynamic approaches (e.g. DCF), avoid in a conclusive way the problem of counters overflow, thus preserving the accuracy of stored information. This makes our data structure suitable for keeping track of multi-sets: inserting the same element several times on a CBF can rapidly bring to overflow, especially with architectures that, as dlCBF, put a lot of effort into reducing the size of the counters (counters can be as small as two bits). With our solution, adding keys just requires adding more layers and results in an increased memory footprint.

Moreover, the introduction of a hierarchical structure allows in ML-CCBF a remarkable decrease of clock cycles for the lookup operation. Indeed, the main structure is stored in local memory, thus enabling lookup by accessing local memory only. Naturally, keeping the whole state required by ML-CCBF in the same cache level reduces the performance boost which is provided by the structure layerization, thus negatively impacting the overall performance: the lookup time is the same of a standard BF and the insertion/deletion time is increased. The membership query is the most frequent operation for these data structures, therefore the reduction of about 83% of clock cycles for lookup is a great outcome. It outweighs the drawback of an increase of 50% of processing for inserting/deleting an element.

Performance results indeed show that ML-CCBF cannot be the best solution when high update rates are requested; this, indeed, is the cost to pay to flexibility that, if on one hand guarantees no overflow, on the other hand requires a few extra operations for inserting (and deleting) entries. Clearly, this suggests the use of ML-CCBF in applications that require very fast lookup but reasonably frequent updates (as in next subsection application)

Finally, note that our HCBF outperforms SBF, in terms of memory consumption and operational complexity. This is an expected result, due to the simplicity of our method and to the use of Huffman code (SBFs are optimized for multi-sets). If compared to the complexity of standard algorithms, HCBF shows a reduction of 13% for lookup and an increase of 45% for insertion/deletion. The different frequency of operations allows to claim that the tradeoff is advantageous.

2.6 iBF: Indexed Bloom Filter

Although BFs have many features that make them attractive for fast and simple applications, their adoption in more sophisticated schemes is prevented by their lack of functionalities and (in some cases) poor performance. As a motivational example, let us suppose we need to classify traffic according to fixed-size substrings of packet payloads at high speed. Let us assume that we are looking for a set of particular strings. We can train a BF with the set of pre-determined strings we are searching, so that the (hopefully large) part of traffic that does not match them can pass through with no additional computation required. However, if the BF returns a match, we need then to check whether it is a false positive and which string has been matched. This requires an additional *exact* filtering stage, which could be implemented as an hash-table. Moreover, while the first stage may help the second hash-table lookup (as, for instance, shown in [93] where a Counting Bloom Filter reduces the number of accesses to the following hash table), the whole lookup remains non-deterministic thus jeopardizing performance if implemented in parallel systems. In order to achieve deterministic lookup times, a perfect hashing scheme ,as shown by Kumar et al. in ([94][64][95]), is very effective. These works propose the adoption of a small fast table of "discriminator" values which, together with the key, are fed to a regular hash function thus removing collisions and achieving perfect hashing. In such a way, in [94] and [64], finite automata are succintly stored and in [95] perfect hashing is achieved with as low as 1 additional bit per key in a double hashing scheme. A BF-like structure is also adopted in the previous section where a Blooming Tree is the basis for the construction of a minimal perfect hashing scheme. However, all these results come at the cost of a quite expensive construction and, unlike the iBF, cannot be adopted in existing applications with minimal effort, as they require major code rewriting of even hardware modifications in order to be effective. Indeed, there solutions require more than a single memory block to be effective as they rely on a number of tables to be accessed at the same time. This implies that, in an existing application, more than just code rewriting is needed: new fast memory blocks and corresponding bus bandwidth must be allocated. The purpose of this work is to show we can use a BF to obtain a perfect hashing scheme by exploiting a certain number of degrees of freedom and relaxing the false probability requirements. In details, we show a quite succinct data-structure, which is a direct modification of a BF that can be implemented in existing applications adopting BF at a negligible cost in terms of code rewriting. The modified BF we construct returns an index for each element of the working set, hence the name *indexed BF* or iBF. The data structure requires $O(\log(n))$ bits per key and k memory accesses per lookup, where k is $O(\log(n))$. A closely related work is the one by Chazelle et al. in [96], introducing Bloomier Filters. Bloomier Filters augment Bloom Filters with the capability of storing any function of the input set. They are therefore more general than our iBF but may require a larger amount of memory.

In short, the main contribution of this work is its novel approach, which exploits a couple of interesting degrees of freedom in BFs, to a widely discussed problem: achieving deterministic perfect hashing in network applications. We believe such degrees of freedom may also be useful for other purposes and the algorithm we propose can be adopted with minor changes in existing applications based on BFs.

Term	Description
singleton bit j	CBF[j]=1
marked bit	a singleton bit cleared to 0. One per element.
index (x)	b bits at the marked bit's left
good BF	BF where each element has at least an singleton bit
well–constructed BF	BF with minimum false positive probability

Table 2.11: Terms and notation used through the work

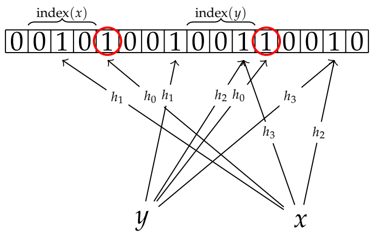


Figure 2.30: The desired data structure

2.6.1 The main idea

The purpose of iBF is to create a perfect hash by simple bit-flipping operations on an already-constructed BF.

As a motivational example, the structure in fig.2.30 shows our desired result: a BF and 2 elements (x and y) are depicted. For each element, one of the 3 hash functions points to a marked bit which, in turn, defines an index at its left. These indexes serve as perfect hash for elements x and y.

The idea of iBF builds upon the following considerations:

1. In a well-constructed BF, if k is large enough (we will show that it must not be larger than $O(\log(n))$) there is at least a hash-function h_i for each item x of the set S that addresses a bit with no collisions (i.e.: where no other $h_j(y)$ falls, $\forall y \in S, y \neq x$). We hereafter refer to BFs with such property as "good" BFs.

2. Bits equal to "zeros" in the BF can be flipped to 1 by only paying a small price in terms of false positives.

These observations basically lead the construction which is, in turn, performed in two steps. The first step marks a bit for each item in the set by focusing on non-colliding bits as suggested by the first observation. The other step exploits the second consideration and flips a number of bits at the left of each marked bit in order to obtain, for each element, a different return value. In the following we describe these operations in greater details.

2.6.2 iBF Construction

2.6.2.1 First step: determine bits to mark

We want to have an univocal index to be returned from the BF for each element x in the set S. To this aim, we take advantage of the first consideration and focus on the "non-colliding" bits in the BF (i.e., bits which have been set by a single element only). In the following we refer to those bits simply as "singleton" (see legend in tab.2.11). A simple way to describe such property is that if we expand the BF to a Counting BF (CBF), singleton bits are those corresponding to counters equal to 1. The first degree of freedom we exploit in this work is used here. By definition, in a good BF, for each element we have at least one of the singleton bits that we can flip to zero, thus marking it. This way, we relax the BF requirements, accepting that an element x belongs to the set if the k hash functions point to k-1 ones and 1 zero. Hence, the false positive probability grows by a factor of 2 (as if we were using a BF with k-1 hash functions), but we earn a way to "mark", for each element x, one of the bits representing it. This is crucial in order to proceed in our construction.

Let us now discuss about the likelihood of the first consideration; in other terms, how probable are good BFs? And what choice of parameters m and k makes a BF good?

The probability for an element x to have at least a singleton bit is simply:

$$\pi = 1 - \left(1 - e^{-\alpha}\right)^k$$

where k is the number of hash functions and α is defined as nk/m. Then, the probability that this property holds for all the n elements (i.e., the probability of a good BF) is:

$$P = \pi^n \simeq e^{-n(1-e^{-\alpha})^k} \tag{2.14}$$

It can be easily demonstrated that the value of k which maximizes P is the same that minimizes false positives: $k = m/n \ln 2$. The reason is simple: let us assume we have $n-1$ items stored in our BF and we add the n-th item. Computing the probability that all the k hash functions point to already-set bits is basically computing the probability of a false positive f. Since we try to avoid this event, maximizing P is the same as minimizing f.

Naturally, we are interested in making P as close to 1 as possible. In this sense, for $\delta \to 0$, we observe that $P \geq 1 - \delta$ if:

$$k \geq \frac{\log n - \log \delta}{-\log(1 - e^{-\alpha})}$$

The main comment is that k must grow like $\log n$ which is quite intuitive, as it makes the structure size behave as Bloomier Filters: $O(n \log n)$. The effect of this inequality is shown in fig. 2.31 where P is reported for different k and as a function of $\alpha = nk/\ln 2$.

Once we assessed the conditions that make well-constructed BFs probable, we can proceed to determine which singleton bit to mark among the ones belonging to

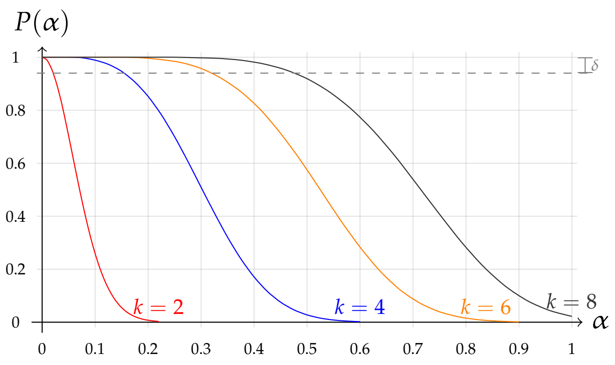


Figure 2.31: Probability of good BFs as a function of α and k.

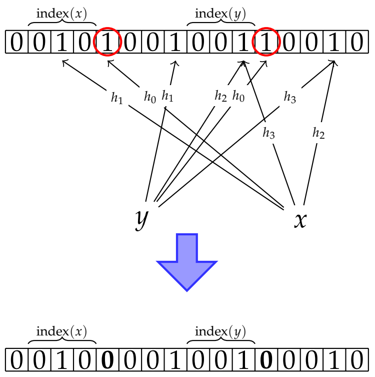


Figure 2.32: Overall scheme. Here the parameters $\varepsilon = 2$ and $m = 16$ are quite over-dimensioned in order to better illustrate the idea.

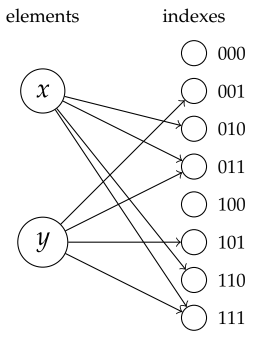


Figure 2.33: The bipartite matching problem.

each item. The choice may be driven according to different metrics which can be combined in order to facilitate the second step. In our experimental tests, we found that a good metric is, for a singleton bit j, the number of zeros at j's left minus the minimum distance between j and other singleton marked bits.

2.6.2.2 Second step: build the index

As a second step, we need to get an index out of the BF for each element $x \in S$. In order to do that, we use the marked bits, and simply choose the index to be defined by a number of $b = \log_2 n + \varepsilon$ bits at the left[1] of them (as shown in fig.2.32 where marked bits are circled). In the following, we will refer to those b bits simply as "indexes". As we have an index for each of the n elements of the set, we can then determine them referring to their corresponding element: for instance, index(x) refers to the b bits at the left of the marked bit for x.

The next step is to make the indexes report different numbers so that we can return them as the result of the perfect hash of the elements we are looking up. This is where the second consideration comes handy: we can exploit the second degree of freedom given by the zeros inside the indexes so that all the elements return a different number. Indeed, by flipping a subset of the zeros within the indexes, each index can provide up to 2^z (where $z = b - \text{popcnt(index)} = \text{no. of zeros}$) different numbers[2]. This problem is an instance of the bipartite graph matching problem (see fig.2.33) which can be easily solved because of the 2^z choices per element that help satisfy the Hall marriage's theorem[97].

We will come back to the theorem after a short discussion on an example describing the idea. In fig.2.33, we present an example of the bipartite matching problem given by the iBF in fig.2.32: the index of the element x is 010, thus we have 2 zeros to flip at will and, in the bipartite graph, we have 4 possible matches (namely $010, 011, 110, 111$); the same goes for y whose possible matches are $001, 011, 101, 111$.

Generally, if p_0 represents the probability of a zero, the mean number of choices per index is:

$$\overline{d} = E[2^z] = \sum_{z=0}^{b} \binom{b}{z} p_0^z (1-p_0)^{b-z} 2^z = (1+p_0)^b \tag{2.15}$$

Of course, for a well–constructed BF, where the number of zeros is the same as the number of ones, the probability p_0 is practically 0.5 and $\overline{d} = 1.5^b$. Since $b \geq \log_2 n$, the mean number of total choices $n \times \overline{d}$ is $O(n^2)$. Therefore the average outdegree of nodes in the bipartite graph is around n. This means we have more links per node than what is needed ($\log n$) to satisfy Hall's theorem with high probability, as shown by Motwani et al. in [98]. Therefore, by means of the Hopcroft-Karp[99] algorithm the bipartite graph matching problem can be easily solved.

In the previous discussion, we have discarded the possibility that two or more indexes could share some bits. For this reason the problem, in real cases, can be

[1]Naturally we could have chosen the right as well

[2]Note that, in a real implementation, we take advantage of the popcnt instruction that computes the number of ones in a register and is available on most architectures.

highly correlated and NP-hard. Indeed, having always more than b bits between two marked bits is an highly unlikely event, and we are definitely going to have super-positions of indexes: two marked bits closer than b bits imply that their corresponding indexes share at list one bit. This means that if we flip those "colliding[3]" bits, then we are actually affecting the match of two or more elements in the bipartite graph, which leads to large difficulties in the construction.

2.6.2.3 Check and restart

It may happen that the Hopcroft-Karp algorithm may not find any bipartite perfect match. This is mainly due to the choices made in the first step. Because of the complexity of the problem, a totally random choice of the singleton bits in step 1 is not a good idea. In our tests, we experimented that *genetic algorithms* are quite useful in this problem. Because of lack of space we do not include all the details of the genetic algorithm we adopted and do not describe the basics of genetic algorithms (interested readers may look at [100]). However, the main step when adopting genetic algorithm for such kind of problems is the definition of fitness. In our scheme, we associated to each iBF a "DNA" of genes defined by a vector of bits of size m. Such a vector D is such that

$$D \oplus BF = iBF \tag{2.16}$$

(where $\oplus$ is the symbol of a XOR operation) and is adopted by the genetic algorithm as starting point for creating a solution. Basically, we start with a vector D which is empty (all zeros). Then we create an "individual[1]" by choosing random singleton bits and setting them in D. Note that, setting a bit in D implies clearing a bit in the iBF (as stated by (2.16)). The individual then passes through step 2 and we compute its fitness and store it. Its fitness is basically defined by the number of matched elements. We repeat this procedure for a number of random individuals which form an initial "generation". For each generation, we adopt a roulette-wheel scheme [100], select individuals according to their fitness and couple them, creating new individuals by means of "cross-over" and "mutation" which, in turn, form a new generation. This means we have other choices of singleton bits to be checked and construct the iBF. The procedure is repeated until an individual with maximum fitness (i.e. a perfect match) is found.

Although the algorithm we adopt is quite general, it provides good results in relatively short time. All experimental tests (with $n \leq 2000$ elements) produced a perfect match in less than 5 seconds on a recent Pentium 4 machine. However this procedure is to be performed off-line and its timing requirements are not strict.

2.6.3 Considerations on iBF

Here we introduce a few considerations on iBFs both regarding their size and their speed.

[3]Please notice the different meaning of "collision" here that refers to bits shared by more than 1 indexes

[1]the individual is the genetic term for a possible solution of the problem

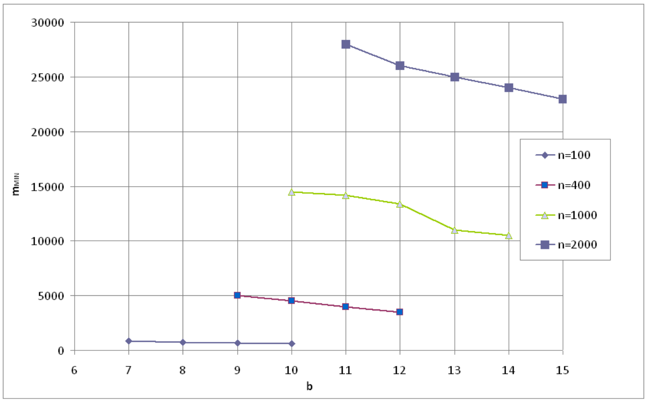


Figure 2.34: Minimal m for the construction of iBF

A first observation can be made on the values of parameters m and k: as discussed above in 2.6.2, k must grow like $\log n$ in order to have a good BF w.h.p. This means that m grows like $O(n \log n)$, which resembles the occupancy of Bloomier Filters[96] but with a lower multiplicative factor. Comparing structure sizes (per item) we have:

- a BF requires $m/n = k/\ln 2$ bits;
- a Bloomier filters needs $k \log_2 n/\ln 2$ bits;
- an iBF needs $m/n = \log_2 n/\ln 2$ bits.

Therefore an iBF requires k times less memory than the corresponding (i.e. same number of items) Bloomier Filter , at the cost of a double false positive rate. Indeed, an iBF behaves as a BF with k-1 hash functions in terms of false positives.

On the other hand, an iBF requires a logarithmical amount of memory accesses for each lookup, which is not optimal but effective in many situations, especially when BFs are implemented in network devices and only few changes may be acceptable to the running code or to the hardware description.

Finally, from the point of view of the overall balance of ones and zeros in the structure, we can say that the two construction stages (first mark some bits by clearing them and then add some zeros to the indexes) somewhat compensate each other. Especially if ε is small and b is hence close to $\log_2 n$, the n indexes are going to provide all the combinations of b bits, which means that, within them, a one is as probable as a zero. This is quite important in order to preserve the false positive probability of a BF with k-1 hash functions, as we see in the experimental results.

2.6.4 Experimental Evaluation

In the following we show the results of the experimental evaluation of iBFs. We first show in fig.2.34 the effect of the number of output bits b on the size m of the

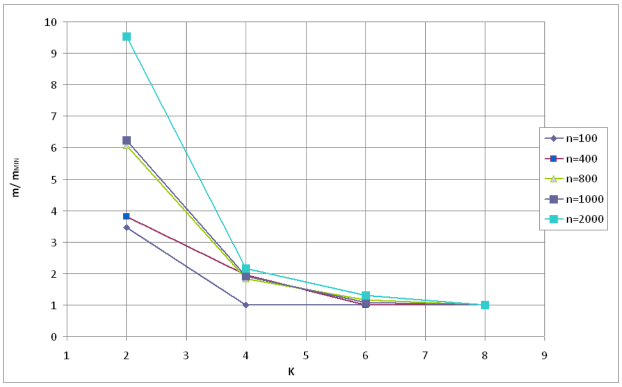


Figure 2.35: Ratio of m over minimal m for the construction of iBF

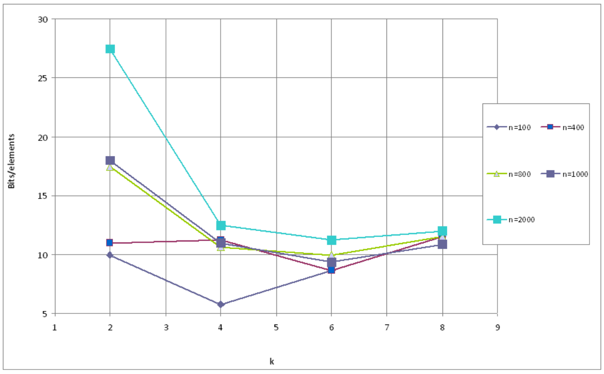


Figure 2.36: Number of bits per element m/n.

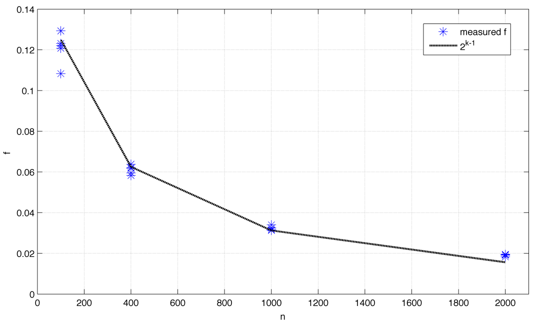


Figure 2.37: False positives in a iBF for $n = 100, 400, 1000, 2000$.

iBF. Enlarging b facilitates the construction of the iBF by allowing smaller structures. Indeed, as b increases, the iBF output grows as 2^b, increasing also the number of links in the bipartite match, which makes the perfect match more probable for small structures. In the graph, values of b are limited as it does not make sense to increase b to value larger than $\log_2 m$. Indeed, if we simple define the output of the iBF as the hash function that points to the singleton bit, we have a fast and simple perfect hash with output domain equal to $m = 2^{\log_2 m}$.

In fig.2.35 we show the behavior of m as k increases. Here the effect of a larger number of choices for a singleton bit is evident as k grows. In the figure, m_{MIN} represents $nk/\ln 2$ which is the value of m that minimizes false positives. As shown in figure, that value of m is also the minimal size of the iBF and it is reachable for values of k that are proportional to $\log n$, as described in sec.2.6.3.

Then figure 2.36 shows the ratio m/n which is the number of bits per element in an iBF. Such a value can be considered as a "cost per unit" for our approach and, again, it reaches its minimal values for values of k which are proportional to $\log n$. This is well justified by the results in the previous figure.

Finally, fig.2.37 shows the amount of false positives we registered by testing the iBF (for $n = 100, 400, 1000, 2000$) with 10 million random queries in 5 different tests. In the graph, the dotted black line represents the theoretical false positive probability (2^{k-1}) and the blue stars are the measured false positive probability value. Measured and theoretical false positive probability overlap for all tested number of elements, confirming our previously stated considerations.

Chapter 3

Network monitoring and testing architectures on hybrid platforms

The extremely challenging nature of network processing often calls for the adoption of dedicated hardware solutions. A number of hardware based appliances for security and monitoring tasks are often to be found in operational networks. Although they meet the desired performance levels, such devices often lack the flexibility which is needed for adapting to fast–changing applications. In order to achieve a better trade off between performance and flexibility, a number of programmable dedicated processors for network appliances has been produced. Different vendors proposed different architectures but, in general, such devices are characterized by high parallelism and a low–level access to hardware resources, including the network interfaces. [101] presents an extensive overview of the network processor architectures. Such processors can usually be accessed as peripherals by a host PC, through a fast PCI bus. Overall, such architectures are appealing for creating hybrid systems, where the actual functionalities are divided between a common CPU and the dedicate engine. A proper partitioning of functionalities, indeed, may allow to achieve a good trade–off between performance and flexibility. We followed such a design principle to implement different types of systems. In section 3.1 we describe the architecture of a traffic generator (named BRUNO) which can support a range of extensible modules for generating synthetic traffic. The generation of packet sizes and inter–departure times happens on the host PC processor, while the actual generation is performed by the network processor, thus guaranteeing maximum performance. In section 3.2, instead, we propose an architecture where a Network Processor is the first capturing layer of a network monitoring architecture, its role being classifying and dispatching traffic across a cluster of software based sensors. This architectural concept is further developed in section 3.3, where a system called "smart probe" is depicted. The cornerstone of such a system is to use a cluster of computers to build a probe which is able to perform complex, application–specific processing on a subset of the captured traffic, so that traffic–related data have to be exported out of the probe only for specific and well–specified reasons. Such an approach guarantees good performance and privacy preservation at the same time. Most of the optimized algorithms proposed in the previous section are an excellent fit for the proposed framework, as they allow to efficiently filter out most of the traffic which is of no interest to the supported applications. Finally, in section 3.4 we describe the architecture of an application–aware switch, which adopts the semantic of regular expressions for specifying switching rules. Such a design choice allows to support the well known Openflow standard and to extend it to fit the needs of specific applications. In order to implement regular expression matching while matching the strict resource requirements imposed by the Netfpga platform, we leverage our optimized compression scheme that we illustrated in section 2.1.

3.1 BRUNO: An Accurate Gigabit Traffic Generator

In the last few years, interest in modern Internet applications has been constantly growing and a significant number of such applications has imposed strict demands on network performance. This has required reliable networks offering high transmission capacity, which in turn has raised the need for network testing to measure performance and reveal possible "weaknesses". Such an evaluation, however, is a very difficult task. Given the high speed of current networks, the simulation of their behavior (for example by means of tools such as the largely diffused *ns2*) is not possible with the proper accuracy: the unavoidable simplifications required by simulations have become unacceptable.

Therefore, the only viable direction to test modern networks is emulation. This requires to generate packet flows which resemble the actual internet traffic, in terms of both data rate and statistical properties. It is obviously a critical task for software tools running on general purpose hardware, especially when dealing with high traffic rates. To address this issue, a very accurate traffic generator, called BRUTE (Browny and RobUst Traffic Engine), has been implemented by our research group [15]. Although such a tool outperform most of its competitors (see section 4.1.1 for a detailed discussion of the state–of–the art in packet generation software), still it's not able to attain line rate.

Network processors (NPs) appear as promising solutions for building flexible yet high–performance traffic generators; systems inspired to such design criterion are presented in works [102] and [103], which have inspired our activity.

This work presents BRUNO (BRUte on Network prOcessor), a traffic generator built on the IXP2400 Intel Network Processor and based on a modified BRUTE version. BRUTE is designed to run on the PC hosting the NP-card and is in charge of computing departure times according to given traffic models. Then, the host PC writes such information in the memory shared with the packet processing units of NP (i.e., the microengines), which, in turn, use these data to generate packets and send them with the right timeliness. The motivation is a smart distribution of tasks according to capabilities and practicality: while it is very easy to program and make a PC "intelligent" and "flexible" enough to provide new functionalities and models, it is quite difficult to do so on a Network Processor which, in turn, has a great brute-force power to sustain and produce high loads of packet rates. The overall application has shown a sustainable rate of 1 Gbps and a great accuracy in models reproduction, guaranteed by a feedback scheme for time correction, thus confirming the goodness of the design.

To date, most of the traffic models implemented in BRUNO are those inherited from BRUTE (Constant Bit Rate, Poisson, Poisson Arrival of Burst). However, the simple APIs provided by BRUTE are also inherited, so that adding custom traffic models is an easy process that involves a minimum amount of programming skills. In addition, a "playback capability" is also available as BRUNO is able to exactly reproduce a libpcap trace and to introduce a scaling factor on the interarrival times for "speeding up" or "slowing down" the real trace.

BRUTE The Browny and RobUst Traffic Engine (BRUTE) [15] takes advantage of the Linux kernel potential in order to accurately generate traffic flows up to very high

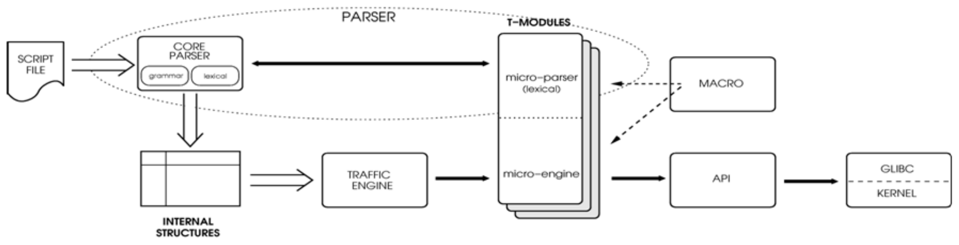


Figure 3.1: Architecture of BRUTE.

bit rates. Because of its excellent flexibility due to a simple scripting language and an extensible architecture, it has been chosen as the basis for the development of our generator BRUNO. BRUTE provides extensibility by means of optimized functions and an interface (API) which enable the implementation in C language of additional traffic sources (named *T-modules*) by users. Because of portability issues (at the expense of a slight loss in terms of latency), it uses POSIX.1B FIFO process type and has been designed as an user space application.

Fig. 3.1 shows the architecture of BRUTE:

- the parser reads script files containing the generation requests;
- such information is then stored into an internal database called *mod-line*;
- the traffic engine examines the *mod-line* entries and instantiates the proper traffic handlers, called *micro-engines*, which are defined into the T-modules;
- all the micro-engines are sequentially executed to generate the requested traffic.

Currently BRUTE is available in [104] along with several traffic patterns: Constant Bit Rate, Poisson, Poisson Arrival of Burst, constant inter-departure time, trimodal ethernet distribution and more. The programming script language is organized in a list of statements, each occupying a single line that consists of an optional label, a command identifier and a sequence of parameters of the traffic class. A little example of the script language is reported in the following:

```
lab: cbr msec=1000; rate=1000;
             daddr=10.0.1.10; len=512;
```

This statement instructs the traffic engine to generate a 1 Kfps CBR traffic flow with 512 bytes long frames for a duration of 1000 ms. When not all parameters are specified, BRUTE uses default values (for instance in this example the default source IP address is assumed). However, in a gigabit ethernet scenario, the highest throughput achievable with BRUTE (1.09 Mpps) is reached only in intermittent bursts (as it will be show in section 3.1.4).

Hardware architectures for generation A few solutions for traffic generation upon specialized hardware architectures have been proposed in last years. Abdo et al. [105] employed an Altera Stratix GX FPGA to develop an OC-48 traffic generator. This tool provides high performance but it presents a lack of flexibility and a restricted set of traffic models. This is mainly due to the difficulties in the definition of new models because of the limited programmability. To the best of our knowledge, only two traffic generation tools have been proposed on Network Processors, both for Intel®IXP2XXX NPs. Such tools are reviewed in sec. 3.1.

This work has been inspired by the need for a generator combining the high flexibility of PC-based tools such as BRUTE and the high performance of dedicated hardware instruments.

Traffic generators on the IXP2400 NP The University of Kentucky developed IXPktgen [102], a generator based on the Intel IXP2400. In spite of the lack of specific informations about this generator, an accurate study of its source code has shown its structure. It employs 4 μEs (working in 8-threads mode) which are used for traffic generation. This implies that, since each thread is statically assigned a single flow, only 32 flows can be generated at the same time. IXPktgen is developed in microcode-assembly and can generate any kind of ethernet frames according to a static parameter file which is read at the startup. Thus the generator is not dynamically reconfigurable.

The Pktgen [103] is a traffic generator proposed by the University of Genova. It is based on the Radysis ENP-2611 board equipped with the Intel IXP2400 NP. It can generate Constant Bit Rate and burst traffic with high throughput. In its design, 5 μEs (working in 4-threads mode with a single flow per thread) are in charge of traffic generations. Therefore it is possible to generate only 20 flows at the same time. The Pktgen code is developed in microC (a C language with several ”intrinsics” for IXP-based specific requirements). Although microC compiler is not as optimized as the microcode-assembler (according to Intel's guidelines [106]), the adoption of a C dialect can simplify a possible porting to other platforms. However, as the previous IXPktgen, in this generator the traffic is statically defined and cannot be changed at run-time.

3.1.1 BRUNO

The target of BRUNO is to combine the flexibility of software-based generators with the high performances achievable only by hardware-assisted applications. Therefore in our architecture we exploit both a general purpose PC and an ENP2611 Radisys pci-board equipped with the Intel IXP2400 NP. The DRAM and SRAM memories on the board, accessible through PCI bus, set up the link between PC host and NP in terms of shared data structures.

The user interface, as well as the parsing process and the creation of flow structures, are assigned to the host PC, while the actual traffic creation is committed to the IXP2400. More precisely, the host PC, through an ad-hoc modified version of BRUTE (that we simply call BRUTE in the following), computes departure times and packet lengths according to the user specifications and stores them in the DRAM. On the

NP side, a μE named Load Balancer (LB) is in charge of reading data from DRAM and applying a correction algorithm on packet departure times, while 4 μEs named Traffic Generators (TGs) create packets for transmission.

Design of BRUNO In fig. 3.2 the design of our solution is depicted. The first μE, represented by the tagged box on the left (Load Balancer), reads the packet timeline that BRUTE (on the PC) writes in DRAM and SRAM. Then it properly modifies and sends it to the μEs called Traffic Generators, through a ring structure. Rings are circular, fast and small FIFO queues allocated into the scratchpad memory of the IXP2400 [92]. Since the scratchpad memory is the only shared memory that is embedded in the NP, such rings represent an optimal solution for the communication among the processing units. Traffic Generators finally send packet transmission requests to the transmitter (TX) μE, which, in turn, is connected to the Load Balancer through the feedback ring.

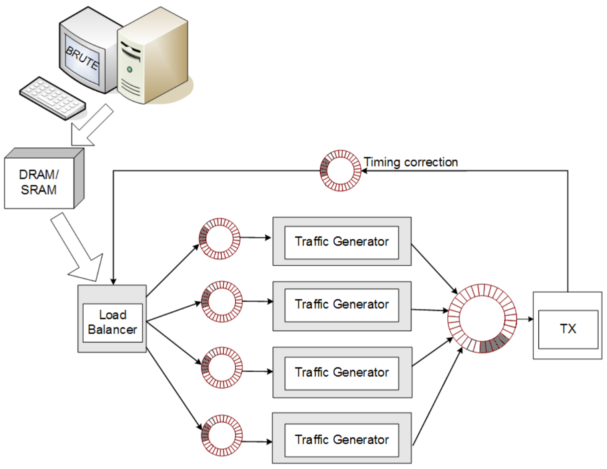


Figure 3.2: Architecture of BRUNO.

The choice of this particular design comes from the need to overcome some limitations that the other NP-based generators have shown. The maximum number of flows that can be simultaneously generated is one of them. This is mainly due to the fixed association between flows and μE threads. For these reasons, in BRUNO a given flow is not strictly associated to a particular thread, thus allowing for an unlimited number of simultaneous flows.

Moreover, if each thread is in charge of a single flow, it is likely that some threads work more than others, or even that all threads on a certain μE work while other μEs just sleep. This is not desirable since a high number of active threads on the same μE could affect the timeliness of packets and hence the precision of the system. In BRUNO each thread processes packets regardless of flows which they belong to and the LB μE guarantees an equal balance of load among TG μEs, by distributing the packet generation requests in a round robin fashion. This way, for instance, whenever a single flow has to be generated, all the threads in the TGs can work for it.

The feedback ring is introduced in order to improve the traffic generation accuracy. Indeed, this mechanism makes the observed real transmission times available to the LB for a comparison with the ideal departure ones. While the packets request are kept in DRAM in a memory window that is continuously refreshed by the PC with new data, the traffic parameters (e.g. L2 and L3 addresses, as we will see later) are kept in SRAM as they need to be accessed very frequently for the creation of each packet.

Load Balancer The LB μE draws data from DRAM related to a packet generation, properly modifies its departure time and then sends them to a TG μE. Fig. 3.3 depicts the structure for a packet generation request (PR), which is loaded in DRAM by the BRUTE application running on the host PC. The first 32 bits contain the interdeparture time, the packet size (16 bits), the pointer to the flow structure in SRAM (*Flow Index*, 15 bits), and the IP version (IPv4 or IPv6, 1 bit) follow.

Interdeparture Time (31-0)		
Packet Size (31-16)	Flow Index (15-1)	Type 0

Figure 3.3: Structure of a packet request (PR).

More precisely, the threads of LB are divided into two groups.

- Even Threads: they convert the departure times from "relative" (as generated by BRUTE) into "absolute" (as required by TGs) and then move PRs from DRAM to the rings in the local memory of μEs. 8 requests are processed at a time since the DRAM is read in blocks of 16 words of 32 bits.
- Odd Threads: they draw PRs from local memory, adjust the departure times according to the feedbacks (the process of departure times correction is accurately explained in section 3.1.3) and forward the new requests to TGs.

Since the timestamp counter in the TGs is limited to 16 bits, the LB should not send to TGs any PRs scheduled more than 2^{16} clock ticks ahead in the future. Therefore odd threads stop when the difference between the time written in the PR and present time is greater than a parameter (*AWAITING_THRESHOLD*), which has to be set at the start of BRUNO application. This parameter can be at most 2^{16} ticks. Since the timestamp counter is increased every 16 clock cycles and the μE clock frequency is set to 600 Mhz:

$$1\,tick = \frac{16}{600 \cdot 10^6} seconds = 26ns$$

The choice of the *AWAITING_THRESHOLD* must be carefully considered. In fact, low values lead to an under-utilization of the Traffic Generators that may not re-

Output_Port (31-23)	Protocol (22-15)	TOS (14-7)	Ind_Type (6-5)	Res (4-0)
Source_Port (31-16)		Destination_Port (15-0)		
Index (31-24)	Total (23-16)	SRC_ADDs (15-0)		
Index (31-24)	Total (23-16)	DEST_ADDs (15-0)		

Figure 3.4: A flow structure.

spond properly to abrupt changes in the traffic. On the other hand, high values of this parameter can easily saturate the scratch rings between the Load Balancer and Traffic.

Traffic Generators As shown in fig. 3.2, 4 μEs are designed for packet creation. The thread of Traffic Generators process a packet at a time, by taking the corresponding PR from the communication ring between LB and TG. Through the field *Flow Index* in the PR, the structure describing the flow which the packet belongs to is accessed. Fig. 3.4 shows an example of such flow structures, which are loaded into the SRAM by BRUTE in the initialization phase, according to the user settings.

Output_Port indicates the physical output port for the flow. *Protocol*, *TOS*, *Source_Port*, and *Destination_Port* provide the corresponding fields of L3 and L4 packet headers. *SRC_ADDs* and *DEST_ADDs* point to two SRAM locations which contain a list of source and destination addresses respectively. *Total* provides the number of these addresses, while *Index* indicates the next address to be read if the choice is made in a linear way (otherwise, in random mode, the proper address is suggested by a random number generator). The two bits of *Ind_Type* indicates the selection mode for both source and destination addresses (which in addition can be IP or MAC addresses).

From these data, a thread is able to create packet metadata and L2, L3 and L4 headers. Then the thread is placed in a state of sleep, until the time to send the packet arrives. At this point, the thread wakes up and sends the packet transmission request to the transmission block, which will provide to transmit the packet.

Transmitter The last stage of the generator is the transmitter. The code used is the one provided by Intel®, as optimized for the transmission. Obviously, the feedback system for time correction has been added. It is executed right before before the last step of the transmission process, in order to measure “real departure times” as truthfully as possible.

System Initialization BRUNO requires that the flow structures and the addresses lists are loaded in the SRAM, as well as the requests for transmission generated by

BRUTE have to be stored in the DRAM, before the application begins to create traffic. In order to obtain a time consistency among the various μEs, which is fundamental for the good feedback functioning, a system synchronization is required. For this purpose, a specific function is included in Load Balancer, Traffic Generators and Transmitter code to force initialization and timestamp synchronization of the various μEs before the regular functioning.

3.1.2 BRUTE-NP communication

The communication between the BRUTE application, which is in charge of generating the packet requests, and the Network Processor, where the traffic generation actually takes place, is performed through the PCI bus. In particular, both the DRAM and SRAM memory banks on the board are accessible through the local PCI bus, which, in turn, is connected to the PCI bus of the host PC through the Intel 21555 non-transparent PCI-to-PCI bridge [107]. Since the address plan referring to the two buses is different (this is the main reason of using a non-transparent bridging), address translation is implemented on such a device (see fig. 3.5): up to three non overlapping intervals in the PCI address space of the host PC (called downstream windows) can be configured to be translated into the corresponding address intervals in the Radisys PCI address space. Every time the 21555 bridge receives a transaction referring to an address falling into one of the downstream windows, it maps such an address into the corresponding address of the Radisys PCI bus and forwards the transaction over it. In a symmetric way, three upstream windows in the Radisys PCI address space can be defined in order to forward transactions from the board bus to the host PC bus.

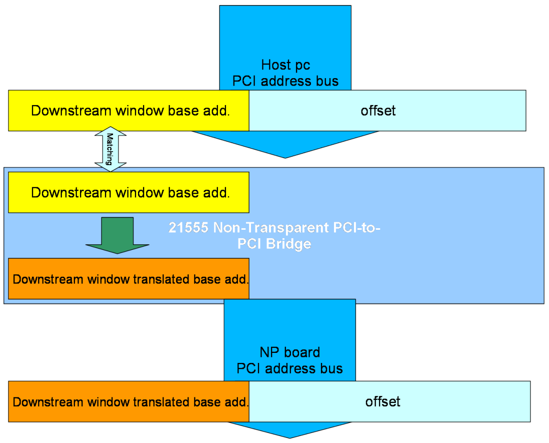

Host pc
PCI address bus
Downstream window base add.
offset
Matching
Downstream window base add.
21555 Non-Transparent PCI-to-PCI Bridge
Downstream window translated base add.
NP board
PCI address bus
Downstream window translated base add.
offset

Figure 3.5: Address Translation.

Different address translation methods are provided by the bridge, but the most simple and efficient one is the direct base translation: a downstream (or upstream) memory window is defined by a base address, and address translation is performed by simply replacing such a base with a corresponding translated base which defines an address region over the target bus. Since the base length is variable, the size of an address window can be defined by the user: in general, the window size may assume values from 4 KB up to 2 GB, thus allowing to completely map each memory bank of the Radisys board.

The memory translation map can be configured by accessing and setting some control registers associated with the non-transparent bridge; in our implementation, this is done by a Linux kernel module inserted in the host PC operating system. After the initialization of such a module, both the SRAM and the DRAM memory banks are accessible as PCI resource regions by the host PC operating system and can be read and written by using system calls referring to memory mapped I/O. In order to offer a simple interface to user applications, our module registers two virtual character devices in the Linux kernel, which are associated to the Radisys DRAM and SRAM banks, respectively. Such devices provide support to the *mmap* access method [108], which allows to register a direct binding between a statically defined physical address region and a user space virtual address region. When a user process accesses a virtual address falling into the mapped area, the virtual memory manager of the kernel directly converts it to the corresponding physical address. This allows user processes to directly access the resources associated with a device, without using any data buffering in the kernel memory. Such a mechanism, which is generally used for accessing high performance devices such as graphical cards, provides the maximum speed for accessing peripheral devices, since no data copying is required. By accessing the two character devices, BRUTE can both configure the parameters defining each generated traffic flow (stored in SRAM) and set the times and lengths for each packet, by writing the corresponding data structure in DRAM.

Synchronization In operational conditions, the DRAM area containing the packet requests must be accessed by both the μEs and the host CPU through the PCI communication mechanism described in the previous section. In order to accurately reproduce the statistical characteristics of a given traffic flow, while the μEs read the metadata and actually generate the packets, new packets lengths and interarrival times must be written in the DRAM memory by the host CPU. Therefore, we need to define a mechanism to cope with the simultaneous presence of readers and writers on the same memory area. In particular, each packet request must be written by the host CPU and read by the NP only once. Let us take as a trivial example a plain Poisson traffic flow: since the packet interdeparture times are independent exponentially distributed random variables, the repetition of a given interdeparture time is not compliant with the flow specification.

For this reason, a synchronization mechanism between the NP and the host CPU has to be implemented. We choose a method that does not rely on the classic interrupt based solutions (typically used for PCI devices) because of the variable and possibly long latencies which are involved in such schemes. On the contrary, we adopt a polling based mechanism. In particular, in our scheme, the CPU takes care of the whole synchronization task: first, ordinary CPUs are generally faster than our NP (the IXP2400 has a 600Mhz clock rate, while ordinary PCs usually work at a frequency of a few Ghz); in addition, the mean rate at which the data structure in the buffer can be read must be of the same order of the packet sending rate (otherwise, in the long run, the internal buffers of the LB μE, where the packet requests are temporarily stored, would overflow). Since, even at full rate, less than a few million packets per second can be sent by the traffic generator, a common PC can easily fill the buffer faster than it is emptied. Besides, in order to avoid contention issues on the host PC, the BRUTE process can be assigned a higher priority in the Linux processor scheduling mechanism, thus guaranteeing that other tasks interfere only marginally with the traffic generation. As a consequence, there is no need for the NP to wait for the CPU, while it is likely that the CPU has to stop to wait for the NP to read the data in the buffer. In addition, the code running on the NP is optimized for maximum performance and implementing waiting mechanisms could lead to a major performance degradation.

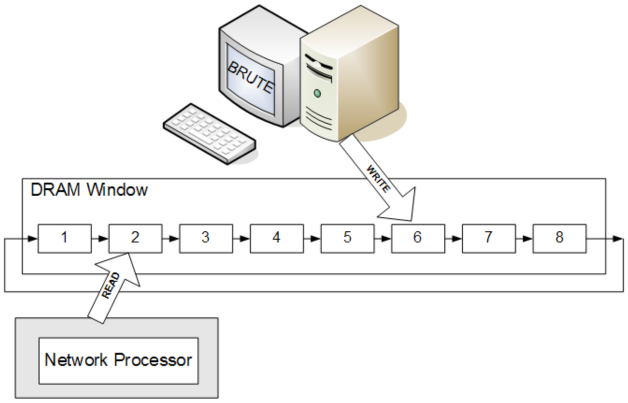


Figure 3.6: DRAM window circular buffer.

The DRAM window containing the packet requests is cyclically read by the Network Processor as a circular buffer. Transferring a large block of data over the PCI bus (and from/to the DRAM) is, in terms of overall delay, more profitable for the host CPU than moving small amounts of packet requests at a time. Therefore, such a circular buffer will be partitioned in blocks containing a given number of data structures (let us say 8); each time either the host CPU or the NP accesses the buffer, a whole block of data structures is read or written. In fig. 3.6, the DRAM window divided into different blocks (arranged in a FIFO circular queue) is depicted.

The NP keeps in its SRAM a pointer to the last block it has read and, in turn, the CPU maintains in its own memory a pointer to the last block it has written. Before performing a write operation, the CPU reads both pointers to check whether the buffer is full. In such a case, the CPU enters a waiting state for a given waiting time and, after that, it checks the pointers again.

We avoid using a polling mechanism that continuously reads the pointer in order to reduce the number of accesses to the NP SRAM. Indeed, contention for access to the memory block could affect the delay of NP accesses to such a memory, and lead to a performance degradation.

The waiting time must be accurately calculated so as to avoid that the NP reads the whole buffer while the CPU is waiting. Since, as already pointed out, the average buffer reading rate must be equal to the average packet sending rate of the NP, good estimate of such a waiting time can be computed by the host CPU as:

$$T_{delay} = \rho \times \frac{B}{R} \tag{3.1}$$

where B is the overall amount of packet requests that can be contained in the buffer, R is the average packet rate produced by the generator (known by the host PC) and ρ is an arbitrary safety parameter smaller than one (if $\rho = 1$ then T_{delay} is the time needed to empty the whole buffer). For very low values of ρ the CPU floods the SRAM with read requests, thus affecting precision. On the other hand, if $\rho \simeq 1$, the CPU may not be able to fill the buffer properly and the system would not be able to respond in time to abrupt changes of the generated traffic. Preliminary experimental results (limited to the PCI communication) seems to confirm that setting $0.1 \leq \rho \leq 0.4$ is a good choice.

The implementation of the busy waiting mechanism over the host PC relies on the real time capabilities which are included in the original BRUTE. It provides busy waiting functions which, by actually counting the CPU clock cycles and by taking advantage of the Linux process scheduling policies, allow to set a waiting period with a fairly good accuracy.

3.1.3 Performance Evaluation

System delays In this section, we analyze the system behavior in order to estimate the goodness of our design in terms of performance. In particular, we will try to understand if the system is able to generate the maximum packet rate for a gigabit ethernet: 1488000 pps (with 64 bytes per packet). As stated by code simulation, the hardest workload among the μEs is in the charge of the Traffic Generators, so we focus on them.

The mean time (hereafter we call "T" the mean times) spent by a thread of a Traffic Generator μE for its overall processing of a packet is:

$$T_c = T_{r,scr} + T_{el} + T_{w,SRAM} + T_{w,DRAM} + T_{wait} + T_{w,scr} \tag{3.2}$$

where $T_{r,scr}$ represents the mean time spent for reading the PR from the scratchring, T_{el} for processing the request, $T_{w,SRAM}$, $T_{w,DRAM}$, and $T_{w,scr}$ for writing metadata in SRAM, the whole packet in DRAM, and the packet transmission request in the transmission scratchring respectively. Finally, T_{wait} represents the time a thread must wait when it is placed in the sleep state, as we have described above.

By using Little's law, we compute the available time budget for the overall processing of a packet by the TGs:

$$T_c \cdot \lambda = n \tag{3.3}$$

where λ represents the load (in our worst case 1488000 pps) and n the number of

$T_{r,scr}$	T_{el}	$T_{w,SRAM}$	$T_{w,DRAM}$	T_{wait}	$T_{w,scr}$
60	200	100	100	250	60

Table 3.1: Mean times for each operation in clock cycles.

entities that take care of packet generation. In particular, in our design, $n = n_m \cdot n_t$, where $n_t = 8$ is the number of threads per microengine and $n_m = 4$ the number of Traffic Generator μEs. With such values, we obtain a time budget for a packet of $T_c \simeq 12900$ clks.

Then we have measured by means of the Develop Workbench the mean times above listed: tab. 3.1 reports these values in terms of clock cycles. Their sum amounts to 770 clks, which is widely within the computed budget. Therefore our system looks able to support the maximum packet rate in a gigabit ethernet, and the experimental results shown in sec. 3.1.4 confirm this analysis.

Timing correction In fig. 3.7 we represent a schematic view of BRUNO as a system with an input (the ideal departure time $t(n)$ for the n-th packet) and an output (the actual departure time $\tau(n)$). This representation comes in handy in order to describe the actual system implementation and also the timing correction algorithm introduced in the Load Balancer μE. The introduction of a correction algorithm is motivated by the large number of phenomena that, in a complex multi-core system such as our IXP2400 NP, could affect the accuracy of the traffic generation. As an example it suffices to say that the latency of each memory access to any SRAM or DRAM is strictly dependent on the number and the state of all the other threads and the number of requests coming from the PCI bus referring to that memory. A large variety of events (that imply memory and bus accesses) also occurs on the XScale core because of the regular OS house-keeping (e.g.: timing interrupts, memory paging and swapping). All these phenomena may affect a number of packet departures because of their duration in time. Therefore they are modeled in our scheme as a noise $\omega(n)$ with a non-null autocorrelation. In addition, we point out that, since the noise represents a sum of different phenomena that introduce delays, its mean value is positive: $\mathbb{E}[\omega(n)] > 0$. Hence the reason for a correction algorithm ($f(\cdot)$ in fig. 3.7).

However, because of the limited instruction set of the μEs and to limit the delay it introduces, our error-correction algorithm must be devised to be fast and simple, requiring the minimal amount of instructions. Therefore we choose an exponential moving average:

$$\varphi(n) = A \cdot \varphi(n-1) + B \cdot [\hat{\Delta}(n-k) - \Delta(n-k)] \tag{3.4}$$

where $\varphi(n)$ represents the correction applied to the n-th packet departure time, $\hat{\Delta}(n-k) = \tau(n-k) - \tau(n-k-1)$ is the measured interdeparture time of the $(n-k)$-th packet (taken from the feedback scratchring) and $\Delta(n-k) = t(n-k) - t(n-k-1)$ is the ideal interdeparture time (kept in local memory) of the same packet. The

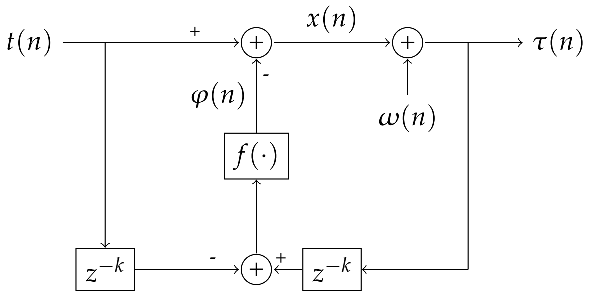


Figure 3.7: Schematic view of BRUNO as a system

parameters A and B are real and positive numbers, with $A + B = 1$, while the term k takes into account the feedback and system delay (shown in fig. 3.7 as z^{-k}). In fact, when the LB is working on the n-th PR, there are a certain number of PRs in the TGs and on the rings, moreover the feedback mechanism is obviously not instantaneous. Notice that the correction function is applied to the difference of interarrival time rather than on the absolute time themselves: indeed, interarrival times must be very precise while the presence of a possible constant offset between $t(n)$ and $\tau(n)$ is not relevant.

In the following, we assume this term k to be fixed and known and we analyze the system in fig. 3.7 as a discrete-time linear system where packet departure times define the time-domain. Notice that, dealing with a discrete time system that evolves on a packet generation basis (i.e., events are not necessarily equally spaced in time), the mathematical approach is still valid though the frequency parameter cannot be interpreted in the standard way and measured in Hertz.

By simple calculations, it is easy do derive the transfer function $H(z)$ that describes the output of the correcting block as a function of the noise process $\omega(n)$ as:

$$H(z) = \frac{\varphi(z)}{\omega(z)} = \frac{Bz^{-k}(1 - z^{-1})}{1 - Az^{-1} + Bz^{-k}(1 - z^{-1})} \tag{3.5}$$

According to fig. 3.7, and by indicating the impulse response of the system $H(z)$ with $h(n)$, one has:

$$\begin{aligned} \tau(n) &= t(n) - \varphi(n) + \omega(n) \\ &= t(n) + \omega(n) - h(n) \otimes \omega(n) \\ &= t(n) + e(n) \end{aligned} \tag{3.6}$$

The error term $e(n)$ associated with the absolute generated times can be calculated as the output of the system:

$$L(z) = 1 - H(z) = \frac{1 - Az^{-1}}{1 - z^{-1} + Bz^{-k}(1 - z^{-1})} \tag{3.7}$$

which receives as an input the sequence of noise $\omega(n)$.

As above mentioned, though, the error sequence of interest is that of the interarrival time, that is:

$$\begin{aligned} \hat{\Delta}(n) - \Delta(n) &= \tau(n) - \tau(n-1) - (t(n) - t(n-1)) \\ &= e(n) - e(n-1) \\ &= \epsilon(n) \end{aligned} \tag{3.8}$$

In other words, we can express the sequence of errors of interarrival times $\epsilon(n)$ in terms of the noise process $\omega(n)$ through the transfer function of the equivalent system:

$$G(z) = \left(1 - z^{-1}\right) L(z) = \frac{\left(1 - Az^{-1}\right)\left(1 - z^{-1}\right)}{1 - Az^{-1} + Bz^{-k}(1 - z^{-1})} \tag{3.9}$$

The effectiveness of the timing correction mechanism can then be evaluated through the characteristics of the equivalent system $G(z)$.

By assuming the noise process as wide sense stationary, it turns out that the average error is null:

$$\mathbb{E}[\epsilon(n)] = \mathbb{E}[\omega(n)] \cdot G(1) = 0 \tag{3.10}$$

and that its power spectral density $S_\epsilon(f)$ is given by:

$$S_\epsilon(f) = S_\omega(f) \left|G(f)\right|^2 \tag{3.11}$$

In lack of any statistical information on the noise, the selection of parameters A and $B = 1 - A$ should be made in order to minimize the energy of the system $G(z)$ so as to minimize the variance of the error $\epsilon(n)$ in the case of flat spectral density of the noise process.

Fig. 3.8 shows the energy of the system $G(z)$ with respect to A for several values of k. From this figure, the choice of $A > 0.5$ seems to be suitable in that the energy approaches 1 and it is very little sensitive with respect to k.

In our experiments (fig. 3.9), though, the noise $\omega(n)$ turns out to be colored as it exhibits stronger components at low frequencies (notice that the peak at $f = 0$ is mainly due to the non zero mean of $\omega(n)$). As shown in fig. 3.10, the frequency response $G(f)$ evaluated for $A = 0.75$ and $k = 12$ (the maximum value of k observed in our experiments) indeed proves a clear high pass behavior, thus effectively filtering out the low frequency relevant components of noise.

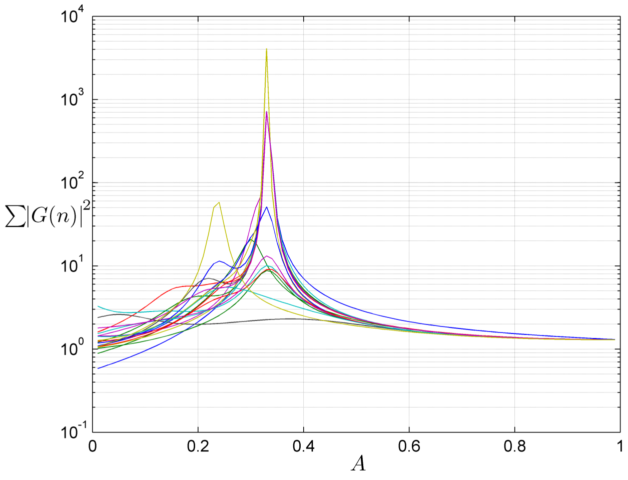


Figure 3.8: Energy of the impulse response of $G(z)$.

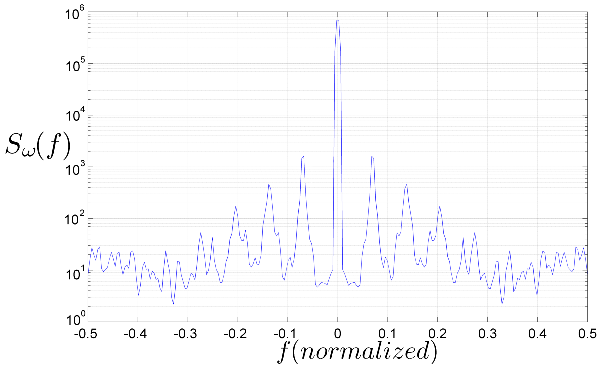


Figure 3.9: Estimated Power Spectral Density of $\omega(n)$.

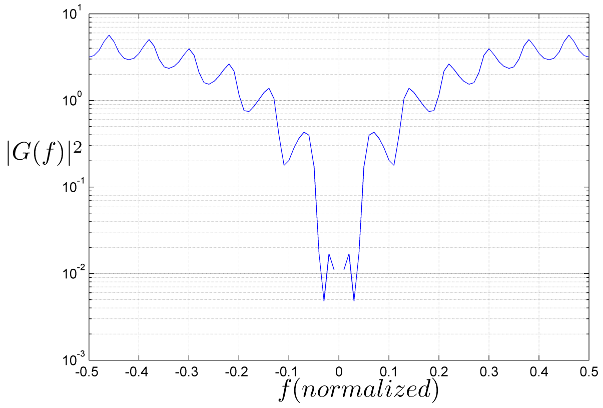


Figure 3.10: Square absolute frequency response of $G(z)$.

3.1.4 Experimental results

We evaluate the actual performance of BRUNO through a wide variety of experimental tests. All the measurements are taken by means of the Spirent AX4000 traffic analyzer [109], which is an ASIC-based tool supporting a precision of the order of nanoseconds.

We test the accuracy of BRUNO in traffic models and synthetic traces reproduction, and illustrate the advantages of the time error correction scheme.

Traffic models To date, the library of synthetic models implemented in BRUNO includes three common traffic profiles with different statistical features and parameters. The modular design of the system, however, allows to add any other models upon need. The traffic models implemented are described in the following along with a performance analysis.

Constant Bit Rate traffic. CBR is the easiest traffic pattern that can be generated as it consists of a sequence of packets with constant interdeparture time. In addition to the common parameters used to build IPv4 packets and UDP headers (i.e., IP addresses, packet size, etc.), the only parameter to be specified is given by the *rate*, that is the number of packets sent per second (or, the inverse of the interdeparture time of packets).

In fig. 3.11, which reports the short term packet rate (calculated as a mean over intervals of 0.10 s), we compare the performance of BRUNO to those of BRUTE in reproducing the CBR model. The improvement of BRUNO is evident, in particular for the accuracy and the maximum achievable throughput: BRUNO is able to generate up to 1488000 pps with a high precision. It is worth noticing that this is the maximum packet rate achievable over a 1 Gigabit Ethernet link with the smallest packet size (64 Bytes); this is clearly the worst case scenario for testing network devices. In such conditions BRUTE provides lower throughput and accuracy.

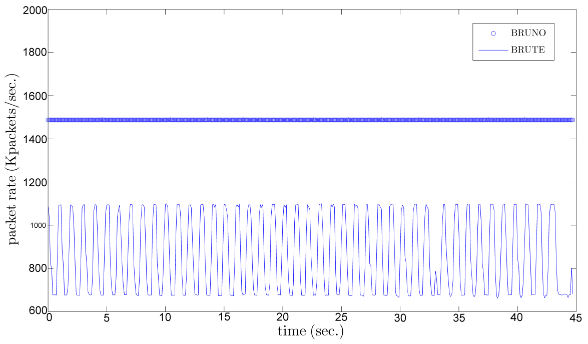


Figure 3.11: CBR traffic: Brute vs Bruno.

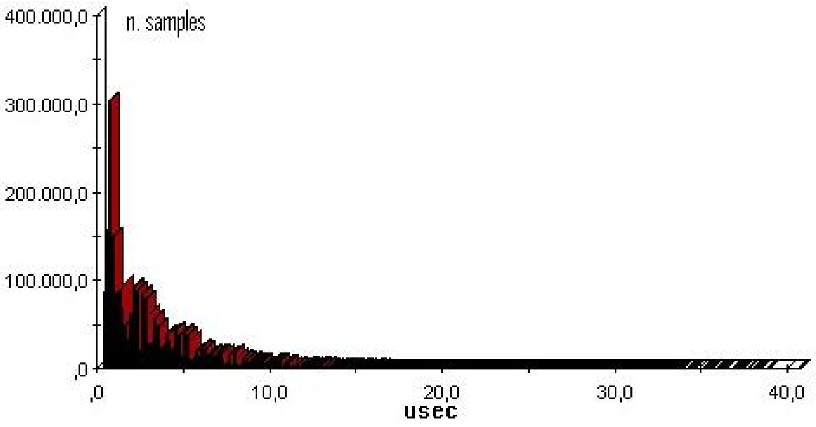
Poisson traffic. The Poisson process is historically one of the most popular traffic models and it is obtained by generating packets whose interdeparture times are independent and exponentially distributed random variables. The parameter λ is used to define the mean number of packets generated per second.

Fig. 12 shows the distribution of a Poisson traffic with throughput= 300 Kpps and $\lambda = 0.03$; the comparison among the traces generated by AX4000, by BRUNO and by BRUTE highlights the improvement of our solution with respect to BRUTE, and especially the capability to properly produce also very small interarrival times (between 0 and 1 μs) thus providing great accuracy at high rates. Moreover, it is worthy noticing that the histogram of interarrival times generated by BRUNO is very similar to that of the commercial AX4000 traffic generator, which is a very expensive hardware solution (hundreds of thousands dollars, while an NP board costs a few thousands dollars).

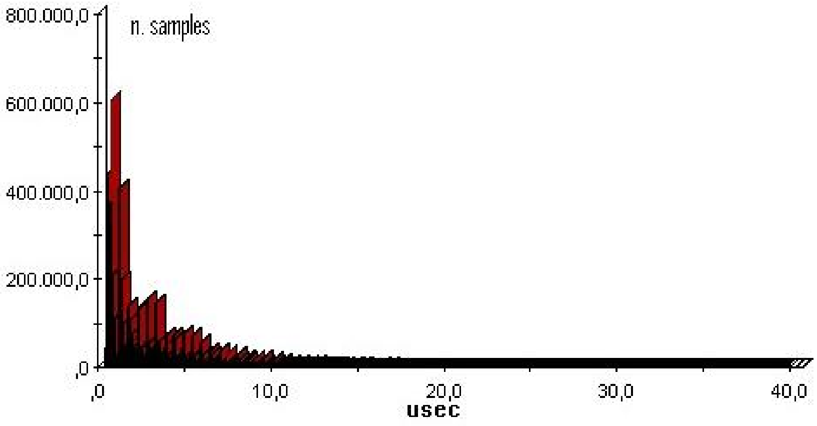
Poisson Arrival of Burst traffic (PAB). The PAB model is the process given by the superposition of CBR bursts scheduled according to a Poisson process of parameter λ, where the duration of bursts are independent and might be modeled as an arbitrary random variable B with distribution $B(x)$. More formally, the instantaneous rate can be written as

$$R(t) = R \cdot N(t) \tag{3.12}$$

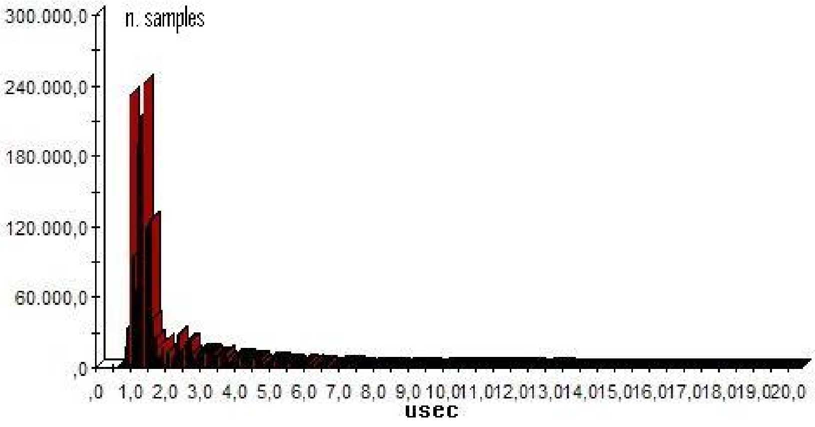
where $N(t)$ is the number of active bursts at time t and R is a scaling factor whose dimension is a data rate (e.g. packet/byte/bit per second). Notice that the random process $N(t)$ is equivalent to the process representing the number of busy servers in a $M/G/\infty$ queue with Poisson arrival rate of parameter λ and service time distribution $B(x)$ (with mean value $\mathbb{E}(B)$).



(a) AX4000



(b) BRUNO



(c) BRUTE

Figure 3.12: Bar chart of interarrival times of a Poisson traffic ($\lambda = 0.03$).

While the marginal distribution of $N(t)$ is given by:

$$\mathbb{P}\left(N(t) = n\right) = \frac{\lambda\,\mathbb{E}(B)\,t}{n!}\,e^{-\lambda\,\mathbb{E}(B)\,t} \tag{3.13}$$

its correlelation features (and so those of the resulting traffic) vary according to the distribution of burst length $B(x)$. In particular, if burst lengths are distributed according to a power–law distribution (e.g., Pareto distribution), such as:

$$\mathbb{P}\left(B > x\right) = 1 - B(x) = \left(\frac{\theta}{\theta + x}\right)^{\alpha} \tag{3.14}$$

depending on the value of α, the resulting traffic process may exhibit either Short Range Dependence ($\alpha \geq 2$ – light tailed distribution) or Long Range Dependence ($1 < \alpha < 2$ – heavy tailed distribution) [110] with Hurst parameter given by:

$$H = \frac{3 - \alpha}{2} \tag{3.15}$$

The parameter θ acts simply as used for time offset. Fig. 3.13 and 3.14 show a slice of about 15 minutes of PAB trace generated with parameters $\alpha = 1.5$ (thus $H = 0.75$), $\theta = 1s$, $\lambda = 300\ \mathrm{s}^{-1}$ and $R = 1000$ packets/s. The associated Variance–Time plot clearly proves the presence of Long Range Dependence. Moreover, the estimated value of H is equal to 0.736, which is pretty close to the nominal value $H = 0.75$.

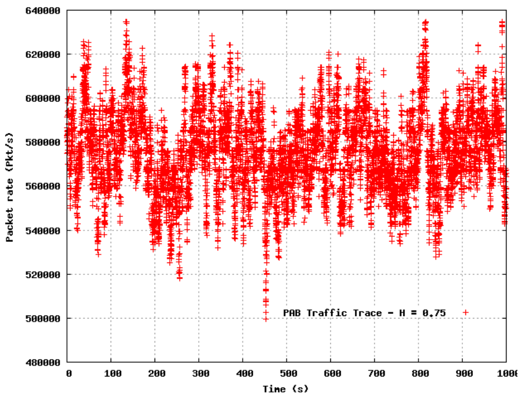


Figure 3.13: PAB traffic profile.

Playback capability The second set of experimental runs aims at illustrating the "playback capability". Our application is able to exactly reproduce a libpcap trace in terms of packet lengths, IP addresses and ports. Moreover, BRUNO gives the pos-

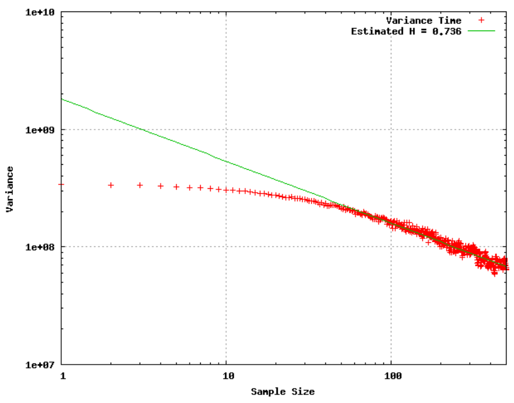


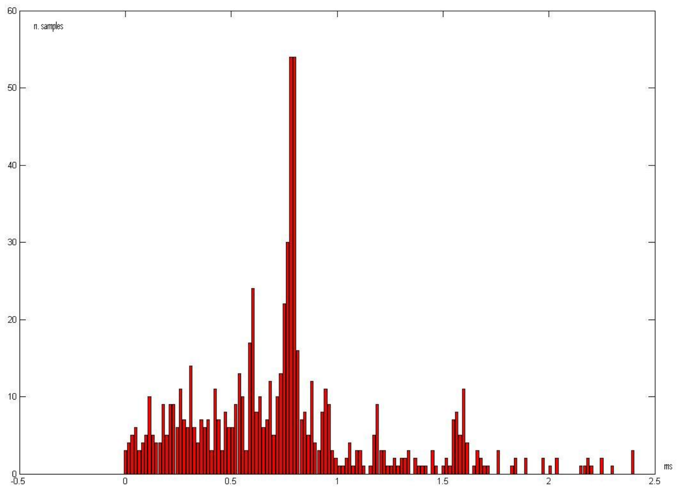
Figure 3.14: Variance time of generated PAB traffic.

sibility of modifying the original speed of the trace by multiplying its interarrival times by a scale factor: the application preserves the time distribution shape of the traffic, while the time scale is "compressed" or "enlarged". This "playback capability" allows at the same time to perform tests with real traffic and stress devices or networks with different traffic loads.

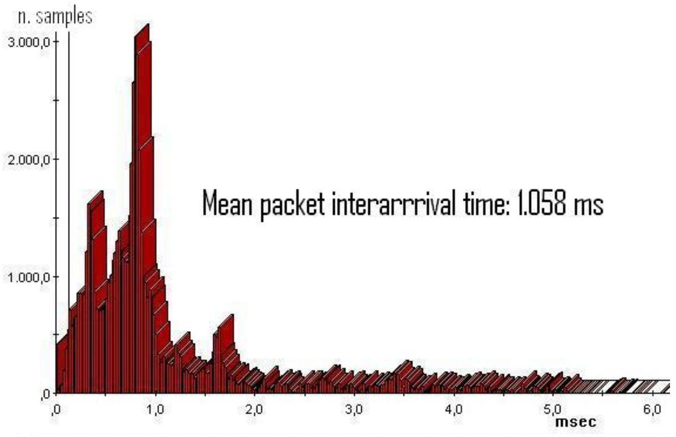
Fig. 3.15(a) shows the interarrival time distribution of a real SIP call (signalling and data) performed through a soft-phone. Fig. 3.15(b) and 3.15(c) show the distribution of the trace reproduced by BRUNO and an "accelerated" version with a scaling factor of 100. It is evident that the shape is almost the same of the original trace and that the time references (time axis and mean packet interarrival time) differ by the factor of 100.

Timing Correction Effect In this set of experiments, the benefits introduced by the error correction mechanism are investigated. We instruct BRUNO to generate CBR traffic flows within a wide range of bit rates, spanning from 100 to 600 Mbps, and we run, for each value of bit rate, both the standard version of BRUNO and a modified version in which the feedback correction mechanism was disabled. The measurements are taken again by means of the AX4000.

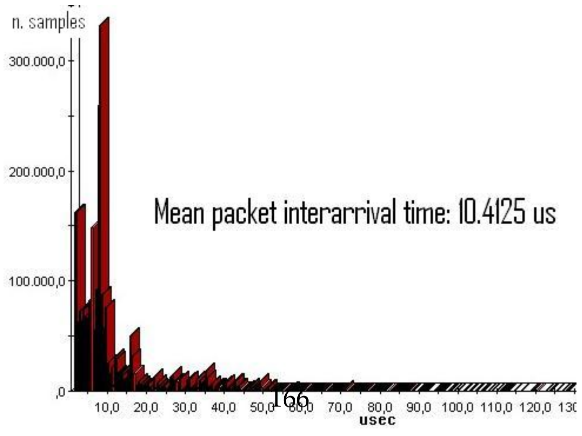
Tab. 3.2 reports the reduction in the interdeparture time variation due to the introduction of the correction mechanism. We have measured the standard deviations of interdeparture times (σ_ϵ for the system with error correction, $\sigma_{n\epsilon}$ for the simple one), obtaining for both versions extremely small values (in the order of hundreds of nanoseconds). However, the use of the timing correction mechanism increases the performance of the system, as shown by the difference $\sigma_{n\epsilon} - \sigma_\epsilon$. In any case, these benefits are more evident with increasing complexity of the traffic model generated, because in a CBR model there are few variable factors that can cause a consistent deviation from the ideal behavior, and, therefore, the variance reduction achievable with an error correction mechanism is limited.



(a) Original trace (*ns*)



(b) Trace reproduced by BRUNO (*ms*)



(c) Trace accelerated by BRUNO (μs)

rate (Mb/s)	$\sigma_{n\epsilon} - \sigma_{\epsilon}(ns)$
100	14.3
200	12.4
300	3.9
400	3.1
500	14.3
600	13.1

Table 3.2: Interdeparture time variation reduction achieved by the correction mechanism.

3.2 A Network Processor based architecture for multi gigabit traffic analysis

In the last few years, the proposal for measurement-based techniques of traffic engineering and management as well as the continuously increasing concern for network security has raised the interest of researchers and network operators towards the development of measurement tools for traffic monitoring/characterization and to support Intrusion Detection Systems (IDSs).

Moreover, the large availability of flexible, easy to use and easy to customize network monitoring software, has proposed the PC as a suitable and cheap platform for network measurement and testing. Indeed, applications such as *tcpdump* [111], *wireshark* [112], *ntop* [113], etc., prove to be very effective and flexible for a large variety of monitoring tasks. Most of these pieces of software are based on the well known *libpcap* API [111], which rapidly became a *de facto* standard for PC based packet capturing. Even though many improvements have been applied to this library [114] [115], it still suffers from performance flaws, that mostly depend on underlying hardware bottlenecks [116] [117].

In particular, two main issues emerge:

1. *packet timestamps:* to sustain a high packet rate, the PC must drive interface cards by using a polling scheme and this results into poor timestamp accuracy;

2. *packet loss:* packet loss can be experienced if either the packet rate is too high and the host CPU cannot allocate/release memory for packets or if the system bus cannot keep the pace of the incoming data.

Furthermore, only off-line processing can often be performed on incoming packets since, typically, no extra CPU power is left for on-line analysis (all the CPU time is used for capturing) [116] [117]. This is mainly due to the lack of packet processing capabilities on the network interface cards which commonly equip commodity PCs. Indeed, these cards are not capable of:

- timestamping the arrival of a packet (avoiding interrupt latency);

- filtering unwanted packets out (avoiding memory allocation/release for unwanted packets);
- feeding the host PC with only a fragment of the packet instead of the entire one (avoiding system bus saturation).

The research described in this work addresses the development of a novel measurement tool that overcome the above listed weaknesses of a purely PC–based architecture by integrating the power of Network Processors (NPs) of the Intel IXP2XXX family in a cooperative and distributed platform. The objective is to combine the high performance of a hardware–oriented solution with the flexibility of general purpose PCs equipped with existing *libpcap*–based applications. The overall target is a powerful system capable of capturing packets on GigaEthernet links with good timestamp accuracy.

Several works on distributed measurement systems have been proposed in the literature in the last few years. In [118], the authors illustrate a distributed architecture for IP traffic analysis composed by a bank of capturing devices which work together and automatically distribute the capturing task among themselves, before passing the data to the bank of processing devices. The purpose is to monitor multiple links within an administrative domain.

The idea of an architecture made up of a primary component that distributes the incoming traffic among several processing units was proposed first in [119]. In the section, the authors outline also the following requirements that must be satisfied by any measurement tools, irrespective of their target application (e.g. attack detection, traffic analysis, accounting or traffic engineering):

- *processing scalability*: the extensibility in terms of packet processing power;
- *flexible flow definition*: the capability of defining a flow by specifying different fields;
- *operational flexibility*: to update the flow definitions on-demand;
- *long-term operation*: the ability of running as long as possible;
- *flexibility of visualization resolution*: the availability of statistics with different resolutions in terms of space and time.

In addition, in [120] the same authors summarize the set of features which a monitoring platform must provide: accurate timestamping, packets storage and/or inspection, packet sampling, flows identification, counting, and high-level statistics computation. Moreover, they propose the use of NPs for the implementation of their system.

Our work originates from the idea of a primary distribution component with the aim of reducing the effort of the subsequent processors and improving the overall monitoring architecture performance. Such a distribution engine is implemented by means of a Network Processor. Thus, the measurement system turns out to be based on a cooperative PC/NP architecture, which provides the flexibility and the power to satisfy the above mentioned requirements as well as to exhibit the features of [120].

Several other research efforts recognize the effectiveness of NP-based devices in aid to common hardware for monitoring purposes. Xinidis et al. [121] propose an active splitter based on Intel IXP1200 for filtering traffic directed to the sensors of a Network Intrusion Detection or Prevention System (NIDS/NIPS). In their scheme, the NP first enforces *Early Filtering* techniques, then it forwards traffic to different sensors, according to *Locality Buffers* or *hash load balancing*.

In [122], Wolf et al. propose to use a distributed architecture, called Distributed Online Measurement Environment (DOME), of passive measurement nodes equipped with Intel IXP2400 NP. Their work includes header anonymization schemes and performance is compared to that of Endace DAG 4.3 cards. Both the previous systems are able to analyze up to 500 Mbit/s traffic flows composed by small packets (64 bytes), while our solution is able to handle up to 1 Gbit/s with the same packet size. Moreover, our system carefully addresses the issue of an accurate packet timestamping.

If compared to hardware solutions (e.g. DAG cards), our traffic monitor turns out to be fairly convenient in terms of price, number of functionalities and flexibility. For instance, the implemented packet classifier supports up to 50000 rules [123], while the DAG 4.3 card, which integrates a simple 7-rule filter, costs as twice as much. Moreover, our solution allows a very flexible and quickly updatable definition of flows.

3.2.1 The System Architecture

The high level view of the measurement system is represented in figure 3.16, where two flow directions of a Gigabit Ethernet optical fiber are both split into two optical signals. The first signal is directed to an output fiber while the second passes through the splitter. Hence, there are two output fibers, one for each direction. This is the best available way to "copy" network traffic though some other solutions are possible (e.g. configuring port mirroring on layer 2 network devices).

The output fibers of the splitter are connected to two of the three optical interfaces (see section 3.2.2) of a Radisys ENP-2611 Network Processor board hosted by a PC (this is our front-end), while a cluster of PC-based Linux boxes is connected to the third interface via a gigabit ethernet switch. The PC cluster and the front-end probe are also connected via a standard 100BaseT Ethernet LAN (which acts as the *control interface* of our system) supporting a standard TCP/IP connection used to issue configuration commands from user interfaces. Therefore, every PC on the LAN can issue configuration commands to the NP via a client/server application (the server resides on the PC hosting the NP board, while each PC runs an instance of the client).

Referring to this scenario, the basic idea behind the proposed architecture is to make the NP board perform the following operations at the wire speed:

1. *packet timestamping:* recording the arrival time of each packet in the standard UTC format;

2. *packet classification and filtering:* selecting the desired packets only and assigning each packet a unique flow identifier based on a rule set;

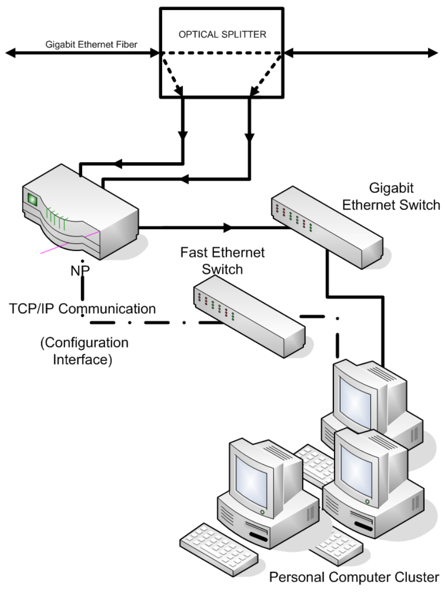


Figure 3.16: Conceptual scheme of the monitoring system.

3. *keeping per-flow counters:* counting the amount of bytes and packets belonging to each flow;
4. *header striping:* getting the necessary information only (e.g. the first *n* bytes);
5. *batch frame crafting:* collecting data in batch frames, each containing the information of several packets;
6. *sending batch frames to commodity PCs belonging to the cluster:* using the third fiber port of the NP board.

On the PC side, the batch frame is received, dissected, and delivered towards any monitoring application, which provides higher level functionalities, such as traffic analysis, intrusion detection and statistics gathering.

The main advantages of this architecture are:

1. *timestamping accuracy*, in that it is performed by the NP card without the interrupt latency typical of a PC;
2. *heavy CPU offload*, as unwanted packets are dropped at the NP level and are not delivered to any PC and since a pre-classification is performed on packets, bringing even more CPU offload (for example in flow identification).

All the functionalities listed above are regulated by a control plane, which is in charge of handling the definition of the traffic flows, to plan the delivery of the batch packets associated to each flow to one (or several) PCs belonging to the cluster and to take care of timestamp association. Such a plane is implemented through user level applications which run both on the PC hosting the NP board and on the PCs belonging to the cluster.

At this stage, the main issue of this architecture would be the incompatibility between the proposed batch frame and all the available *libpcap*–based applications. Next sections describe the implementation design of the entire architecture made up of an NP-side timestamping and classification application and a PC-side kernel space abstraction layer which guarantees the compatibility with any libpcap-based application.

3.2.2 Network Processor Side

The IXP2400 is a fully programmable NP, which implements a high-performance parallel processing architecture on a single chip suitable for processing complex algorithms, detailed packet inspection and traffic management at the wire speed. It combines a high-performance Intel®XScale core with eight 32-bit MEv2 packet processing engines called microengines (μ-Es) which cumulatively provide more than 5.4 giga-operations per second (capable of processing, namely, up to 3.6 mega packets per second). Each μ-E has eight hardware-assisted (i.e. zero-overhead context switch) threads of execution. The Intel XScale core is a general-purpose 32-bit RISC processor (ARM Version 5 Architecture compliant) used to manage the NP, to handle exceptions and to perform *slow data path*. The XScale processor and the whole set of μ-Es run at 600 MHz. Microengines provide the processing power necessary to perform *fast data path* tasks that would require very expensive high-speed ASICs.

The IXP2400 NP is hosted by a third-party board. We adopt the Radisys ENP-2611 board, equipped with 8 MB of SRAM and 256 MB of DRAM. This board provides three Gigabit Ethernet optical interfaces and one Fast Ethernet interface for remote control. Moreover it supports MontaVista Linux [124] operating system running on the XScale CPU. The board is plugged into a PCI-X slot of a host PC; a non-transparent PCI to PCI bridge allows fast communication between the host PC and the Network Processor board.

The NP side of our traffic monitor application reflects the IXP processor hierarchy: μ-Es are in charge of packet timestamping, classification, per-flow counting and batch frame crafting, while the XScale deals with classification table setup and update and parameter reconfiguration, according to the configuration data which are provided by the host PC.

The entire NP-side application is depicted in fig. 3.17: the circles represent rings, which are on-chip circular FIFO queues (used for inter-μ-E communication), the external rectangles represent processors, the internal ones represent the pieces of code that implement specific functions. In a μ-E, these pieces are named μ-blocks, while at the XScale level they are called "core components". White μ-Es contain the driver-blocks, directly provided by Intel and strictly hardware-dependent, dealing with low level functionalities.

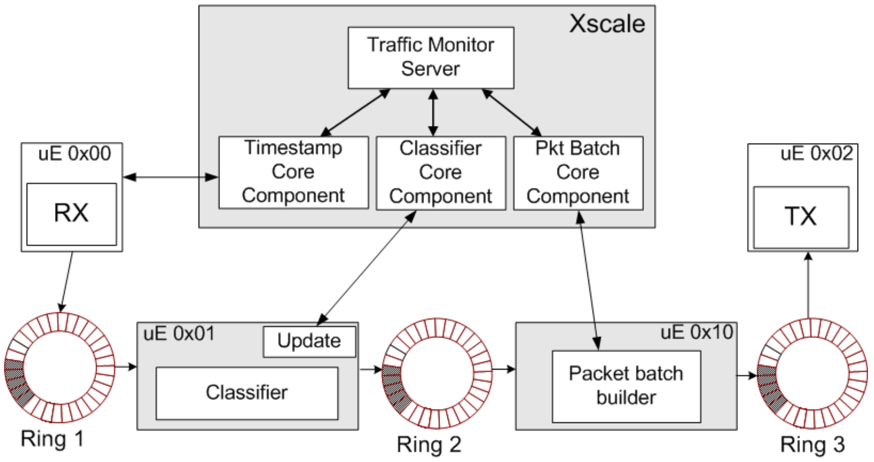


Figure 3.17: Functional scheme of the entire NP-side application.

3.2.2.1 Microengines Application Scheme

Referring again to Figure 3.17, the whole application can be summarized as follows. The *RX* μ-E (0x00) retrieves packets from the interface and places them in Ring 1. For each packet, the arrival time (actually the arrival time of the first mpacket, cfr. section 3.2.5), the entire length and the first n bytes of the packet are recorded. The second μ-E (0x01) classifies the packets it receives from Ring 1 (by either assigning them a *flow identifier (flowID)* between 0 and $2^{16} - 1$ or by simply dropping them), increments the corresponding per-flow counters and sends them to the next μ-E (0x10) which copies all data buffers (each one containing flowID, length, timestamp and the first n bytes of a packet) together to create a batch frame (whose format is depicted in fig. 3.18). Finally, the batch frame is passed to the TX μ-E (0x02) to be sent to one of the PCs belonging to the cluster. The batch frame header has the source address set to the MAC address of the outgoing interface, the destination address set to the MAC address of the correspondent cluster's PC (which is in charge of analyzing that flow) and the type field set to an unused value (0x9000). As shown in fig. 3.18, the payload filled with a variable number of *packet digests* each made up of all the packet information (flowID, length, the n-bytes fragment and the arrival timestamp). The length of the fragment can be different among different flowIDs. The code running on the 0x10 μ-E (Packet batch builder) contains a table with the correspondence between flowIDs and the MAC addresses of the PCs processing those flows. At a given time, the application maintains up to one batch frames for each PC of the cluster and each packet digest is copied onto the batch frame correspondent to its flowID. Packet classification is performed according to the scheme described and in [125][123]. The adopted classifier is reconfigurable and capable of sustaining a very high packet rate.

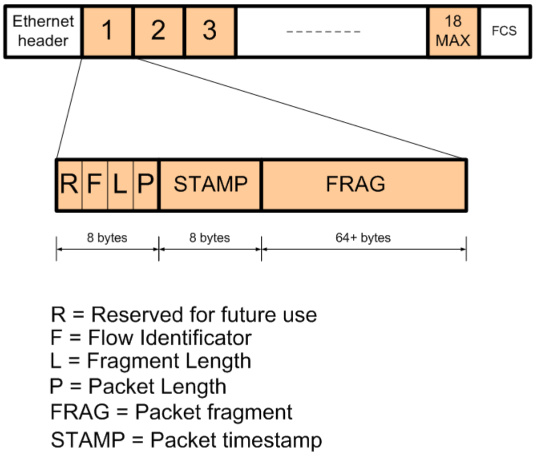


Figure 3.18: Batch frame and packet digest specification.

3.2.2.2 XScale Application Scheme

In our scheme, the XScale processor reads the configuration parameters which have been written by the control plane software at well known addresses of the Radisys board memory banks and translates them into a decision-making data structure, which is stored in the SRAM block; this is performed by the Classifier Core component [123]. In addition, the XScale also manages the dynamic reconfiguration of the rule set, which allows for updating the flow definitions on-demand. This is done through a message exchange between its *core components* and μ-Es, which keeps data structures consistency by properly switching between the classifier table and the cache table. Such a mechanism allows for a fast update of the table and, besides, no packet loss is experienced, since the traffic arriving while the data structure processing is in progress can still be classified by using the cache table. Furthermore, the XScale is in charge of loading the microcode on the microengines when the monitoring application starts.

3.2.3 Host PC side

The communication between the host PC CPU and the Radisys board takes place over the PCI bus and leverages the same mechanisms that we described in subsection 3.1.2.

Although, at the present state, the PCI bus is only used to convey control plane information and, therefore, communication speed is not a sensitive issue, the possibility of fast data transfer offers an interesting opportunity to expand our architecture by implementing additional functionalities on the host PC.

Furthermore, since the use of a gigabit ethernet interface of the NP-board to send batch frames to the cluster can represent a limitation for the performance of our system, a less expensive choice would be to use the PCI-X bus to transmit batch frames to the host PC and make it forward them to the cluster via a standard gigabit ethernet PCI-X NIC.

A server application running at the user space of the host PC is in charge of the implementation of the control plane; such an application communicates through TCP connections with the PCs of the cluster and it handles the functionalities concerning the timestamp synchronization (as it will be described in section 3.2.5.3), the definition of the different flows (including the number n of bytes to be captured for each packet) and their association with a unique flowID and a given PC of the cluster. Besides, it periodically reads the per-flow counters updated by μ-E 0x01 and makes it available on-demand to the client application together with the associated timestamp. Since this software takes advantage of the different libraries which have already been developed, encryption and authentication functions can be easily integrated in our control plane in order to protect the system from malicious users.

3.2.4 Cluster PCs side

The PC-side application is composed by two components. The first is a Linux kernel module which implements a compatibility abstraction layer, while the second is a user space application which consists of a front-end (user interface) and a back-end which passes user's configuration commands to the kernel module (via *ioctl* system calls) and to the NP (via the above mentioned TCP/IP connection established on the Fast Ethernet control interface).

3.2.4.1 Kernel space: the compatibility abstraction layer

This module acts as a compatibility layer between the NP-PC communication protocoland the standard packet processing chain of the Linux kernel on which the *libpcap* API is based.

The module registers itself as a virtual "network layer" capable of processing ethernet frames with the type field equal to 0x9000. The module also creates up to 2^{16} virtual interface cards *mon0* to *mon65535* (one for each flowID), thus implementing an abstraction layer toward the system. Every time a batch frame is received by the kernel, it is steered to this layer which, in turn, extracts from its payload all the packets together with their timestamp and flowID. For every extracted packet digest, a new correctly timestamped packet is generated and transmitted on the virtual interface indexed by *flowID*.

Hence, for example, a *libpcap*–based application configured to monitor the interface *mon5* (with the command *tcpdump -i mon5*) will see all (and only) those packets with flowID 5, as if it were directly connected to the fiber (to which actually the NP is connected).

Therefore, this layer makes it possible to instruct the NP to mark an arbitrary microflow with a specific flowID, and to analyze this flow by simply connecting an application, such as *wireshark*, to the corresponding virtual interface.

As experimentally shown in section 3.2.6, the computational overhead introduced by this piece of code is negligible since it is implemented in a *zero-copy* fashion.

Probably, the most important advantage of this abstraction layer is the full compatibility with existent software: packets arrive at the kernel as they were captured on the wire, making unnecessary any modification to applications and libraries.

3.2.4.2 User Space – the user interface

The user interface is made up of a back-end for:

- configuring the NP classifier via a TCP connection whose peer is the host PC application;
- instructing the NP to capture the desired number n of bytes from every packet (via the TCP connection);
- reading the association timestamp-UTC;
- retrieving per-flow counters;
- configuring the abstraction layer via *ioctl system calls.*

The front-end module simply implements a user-interface from which the user can configure the entire system.

3.2.5 Timestamping

The timestamping operation consists of recording the arrival time of each packet. The arrival time is intended as the time t_a at which the first bit of the packet reaches the network interface. Unfortunately, packet reception (the action of retrieving packets from the wire to the CPU, which is the first place where timestamping can be performed) is a compound operation (made up of many stages): to derive the timestamping accuracy of the system, we need to accurately examine what actually happens whenever a packet arrives at the board.

The Gigabit Ethernet interfaces of ENP-2611 are controlled by a Sierra PM3386 and a PM3387 Gigabit MAC devices (see figure 3.19). Those devices forward received frames to an FPGA bridge connected to the Media Switch Fabric (MSF) interface of the IXP2400. The MSF operates in POS-PHY Level 3 (aka SPI-3, aka PL3) mode and splits packets in fixed-sized chunks called *mpackets* (whose size is configurable as 64, 128 or 256 bytes). To avoid contention on the PM3386, in our application one of the two interfaces connected to this chip is used for transmission, while the remaining one, together with the one connected to the PM3387 chip, is used for packet capturing.

At start-up time, all the RX μ-E threads place themselves on a freelist (RX_FREELIST), thus announcing they are ready to handle a new mpacket. Each time the MSF receives an mpacket, it awakes the first thread in the list and delivers the data to it. Then *RX* threads gather the set of incoming mpackets from MSF and merge them, thus reassembling original packets.

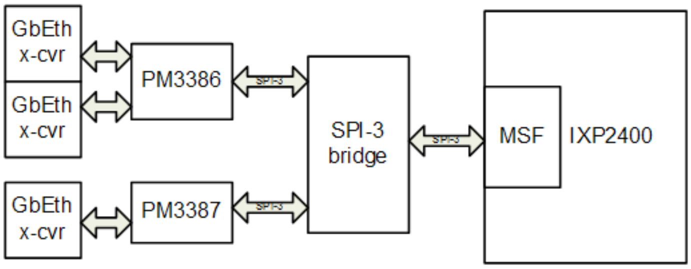


Figure 3.19: Hardware packet receiving chain.

3.2.5.1 Time Budget

When dealing with timestamp operation, the main concern is the jitter of the delay each stage introduces (a fixed and known amount of delay between the real and measured time can be simply subtracted to the measure). In the following, we show that the delay between the arrival time and the timestamp operation is almost constant on the ENP-2611.

Both the PM3386 controller and the SPI-3 bridge forward incoming frames as soon as a certain amount of bytes (hereafter we will call it the "forwarding threshold") is received. This threshold can be configured to 64, 128 or 256 bytes. Since the minimum packet size on Ethernet is 64 bytes, in order to avoid timestamp jitter due to different packet lengths, we set the threshold and the mpacket size to 64 bytes.

This way, the first thread in the RX_FREELIST is awaken at time t_x with a fixed delay from the arrival time t_a. The delay $t_x - t_a$ consists of the sum of three latencies corresponding to the three interfaces that data has to cross (see fig. 3.19):

1. the time to reach the "forwarding threshold" within the PM3386, given by $d_0 = 512\text{bits}/1\text{Gbps} = 0.512\mu s$;

2. the time required by the PM3386 to transfer data across the second interface toward SPI-3 bridge, given by $d_1 = (512\text{bits}/32\text{bits})/104Mhz \cong 0.154\mu s$;

3. the time required by the SPI-3 to transfer data across the third interface, which operates at the same speed as the second one, thus adding an equal delay $d_2 = d_1$.

Therefore, provided there is an available thread ready to timestamp the packet as soon as it arrives, the time lag $t_x - t_a = d_0 + d_1 + d_2 \cong 0.820\mu s$ is fixed and known.

3.2.5.2 The Accuracy of Timestamps

We now have to prove that there is always such a thread. The operations performed by a thread of the RX μ-E when a packet arrives are timestamping (i.e., reading a timestamp counter and storing its value into an internal register), copying packet data and timestamp into the DRAM and context switching.

The first two steps together are very fast and take about 80 clock cycles (cc) to be executed. In the third step, the μ-E puts itself in an idle state until the memory executes the requested operations and switches to the first ready thread in the RX_FREELIST; when all the memory operations are completed, the thread is signaled by the memory hardware itself and can restart its operation. Clearly, while the memory controller is executing the requested operations, the μ-E can be used to perform other tasks by means of other threads.

The worst case occurs whenever all packets are 65 bytes long: in this case, we have a 64 bytes long mpacket plus one extra 1 byte long mpacket. The total amount of time it takes to process this packet is twice the time needed for one mpacket (i.e. $2 \times T_{proc}$), while the interarrival time is slightly larger than the single-mpacket case: $408cc$ instead of $364cc$.

As reported in the IXP2400 data sheet, the signaling delay to awake a RX thread is constant and very small. Thus, by using N_{th} threads per port, we make sure that the RX μ-E threads can receive and timestamp the first mpacket of a packet with a fixed delay from the real arrival time if the following inequality holds:

$$T_{proc} \leq 204cc \times N_{th} \tag{3.16}$$

Since T_{proc} depends on a very large number of factors (accesses in memory, number of threads, instantaneous conditions, etc.), it has been experimentally measured. As shown in fig. 3.20, either for 4 and 8 threads it largely satisfies (3.16).

As for packet batch creation, both the amount of data taken from each packet and the packet batch total size are configurable. Once the amount of data in the packet batch reaches the configured size, it is sent to the TX μ-E. Moreover, a timeout is provided to make sure that non-full packet batches are transmitted if no more packets arrive.

Timestamp is provided by the use of 64-bit timestamp registers within the RX μ-E. Such registers are increased by one every $16cc$ (we shall call it "NP-tick" or simply "tick"). Then each packet is timestamped with a value given by $t_x - d_0 - d_1 - d_2 = t_x - 492\text{cc} = t_x - 31\text{ticks}$.

In order to quantify the goodness of timestamp accuracy, it is worth reminding that the most error sensitive application is traffic characterization; in this application the measure that has to be very accurate is the inter-arrival time of packets. Since the minimum inter-arrival time on a Gigabit Ethernet link is $0.68\mu s$, the "tick" granularity represents a very good maximum error of 4%.

If packets are concurrently captured from two interfaces, a timestamp error occurs whenever two mpackets are sent to the SPI-3 chip by PM3386 and PM3387. An upper bound of the timestamping error is obtained in the worst case which takes place when two mpacket arrive exactly at the same time to the SPI-3. In this case one of the two mpackets has to wait $E_{max} = d_1 \cong 0.154\mu s$ before being timestamped.

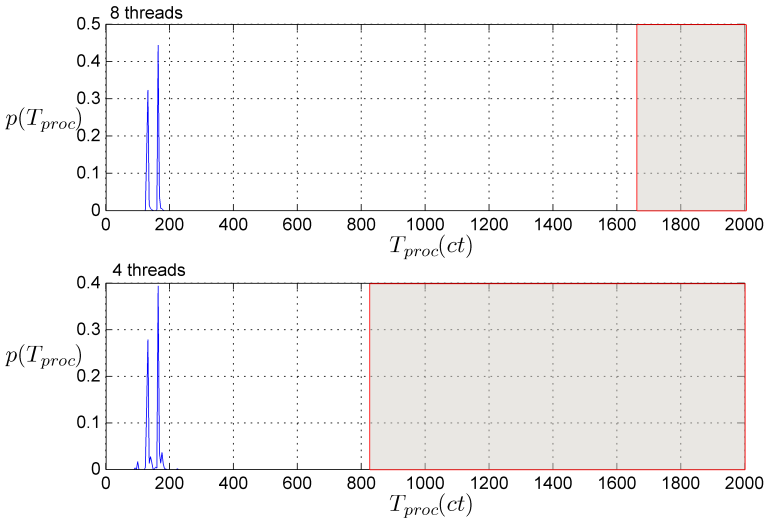


Figure 3.20: Histogram of measured T_{proc}. Inequality (3.16) is satisfied for 4 threads ($T_{proc} < 816cc$) and 8 threads ($T_{proc} < 1632cc$).

Comparing this error to the minimum inter-arrival time, we obtain a maximum error of 22.6%, which is much larger than the 4% due to the clock granularity.

3.2.5.3 Timestamp synchronization

As above described, the timestamps included in the batch frames are expressed in ticks ($26ns$) and represent the time offset of the packet arrival with respect to the instant when the monitoring application has been loaded; such a time instant will be referred to as $UTC0$. Despite its accuracy, such a time measure cannot be directly converted into an absolute timestamp; therefore, measurements taken by two different monitoring probes cannot be correlated, because of the lack of an absolute synchronization. However, the need to correlate measurements taken by different probes is crucial for many network applications: for example, network monitoring applications may require the knowledge of the delay experienced by packets belonging to a given flow while crossing a network. For this purpose, it is necessary to deploy a probe at the network ingress and one at the egress, and to compare the timestamps given by the two devices to the same packet. For such a purpose, we provide our monitoring device with a mechanism to convert the timestamp in the batch frame to an absolute timestamp. Such a mechanisms is based on the cooperation between the PC hosting the NP board and the PC receiving the batch frames. The XScale routine in charge of loading the monitoring software, immediately after resetting the tick counter register, sends an interrupt to the host PC, by setting a bit of a register belonging to the non-transparent PCI bridge which connects the NP board

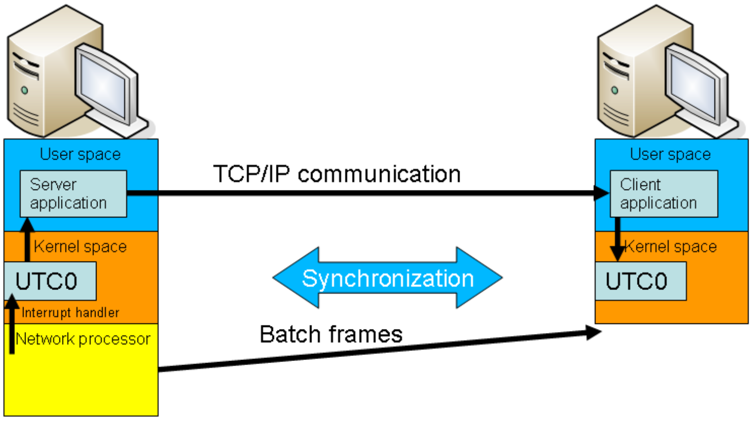


Figure 3.21: Synchronization mechanism.

PCI bus to the host PC bus. The latency introduced by this operation is negligible. The interrupt handler on the host PC, which is implemented as a Linux 2.6 kernel module, records the timestamp associated to the interrupt directly from interrupt context; such a timestamp is considered as the $UTC0$ associated to the probe and is conveyed to the PC receiving the batch frames by means of a user level application. Subsequently, such a value is passed, via an *ioctl* system call, to the kernel module handling the batch frames, which, by combining it to the offsets conveyed by the frames, is able to associate to each incoming fragment an absolute UTC timestamp. In order to cope with the clock skew between the NP and the PCs, a mechanism to periodically update such a timestamp has been designed. The whole mechanism is illustrated in figure 3.21.

Our design provides the maximum flexibility for the synchronization of different probes: since the reference UTC is measured with respect to the clock of a general purpose PC, any available synchronization methods can be used, depending on the precision which is required by the specific application. If an offset of a few milliseconds is acceptable, the synchronization of the probes can be achieved by using the NTP protocol; on the other hand, if a very precise synchronization is needed, GPS receivers can be connected to the PCs hosting the monitoring probes. Note that a small timing error may be due to the latency associated with the interrupt handling by the host PC. However, since such a system does not have to perform any data path function, it is likely not heavily loaded and, therefore, if the interrupt is given high priority, the delay associated with interrupt preemption will be in general very small.

3.2.6 Packet Capturing Evaluation

3.2.6.1 Experimental setup

In the experimental testbed, the NP-based capturing device is connected to a high-end personal computer equipped with two Intel Xeon 2.8GHz CPUs (with hyper threading activated), 1 GByte of rambus RAM and a 3COM Gigabit Ethernet optical fiber network interface using the *tg3* driver. The installed Operating System is Ubuntu Linux 7.04 with a 2.6.18 vanilla kernel. Unfortunately, the *tg3* driver, similarly to the majority of the drivers for gigabit interfaces available for Linux, does not support the *polling* working mode (NAPI). Nonetheless, the interrupt mitigation mechanism supported in hardware by this 3COM interface proved to be sufficient to avoid PC livelock.

In order to perform packet capturing, a standard *tcpdump* and *libpcap* distribution is used. Data streams are generated by *Spirent* ADTECH AX4000 hardware packet generator and analyzer [126].

3.2.6.2 Experimental runs

In the first experiment, a bulk traffic stream is generated and sent to the personal computer either directly or through the NP. The main purpose of this experiment is, on one hand, to evaluate the processing overhead introduced by the abstraction layer and, on the other hand, to assess the benefits introduced by the packet batching operation performed in the NP (due to the lower packet rate which means a lower rate of calls to the driver function). The NP has been set up to mark all the traffic with the flowID 3, thus making it available through the *mon3* virtual network interface on the receiving PC.

The stream is captured in both cases by using the *tcpdump* raw capturing features. Hence, for the first experiment the command line is:

```
user@hostname# tcpdump -i eth4 -w file1
```

while for the NP-driven one the command line is:

```
user@hostname# tcpdump -i mon3 -w file2
```

The second experiment aims at showing the capabilities of the system in extracting and processing a *mouse* flow in presence of an *elephant* one. Therefore two flows are involved: the *mouse* flow, from IP host 100.3.3.3 to 10.3.3.3 with TCP source port 100 and destination port 3357, and the *elephant* flow with a different source port (3). The first flow is generated at a rate of 50$Kpps$, while the second one is generated at increasing packet rates. The compound flow is once again captured by the PC alone and through the NP. The NP is configured to mark the *mouse* with flowID 4 (available through *mon*4 at the receiving PC). In both cases *tcpdump* is simply used to decode and dump packets in a trace file (in a real context, this is the minimum real-time packet processing). The issued command line is the following:

```
    user@hostname# tcpdump -nttv -i eth4 -w file1 src host 10.3.3.3
and dst host 100.3.3.4 and src port 100 and dst port 3357
```

while, in the NP-driven experiment:

```
user@hostname# tcpdump -nttv -i mon4 -w file1.
```

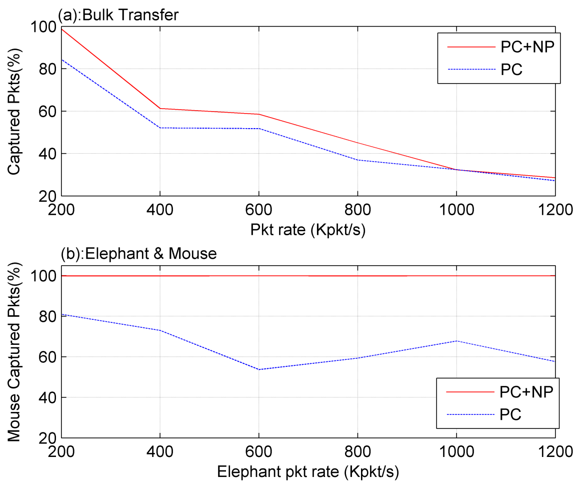


Figure 3.22: (a): Packets rawly saved to trace file. (b): Packets captured from the mouse flow.

In both experiments, packets are minimum sized (i.e. 64 bytes). Notice that this is the worst case scenario for our system since every single byte of the incoming packets is included in the batch frame; therefore there is no bandwidth reduction between incoming and batch frame traffic and the only relief for the PC is given by interrupt mitigation. However in realistic traffic scenario, where average packet lengths are likely higher than 64 bytes, the bandwidth reduction represents a major advantage for our system.

Fig. 3.22.a refers to the first experiment performed with CBR traffic and reports the amount of packets captured by our system and by the PC alone. As shown in figure, the NP-based system outperforms the PC alone, meaning that the benefit of a lower number of calls to the driver is greater than the processing overhead introduced by the abstraction layer. Fig. 3.22.b shows the full advantage obtained by using the NP in the flow extraction. The totality of the *mouse* flow is captured by the NP-based system while it shares the fate of the *elephant* flow when captured by the PC alone. In this context, the PC shows all its architectural flaws in that it looses a huge amount of packets, while the NP-based system performs this operation with no loss.

3.2.7 Timestamp accuracy

We performed a broad experimental campaign in order to assess the capability of our tool to provide precise timestamps with different amounts and characteristics of traffic load; in particular, we focused on the precision of interarrival time measurements, since the precision of UTC timestamps heavily depends on the performance of synchronization protocols, which, in turn, is influenced by factors (propagation delay, jitter, etc.) which are external to our system. Since, in this section, we focus on timestamp accuracy, we ignore the packet loss issues (extensively discussed in section 3.2.6) and computations are made on the available interarrival times only. Unfortunately it was not possible to compare the performance of our system with that of other NP-based solutions, since the source files of such applications are commonly not publicly available.

We point out that, unless explicitly mentioned, the generated traffic is always composed by 64 bytes long packets; as already discussed, this constitutes the worst case scenario for our system. Furthermore, except for the last experiments, we generated constant bit rate traffic; however, since we pushed our analysis up to very high traffic rates ($1200Kpps$), such scenario can be considered as the worst case for the system (like a non-stopping burst).

3.2.7.1 Testbed description

The experimental layout is the one previously described in section 3.2.6.1. In addition, to evaluate the accuracy of timestamps, we used as a primary benchmark the timestamps provided by an Endace 4.5G2 DAG card, which can reach a time granularity of $20ns$. Therefore, we connected the Network Processor or the optical NIC of the PC to a port of an optical splitter. The DAG card was connected to the other output port while the input signal was generated by the *Spirent* ADTECH AX4000. This way, both measurement devices were loaded with exactly the same traffic.

Notice that the data structure used by the Linux kernel forces the timestamp granularity of both our tool and the PC to be $1\mu s$. Therefore, if compatibility with existing user space software has to be maintained, quantization errors are not avoidable.

We generated traffic flows at different rates and calculated the interarrival time measurement errors by using the timestamps provided by the DAG card as reference; since such an error has typically zero mean, we considered its absolute value. The results are illustrated in figure 3.23, where the distribution of the measurement error is reported for different rates of the incoming traffic. The plots show that the accuracy achieved by our system is significantly better than that of an ordinary PC: while the measurements provided by the former are almost always within one microsecond from the actual values (a difference which can be ascribed to the coarser granularity of the timestamp), those provided by the latter can differ for even 100 μs with non negligible probability. In addition, while, for the measurements provided by the PC larger error values emerge at higher traffic rates, the performance of our tool is not significantly influenced by the speed of the incoming flow.

Figure 3.24 summarizes the performance achieved by both tools in terms of mean value of the absolute measurement error. The results confirm that the performance of our tool does not suffer from higher traffic rates and generally outclasses that of a PC by at least one order of magnitude. In addition, the granularity of the timestamp appears to be the main source of measurement error, since, as it appears from the analysis, such an error is always smaller than $1\mu s$ and keeps almost constant despite the significant increment of traffic rate. Such an analysis is also confirmed by the fact that the best performance is achieved for a packet rate of 200000 packets per second, when the interarrival time ($5\mu s$) is an exact multiple of the timestamp granularity.

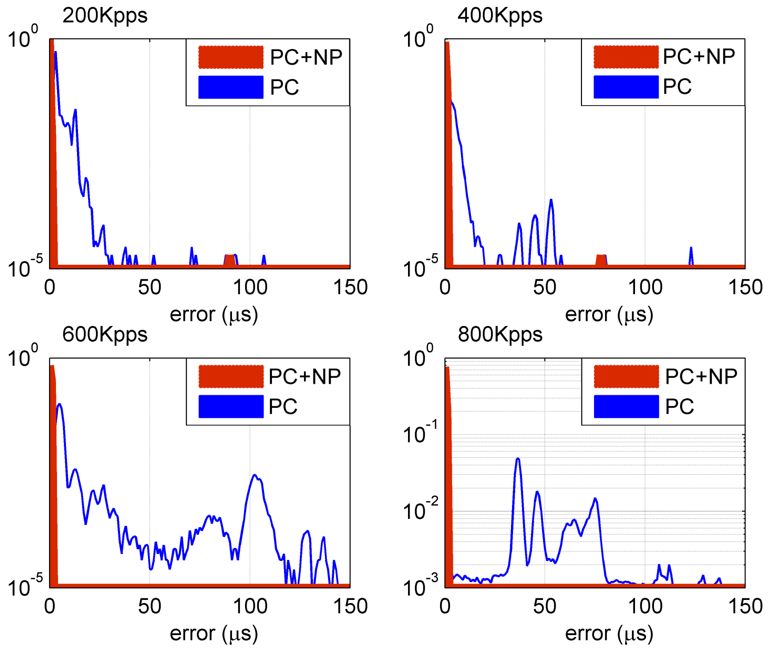


Figure 3.23: Distributions of measurement errors of interarrival times: comparison between our system and an ordinary PC.

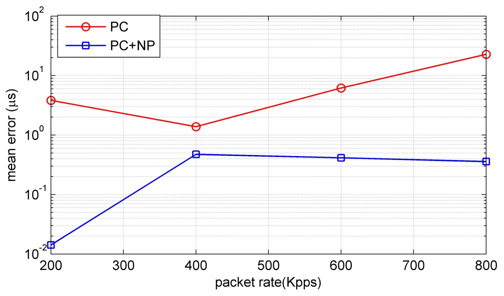


Figure 3.24: Comparison between our system and an ordinary PC in terms of mean value of the absolute measurement error.

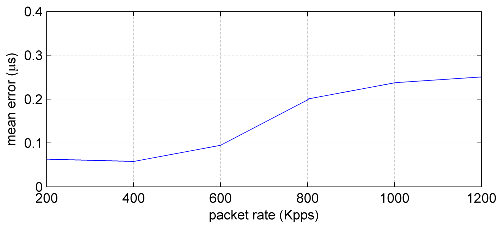


Figure 3.25: Mean absolute error of timestamps without interfering flows.

3.2.7.2 Finer grain performance analysis

In order to evaluate the best performance that can be achieved by our system, we slightly modified the kernel module running in the back–end to retrieve the original timestamps provided by the NP, thus reaching a granularity of 1 tick ($26ns$). Furthermore, such a higher resolution allows to analyze the impact on the system performance of the variation of some operating parameters, which are usually hidden by the coarser granularity of the timestamp. First, we considered a single traffic flow with increasing bit rate, and evaluated the mean absolute error; the results are shown in figure 3.25 and confirm that the system is able to handle high rate traffic flows, while still preserving a good timestamp accuracy. The error, although slightly growing with the packet rate, is always well below $0.3ms$.

After that, we evaluated the effect of concurrent traffic flows on the timestamping of packets belonging to a given flow. In particular, we first kept constant the packet rate of the flow of interest ($200Kpps$) while gradually increasing that of the interfering traffic; subsequently, we generated a constant interfering flow ($200Kpps$) and a flow of interest with increasing bit rates. Figure 3.26 shows the results of the experiments as a function of the overall traffic rate: from the comparison between the two graphs it appears that the accuracy is only marginally influenced by the composition of the incoming traffic; such a result is coherent with the timestamp being taken before the classification of the packet.

In a further experiment, we evaluated the influence of incoming packet length over the timestamp accuracy. We therefore generated traffic flows with the same packet rate (i.e., $80Kpps$), but with a different packet length. As a consequence, the traffic flows are characterized by different bit rates; the results, reported in figure 3.27, show that the mean error rises very slowly with the packet length, as a plain consequence of the increased bit rate. Indeed larger packets, composed by several *mpackets* (see section 3.2.5), require the RX μ-E to perform a greater amount of work, thus slightly affecting the timestamp accuracy.

Finally, we evaluated the timestamp accuracy in the case of incoming traffic with variable interarrival time of packets. We generated traffic according to the Markov Modulated Poisson Process (MMPP) [127, 128], a well known traffic model for aggregate sources, widely adopted to characterize packetized multimedia communications. In this case, we evaluated the distribution of measurement error with two kinds of traffic flows, one characterized by an average rate of $350Kpps$, the other by

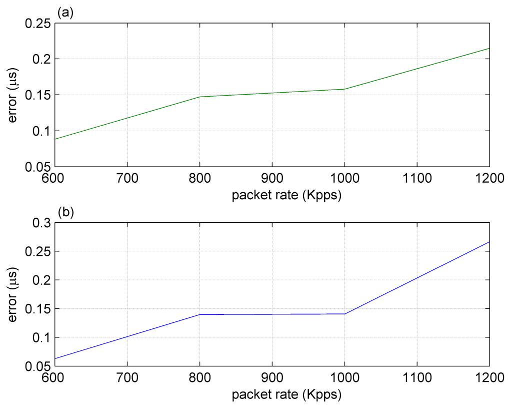


Figure 3.26: a) Timestamps with variable interfering flows; b) Timestamps with fixed interfering flows.

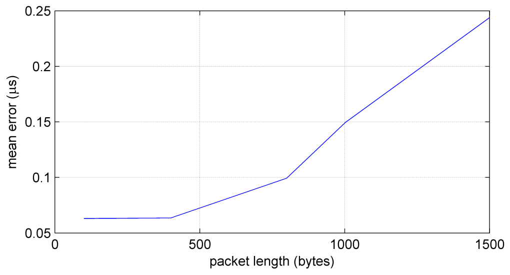


Figure 3.27: Timestamps with variable packet length.

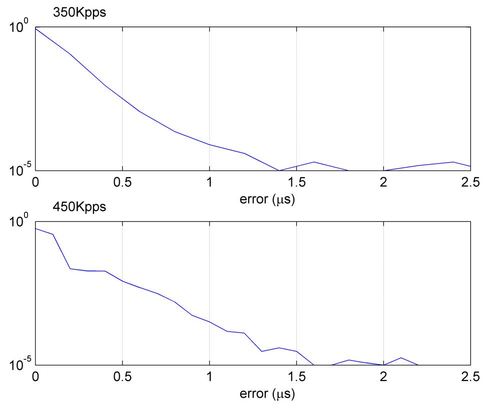


Figure 3.28: Timestamps with MMPP traffic.

an average rate of $450Kpps$; in both cases the average packet rate associated with the higher speed state of the correspondent Markov chain is $800Kpps$. The results are reported in figure 3.28 and show that, even in this case, the measurement error is well constrained below 1 microsecond with mean values of $60ns$ and $80ns$, respectively. Therefore, the accuracy of our system is preserved even in more realistic traffic scenarios.

3.3 Towards smarter probes: in-network traffic capturing and processing

As already discussed, analyzing and checking out the traffic flowing over a high capacity link is still a very challenging technological issue, due to the huge amount of data stemming from such a process. Furthermore, current national and international legislation is imposing stricter and stricter limits on the storage and utilization of potentially privacy-sensitive data that may be generated from monitoring applications. We argue that both of these problems can be effectively addressed by increasing and extending the capabilities of traffic capturing devices beyond plain packet capturing and flow metering. Therefore, we envision a new generation of smart probes that support traffic pre-processing according to the needs of the specific application that is expected to provide the final results of the monitoring activity. The benefits of such an approach are two-fold: on one hand, *in–network* traffic filtering allows to discard a huge amount of information which is not relevant at all to the selected application, thus relaxing the performance requirements of the application itself. On the other hand, traffic pre-processing can be used to hide personal information that may be made available only to a user in possession of the required privileges upon verification of a given condition. Optimized and effective packet processing algorithms, as those we described in the previous chapter, are a very nice fit for this kind of processing. Following such a general approach we propose a modular architecture that allows application specific traffic pre-processing to be carried out in a scalable and performance-effective way. Such an architecture interacts with the external network by enforcing strict role-based policies, thus allowing selective and proportional information disclosure; the architecture as it is can be easily integrated with a standard access control infrastructure. An example application is demonstrated in order to prove the effectiveness of the proposal.

3.3.1 Introduction

3.3.1.1 Why smart probes?

State–of–the–Art approaches do not scale. Many currently adopted monitoring applications (Snort [45] is just the simplest example) are built as a unique block, which takes as input a stream of raw packets (a trace, which can be made up of live traffic or traffic which has been previously captured by a probe) and returns the desired output. Usually, such systems leverage the standard PCAP interface, which provides a similar kind of access to stored and live data. Several examples of such solutions have been proposed in the literature. Coralreef [129] provides an API implementing two stacks to retrieve data from hetherogeneous sources: one of the stacks is used to import traces from different kinds of links while the second one enables working with flow records. The work [130], instead, proposes a large scale measurement infrastructure which is more tailored for active and performance measurements.

Nowadays, such a design paradigm shows several limitations. On one hand, with the current fast growth of link capacities and traffic volumes [131], having a full fledged monitoring application inspect every single packet on a multi–gigabit link raises huge performance issues. Common general purpose hardware hardly keeps the pace with the packet rates characterizing current core links, even when minimal per–packet processing is required. Hardware-based implementations, in turn, usually lack the flexibility and expressiveness which are required to implement complex traffic analysis applications. Moreover, according to the current technological trends, traffic speed is growing faster than processing power, so that having the application monitor a complete traffic stream on a core link will be increasingly problematic. Such a scaling problem surfaces, in particular, when dealing with distributed monitoring applications, which have to deal with data captured by multiple vantage points scattered across the network. This kind of applications is likely to become more and more popular, as distributed anomalies and cyberthreats (of which botnets are a major example) require a detection/mitigation infrastructure which correlates events and alerts from several probes, possibly belonging to different domains.

Other monitoring applications address this issue by taking as an input pre–processed reports formatted according to NetFlow [132] or IPFIX [133] protocols. Such reports encompass summarized per–flow information (usually cumulative packets or bytes counters, duration and TCP flags) which are usuful for a number of applications (billing is a common example). IPFIX, in fact, offers a significant degree of flexibility in letting the user define the data types it needs to convey.

Such an approach is nowadays very popular in the field of distributed monitoring: a scenario where several NetFlow probes report to a centralized collector is common in many operational scenarios. However, despite achieving a significant reduction over packet traces, the practice of exporting a standard information for every flow as an input to the monitoring application still presents significant issues. On one hand, collecting per flow data on a full operator network is likely to raise a huge scalability problem, as the collector represents a serious performance bottleneck and is likely to get congested: the rate of new flows entering the network is likely to be one or two orders of magnitude lower than the packet rate but, in a large operator network, can easily rise to prohibitive figures. Indeed, much of the information which is conveyed by per–flow records is of little or no interest to the application, especially when it deals with detecting anomalies and security breaches: a famous Van Jacobson quote reports that "*...If we're keeping per–flow state, we have a scaling problem, and we'll be tracking millions of ants to track a few elephants*". In addition, per flow information is, in many cases, not detailed enough for the application's needs: as an example, Snort requires scanning the packet payload for malware signatures, while applications dealing with network path delays needs precise timestamps of certain packets.

In many cases, performance problems have been addressed by implementing some critical application primitives in hardware. While this solution is certainly effective in reaching the required throughputs, it significantly impacts on the flexibility of the monitoring devices: once the desired functionalities have been committed to the silicon, there is no way of updating them. DAG cards [134] are effective in capturing packets at high–speed, but usually only provide limited on-board filtering functionalities. Other probes export flow–data through NetFlow or IPFIX protocols; such information is sometimes not enough for certain applications. Some devices [135] use special purpose hardware in order to perform some specific monitoring tasks on high–speed links. However, as previously mentioned, they are intended for a specific task only and are not able to support a wide range of applications.

Privacy preservation is not an option. Another big issue that current network monitoring practice needs to address is the compliancy with current legislation in terms of privacy preservation. In particular, due to current legislation trends ([136] effectively explains the constraints imposed by EU legislation), much of the information which is retrieved from the captured traffic is considered privacy sensitive, and its disclosure and storage is subject to strict rules. Such constraints, besides further preventing the practice of trace based monitoring, reflect on almost all of the above described approaches. Not only the packet traces, but a huge number of derived metadata (including per–flow reports) are considered to contain privacy sensitive information and therefore their export and storage is subject to very strict rules (if not completely forbidden).

A classical approach to privacy–aware network monitoring has been to run specific anonymization tools over the packet traces or the metadata to be exported and to hand them over to the legacy analyses, so as to have them work with "sanitized" input. A broad range of such techniques has been proposed in the literature (see [137] for a detailed survey), including blackmarking (complete removal of given fields), pseudonymization (substitution of a given identifier with an anonymized alias), timestamp approximation or prefix preserving anonymization (a technique allowing to replace IP addresses with a pseudonym which preserve their common prefix length relationships). Several tools have been published claiming to be able to sanitize a packet trace in a user configurable manner. However, several studies show that classic anonymization schemes may be easily reversed by skilled attackers using statistical analysis techniques (see, for example [138] or [139]). A theoretical study [140] showed that there is a clear trade–off between the information content of a trace and its ability to preserve privacy: as a consequence, traces which are well anonymized turn out to be almost useless for a monitoring application. In addition, anonymizing the data before handing them over to the monitoring application increases the burden of traffic capturing, so making the performance issues even more serious.

Finally, particular care has to be taken in distributed monitoring scenarios, especially when dealing with multi–domain applications: in the latter case, indeed, beside complying to the law, the application must ensure that no business confidential information is leaked to a possible competitor.

3.3.1.2 The *analyze while capturing* paradigm

As opposed to traditional approaches in monitoring application design, in this work we propose to address both of the above illustrated major issues through in–network traffic processing performed by smart probes. In fact, traditional probe devices are usually designed with a strong focus on capturing performance but with little flexibility in terms of packet processing and exporting. On the contrary, we argue that the probe should be a flexible and programmable device that can filter and export the information it handles in a way that is specifically "matched" to a certain monitoring application. Of course, this means that a modular and extensible probe design is needed in order to accommodate different monitoring applications concurrently. An application specific filtering module directly on the probe can select and export the only information which is relevant to that particular application, thus enforcing immediate data reduction.

Such basic principle of "processing while capturing" is beneficial to the overall monitoring infrastructure in that it addresses at the same time the two main issues which have been described. In particular, it allows to support:

- performance scalability through data reduction: information which is of no interest to the application is discarded, thus aggressively reducing the amount of data to be processed;
- selective data protection: personal information is hidden and is not allowed to leave the probe, unless some particular condition is met that makes information disclosure necessary and therefore legal.

As for the latter point, our approach effectively leverages the so-called "proportionality principle", which is common in privacy related legislation: an application is allowed to receive only the data which is strictly necessary to its operation. On-probe filtering, therefore, allows the data which are required by a given application to be legally exported out: the privacy of the users is preserved as most of the regular and legitimate traffic (which is of no interest to the applications) will never leave the probe, thus preventing any possible leak of sensitive information.

A simple example of how these principles are implemented is that of a common intrusion detection application: in that case the filtering module on the probe will perform a fast scan of every packet, in order to separate legitimate traffic (the vast majority), from suspicious flows, which will be sent to the actual application for more detailed inspection. Of course, such a fast filtering activity requires proper algorithmic, as it will be illustrated in subsection 3.3.5.

A smart probe, in addition, can export information in very compressed and anonymity-preserving data structures, which allow to detect a certain class of events without leaking information about the single users; *sketches* [141] and *Bloom filters* are good candidates for this kind of solutions.

Such a general principle is embodied in a flexible monitoring probe architecture that will be described in the next subsections. Such architecture allows to encompass several instances of application–specific processing running in parallel in a scalable and effective way. This is accomplished by means of several architectural choices:

- decoupling of control functions (access control, configuration etc.) from traffic processing functions;
- distribution of the traffic processing workload among several (possibly heterogeneous) processing units;
- dynamic allocation of the computational resources;
- strict access–control mechanisms with role-purpose specifications.

Such a novel architecture has been adopted and deployed within the integrated prototype built as a result of the FP7 European research project PRISM [142] (under which this research activity has been carried out), where it was used as the base architecture of the Front–End module [143]. The goal of such a research was to design a framework allowing to deploy heterogeneous off–the–shelf monitoring applications while respecting user privacy constraints.

The rest of the chapter is organized as follows. Section 2 describes the overall probe architecture while subsection 3 and 4 deals more specifically with the data and control planes respectively. Section 5 is devoted to advanced algorithmic for on–the–probe traffic processing while subsection 6 reports a practical use case deployment of the described probe architecture. Finally conclusions end–up the contribution.

3.3.2 The probe architecture at-a-glance

The probe design which embodies the overall previous discussion is represented in Figure 3.29. The architecture reflects the idea of an advanced logical component that is in charge of both capturing data at gigabit speed and performing a set of basic operations on–the–fly in an efficient way in order to:

- isolate relevant flows out of the set of all traffic flows;

- extract relevant information from the observed traffic and eliminate irrelevant information (data reduction);
- protect relevant information to force compliancy with end–users privacy requirements.

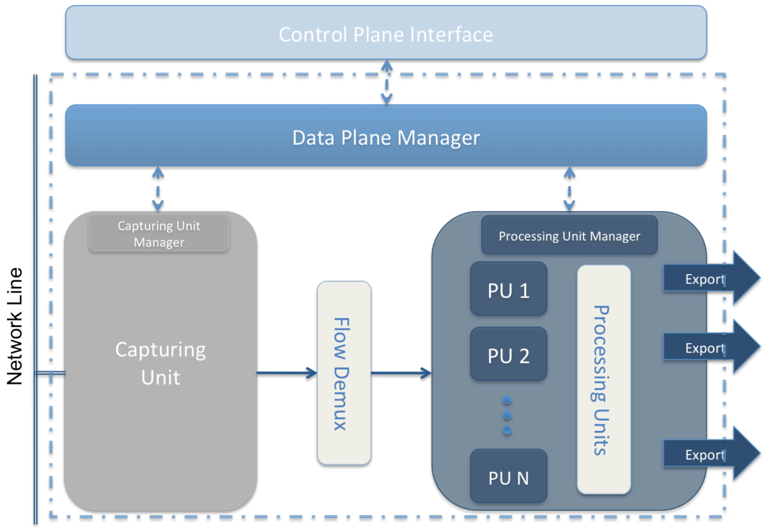


Figure 3.29: Overall probe architecture

The device directly connected to the wire is the Capturing Unit (CU). Its main functions are: capturing traffic, timestamping packets, classifying packets and sending snapshots of the captured packet to multiple destinations (Processing Units). As Figure 3.29 highlights, multiple Processing Units (PU) are supported. They are deployed as commodity PCs or dedicated HW devices and they receive data from the CU and are in charge of actually implementing the application-matched data processing and protection (*analyses*). After their operations, PUs export results towards the further stages of the monitoring applications. The overall system configuration is controlled by the Control Plane Interface that issues commands to the Front-End Data Plane Manager (DPM), which, in turn, communicates with the other components. The DPM, in particular, enforces the received directives on the CU and the PUs by instantiating commands to the Processing Unit Managers and the Capturing Unit Manager, which constitute the control components of the traffic processing blocks. Notice that the prototype implementation of the proposed architecture is largely based upon the system that we describe in section 3.2. In particular, both the CU and the abstraction layer of the PU are based on the implementation described in the previous section.

3.3.3 Probe Data Plane

3.3.3.1 Capturing Unit

As already mentioned, this is the component which is in charge of capturing the monitored traffic and demultiplexing it among the analyses performing application–specific processing. In particular, packets can be dropped or selected for further processing, in which case a portion of them will be forwarded to one or multiple PUs, according to the rule table. As it is in charge of handling a potentially large data flow, it is likely to be implemented as a hardware accelerated device. Depending on the monitoring application requirements, this unit may extract header values by stripping only the necessary information from packets (in the simplest case this might be even just packet truncation after the layer 3 header), thus alleviating the workload on the upstream PUs. In order to deliver the captured traffic to the proper processing block, an internal interconnecting network is used. A simple and effective implementation consists of an Ethernet network where *batch frames* are transmitted, consisting of snapshots of captured traffic data plus some extra meta-data information.

3.3.3.2 Processing Units

As far as the data plane concerns, processing units are physical devices that receive data from the capturing unit through standard interfaces (standard libpcap interfaces), process them according to the analysis function(s) installed on them, and finally deliver encrypted data to the external world. PUs are usually software–based devices, where different kinds of processing are dynamically allocated. However, in case processing intensive analyses are required (e.g. deep packet inspection), they can use special purpose hardware. Such particular features are taken into account by the control plane when resource allocation has to be performed.

A typical PU has to implement several functionalities, which are described in the following.

Abstraction layer. In order to provide a standard interface between the proprietary protocol implemented by the CU exporting process and the analysis functions, a proper abstraction layer is installed on the PUs. Such an abstraction layer restores the compatibility between the proprietary batch frame format and the standard libpcap capturing interface. The abstraction is typically implemented by leveraging the concept of virtual capturing interface: all of the packets belonging to the same group (matching the same set of flow definitions) can be received by the analysis function through a virtual network interface, just as if the traffic were captured by the PU itself. The major advantage of this approach is that it totally hides the underlying batch framing process and allows compatibility to existing software. Therefore, brand new applications that reside on PUs can be designed and implemented in a completely independent manner, as they will just need to rely on libpcap.

Analysis functions. Analysis functions are applications developed at the user space that *i*) read (possibly truncated) packets from the virtual monitoring interfaces made available by the abstraction layer, *ii*) process data according to their specific function and *iii*) export their outcome to the further processing stages of the monitoring application.

The development of analyses actually implements the overall design philosophy of further reducing data directly at the probe (the first application-agnostic stage of reduction occurs at the capturing unit level as packets are truncated to a customizable size) and delivering to the following stages the minimum necessary information only. Although the functional interface of such analyses is quite simple, they are subject to strict performance requirements, as they must process packets in real time and by keeping a very limited amount of state.

As it will be elaborated upon in subsection 3.3.5, the use of probabilistic data structures, such as *Bloom filters*, that keep state in a compressed and quickly accessible way, is envisioned at this stage of the application.

Besides processing traffic, the PUs are also directly responsible for conveying it out of the probe to the further stages. The destinations of such reports (as well as some formatting options) are communicated to the analyses through the control plane. In principle, any analysis function may use its own protocol, even if the use of standard formatting (as the IPFIX protocol, that was the choice for the PRISM project) is recommended.

3.3.3.3 Data Plane Performance

The probe data plane is subject to very strict real-time constraints, as a large portion of the traffic flowing over the monitored link has to be conveyed through it (potentially, as some flows may need to be duplicated to several PUs, the traffic rate might be even higher than that on the link). In order to prove that such a component can be actually implemented and meet such performance requirements, we report here the experimental results as obtained for the PRISM implementation of the smart probe. As for an evaluation of the basic low–level capabilities (timestamping, capturing etc.), please refer to the exstensive evaluation reported in section 3.2 (as already stated, the implementation of the performance–critical components relies on the system we described in that section).

3.3.4 Probe Control Plane

The high level scheme of the probe control plane is depicted in Figure 3.29. An analysis function is dynamically set up on the probe upon request from an external entity (be it the monitoring application itself or a further processing stage). All kind of requests are received through the Control Plane Interface (CPI), which acts as the border module between the probe itself and the rest of the system. Typical requests being served at the CPI and properly mapped and delivered to the Front–End include *setting up*, *tearing down*, *stopping* and *restoring* an analysis, as well as *updating* its parameters (output format, report destination, etc.). Naturally, strict authentication and authorization mechanisms have to be enforced within this block, ranging from standard X.509 identity and privilege management infrastructure to more involved schemes, such as the *purpose-role* based mechanisms used within the PRISM project [144].

Once a request has been authorized, it is taken over by the Data Plane Manager (DPM). The DPM is the central component in charge of managing the capturing unit and the available processing units. Its main purpose is to arrange and launch the analyses on the various PUs (by communicating to the Processing Unit Manager installed on all PUs), and to configure the CU (through the Capturing Unit Manager) to capture and forward them the portions of traffic that will be the subject of the analyses. Since DPM has a perfect knowledge of the status of all the processing units, before setting up new analyses it first checks for resource availability (*admission control*) and may enforce load balancing in order to optimize resource usage.

Any new request is accompanied by a tuple (source and destinations networks, ports, protocol) that specifies the traffic flow subject of the inspection. Such information is used by DPM to dynamically configure the CU, in order to classify and forward the traffic to the selected PU. It is worth noticing that, since the traffic classification rules may overlap (i.e.: two distinct analyses running on top of the same PU may have a network address range in common), set theory results applied to collections of ranges have been efficiently used to minimize the amount of traffic to be forwarded from the CU to the various PUs. Indeed, DPM relies on a generic multidimensional range algorithm to expand and reduce the classification rules in term of generic tuples, as well as to arrange them to fit with the longest prefix match algorithm implemented in the CU.

3.3.5 On–The–Probe Advanced Processing Techniques

The development of a probe that plays an active role in the overall monitoring process increases the burden on such component, and therefore requires the adoption of a novel design paradigm where methodologies and functionalities are strongly aware of the available hardware. Indeed, in order to accomplish operations like capturing, classification, anomaly detection, flow discrimination and isolation, etc., *at wire speed*, a detailed knowledge of the hardware capabilities/bottlenecks, as well as the fine grained analysis of the available time budget for each micro–operation involved are required.

The attempt to come up with performance effective solutions to be integrated into the front–end stage must then pursue the investigation of *stateless* and *memory saving* approaches with *constant look–up time* in that they tightly reflect into faster operations since they can take advantage of layered caches available in today's off–the–shelf multicore processors.

In particular, the use of Counting Bloom Filters for statistical data processing turns out to be extremely flexible although the fixed size of bins may cause memory inefficiency. A significant improvement can be obtained by allowing dynamic size of bins, compression, and multi–layering, as reported in section 2.5. These modifications appear applicable to the data processing performed by the probe.

For example, let us consider a set of rules used to classify flows at the front-end: a CBF can easily be used to represent the set. In order to verify whether a packet obeys one of the rules of the set, a simple lookup operation consists of evaluating k hash functions and comparing the values in all resulting bins to zero. If the result is positive, the packet satisfies the rule with small and predictable error probability, and can be exported for further processing.

Several papers have been published that describe in detail the application of such technique to specific monitoring applications. Among them, [145] and [146] have been devised within the PRISM research project.

3.3.6 Actual Monitoring Applications: A Practical Use Case

In order to better illustrate the above discussed features, we describe here a possible use case scenario where the proposed architecture allows to meet high–performance demands while being privacy preserving at the same time.

Let us assume the smart probe is used to protect a company network from external attacks by monitoring its gigabit ingress/egress link. In particular, let us suppose that two monitoring applications are used: a *scan detection* application that flags anomalous behaviors, and a TCP SYN *flooding detection* application. Both applications would raise significant performance issues if they had to process the whole traffic flowing through the link. In order to avoid that, special pre–filtering functions are installed on the Capturing Unit: the scan detection application will receive only the headers of the traffic entering the network, while the TCP SYN flooding detector will be fed with the headers of both incoming and outgoing TCP segments. The classified traffic is shipped by the CU onto batch frames and forwarded to separate PUs; at this stage, the data rate turns out to be significantly reduced. With a standard trimodal packet length distribution, a quick back–of–the envelope calculation shows that each processing unit needs to process less than 20 MBps of traffic, which is affordable with current off–the–shelf hardware.

The amount of flows to be processed, however, has not been reduced and keeping per–flow state is still unfeasible, due to its excessive memory footprint. To this end, PU processing should be carried out in a quick and stateless manner, which can be achieved by using probabilistic data structures. In particular, the method proposed in [147] provides a good heuristic for TCP SYN flooding detection while the one proposed in [146] can be adopted for fast stateless scan detection. Such a second–stage filtering operated at the PUs, is used to select suspicious traffic, which can be legitimately conveyed to an external collector for further analysis (as the volume of such data is likely to be very low, stateful and more complex analyses can be carried out), and discard legitimate traffic, which is likely to contain privacy sensitive information and will never leave the probe.

3.4 Design and Development of an OpenFlow Compliant Smart Gigabit Switch

OpenFlow [148] has recently been proposed as a switching paradigm that allows a network or data center operator to arbitrarily control routing without being constrained by the existing protocols. The applications of such mechanism are manifold: a data center operator can decide to move an instance of a server (running on a virtual machine) from one physical node to another without changing its IP address and without even interrupting the current tcp connection. This can be done in order to ensure reliability (in case of a node fault) or to save energy by switching off a portion of the network in a light load condition. Other application of Openflow include network virtualization and experimentation of new routing protocol. Indeed, despite its original goal was to allow researchers to evaluate new network architectures, Openflow is receiving more and more interest also in the industrial fields, and big equipment manufacturers such as NEC and CISCO are designing and deploying Openflow switches. The big advantage of the OpenFlow paradigm is the separation between the data plane (whose functionality is fixed and which can be optimized for high performance) and the routing intelligence, which is left for the user to implement through a well defined standard interface. In particular, the user is free to make a-priori switching decisions based on a flexible definition of flow or to take decisions on demand when a packet belonging to an unknown flow is detected. However, despite it offers a wide flexibility for intelligently tweaking network routing, OpenFlow seems to be slightly limiting with respect to other classes of network functionalities which may equally benefit from the smart switches. An example thereof are network monitoring applications, which may use an OpenFlow switch as a demultiplexer in order to dispatch packets and flows to an array of software based sensors. Another useful appliance would be an application-aware switch, which may demultiplex packets based on a the presence of a certain pattern in their payload (in turn, revealing a particular network based application). Such use cases cannot currently be handled by the standard OpenFlow protocol, which limits the definition of a flow to a 10-tuple of fields extracted from layer 2-4 headers. For these reasons, we propose a novel switching architecture which, unlike OpenFlow, is based on regular expressions.

A regular expression, also referred to as regex or regexp, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification. Chapter 2 reports a more thorough discussion about pattern matching and finite automata.

In our case, instead of defining a flow in terms of a tuple (with possibly undefined values), we define it in terms of a pattern described as a regular expression. Such a different approach allows to define a flow in a very flexible way: each field of the packet can be "wildcarded" or assigned a set of alternative values (by OR–ing together several expressions) and, if needed, the definition may also describe patterns observed in the payload. Let us for example assume that all RTP traffic needs to be forwarded through a given port: as RTP port numbers are notoriously dynamically assigned, that cannot be achieved by just observing the OpenFlow 10-tuple. However it is easy to specify in terms of regular expressions the patterns in the payload that reveal the presence of RTP streams. In this section we will describe in more details two specific use cases of the architecture we propose: a first one describes how our architecture can be used to implement a full-fledged standard-compliant OpenFlow switch (that implies that OpenFlow is somewhat a subset of the cases our proposal is able to handle). We illustrate a prototype of a node implementing our architecture in hardware over a NetFPGA [149] board. In this case, we also show that our solution provides a relevant performance enhancement over the current state–of–the–art implementation of an OpenFlow switch over the same architecture, in that a higher number of flow definition rules can be supported, while still reaching line-rate speed. In a second scenario we describe how, by leveraging common statistical properties of the packet distribution, our architecture can be used to implement a traffic load balancer.

3.4.1 Related Works

In the last years, a large number of solutions have been proposed in order to lead researchers and developers to evolve networks, and so accelerate the deployment of improvements, and create a marketplace for ideas. OpenFlow protocol is the most promising one. It was born as an abstraction of Ethane's datapath [150] but it guarantees more flexibility thanks to its modular approach. Moreover, we believe that it could be possible to obtain more flexibility defining a flow tuple in terms of a pattern described as a regex. Naus et al. [151] propose a possible implementation of the OpenFlow datapath on NetFPGA. The whole NetFPGA SRAM is used for the rules and the output queues are placed in the DRAM. This system allows to create up to 64000 exact match rules and up to 32 wildcard rules while our system guarantee up to 100000 wildcard rules using half-SRAM. In this way, it is possible to use the other portion of SRAM for the output queues avoiding a loss a performance due to the bigger latency of DRAM with respect to the SRAM memory. If we move the output queues in DRAM and we use the whole SRAM for our data structure we could reach approximately up to 200000 rules considering a linear growth of the memory consumption with respect to the number of rules. Greenberg et al. [152] propose a clean slate approach to network control and management called 4D. In the 4D architecture, the routers and switches simply forward packets at the behest of the decision plane, and collect measurement data to aid the decision plane in controlling the network. However, 4D does not provide fine-grained, per-flow control over the network. Molinero-Fernandez and McKeown [153] propose a technique called TCP switching in which each application flow triggers its own end-to-end circuit creation across a circuit switched core. Based on IP switching, TCP switching incorporates modified circuit switches that use existing IP routing protocols to establish circuits. Routing occurs hop by hop, and circuit maintenance uses soft state, that is, it is removed through an inactivity timeout. OpenFlow is more powerful because delegates decisions to a controller and the decisions are made once per–flow, not hop–by–hop. In addition, with our architecture we could also implement this system by using regexes without significant changes in the source code. Finally, Click [154] is a software architecture for building flexible and configurable routers. Using the NetFPGA hardware we are able to support the full line-rate with respect the Click software implementation that achieves a maximum loss-free forwarding rate of 333,000 64-byte packets per second.

3.4.2 NetFPGA board

NetFPGA [149] is a low-cost platform, developed by the High Performance Networking Group at Stanford University, primarily designed as a tool for teaching networking hardware and router design. It is a standard PCI card that plugs into a standard PC. The card contains a Field Programmable Gate Array (FPGA) by Xilinx (Virtex-II pro) which is programmed with user-defined logic and has a clock of 125 MHz. The PCI interface connecting the host PC to the NetFPGA is managed by a small Xilinx Spartan II FPGA. Four 1GigE ports, 4.5MB of Static Ram (2 banks) and 64MB of DDR2 Dynamic RAM are also on board in the card. A reference package containing verilog source code for the FPGA, C code for the host PC and java code for the graphical interface can be downloaded from the NetFPGA website in order to run NetFPGA with basic networking functions such as Network Interface Card (NIC), PW-OSPF IPv4 Router and Layer 2 switch. The basic target for this board is the adoption of an FPGA as a networking accelerator in order to take advantage of the host PC flexibility to implement the control plane of the project. In this scenario, for example, the user could implement the forwarding plane of an IPv4 router in FPGA and the control plane (with its routing algorithm) in the host PC connected to the card via PCI. Thanks to its modularity, NetFPGA is a very useful system to test new ideas for next generation networks.

3.4.3 Pattern Matching Engine as a Gigabit Switch

Switches may operate at one or more OSI layers, including physical, data link, network, or transport (i.e., end-to-end). A device that operates simultaneously at more than one of these layers is known as a multilayer switch. The process of IP packet forwarding depending on arbitrary metadata (i.e., one or more OSI layers) contained in the packets themselves is logically (and practically) equivalent to perform *pattern matching*. The search in the forwarding table can therefore be obtained by simply applying pattern matching algorithms upon the associated fields of the IP packets as in classification problems [155]. However, our scheme supports forwarding rules defined over arbitrary metadata. As pattern matching is a widely addressed topic in literature, the above observation opens a wide horizon of theoretical and practical solutions to address the problem of lookup and classification. In recent years, due to the increasing interest focused on *deep packet inspection*, the use of regular expressions (regexes) has become more and more popular because of their high expressiveness in describing sets of strings [44]. Typically, finite automata (FAs) are employed to implement regular expression search, but for the current string sets they need a memory amount which turns out to be too large for practical implementation. Many recent works have proposed improvements to address this issue. They are adopted by well known IDS tools, such as Snort [45] and Bro [46], and in firewalls and devices by different vendors such as Cisco[47]. However, Finite Automata suffer from either speed issues (if they are non deterministic) or size ones (if, on the contrary, they allow deterministic lookups). For these reasons, many works have been recently presented with the goal of memory reduction for DFAs, by exploiting the intrinsic redundancy in regular expression sets [48, 49, 50, 51]. We have chosen for our scheme the δFA (which has been already described in section 2.1) for its interesting performance characteristics. In particular, in addition to maintaining a data structure which is much more compact that the standard automaton, it needs a lower number of memory accesses than most compressed automata.

3.4.4 The Smart Switch

Our aim is to develop an OpenFlow compliant smart architecture on NetFPGA that could be also used in systems where load–balancing is needed. Our implementation can hold more than 100000 flow entries and it is capable of running at line-rate across the four NetFPGA ports. We stored the δFA structure in half-SRAM of NetFPGA, leaving the rest of SRAM for the output queues. In this way, with respect to the original implementation of OpenFlow on NetFPGA [151], we prevented from risks of queue overflow (i.e., putting the output queue in BRAM, the fastest on-chip memory usable in FPGA design) and, at the same time, we tried to guarantee the line rate avoiding to put the output queues in DRAM. While the hardware takes care of finding the longest prefix match in the δFA structure, the software level manages this data structure, by inserting and removing the rules in the forwarding table. When a packet does not match any rules it is sent via PCI to the PC-Host of the NetFPGA that will take care to send it to the Controller through an SSL connection. Moreover, if no OpenFlow compatibility is requested, a local running software on the PC-Host could provide a simple command line interface for the user in order to manage the forwarding rules.

3.4.4.1 Software Plane

As above mentioned, the software level of the Smart Switch takes care of creating and managing the δFA data structures as well as storing them into the NetFPGA SRAM. We extended the OpenFlow reference software implementation in order to map the rules created by the OpenFlow protocol in a δFA data structure and to store it in the NetFPGA SRAM. Any time a new change occurs in the OpenFlow rules table, a new recalculation of the δFA data structure is required and so is the consequent write operation in the NetFPGA SRAM. If OpenFlow compatibility is not requested, the user could write a simple text file to specify the rules and the associated output port. A software, then, is in charge of creating from the rules the associated standard DFA and convert it to the δFA structure in order to store it in the NetFPGA SRAM.

3.4.4.2 Hardware Plane

The core module of the switch is shown in figure 3.30. The first operation performed on the incoming packet is parsing the fields of the packet in order to compose the string which will be fed into the δFA state machine. The "Metadata Exctractor" block extracts the right fields (i.e.: in the implemented prototype the OpenFlow 10-tuple). The obtained string, then, is used for querying the cache (see subsection 3.4.4.2). If a miss is obtained, the "Pattern Matching Engine" starts walking through the δFA data structure stored in SRAM using one character at a time. The result (i.e., the output port associated with the flow) is then written in the Cache. If the system is configured to work in an OpenFlow version, when a packet does not match any rules is sent to the PC-Host where the OpenFlow software is running. Then, thanks to the Controller, a new entry related to that flow will be inserted in the flow table. Otherwise, if no OpenFlow compatibility is requested, the hardware plane will send the packet without any rules associated with a default port. As already

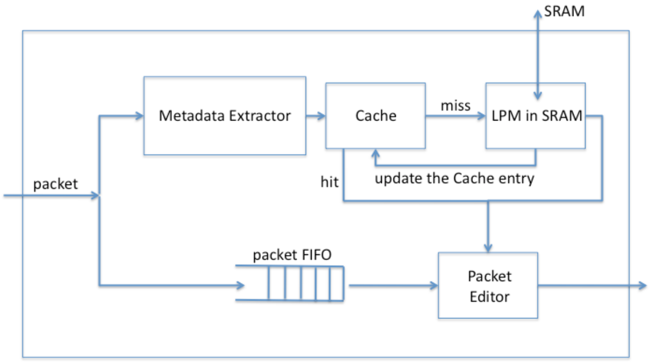


Figure 3.30: Core module of the Smart Switch.

mentioned in subsection 3.4.4.1, the software plane is in charge of updating the δFA data structure in SRAM. To avoid loss of data during the rules update in hardware, while the software plane is writing in SRAM only the Cache is enabled.

Caching An ordinary way to speed up packet forwarding is caching flows. Packets belonging to the same flow are likely to exhibit good temporal locality, and the result of the search in the forwarding table can be cached and used for the forthcoming packets. Therefore, it is useful to introduce a flow–cache, where a new entry is added when the first packet of a new flow enters the system. In this case, the switch performs a lookup in the δFA as mentioned in subsection 3.4.4.2 and stores the result in the flow cache. Otherwise, for each packet belonging to a known flow, the forwarding result is already in the cached data and the amount of memory accesses is reduced. Since the number of flows can be very high, a hash table is an efficient way to implement such a cache. To avoid collisions, we implemented a Perfect Hash Function (PHF) through double hashing. The basic idea to create a PHF is using a two-level hashing scheme with universal hashing at each level. In the first level, the n keys are hashed into m slots by using a hash function h carefully selected from a family of universal hash functions. To handle collisions in a slot j, a small secondary hash table S_j with an associated hash function h_j is used. By carefully choosing the hash functions h_j, we can guarantee that there are no collisions at the secondary level.

However, we need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j. While having such a quadratic dependence of m_j on n_j may seem likely to cause the overall storage requirements to be excessive, it has

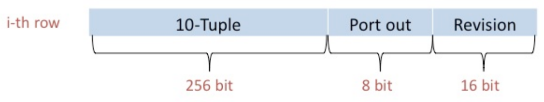


Figure 3.31: A row in the BRAM implementation of the Cache.

been shown that by properly choosing the first level hash function, the expected total amount of space used is still $O(n)$. In our current implementation, such a structure is kept in two different blocks of BRAM memory. In order to provide on–the–fly reconfigurability, we also inserted the field "revision", as shown in fig. 3.31. Every time an update in SRAM is done, the global variable revision is incremented. This way, by comparing the global revision value to the one associated with a given flow, it is possible to check whether the entry needs to be updated or not.

3.4.5 Load Balancing

In this subsection we describe how our architecture can be used to implement a traffic load balancer by leveraging common statistical properties of the packet distribution. The analysis of many forwarding tables of real devices reveals that the most significant 16 bits of rules are almost always completely specified and do not present wildcards: in other words, the prefix lengths are greater than 16 [156]. This result is not surprising because of supernetting; however it points out that a "smart" switch (it is worth reminding that we indicate with smart switch a networking device that works not only at layer 2) must be able to manage a large number of packets with destination IP addresses that broadly differ in their less significant bits while they do not differ significantly in the most significant ones. For this reason, we focused on the design of a load balancer by just exploiting this heuristic and by using the wildcards in the most significant bits only. In a first step, we performed some analysis on real traffic traces in order to verify if such idea provides a uniform distribution of the less significant bits of the IP destination address. To this aim, we read traffic from a 9GB long *pcap* trace collected from a local network and we evaluate the statistical properties in order to verify if this idea could be used.

Figures 3.32 and 3.33 show the histogram of destination IP addresses with the 6 and 8 less significant bits fixed, respectively. These first results show a good uniformity and allows us to test our system using the wildcards in the most significant bits. The experimental results will be shown in subsection 3.4.7.1. Notice that, being the regex aligned on bytes, if the number of fixed bit used is not multiple of a byte we can just OR-ing two different regexes in order to obtain the desired granularity.

3.4.6 Device Utilization

We compared the complexity of our Smart Switch working as an OpenFlow compliant device to the original OpenFlow running on NetFPGA as implemented in [151].

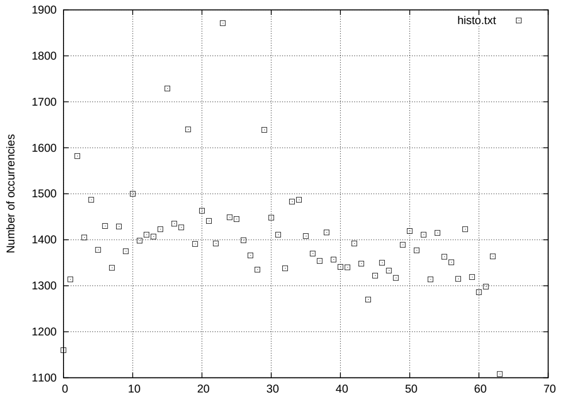


Figure 3.32: Histogram of distribution of destination IP addresses with the 6 less significant bits fixed.

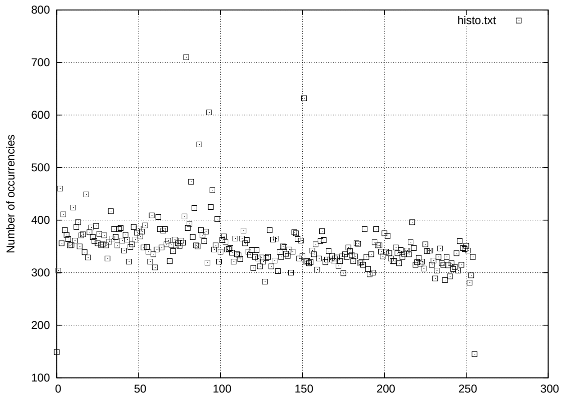


Figure 3.33: Histogram of distribution of destination IP addresses with the 8 less significant bits fixed.

The build results for the two designs are shown in table 3.3. These results were obtained by using Xilinx's implementation tools from ISE10.1.03. We report the values of logic utilization and logic distribution considered in percentage. These results suggest that in terms of logic, our implementation is better than the original. This is a rather satisfactory result, even considering that our solution can handle a larger number of rules. The core processing of the smart switch resides in SRAM, where all the δFA structure is stored. This way, the FPGA chip only takes care of i) extracting the right fields from the packets ii) walks the automata, and, at the end, iii) stores the obtained result in Cache. In order to implement a very good cache with a perfect hash function (as described in 3.4.4.2), we needed a large amount of BRAM. For this reason our percentage equals that of the original solution (that employs 8 parallel TCAMs using Xilinx SRL16e).

Resource	Logic Utilization (%)		Logic Distribution (%)		
	Slice Flip Flop	4 input LUTs	Occupied Slices	Total of 4 inputs LUTs	RAMB16s
Smart Switch	32	42	66	49	65
OpenFlow on NetFPGA	47	75	91	80	65

Table 3.3: Comparison in resource utilization.

Load Balancing	Trace	Total length (MB)	Total # of pkts (Kp)	Interface 1		Interface 2		Interface 3		Interface 4	
				Mbytes	pkts (Kp)	Mbytes	pkts (Kp)	Mbytes	pkts (Kp)	Mbytes	pkts (Kp)
Our Solution	Trace 1	22.17	36.02	8.1	10.38	3.01	5.71	2.52	6.39	8.52	13.54
	Trace 2	9041.98	15471.47	2945.68	4515.77	1851.51	3255.43	1667.36	3362	2577.41	4338.25
Hash	Trace 1	22.17	36.02	2.79	5.85	5.81	9.43	3.27	6.45	10.3	14.28
	Trace 2	9041.98	15471.47	2071.72	3850.88	2473.51	4180.01	1925.17	3395.57	2571.56	4044.99

Table 3.4: Load balancing feature.

3.4.7 Experimental Results

3.4.7.1 Load Balancing

As motivated in subsection 3.4.5, we hereafter describe a set of experiments where our architecture is used for load balancing purposes. We used TCPreplay to reproduce two different real traffic traces collected from a local network. Tab. 3.4 shows the results on the number of packets (as well as their volume in MB) sent for each physical port of the NetFPGA board, using rules with the 24 most significant bytes specified. We compared the results of our solution to those that can be obtained by means of a common load balancer based on a hash function (i.e., we used the one in the std library of the C++). Our solution works, approximately, as a hash-based solution and does not require any particular implementation.

Despite the traces that we used are affected by address locality (indeed, they are not taken from core links) the obtained results are encouraging. In this sense, we tested the load balancer in a worst case scenario: indeed, we expect that its use in a core network context (where addresses locality does not occur) would reflect in much higher performance.

3.4.7.2 Throughput

In this sets of experiments we evaluated the throughput of the smart switch. We carried out several tests by using a Spirent AX 4000 hardware based traffic generator. Such a device is able to completely saturate a Gigabit link with minimum sized packets, thus recreating the worst case scenario for a network device performing packet–by–packet processing. For this reason, we actually performed all tests with minimum sized packets. We inserted static rules that covers all the possibilities because we were not interested in the PCI throughput (i.e., a packet that does not match any rules is sent to the PC-Host via PCI) and we registered the processing time of our architecture. As the performance of our smart switch is strictly dependent not only on the packet rate, but also on the number of flows, which, in turn, reflects the speed-up introduced by the cache, we used the API generator in order to produce a high number of flows. In particular, the AX 4000 generator can inject packets whose addresses are randomly selected within user defined ranges, thus generating traffic where different flows are randomly interleaved. We point out that this scenario is probably more challenging than that of real traffic, as in the latter packets from the same flows are close to each other and the caching mechanism provides a significant speed up. We increased the generation rate until the link was completely saturated. As illustrated in figure 3.34 our switch is able to process all of the packets with negligible losses.

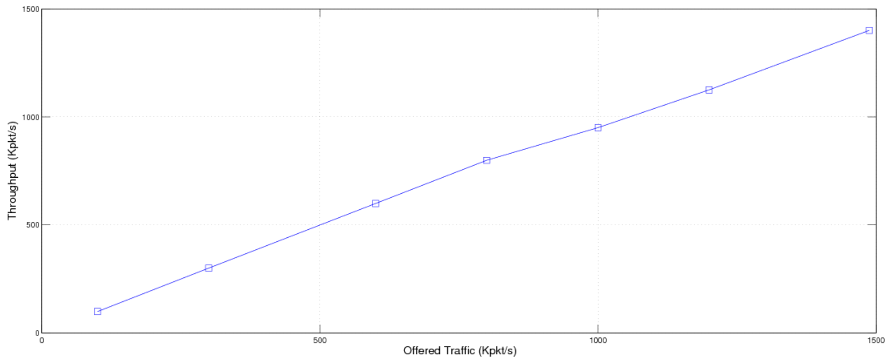


Figure 3.34: Throughput of the smart switch with growing rates of the traffic of interest.

Chapter 4

High performance packet processing on general purpose parallel platforms

General purpose processors provide the most suitable architecture in terms of flexibility and usability for building network monitoring systems. However, due to their performance limitations, commodity CPUs have always been deemed not to be able to keep the pace with incoming traffic on high speed links. In particular, it has often been claimed that the growth rate of the CPU clock would be slower than that of the link capacity and, in any case, limited by physical factors. Despite that, the recent evolution of general purpose hardware towards an increasing degree of parallelism may change such a commonplace and make generic CPU apt to high–speed monitoring again. In particular, commodity CPUs are providing an increasing number of cores and networking hardware is adapting to this trend by explicitly supporting concurrent access. New forecasts claim that the overall processing power of the cores in a CPU will grow again according to Moore's law. Although this trend opens a wholly new space of opportunities for the evolution of high performance network software, several challenges emerge. Most of the current packet processing software is still designed in a single–core perspective. Very often, even if concurrency is formally supported, this is ensured by means of *mutexes* and *locking*. While such mechanisms do guarantee correctness, they are not fit for high speed parallel processing and prevent the software from effectively scaling. In this chapter we intend to propose some new architectures that would allow network monitoring and testing applications to benefit from parallel hardware in a way that is as transparent as possible for the application writer. This is mainly done by hiding within our architecture *lockless* solutions for efficient handling concurrency and data sharing. In particular, in section 4.1 we sketch the architecture of a traffic generator which can produce synthetic traffic at very high rates by leveraging parallel hardware, while still retaining an extensible and modular architecture for supporting arbitrary traffic models. In section 4.2 we describe a novel packet capturing engine that is able to leverage the full potential of hardware parallelism while allowing the application developer to choose the degree of parallelism of the user–space software. In section 4.3, instead, we propose a complete framework for composing modular monitoring applications, named Blockmon, which provides the user with several mechanism for easily exploiting processor parallelism. Such a system is based around the block abstraction (a block being an independent and self–contained processing functionality) and a message passing paradigm. Blockmon allows the user to define a monitoring application as a composition of blocks and to easily reuse a base of highly optimized thread–safe code.

4.1 Multi–Gigabit Packet Capturing With Multi–Core Commodity Hardware

As already discussed in 3.1, reliable tools that can generate a realistic and verifiable traffic load are needed in order to discover weaknesses and faults in the network infrastructure before they become a cause of major damages, and to assess the capability of the equipment to keep working correctly even in the worst possible condition. On one hand, flexibility is needed in order to be able to mimic very specific traffic patterns, while on the other hand an ever increasing level of performance is required. Indeed, as the overall amount of the traffic carried by the internet is growing exponentially, the capacities of the links deployed on production networks is rapidly increasing (10 Gigabits are becoming more and more common). Therefore, a reliable testing tool needs to be able to generate an ever–growing amount of traffic. In spite of that, high performance traffic generation is a surprisingly understudied topic in the research community.

In many cases, the task of generating traffic to assess the performance of prototypes is delegated to hardware commercial devices. In order to reproduce different traffic models, such generators usually either provide a well bounded set of possible models (as in the Spirent AX 4000) or reproduce real traffic traces captured over the network (like Napatech). Both approaches show severe limitations: in the former there is an evident lack of flexibility, as the models are implemented in hardware and therefore there is no way of introducing new features. The latter approach, however, despite being more flexible, still has several limitations: first, relevant traffic traces are treated by the law as personal data and are hardly available for the purpose of testing (besides, storing such traces is officially forbidden). In addition, a real traffic trace is usually not suitable when there is the need of reproducing particular traffic patterns (e.g. the worst case scenario) which are not common in real traffic.

As opposed to the limited flexibility of hardware based generators, a broad range of software based generators are available, which usually allow a very easy configuration of the generated traffic stream (often with graphical interfaces) and are able to support a wide range of protocols. However, those generators usually provide relatively low performance: most of them are able to fill only a fraction of a 1 Gb link. The currently available software tools, in addition, were mostly designed to run on single core commodity machines and are not able to make the most of the latest commodity hardware.

Indeed, the evolution of commodity hardware is pushing parallelism forward as the key factor that can allow software to attain hardware-class performance while still retaining its advantages. On one side, commodity CPUs are providing more and more cores (the next-generation Intel Xeon E 7500 CPUs will soon make 10 cores processors a commodity product), with a complex cache hierarchy which makes aware data placement crucial to good performance. On the other side, server NICs are adapting to these new trends by increasing themselves their level of parallelism. While traditional 1Gbps NICs (such as the very common Intel pro 1000 cards) exchanged data with the CPU through a single ring of shared memory buffers, modern 10Gbps cards (such as those based on the Intel 82599 controller) support multiple queues: multiple cores can therefore receive and transmit packets in parallel. In particular, incoming packets can be demultiplexed across CPUs based on a hash function (the so-called RSS technology) or on the MAC address (the VMD-q technology, designed for servers hosting multiple virtual machines). The Linux kernel has recently begun to support these new technologies (the 2.6.32 kernel is already multiqueue–aware).

In this section we propose a novel architecture for packet generation, which effectively leverages NIC and CPU parallelism in order to combine the flexibility of the software based generators with hardware–class performance. In particular, our system is modular and allows a user to plug in new traffic models by writing a very limited amount of code; the massively parallel nature of the underlying system is completely hidden to the model developer and handled by the framework. The ability of effectively leveraging parallel processors guarantees that our architecture will keep scaling with the newer generations of CPUs, thus being able to attain higher and higher rates, the bottleneck of the single processor being eliminated. Multi-core aware design and implementation allow our generator to produce as much as 13 million packets per second on a commodity server, thus almost reaching wire-rate on a 10Gb link. In addition, differently from most of the solutions which need modified device drivers, it achieves such performance with vanilla drivers.

4.1.1 State–of–the–Art in software–based packet generation

One of the most crucial performance show-stoppers in packet generation is the capacity of the socket which is used for sending packets down to the network interface. Therefore, we will briefly point outs the limitations of the state–the–art solutions.

Packet transmission sockets. Most of the current software based generators use the PF_PACKET socket, which comes with the standard Linux distribution and also supports a memory-mapping mode that allows to increase performance. However, such a socket has been designed in a single–core perspective and shows some limitations with respect to the modern architectures.

By accurately delving into its source code, we found several bottlenecks that we will describe in the following. The most evident weakness of `PF_PACKET` is that it does not allow to select a specific hardware queue for transmission when used on top of multi-queue NICs; this results in thread serialization when multiple threads send packets on the same device (no matter if they share the same socket or not) and it clashes with the base purpose of queue parallelism. In addition to this, `PF_PACKET` is based on a per-packet system call, like most of the classic I/O mechanisms. Such a dated design represents a remarkable overhead. Although the cost of the system–call can be amortized by means of batch–transmission (recent kernels introduce `sendmmsg`), such a system–call is hardly useful for a traffic generator, as there is no way to specify the inter–departure times for packets in the batch.

Another bottleneck of `PF_PACKET` is represented by the `copy_from_user` function, commonly used to transfer the packet payload to the kernel, and whose overhead is well-known to be higher than that of a normal `memcpy` performed to a memory-mapped region. However, this limitation can be worked around by using the recently introduced memory mapped version of the socket. Furthermore, packets transmitted by this socket are not directly conveyed to the NIC device driver but go through a series of mechanisms which brings several additional overheads. In particular, packets are also sent to registered sniffers, and therefore toward other open sockets. Since there is no way for a `PF_PACKET` socket to be used exclusively for transmission, this results in a severe performance penalty, especially in a multi–core scenario, where several sockets are in-use for parallel transmission (i.e. one socket per thread). Furthermore, when packet transmission is performed, the Linux traffic control (TC) also comes into play. This, in turn, involves an additional overhead even if no TC egress class is specified for the network device. This forces the packet transmission to be performed asynchronously by a different context, impacting adversely on the precision of the transmission time.

Recently, different solutions for improving the efficiency of Linux networking I/O have been proposed. Such solutions do provide good performance and are usually able of saturating a link with minimum sized packets. However, their scope is essentially different from that of our work, as they do not come with an integrated framework for synthetic traffic generation and are based on heavily patched drivers.

In [157], the authors present Packetshader, an extremely well performing software router, which is built around GPU acceleration of computation intensive and memory intensive functionalities (such as address lookup). Also, it relies on a heavily modified driver which introduces several optimizations, such as using a reduced version of the socket buffer structure and preallocating huge buffers to avoid per–packet memory allocations.

Netmap [158], a BSD based project, integrates in the same interface a number of modified drivers mapping the NIC transmit and receive buffers directly into user space. Deri recently released a heavily patched driver to be integrated into the well known PF_RING architecture [159], which allows to reach wire speed both in generation and in transmission, when simple test programs are used. However, as reported in [160], integration of such a system with a non multicore–aware traffic generator brings a significant performance decrease (the maximum packet rate being about 6 millions).

Software based traffic generators. Several open-source tools for traffic generation on commodity PCs have been proposed over the years, most of them designed for the Linux Operating System. KUTE [161] (an evolution of the former UDPgen) is an UDP traffic generator which is designed to achieve high performance over Gigabit-

Ethernet. It is based on a Linux kernel module that operates directly on the network device driver bypassing the Linux kernel networking subsystem. However its performance is reported to be low and the project has not been supported for some years. RUDE [162] is able to instantiate simultaneous patterns of traffic, but it does not provide any explicit support for extensible interfaces and is not suitable to work at high rates, especially with small frames, as shown in [15].

MGEN provides both a command line and a GUI for user-friendly traffic generation in user-space. It runs on different Operating Systems such as FreeBSD, Linux, NetBSD, Solaris and Windows but its accuracy is limited by the system timers it is based on (e.g.: in the Linux kernel on PC-platforms, the timer resolution used by MGEN is only 10ms [162]). The Internet Traffic Generator (ITG) [163, 164] aims at reproducing TCP and UDP traffic and replicate appropriate stochastic processes for inter–departure time and packet size. It is based on several distinct processes that are connected through Inter Process Communication and can actually support parallel generation. It is able to achieve a performance level comparable to that of RUDE and MGEN but provides more traffic patterns and runs also under Windows^{TM}. A version thereof [165] has been proposed for distributed measurements, but the authors reports its generated traffic to be below 650 Mb/s. Brute [15] is probably the best–performing software based traffic generator among the currently available ones and has an extensible modular architecture. However, as the other competitors, its design does not take parallelism into account and, therefore, it cannot scale properly on multi–core platforms. In particular, we measured its peak rate to be around one million packets per second. Ostinato [166] is a very recent traffic generator, which enables very flexible definition of the traffic flows via a graphical interface. However, its performance is reported to be quite poor when it comes to generating high traffic rates [167]. As already mentioned, [160] shows that, even when the packet transmission bottleneck is removed by using a properly modified driver, the design of Ostinato (which is not multi–core aware) limits the overall obtainable performance. Pktgen [168] is a software based traffic generator that runs within the Linux kernel directly, thus avoiding the overhead of communicating with a user space application. However, such a design choice limits its flexibility, as the range of traffic patterns it can generate is fixed and quite limited.

[169] gives an interesting insight about the level of performance and accuracy software based packet generators can reach. In particular, it compares MGEN, ITG and RUDE and shows that they cannot comply with the requested packet rate even at reasonably low speeds (well below 1Gbps). The paper also gives an interesting insight about how the OS scheduling interferes with packet generation and suggests polling as a way of improving both accuracy and performance (in fact, this is the solution we use in our architecture). [170] investigates further the correlation between OS scheduling and traffic generation accuracy.

Our approach. Our approach, whose cornerstone is the integrated co-design of kernel–space and user–space components, shows several significant contributions. In particular, with respect to the latest generation sockets:

- it adds a modular architecture which allows to transparently leverage parallelism for generating arbitrary traffic models. This is not trivial because, as reported, a non–aware design of the user space portion can cause a huge performance loss even with highly efficient sockets;

- it allows to attain high rates with non modified drivers, thus being able to cover a much broader range of hardware platforms.

With respect to user–space only traffic generators, instead, our design allows to attain much better performance, while still retaining a total flexibility with respect to the traffic models that can be generated. In particular, we propose an open architecture that users can extend to fit their needs.

4.1.2 The generator architecture

Our modular architecture is made up of several components. First of all, a set of parallel traffic transmitters is in charge of actually sending the packets to the NIC. Such transmitters are implemented as a novel socket named `PF_DIRECT`, along with an active context implemented as kernel–space thread. The context is in charge of either managing a specific device, or a hardware queue (for devices supporting them – notice that most of modern 10G NICs are equipped with hardware queues). `PF_DIRECT` sockets are fed with data to send by a set of traffic generation engines, which represent the user space threads that generate the global traffic streams out of a set of independent models. From a certain perspective, an engine is nothing but a discrete event simulator, which keeps ordered and updates the events generated by the traffic models; however, in order for the transmitters not to run out of data, the engine has real time requirements.

The traffic models are plug-ins that generate an ordered sequence of packets; they can set at will both the inter–departure times and the packet payloads, thus leaving the maximum freedom for implementing new models. In order for the system to avoid any data contention (which would impact on the performance), the described entities are associated into independent groups (user–space threads): each engine has a separate set of models to handle and a separate set of transmitters to feed. Notice that models served by the same engine are guaranteed to be strictly ordered with respect to the inter–departure times of packets (unless multiple hardware queues are used by the same engine, which may involve occasional reordering). Instead, models associated with different engines are independent and their synchronization relies on a common timer only. This architectural constraint is required to avoid the high cost of handling a shared resource among multiple cores. However, we point out that this is perfectly acceptable: as a use case, an engine could be serving a set of models which create an attack pattern, while a parallel one could be in charge of simulating a background traffic. Our architecture leaves a high degree of freedom in choosing different configurations, which can be specified as XML files.

4.1.2.1 Traffic Transmitters: PF_DIRECT

As already discussed, the Linux kernel places several bottlenecks on the path between the user space application and the NICs. In order to avoid this limitation we developed a novel socket, named `PF_DIRECT`, which allows efficient and scalable packet transmission.

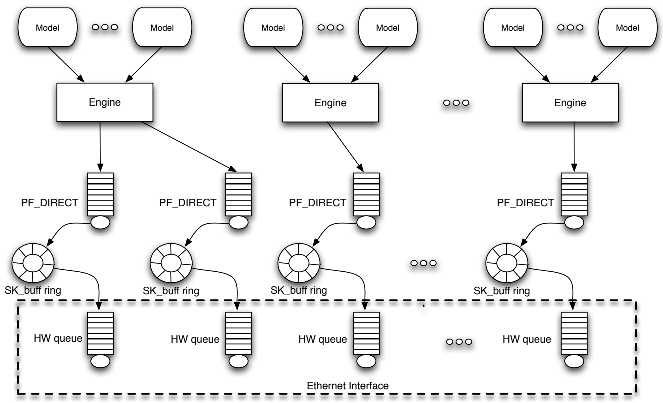


Figure 4.1: Traffic generator architecture

The novel PF_DIRECT socket. The internal architecture of our socket is depicted in Figure 4.1 and is made up of the following components:

- a memory–mapped queue for payload and meta–data;
- a pool of pre–allocated socket buffers;
- a direct interface to a hardware queue.

We point out that a PF_DIRECT socket is bound to a network device or to a specific hardware queue and, being asynchronous, it has its own thread of execution in charge of transmitting packets. In addition to this, it is supposed to perform active waits in order to precisely reproduce the inter–departure times generated by the models. The SPSC (Single Producer Single Consumer) queue is the communication channel between the application and the socket. The queue consists of a memory–mapped area and two indices, specifying the last read position (which is written by the engine context and read by the socket context) and last written position (which, conversely, is read by the engine and written by the kernel context). Such a simple implementation gives a two-fold advantage: on one hand it avoids the performance cost of a system call, on the other it provides a wait–free mechanism for data sharing. The queue is used to convey both the packet payload (which is generated by the models) and some associated meta-data: packet length and, most importantly, transmission time.

Notice that the timing of the whole system is based on absolute timestamp-counters (TSC) available as a 64 bits register in modern CPU. TSC provides a number of advantages: on one hand, reading the value of the TSC register is much quicker (a few cpu cycles) than any other call/mechanism for reading the current time; besides, working with absolute time–points allows an easier and flexible dispatching of packets across multiple cores, which can follow a precise ordering of transmission without requiring inter-core communications. On the other, absolute time–points optionally leaves to the socket the possibility to recover from a late transmission by anticipating the following one, which would not be possible if the inter–arrival policy were specified. However, for multiple cores to work with absolute times, a common source of clock is required. Most server–class Xeon processors, for example, support the INVARIANT_TSC capability, which guarantees the timestamp counters on the different cores to be consistent. In case such feature is not available, an initial calibration procedure can be used to compute per–core offsets: in particular we can have all of the cores actively polling for a given atomic variable, then immediately read their own timestamp counter. Of course this may involve a small calibration error and we plan to devise a correction method based on multiple subsequent measurements as a future work.

The second component of the socket is a ring of pre–allocated socket buffers (`sk_buff`). While [157] reports allocation and initialization of the sk_buff structure to be one of the main bottlenecks in the Linux networking subsystem, using such a structure is mandatory in order to work with vanilla device drivers. Therefore, `PF_DIRECT` allocates a pool of such structures at initialization time and cycles through them at run time: in particular, the payload and part of its metadata are copied from the shared queue into a socket buffer of the pool (at a negligible cost). The usage count of such a socket buffer is then forcibly incremented before the transmission, so that it is not deallocated by the device driver once transmitted.

The ring of socket buffers is required to amortize the latency of the clean-up routine: commonly it is used in drivers to free the socket buffers, while, in the case of `PF_DIRECT`, it just notifies that such a buffer is ready to be reused. The last component of the `PF_DIRECT` socket is the transmission routine, a direct interface to the device driver: it allows to skip the traffic control machinery that, as already pointed out, has a negative impact on the precision of the inter–departures time of packets.

4.1.2.2 Traffic Engines

The traffic engines are completely user–space threads of execution whose main function is generating an ordered stream of packets from a model set and dispatching them across a set of `PF_DIRECT` sockets (packet transmitters). For this reasons each engine keeps the models ordered in a heap according to the scheduled transmission time; they continuously extracts the first model from such a heap (i.e. the one with the closest transmission time) ans push its associated packet into the queue of one of the transmitters. The selection of most appropriate transmitter can be made according to a number of different criteria; we recall here that the packet transmitter perform active waiting until the intended transmission time for the packet is reached. The implemented criteria are:

- round robin: this policy provides good load balancing in terms of number of packets and is very simple;
- affinity: as the transmitters are synchronized and transmission is based on an absolute time–point, the packets generated from the models should be transmitted in order (i.e. in the order they are popped from the heap). However, if

transmission times are close to each other and packets are assigned to different transmitters, minor timing errors can result in packet reordering. If a traffic model can by no means tolerate reordering, it can specify an affinity value, so that all of its associated packets are assigned to the same transmitter and strict ordering is enforced.

As above mentioned, in this preliminary prototype we only implemented these two policies: the implementation and evaluation of more involved schemes are left to future work.

4.1.2.3 Traffic Models

The traffic models are the components which are in charge of defining the traffic flows generated. Models are in fact plug-ins conforming to a simple interface and can be easily added by the user through a factory pattern implemented in C++. The interface is defined as an abstract base class. Such a base class provides three essential protected methods to the derived classes (i.e. the actual models): one for accessing a buffer where the data to be transmitted is stored, one for setting the packet length, and another one for defining the inter–departure time of the packet. For the former, notice that defining a new packet involves no data allocation: the model just needs to overwrite the fields that change on a packet by packet basis. As for the latter, the choice of having a model work in terms of relative inter–departure times is intended to provide the maximum degree of abstraction to the model developer; conversion to an absolute time–point expressed in clock cycles is automatically performed by the framework. In order to define a new model, a developer has to implement two simple methods: one intended for configuration of the model's internals (e.g. a Constant Bit Rate – CBR – model needs to be provided a rate value) and called at start-up time, one called to schedule the transmission of a new packet when the previous one has been dispatched to a transmitter. We point out that our simple interface allows to reproduce even complex traffic model (we are currently working on a TCP emulator that should mimic two state machines on a network path). However, due to its performance penalty, closed loop generation is not supported.

As the packet–by–packet update method is on the fast data path, it is up to the developer to make it as quick and efficient as possible: a slow method can result in the transmitter queues to become empty and, in turn, in the generated traffic stream to have huge gaps and bursts (if the recovery algorithm is enabled). If a model needs an unavoidable degree of complexity, it is a good configuration choice to segregate it on a specific engine, so that it can have dedicated resources and cannot interfere with the rest of the traffic.

4.1.3 Experimental results

In order to assess both the precision and the performance of our traffic generator, we carried out a number of tests by using different traffic analysis tools. Our generator always ran on a server–class machine, whose cost is below 2000$, which can be reasonably considered to be a commodity platform. Such a machine comes with a 6 cores Intel X5650 Xeon (2.66 Ghz clock, 12Mb cache), 12 GB of DDR3 RAM, and an

Intel E10G42BT NIC, with the 82599 controller on board. In order to test our system with the maximum degree of parallelism, we kept Intel Hyperthreading enabled, thus carrying out the experiments with 12 virtual cores. The server runs Linux with the latest 3.0.1 kernel and the *ixgbe* 3.4.24 NICs driver.

In order to analyze the traffic produced by our generator, we used both commodity and special purpose hardware. A first set of tests is performed with the Spirent AX4000 hardware based protocol analyzer, which provides high resolution packet timestamping but, unfortunately, is not equipped with a 10Gb interface, thus limiting the maximum traffic rate to 1Gbps. We used it in order to obtain a very reliable characterization of the inter–arrival times of our generated traffic at relatively low rates (under 1 Mpkts/sec). As for 10Gb measurements, we had to rely on a software solution running on a server which is identical to the one we used for generation. We measured the overall rate of the generated traffic by means of PFQ, a Linux kernel module designed for packet capture on multi-core architectures whose details will be described in section 4.2. We captured the traffic generated by our system on a separate host (its hardware configuration being identical to the one of the generator); we also configured PFQ in order to timestamp the packet as soon as it was captured, in order to achieve the best resolution available on a software platform. Overall, the performance evaluation here carried out will be further expanded in future works. In particular we plan to perform a more detailed investigation of the timing precision of our generator and to develop further its calibration mechanisms. In addition, the generator will be compared to a broader number of other generators on the same platform.

4.1.3.1 Up to 1 Gb/s rates

The first set of tests aims at assessing the precision of the traffic generator in terms of packet inter–departure times and packet rates produced by CBR traffic models. We point out that in this subsection, unless it is explicitly remarked, we always used a single generation engine and a single hardware queue; also, generated packets are always 64 bytes long. Figure 4.2 shows the histogram of the packet inter–arrival times of CBR traffic at 100 Kpkt/s rate with a clear mode at 10 μs that, indeed, represents the constant inter–departure times of packet at that rate: in fact 99% of the samples lay in a range of 0.4 microseconds from the expected value. Figure 4.3, instead, is obtained by increasing every 10 seconds the generated packet rate, starting from 100 Kpkt/s up to line rate. The picture shows that the measured packet rate corresponds almost perfectly to the selected values for transmission; in addition, no significant rate fluctuation is reported, proving the high stability of the traffic generation process.

Figure 4.4 and 4.5 represent the histogram of inter–arrival times of two Poisson processes, with average rate of 100 Kpkt/s and 1 Mpkt/s respectively. The exponential behavior is clear in both cases (values of Figure 4.4 are plotted in linear scale while those of Figure 4.5 are reported in log scale) with values of rate consistent to those selected in generation. Only with small inter–departure times (under one microsecond) does the distribution in 4.5 show some spikes. This is most likely due to the quantized latency of the in–kernel polling cycle: indeed, this involves retrieving the timestamp value, which involves a non–negligible amount of clock cycles, thus

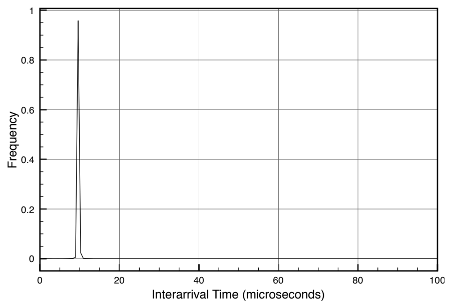


Figure 4.2: CBR traffic – Rate: 100 kpkt/s.

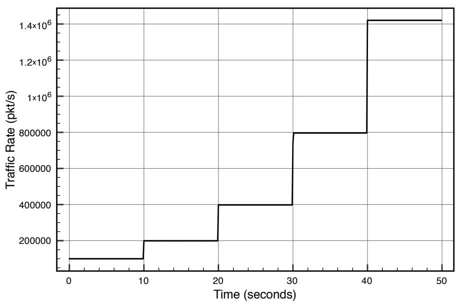


Figure 4.3: CBR traffic – Increasing rates

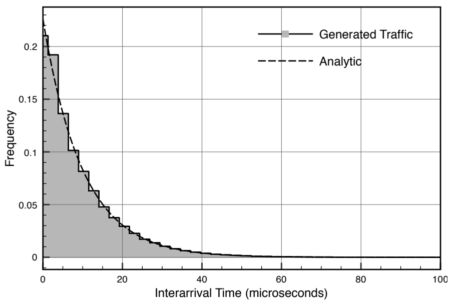


Figure 4.4: Possion process – Rate: 100 Kpkt/s.

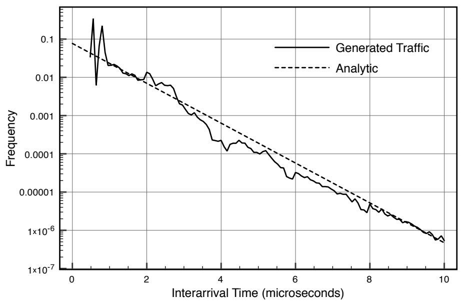


Figure 4.5: Poisson process – Rate: 1 Mpkt/s.

leading to a discretized duration of the polling operation. The same phenomenon can be noticed in Figure 4.7.

Figure 4.6 reports the histogram of inter–arrival times of packets generated according to a Poisson process at 100 Kpkt/s rate by varying the number of hardware queues used in packet transmission. The histograms so obtained do not exhibit significant differences (they actually are almost perfectly overlapped), thus proving that the statistical properties of the traffic model are not affected by the number of active transmitters. This is an important result, as it shows that reordering of packets yield by the same model (which is theoretically possible when more than one hardware queue is used) is unlikely and does not affect the generator accuracy significantly.

Finally, to validate the overall architecture when multiplexing multiple traffic

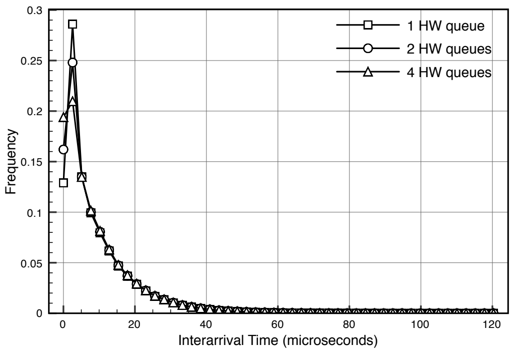


Figure 4.6: Poisson processes generated with different number of HW queues (transmitters)

models, we compose together three independent Poisson processes with rates of 1 Kpkt/s, 4 Kpkt/s and 14 Kpkt/s respectively on the same generation engine. As well known from traffic theory, the resulting process must be again a Poisson process. Figure 4.7 clearly proves that the histogram of inter–arrival times is exponential (a straight line in log scale).

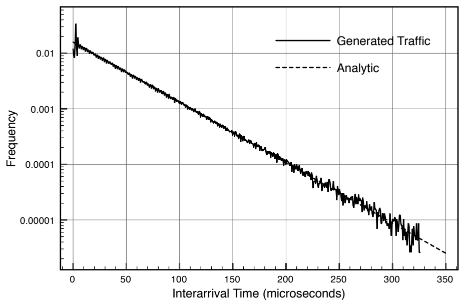


Figure 4.7: Composition of three Poisson processes

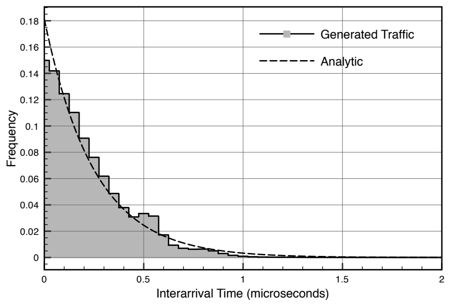


Figure 4.8: Poisson process – Rate: 4 Mpkt/s.

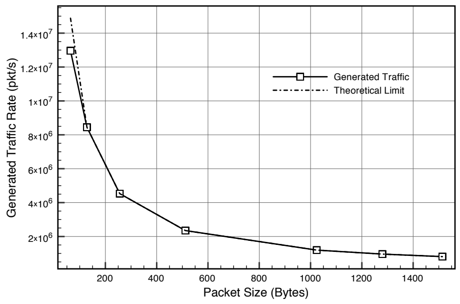


Figure 4.9: Traffic packet rate vs. packet size

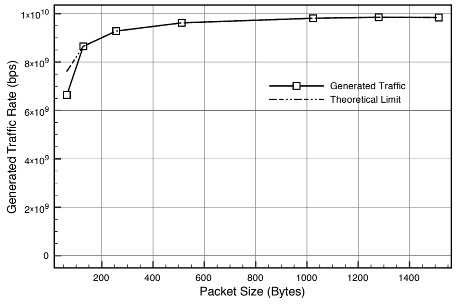


Figure 4.10: Traffic bitrate vs. packet size

4.1.3.2 Towards 10 Gb/s rates

In this subsection we report the results of several tests carried out at traffic generation rates higher than 1 Gbps. Figure 4.8 reports the histogram of inter–arrival times of a Poisson process with 4 Mpkt/s. rate (in this case we used multiple hardware queues to generate the distribution). As previously mentioned, in these cases measurements are taken by means of a software application (`pfq-isto`) running on top of PFQ: although we cannot make definitive statements on time precision, the figure shows a good exponential behavior. Notice that, unlike analogous tests carried out with lower rates, for the system to produce high packet rate multiple engines must be used on different cores. In the specific case shown in Figure 4.8 we used two engines each of them, in turn, feeding two transmitters (for an overall number of four transmitters involved in generating traffic). As a result of the last set of experiments, Figures 4.9 and 4.10 show the cumulative amount of traffic that could be generated with our platform. To this end, we used a simple constant bit rate model and we adopted a configuration with 1 traffic engine and 4 packet transmitters (PF_DIRECT sockets); notice that in this case a single engine is enough to feed the transmitter due to the extreme simplicity of the CBR model, which does not involve random number computations. Figure 4.9 reports the maximum cumulative packet rate that we can produce for different packet sizes. As we can see from the graph, our generator hits the line rate with 128 Bytes long packets and stays very close to it for minimum size (64 bytes) packets: indeed, it generates up to 13 million packets per second. The correspondent values in terms of bit rates are reported, instead, in Figure 4.10.

4.2 Flexible High Performance Traffic Generation on Commodity Multi–Core Platforms

In the previous section we showed how hardware parallelism can be highly beneficial for building fast and scalable traffic generators. Unfortunately, current network monitoring and security software is not yet able to completely leverage the potential which is brought on by the hardware evolution: even if progress is actually being made (multiple queue support has been included in the latest releases of the Linux kernel), much of current monitoring software has been designed in the pre–multicore era. The aim of our work is to make the full power of parallel CPUs available to both traditional and natively parallel application, through efficient and configurable in–kernel packet flow aggregation. Therefore, we designed a novel packet capturing engine, named PFQ, that allows to parallelize the packet capturing process in the kernel and, at the same time, to split and balance the captured packets across a user–defined set of capturing sockets. This way, the application writer can arbitrarily choose its level of parallelism with PFQ, hiding within the kernel the full parallelism of the system. In particular, an application can either use a single capturing socket (as in the case of legacy applications) or have PFQ balance incoming frames across a configurable set of collection points (sockets) or even use a completely parallel setup, where packets follow parallel paths from the device driver up to the application. In all of those cases, PFQ yields better performance than its competitors, while burning a lower amount of CPU cycles. Differently from many existing works for accelerating software packet processing, PFQ does not require driver modification (although a minimal few–lines patch in the driver can further improve performance). Scalability can be achieved through batch processing (which, in turn, leverages the hierarchical cache structure of modern CPUs) and through lockless techniques, which allow multiple threads to update the same state with no locking and minimal overhead. In particular, we designed a novel double buffer multi–producer single–consumer lockless queue which allows high scalability. PFQ is open–source software released under GPL license and can be freely downloaded at [171]. The package consists of a Linux kernel module and of a C++ user–space library.

4.2.1 State–of–the–Art in packet capturing

Several solutions have been proposed to speed up the packet capturing capabilities of commodity PCs. nCap [172] uses memory mapping to directly expose to the application the memory areas where the NIC copies incoming frames. Also PF_RING [173] uses a memory mapped ring to export packets to user space processes: such a ring can be filled by a regular sniffer (thus using the standard linux capturing mechanisms) or by specially modified drivers, which skip the default kernel processing chain. Those can be both drivers with minimal patches (aware drivers) or heavily modified ones. Memory mapping has also been adopted by the well-known PCAP capturing libraries [174]. In the past years, the capturing stack of Free-BSD has been enhanced by a double–buffer mechanism, where packets are written into a memory–mapped buffer which is first filled within the kernel and then switched over to the application for reading. This is different from PF_RING, where applications and kernel work on the same ring concurrently. Although our proposed architecture also adopts a double buffer solution, it brings it further by introducing other optimizations (like batch processing) and by explicitly tailoring it to a multi–core scenario. Many works (most of them on software based routers) have obtained good results in accelerating software packet processing by extensively patching the device drivers. TNAPI [175] effectively addressed the topic, but the proposed solution is based on a heavily customized driver, which detaches parallel polling threads instead of relying on NAPI. Besides, its heavy use of kernel level polling leads to high CPU utilization. The authors in [176] focus on how to distribute work across cores in order to build high performance software routers. Although the results are certainly interesting, it relies on the Click modular router [154] and its modified polling driver to deliver good performance. Again (like in 4.2), our work is somewhat orthogonal to those based on modified drivers, as PFQ is a general architecture that can be beneficial to both vanilla and modified drivers. Other architectures for accelerating network IO on general purpose systems are described in 4.1.1.

4.2.2 PFQ capturing engine

The system as a whole is depicted in Figure 4.11 and is made up by the following components: the packet fetcher, the demultiplexing block and socket queues. The fetcher dequeues the packet directly from the driver, which can be a standard driver or a patched "aware" driver, and inserts it into the batching queue. The next stage is represented by the demultiplexing block, which is in charge of selecting which socket(s) need to receive the packet. The final component of PFQ is the socket queue, which represents the interface between user space and kernel space. All of the kernel processing (from the the reception of the packet up to its copy into the socket queue) is carried out within the NAPI context; the last processing stage is completely performed at user space, thanks to memory mapping. In the following we will describe in more detail each building block.

4.2.2.1 Building blocks

Aware driver. The concept of driver awareness has been first introduced by PF_RING: an aware driver, instead of passing a packet up the standard linux networking stack, highjacks and forwards it directly to the capturing module. This implies that, on one hand, the message does not have to go through the standard network stack processing, thus improving performance. On the other hand, the capturing module has exclusive ownership of the packet, which is invisible to the rest of the kernel (including the sniffers). We developed a patched version of the ixgbe driver that just involves minimal code modifications (around a dozen lines of code); such a simple patch can be easily applied to new and existing drivers. We point out that such a block is completely optional and PFQ shows good performance with vanilla drivers too. Moreover, an aware driver managing multiple interfaces can handle in aware-mode only the packets coming from a monitoring interface, while exposing the others to the kernel stack.

Packet fetcher. The packet fetcher is the only component which acts on a packet by packet basis. It receives the packets and inserts the associated pointer into its

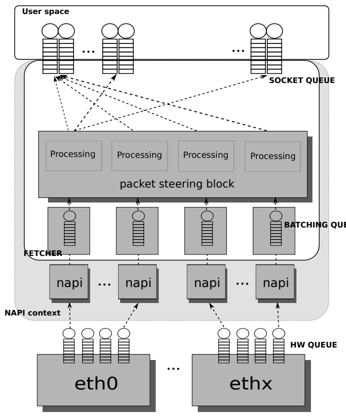


Figure 4.11: PFQ scheme at–a–glance

batching queue. Once such a queue (whose length is configurable) is filled, all of its enqueued packets are processed by the next block in a single batch. Batch processing turns out to be more efficient in that it improves the temporal locality of memory accesses, thus reducing the probability of both cache misses and concurrent access to shared data. In particular, a significant advantage comes from deallocating packets in batches that, according to our measurements, can reduce the deallocation cost by as much as 75%. Our measurements reveal that the optimal queue length is of the order of a hundred of packets. Notice that, as the packet is timestamped before queueing, this component does not influence timing accuracy.

Packet steering block. The main function of the steering block is to select which sockets need to receive the captured packets. Notice that, although it is a single functional block, the steering block processing is completely distributed and does not represent a serialization point (in fact, it only deals with read–only state). Such a block consists of a routing matrix that allows to flexibly dispatch the incoming packets across multiple capturing sockets. In particular, such a matrix associates each reception queue of each handled card with one or more capturing sockets. Such sockets can be independent from each other (thus receiving one copy of the packet each) or can be aggregated into a load balancing group. In this latter case, a hash function is computed for each packet and only one socket in the balancing group is chosen. An additional advantage of such an approach is the possibility of performing a bidirectional load balancing. Indeed, RSS performs its native form of load balancing by computing a hash function over the 5–tuple of incoming packets. However, such a scheme may not be appropriate for some applications, as RSS is not symmetric. For example, applications that monitor TCP connections need to observe packets from both directions which RSS would dispatch to different cores. For this reason, the packet steering block recomputes a symmetric hash function that will rebalance the packets with small overhead. Notice that load balancing and copy are not mutually exclusive: packets from the same hardware queue can be copied to a set of sockets and load–balanced across another one. In greater detail, the demultiplexing block is composed by a bit–field matrix and a load balancing function. The switching matrix stores, for each queue, a bitmap specifying which sockets have to receive its packets. Such a design allows dynamic insertion and removal of sockets with no need for mutexes on the fast data path.

Socket queue. It is the last component of our architecture and the only one which is subject to inter–core contention. Our design shares some similarities with that of the FreeBSD zero–copy packet filter, but it improves the state of the art by introducing a wait–free solution which is optimized for a multi–core environment. Indeed, the whole mechanism implements a multiple producer – single consumer wait–free queue. The main components of this block are two memory mapped buffers: while one of them is being filled with the packets coming from the demultiplexer, the other one is being read from the user application. The two buffers are periodically swapped through a memory mapped variable (named *index* in the pseudocode of algorithm 14) that stores both the index of the queue being written to and the number of bytes that have been already inserted (in particular, its most significant bit represents the queue index). Each producer (i.e. a NAPI kernel thread) reserves a portion of the buffer by atomically incrementing the shared index; such a reservation can be made on a packet by packet basis or once for a batch. After the thread has been granted exclusive ownership of its buffer range, it will fill it with the captured packet along with a short pseudo header containing meta–data (e.g. the timestamp). Finally, it will finalize it by setting a validation bit in the pseudo–header after raising a write memory barrier. Notice that, when the user application copies the packets to a user space buffer, some NAPI contexts may still be writing into the queue. This will results in some of the slots being "half filled" when they reach the application; however, the user–space thread can wait for the validation bit to be set. On the application side, the user thread which needs to read the buffer will first reset the index by specifying another active queue (so as to direct all subsequent writes to it). Subsequently, it will copy to the application buffer a number of bytes corresponding to the value shown by the old index. Such copy will be performed in a single batch, as, from our past measurements, batch copy can be up to 30% faster. Alternatively, packets can be read in place in a zero–copy fashion. The access protocol is described in greater detail by the pseudocode in algorithm 14. Notice that, the first read of the index is not functionally necessary, but prevents the index from overflowing in case the consumer is not swapping for a long period. Finally, we point out that PFQ comes with a C++ user-space library which hides the complexity of the lockless queue while still transferring packets in batches.

4.2.3 Experimental results

We assessed the performance of our system under several configurations and we compared it mainly against that of PF_RING. The latter is the obvious competitor for PFQ, in that it is a general architecture that increases the capturing performance with both vanilla and modified drivers. Unfortunately we could not consider PF_RING TNAPI [175] in the comparison as it is not publicly available for download. We also show some results obtained by the well–known PCAP library (version 1.1.1 with memory mapping enabled), that only works with vanilla drivers; however,

Algorithm 14 Pseudo-code for the NAPI context inserting N packets into the double–buffer queue.

```
function insert_packet(bytes, packet)
 1: if $QLENGTH(index) < BUFFER\_LEN$ then
 2:     queue full, exit                                  ▷ this first read is only to prevent overflow
 3: end if
 4: $curr\_index \leftarrow atomic\_incr(index, bytes + PSEUDO\_HEADER\_LENGTH)$
 5: $curr\_bytes \leftarrow QLENGTH(curr\_index)$
 6: $curr\_buffer \leftarrow QACTIVE(curr\_index)$
 7: if $curr\_bytes < BUFFER\_LEN$ then
 8:     $queuefull, exit$
 9: end if
10: $my\_buffer \leftarrow buffer\_pointer[curr\_buffer] + curr\_bytes - (bytes + PSEUDO\_HEADER\_LENGTH)$
11: copy packet and compile pseudo header
12: $write\_memory\_barrier()$
13: set pseudo header validity bit
function read_packets()
 1: $active\_queue \leftarrow QACTIVE(index)$
 2: $next\_index \leftarrow complement(acive\_queue) << INDEX\_BITS - 1$
 3: $index \leftarrow next\_index$                                       ▷ atomic swap
 4: $my\_buffer \leftarrow buffer\_pointer[active\_queue]$
 5: for all packet in $my\_buffer$ do
 6:     wait for valid bit to be set
 7:     read packet and pseudo header
 8: end for
```

as PCAP does not explicitly support hardware queues, its results can be shown in a few layouts only. We wrote a simple packet counting application for PFQ, while for PF_RING we used the *pfcount* application that comes with the project distribution. We took two main performance metrics into consideration: number of captured packets and average CPU consumption. While the first one is the most obvious performance index, the second one is important as well: if the capturing engine is consuming a very high fraction of CPU cycles, a monitoring application will hardly have resources to do any significant processing. The testbed for experiments is made up of two identical machines, one for generating traffic, the other in charge of capturing. Both of them come with a 6 cores Intel X5650 Xeon (2.66 Ghz clock, 12Mb cache), 12 GB of DDR3 RAM, and an Intel E10G42BT NIC, with the 82599 controller on board. In order to test our system with the maximum degree of parallelism, we kept Intel Hyperthreading enabled, thus carrying out the experiments with 12 virtual cores. We will show that such a choice yields performance improvement in all scenarios. Both servers run Linux with the lates 3.0.1 kernel and the *ixgbe* 3.4.24 NICs driver. Due to the high cost of hardware based traffic generators and to the limited performance of software based ones, we chose, as the authors also did in [157], to write our own generator. Such a software [177] which, again, leverages platform parallelism, is able to generate up to 12 Millions minimum–sized packets per second. We validated its performance by means of a borrowed Napatech hardware based traffic analyzer (courtesy of Luca Deri). In particular, we verified that the maximum generated rate advertised by the generator itself was the same rate measured by the Napatech board. Moreover, in order to leverage the RSS load–balancing mechanism, we randomized the IP addresses of each packet.

Finally, we remark that due to the use of hyperthreading, we can display up to 12 capturing cores; however, from the performance point of view, this is not the same of having 12 real CPUs. The CPU numbers are arranged as follows: real core number x corresponds to two virtual cores x and $6 + x$, respectively. Therefore, if we increase the set of capturing cores in a linear manner starting from 0, we expect the contribution of the first six cores to be significantly higher than that of the others (as it actually appears in our results). Therefore, we expect an ideal graph to scale linearly from 1 to 6 and to show a discontinuity in 6 and to grow linearly again, but with a much less steep slope, from 7 to 12 cores.

4.2.3.1 One–thread setup

In this first layout, which is the most relevant for legacy applications, we used a variable number of hardware queues for fetching packets and we only used one socket to bring them to user space. Indeed, we hid the system parallelism within the kernel while still exposing a standard interface to the application. In particular we used a layout that we showed to be beneficial in [178]: we captured the packets on all the physical cores but one, and on that one we bound the user–space process. The results shown in Figure 4.12 report the number of captured packets for both modified and aware drivers: the behavior of PFQ with an increasing degree of parallelism is piece–wise linear (due to the expected discontinuity around 6) while PF_RING, that handles contention through traditional lock–based mechanisms, and PCAP do not manage to scale with the number of cores. Besides, the scalability of our architecture does not depend on the driver: using an aware or a vanilla driver just reflects on the slope of the graph, but linearity is preserved. Notice that, as anticipated, we did not capture packets on the physical core where the user space process is bound: therefore, the number of available capturing cores is limited to 10.

Figure 4.13 reports the bit rate of the captured traffic for several packet sizes and by using 12 hardware queues. PFQ always captures all of the traffic our generator can provide (although this does not always correspond to the nominal maximum bit rate).

4.2.3.2 Parallel setup

In this scenario each hardware queue is associated with its own user space thread, so that the processing paths of packets are completely parallel. Notice that in this scenario we used PF_RING with the recently introduced *quick mode* option, which allows avoiding per–queue locks. The results are shown in Figure 4.14 and show that, although PF_RING manages to achieve good performance by preventing locking, PFQ still outperforms it. Besides, PFQ shows the same behavior with both vanilla and aware drivers (apart from a scale factor), while PF_RING only scales well with aware drivers. Notice that PFQ is able to capture all of the incoming packets with 10 cores (its throughput steadies because there is no additional traffic to capture);

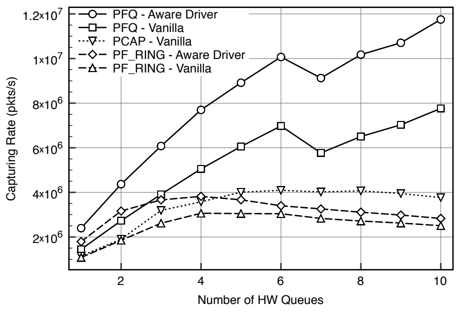


Figure 4.12: One capturing thread

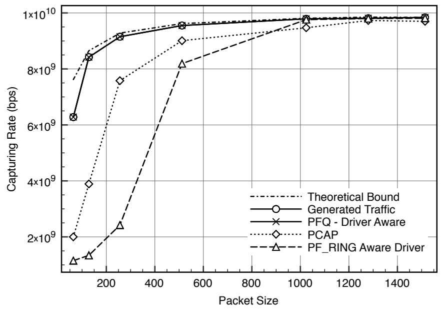


Figure 4.13: Throughput vs. Packet Size

unfortunately, our generator is not able to produce more input traffic and, therefore, we can only obtain a lower bound of PFQ's performance.

We also report the CPU utilization (in the case of aware drivers) in Figure 4.15: while PF_RING saturates the CPU, the global CPU consumption in the case of PFQ is roughly constant and well below 20%.

4.2.3.3 Multiple capture sockets

Besides high performance, one of the strengths of PFQ is the ability of decoupling parallelism between the application level and the kernel level. In this set of tests we measure the performance of such a feature by always using the maximum number of available contexts in the kernel (i.e. 12) and by varying the number of parallel user–space threads. First, we report the overall throughput when incoming packets are load–balanced across the application threads. In order to have a benchmark, we compare our result with that of PF_RING using the recently introduced RSS rehash functionality. However, the balancing functionality in PF_RING slightly differ from that of PFQ. The results are reported in Figure 4.16 and show that, with an aware driver, PFQ is able to capture all of the incoming traffic with just 3 user–space threads while, with a vanilla driver, the behavior is the same but the overall throughput is lower.

We also evaluate a scenario where multiple applications are requesting a copy of the same packet: the results are shown in Figure 4.17 and show the *cumulative* number of packets brought to user–space. In this case, we also show the results for PCAP. Ideally this graph should scale linearly, as the same traffic is being copied to more and more threads; however the overhead of copy and concurrent access to the socket queues has a relevant impact on performance when the number of copies is high. Notice, however, that such a large number of copies is unlikely in a practical setup. Interestingly, this figure also provides an upper bound of the number of packets the system may be able to process with a faster driver with no allocations

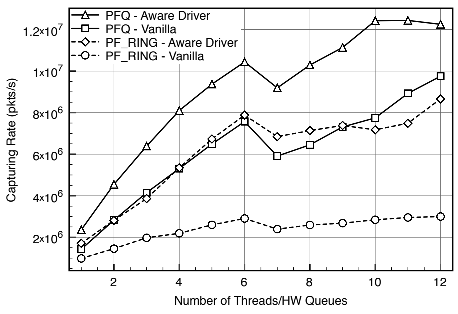


Figure 4.14: Completely parallel processing paths

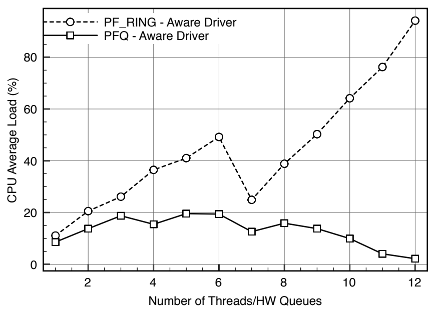


Figure 4.15: Completely parallel processing paths: CPU consumption

or multiple capturing cards: PFQ is able to enqueue and make available to user space over 42 Mpps, thus outperforming by far both competitors.

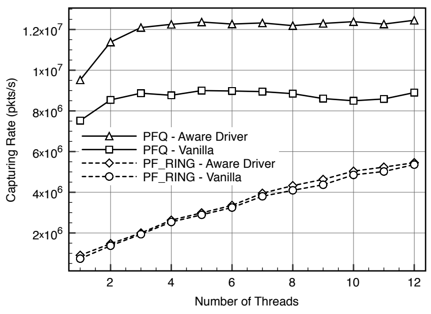


Figure 4.16: Load balancing across a variable number of user–space threads

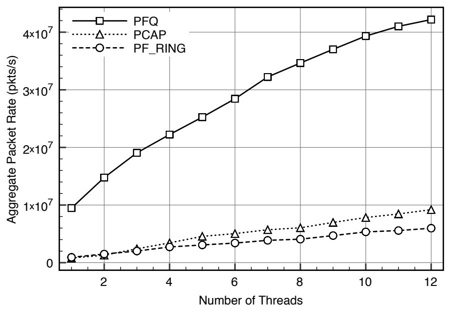


Figure 4.17: Copying traffic to a variable number of user–space threads

4.3 Blockmon: A Modular System for Flexible, High-Performance Traffic Monitoring and Analysis

As already discussed in the introduction, the monitoring infrastructure is called to face two different and opposed trends: on one hand the growth of internet traffic makes the performance constraints on the monitoring points more and more tight, while on the other the great variety and quick evolution of threates requires high flexibility and quick adaptation of the analysis.

These challenges point to the need for a high-performance, yet easily-extensible solution. In this section we present Blockmon, a system for flexible, high-performance traffic monitoring and analysis that allows concurrent use by independent applications or users.

Blockmon builds upon previous work in programmable on-line network measurement and composable networking. CoMo [179], a passive monitoring system, introduced the concept of a monitoring plugin: a monitoring application is written as a set of callback functions and the system is in charge of calling them. However, these callback functions are limited and strictly pre-defined, reducing the system's flexibility.

The modular principles in Blockmon were inspired by the Click modular router [154]. However, Click is intended only for packet processing, with its modules only able to communicate in terms of packets. Further, Click cannot easily do more advanced types of processing such as maintaining TCP connections, inter-acting with databases, or any number of other actions relevant to monitoring and data analysis.

In essence, Blockmon takes some of the design principles of previous approaches, enhances them to allow for a wider range of monitoring and analysis applications, and provides tuning mechanisms that allow it to yield high performance when running on modern, multi-core, multi-queue architectures. Further, by allowing runtime reconfiguration of the connections among its modules, Blockmon enables on-line data analysis of traffic which adjusts to current network conditions. Our contributions are as follows:

- A flexible, multi-user, high-performance system for monitoring and analysis, leveraging recent work and new technologies in high-performance software development.
- A set of common *blocks* (small units of processing) for several application areas, leveraging libraries implementing probabilistic data structures, hashes and abrupt change detectors.
- The ability to extend an application across multiple nodes.
- The ability to dynamically reconfigure a running application.
- The release of Blockmon as open source.

4.3.1 Related Work

We are certainly not the first to tackle the problem of modular, high performance traffic processing on commodity hardware. As mentioned, both CoMo [179] and Click [154] are in this space, but neither provide the mechanisms nor flexibility needed for monitoring and data analysis tasks.

The work in [180] explores the scalability of software routers on general-purpose hardware, and investigates the bottleneck in packet forwarding performance; such results were important in informing the decision of how to parallelize and scale processing in Blockmon. RouteBricks [181] explores the scaling of software routers by enabling parallelism across multiple servers. This complements our work, since such an approach can be used to scale Blockmon's capacity by adding more servers.

Considering programmable measurement, ProgME [182] specifies a runtime-programmable network flow aggregator configured using a declarative language based upon set algebra. An approach that provides a framework for building monitoring applications is RTC-Mon [183], though the architecture it provides is somewhat inflexible.

There has been significant work on the subject of fast packet capture, as in [184] and PF_RING [185]; this latter even contains a basic, if inflexible, programmable measurement system. For 10Gb and faster links, packet capture is often enhanced through hardware acceleration [186, 187]. NetworkDVR [188] follows another approach to packet capture by deciding early in the process which packets to capture and which to ignore; this can also be performed by packet capture and offload cards [189]. As we will show, Blockmon can easily take advantage of such hardware acceleration and combine it with the flexibility provided by its software-based blocks.

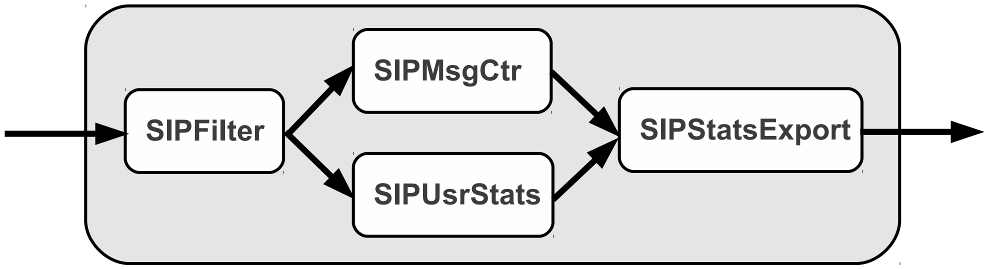
4.3.2 Base System

At a high-level, Blockmon provides a set of units called *blocks* each carrying out a certain type of processing, for instance counting the number of distinct VoIP users on a link. The blocks are then inter-connected via a set of input and output *gates*; the number of gates and what they are used for are defined by the developer of the block. A set of inter-connected blocks representing a monitoring and data analysis application is called a *composition*, which is specified in XML. The Blockmon core and the blocks themselves are implemented in C++, and the system is controlled at runtime using a simple, Python-based command-line interface.

Figure 4.18 shows an example of a composition, first drawn as a logical graph of connected blocks and then as the actual XML file that would be given to Blockmon for installation. In this case, the composition filters traffic for SIP messages, keeps per-message and per-user statistics and periodically exports results.

Blocks can be configured so that two blocks of the same type may be initialized differently; for instance, a block that captures traffic could be configured with different capture filters by different users. Should the available blocks not cover the functionality required by a particular application, a user can easily implement additional blocks and connect them to existing ones.

To support different users and applications simultaneously, Blockmon compositions have *external interfaces* that are defined in terms of input and output gates and



(a) Composition in graph form.

```
<composition id="1" app_id="sipstats">
  <block id="filter" type="SIPFilter">
    <params>
      <source type="live" name="eth0"/>
    </params>
  </block>

  <block id="ctr" type="SIPMsgCtr"/>
  <block id="stats" type="SIPUsrStats"/>
  <block id="exp" type="SIPStatsExport">
    <params>
      <dst ip="192.168.0.200" port="5000"/>
    </params>
  </block>

  <connection src_blk="filter" src_gate="1"
              dst_block="ctr" dst_gate="1"/>
  <connection src_blk="filter" src_gate="2"
              dst_block="stats" dst_gate="1"/>
  <connection src_blk="ctr" src_gate="1"
              dst_block="exp" dst_gate="1"/>
  <connection src_blk="stats" src_gate="1"
              dst_block="exp" dst_gate="1"/>
</composition>
```

(b) Corresponding XML composition file.

Figure 4.18: Blockmon sample composition.

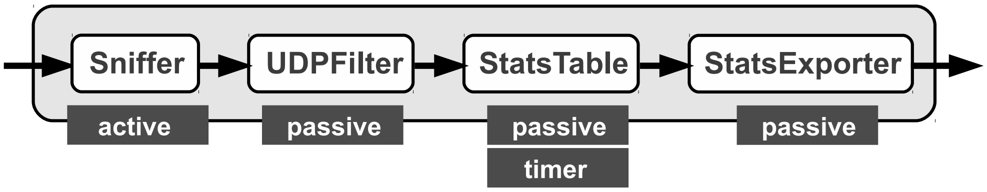


Figure 4.19: Simple composition showing block scheduling types.

the types of messages they expect. For instance, consider the likely case of several applications needing to capture traffic, via access to the same *Sniffer* block. Each application has its own composition, and each of these uses its external interface to connect to the external interface of the composition containing the sniffer. This approach also permits the internals of compositions to change without affecting other compositions, as long as the interface does not change: for example, changing the sniffer to capture on "eth1" instead of "eth0" in response to a routing change would not disturb running compositions using the sniffer.

In the rest of this subsection we describe the various parts of Blockmon in greater detail.

4.3.2.1 Blocks and Scheduling

As mentioned, a block represents a small unit of processing and can implement a wide range of functionality including packet capture and filtering, monitoring, anomaly detection algorithms and export capabilities (see subsection 4.3.4 for a description of blocks available in the Blockmon distribution).

All blocks are derived from a common superclass. In order to create a new block, developers simply inherit from this class and implement at least two methods: `configure`, which receives XML representing the block's configuration parameters, and `receive_msg` which is called when a message arrives at the block.

Blockmon supports *active* and *passive* block invocation. Active blocks are called by Blockmon's scheduler, while passive blocks are activated directly within the same thread as the block which sends them a message. Blocks can also be scheduled to run on a timer.

To make things more concrete, figure 4.19 shows an example of how the different block and scheduling types are used in a composition. In this case, *Sniffer* is actively scheduled, capturing packets from a network interface and sending them to *UDPFilter*. This is a passive block, and runs whenever the sniffer sends a message to it (calling to its `receive_msg` method). *UDPFilter* then filters for UDP packets, and sends results to *StatsTable*, which is also passive and registers a timer. Periodically Blockmon's scheduler activates *StatsTable*, causing it to send data to *StatsExporter*, which exports them to an external consumer.

4.3.2.2 Gates and Messages

Blocks use gates to communicate with other blocks, with each block defining a number of input and output gates. For instance, a sniffer block might have no input gates and an output gate per given filter, while a packet counter block might have several input gates but a single output gate.

Gates are implemented as simple message queues. For an active block, the message is simply stored in the queue; the block will then retrieve it when it is run by the scheduler. For a passive block, the message is not stored, but rather sent directly as a parameter to the block's `receive_msg` method.

The actual communication is in terms of messages derived from a common superclass, which provides a basic interface for identifying message types, and for supporting marshaling and demarshalling of messages for the proxy block facility described below. Blockmon provides a *base data model* for common monitoring and analysis tasks. This data model is designed to ease the composition of blocks by ensuring a single implementation of commonly used messages. The message types in this base data model include:

- **RawPacket**: represents a raw captured packet and a timestamp (equivalent to `struct pcap_potheads` in libpcap-based applications).
- **Packet**: extends a RawPacket to add parsed packet header information.
- **Flow**: represents a set of packets sharing a common 5-tuple (source and destination address and port, plus protocol), and maintains start and end timestamps, optional partial or complete payload reassembly, as well as byte, packet, and "accumulator" counters for each;
- **PairMsg**: a class template which represents a two-tuple of values, intended to represent key-value pairs.

The first three of these are designed for the early stages of processing, when the compositions are still handling relatively raw data. The last of these is intended for mostly-processed data and export of results for presentation and storage.

4.3.2.3 Dynamic Reconfiguration

The environment in which an analysis system runs is rarely static. For example, routing changes could require traffic capture from a different interface, or information produced by an application could lead to a refined query. Blockmon supports dynamic reconfiguration, changing block parameters or the connections among running blocks, or adding or removing blocks from a composition. This allows for the implementation of control loops, where compositions can be changed based on analysis results they produce themselves. This is particularly useful in monitoring protocols with application-layer control, such as in VoIP traffic, where RTP flows are identified by the payload of SIP messages.

Blockmon's Python-based command-line interface allows changes of compositions on the fly, without losing per-block state. At reconfiguration time, messages flowing through a composition are kept in flight in circular buffers until the reconfiguration is complete, then processed before new messages from upstream in the composition.

4.3.2.4 Multi-Node Blockmon

Blockmon supports the ability to extend a single composition across a set of nodes. This allows additional scalability, by leveraging multiple nodes as well as multiple cores for processing, and flexibility of deployment. To implement this, Blockmon provides a *proxy block* facility. This represents each end of a connection between Blockmon nodes with a block which acts as a proxy for the remote end, handling marshaling and demarshaling of Blockmon messages on the wire. Two wire protocols are supported. The first is called raw structure marshaling, which allows direct copying of message contents from machine to machine, assuming identical binaries, language runtimes, and architectures on each.

This facility also supports the IETF IPFIX [190] standard, which provides a templated format and transport protocol for network traffic data. In addition to adding flexibility in multi-node Blockmon installations, by allowing compositions to be built from nodes of different architectures and versions of Blockmon, IPFIX proxy blocks also significantly increase the interoperability of our system. For example, by leveraging existing IPFIX flow meters (e.g., YAF [191]) Blockmon is applicable as a flow analysis tool in addition to its native packet analysis capabilities. Fully exploring the possibilities of this interoperability are the subject of future work.

4.3.3 Performance Mechanisms

Blockmon has been designed to leverage the potential of modern multi-core commodity server hardware and network interfaces. In this subsection we describe the optimizations used by Blockmon for high performance traffic processing; evaluation results for these appear in subsection 4.3.5.

4.3.3.1 Thread Pools and CPU Pinning

Blockmon is multi-threaded in order to take advantage of multi-core CPUs. The assignment of activities to threads and threads to CPU cores can have a large impact on performance [180]. To leverage this, Blockmon schedules work in thread pools. Each block is assigned to a pool via the composition, and pools can be pinned to specific cores.

This model allows flexibility in terms of which block is executed on which CPU core. In the example in figure 4.20, `pool1` runs a single thread on its own dedicated CPU (for performance critical tasks), `pool2` shares three threads across three cores with no fixed mapping, and `pool3` provides 10 threads running on any available core. Lastly, it is worth pointing out that blocks can make use of a default mechanism providing thread-safeness, as is the case with `pkt_counter3`; we provide figures on the impact of using this in the evaluation subsection.

4.3.3.2 Lockless Queues

One potential performance bottleneck are the queues within a block's gates, particularly for compositions where several blocks send messages to a single active receiver

```
<composition id="1" app_id="test">
 <threadpool id="pool1" n_threads="1" cores="2">
 <threadpool id="pool2" n_threads="3" cores="0-1,4">
 <threadpool id="pool3" n_threads="10">

  <block id="pkt_counter1"
         type="PktCounter"
         threadpool="pool1" />
  <block id="pkt_counter2"
         type="PktCounter"
         threadpool="pool2" />
  <block id="pkt_counter3"
         type="PktCounter"
         thread_safe_mode="on" />
</composition>
```

Figure 4.20: Excerpt from a composition specifying mappings of blocks to thread pools, CPU pinnings and thread-safeness.

Block Name	Description
Sniffer	Captures traffic from a local interface or pcap trace file into RawPacket messages.
ComboSniffer	Captures traffic from an INVEA-TECH COMBO card into RawPacket messages.
MQPfringSniffer	Capture traffic from PF_RING sockets into RawPacket messages; supports multi-queue NICs.
PFQSniffer	Captures traffic using PFQ into RawPacket messages; supports multi-queue NICs.
TCPDemux	Segregates TCP and non-TCP packets.
IPAnon	Parses the headers of RawPackets and anonymizes IP addresses.
PktCounter	Counts and periodically logs number of received messages.
CDF	Tracks and periodically logs cumulative distribution of values in received messages.
FlowStats	Counts bytes and packets per flow based on a 5-tuple flow key.
HeavyFlowSelector	Computes heavy hitters in terms of packet counts from flow statistics.
SYNCounter	Counts the number of TCP SYN packets received from each IP using a count min sketch.

Figure 4.21: Sample of provided blocks.

block. In order to remove contention in these cases, BlockMon provides a multi-producer, single-consumer queue using double buffering, reserving one buffer for the writing and one for reading at any given time. The design of this queue is similar to the one we describe in subsection 4.2.2.

This design makes synchronization fairly simple: producers reserve a slot by incrementing an atomic index, while the consumer simply swaps the active buffer (by atomically changing the index) and is granted exclusive access to the written data. Writers signal they are finished via a per-slot flag; this allows writes to complete after a buffer swap.

Although both double buffers [192] and lockless queues [193] are covered in the literature, the design of this queue is, to the best of out knowledge, novel: Blockmon's lockless queues are wait-free for producers, while consumers only have to wait for writes to complete after a swap. Experimentation shows this is a low-probability occurrence with negligible impact on performance.

4.3.3.3 Batch Allocation

BlockMon reduces dynamic memory allocation overhead [194] by batching memory allocations of buffers used by each block into larger blocks. The batch allocations are reference-counted, such that they are automatically freed with the destruction of the last buffer in the batch. Note that this optimization is only possible with the new shared ownership constructor of the shared pointer class supported by C++11 [195], which avoids the allocation of a reference-count metadata structure for each buffer.

4.3.3.4 Efficient Message Transfer

Blocks pass messages between themselves via C++11 shared pointers, so that the same message can go through different processing paths in a composition without spurious allocations or copies, with automatic reference counting. However, copying a shared pointer involves atomically decrementing and incrementing the reference counter, which can lead to high contention when a message moves from core to core. Therefore, Blockmon adopts the new C++11 object-move semantic, which allows for the transfer of shared pointers without reference count updates.

4.3.3.5 Pluggable Schedulers

How one schedules the various threads to run (and the blocks within them) can have a major impact on performance. Rather than trying to design an ideal scheduler for all possible compositions, Blockmon's scheduler, like that of the Linux kernel [196], is pluggable. It provides a standard scheduler that provides good performance, but allows advanced developers the ability to easily plug-in custom-built schedulers by implementing a simple interface.

4.3.3.6 Fast Capture Blocks

Fast packet capture is crucial to systems like Blockmon aimed at doing high-rate monitoring and data analysis. To this end, Blockmon provides three software-based packet capture blocks: a standard pcap-based `Sniffer` block that can capture packets from a network interface or a packet trace; a `MQPfringSniffer` block based around the PF_RING network socket for higher performance; and a `PFQSniffer` block, which implements an adapter for a novel engine called PFQ [171], which better leverages multi-core architectures. Hardware-accelerated capture using INVEA-TECH Combo cards is supported by a fourth `ComboSniffer` block.

4.3.4 Blocks and Libraries

Blockmon provides flexibility for users to create arbitrary monitoring and data analysis blocks. In addition to the standard blocks included in Blockmon, a sample of which are listed in figure 4.21, Blockmon provides a set of libraries to ease block development as well. The subsubsections below give an overview of the available libraries for implementing new blocks.

4.3.4.1 Hash Library

Network monitoring applications running on high speed links need to use efficient data structures that can be quickly updated independently of the number of elements stored. Hash functions are often used to satisfy these requirements and this is why Blockmon provides a simple and extensible Hash Library. Application developers can use the hash functions already provided (e.g., MD5, SHA1, and SHA256, among others), or create new ones by extending a simple prototype. In addition, the library provides a hash data structure as well as a D-Left hash that provides better performance.

4.3.4.2 Probabilistic Data Structures Library

A common challenge in network monitoring is data storage: many algorithms have to store counters or information on a per-flow basis. To this end, the probabilistic data structures library contains a Count Min Sketch (CMS) which is simple to use: choose a width and depth for the sketch (to determine the memory usage and the error probability), choose which fields of the 5-tuple compose a counter's identifier, and then increment, decrement or read a counter by calling the data structure with a 5-tuple identifier.

Further, the library implements both a Bloom filter data structure as well as a counting Bloom filter one. Both of these can be easily configured based on table size, bucket size, and number of hash functions, among others.

4.3.4.3 Abrupt Change Detection Library

For certain algorithms, a common task is to detect an abrupt change in the distribution of a value over time, since such a change may indicate an anomaly due to an attack. This is for example required to detect a flooding attack: an abrupt increase in the number of packets of a certain type (TCP SYN for example) can be an indication of an attack. It may also be used for Botnet detection by detecting an increase in the frequency of some payload characteristics.

To aid in this kind of detection Blockmon provides a library that implements the CUmulative SUM control chart (CUSUM) algorithm, which detects anomalies by looking at abrupt time changes in the distribution of a random series. The recursive implementation of the algorithm updates CUSUM statistics at each new observed value, and an alarm is raised as soon as these statistics pass a threshold. Blockmon also provides NP-CUSUM, a non-parametric version of CUSUM.

4.3.5 Evaluation

Blockmon contains a number of performance optimizations aimed at providing high-rate traffic capturing and analysis. In this subsection we present an experimental evaluation of these mechanisms in isolation. Section 4.3.6 focuses on the evaluation of two applications implemented using Blockmon.

4.3.5.1 Experimental Setup

For our experiments we use a pair of servers, one running a traffic generator and the other one running Blockmon, directly connected via 10Gb wired interfaces. Each of these computers has a 2.66Ghz 6-core Intel Xeon X5650 with HyperThreading enabled, 12GB of DDR3 RAM, an Intel 82599EB network interface, and run Linux kernel version 2.6.39. A third server is used for experiments on the COMBO card; this has a 2.50GHz 4-core Intel Xeon E5420 CPU and 4GB of RAM; test traffic in this scenario comes from a hardware traffic generator. Unless otherwise stated, all experiments in this subsection are performed with 64-byte packets, since these maximize strain on the system. Throughout we use Mp/s to mean million packets per second.

4.3.5.2 Performance Experiments

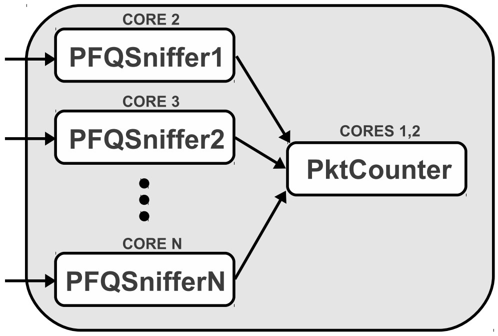
To test the effect of the batch allocation and lockless queue optimizations, we created the composition shown in figure 4.22(a). Each of the multiple capture blocks services one of the hardware queues on the Intel NIC, and feeds packets into a single counter block. We further assigned one CPU core to each capture block, and one for the counter. The single counter creates a bottleneck that allows us to measure the effects of the lockless optimization.

Running this set-up produces the results in figure 4.23, showing packet rate in Mp/s depending on the number of logical CPU cores used for capture. Two logical cores are reserved for the counter block. We start our measurements with 2 cores (i.e., 2 sniffers) since setting RSS [197] on the Intel NIC to 1 causes the driver to use all of the NIC's hardware queues instead of a single one.

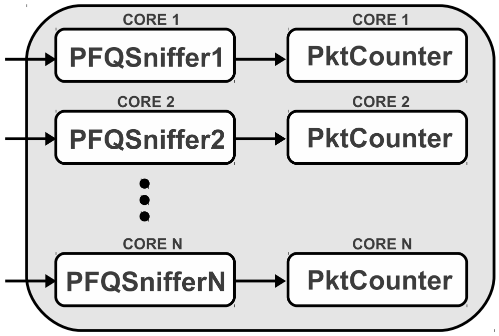
Applying each of the two optimizations in turn, as is the case for the "batch only" and "lockless only" curves, results in sub-optimal performance, as each optimization removes only one of the two performance bottlenecks: performance is still bound by the remaining bottleneck. Thus, the lockless optimization curve does not show much improvement as the number of sniffers increases and contention on the single counter becomes more severe since the memory allocation bottleneck remains. Conversely, the batch optimization curve holds steady but decreases slightly as the contention on the packet counter's queue increases with the number of capture blocks. This effect is confirmed by the top curve: removing both bottlenecks provides a significant bump in performance. Note that the slight dip in performance at 6 cores is due to the fact that the last 6 cores are not physical cpus, but rather emulated by means of the Intel HyperThreading technology. As a result, their contribution is lower with respect to that of actual cores.

The graph also quantifies the overhead of the thread safety mechanism. Here, we enabled thread safety for the counter block, and enabled the batch and lockless queue optimizations. As can be seen, the mechanism still yields decent packet rates (in the order of 2.5 Mp/s), but hinders scalability as the number of cores increases. This shows that disabling thread safety for a block can yield an approximate doubling in capacity, at the cost of increased implementation complexity for the block developer.

Parallelizing packet counting, as shown in figure 4.22(b), removes the single counter bottleneck. Here we measure only the capacity increase due to the batch



(a) Non-parallel setup.



(b) Parallel setup.

Figure 4.22: Blockmon packet capture and counter compositions.

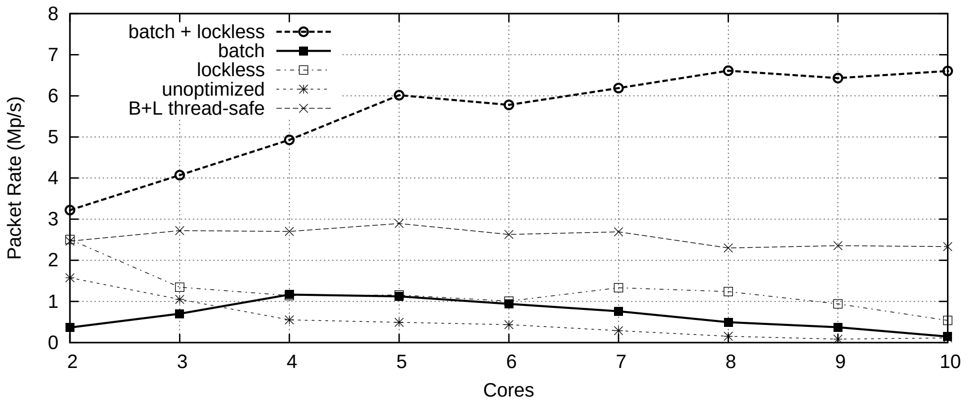


Figure 4.23: Effect of batch allocation and lockless queues optimizations and of the thread-safe mechanism.

optimization, as there is no longer any queue contention. Taking advantage of parallelization, NIC hardware multi-queuing, multiple processor cores and the batch optimization produces very good results: up to 12 Mp/s captured and counted, as shown in figure 4.24. This rate is equal to the maximum rate offered by our traffic generator; further measurement of Blockmon will require future optimizations to the traffic generator used in the test.

To show the dependence of performance on packet size, we tested the composition in figure 4.22(b) with varying packet sizes. As shown in figure 4.25, two cores are sufficient to capture 512-byte packets at line rate; three cores for 256-byte packets; and five cores for 128-byte packets.

To quantify the overhead introduced by Blockmon itself, we created a simple stand-alone test application that uses the PFQ engine to capture and count packets, and compared it to Blockmon installed with the parallel composition in figure 4.22(b). The results, shown in figure 4.26, illustrate that the flexibility provided by Blockmon costs less than 17% in the worst-case for this simple application.

As described in subsection 4.3.3.6, Blockmon provides a number of software capture blocks. We compare their performance in figure 4.27, showing that PF_RING and PFQ handily outperform pcap, with PFQ slightly faster.

Hardware, however, outperforms all. To demonstrate Blockmon's ability to integrate specialized hardware into a composition, we replaced the *PFQSniffer* blocks with *ComboSniffer* ones that encapsulate an INVEA-TECH's COMBO-10G2 FPGA hardware card [189]. The results in figure 4.28 show that using 4 cores results in line-rate processing for any packet size, showing that Blockmon's performance can be even further enhanced by using specialized hardware.

Blockmon utilizes new language features in C++11 to maximize performance. It is difficult to quantify the performance attributable to this design choice, specifically with respect to the move semantic used for message passing (as described in

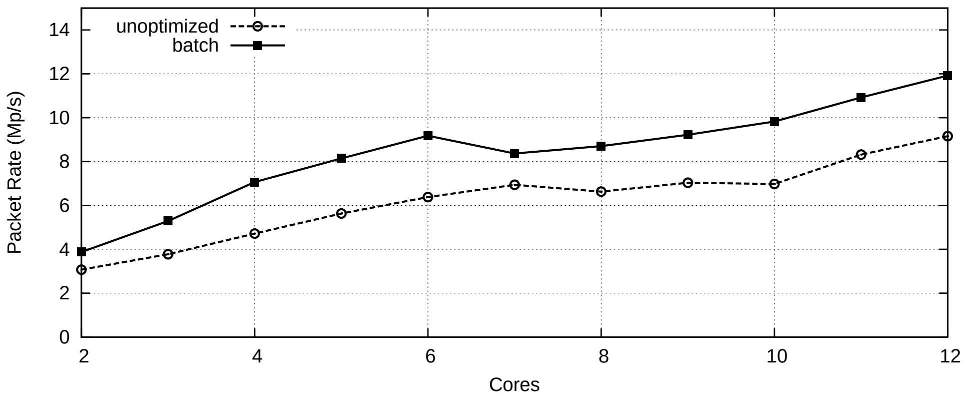


Figure 4.24: Scaling performance with a parallelized composition.

subsection 4.3.3.4), as it is deeply integrated into Blockmon and cannot be disabled. To avoid having to implement a different, lower-performance Blockmon to measure this, we implemented a simple test application that allocates data (a simple integer) and binds it to a shared pointer. Multiple consumer threads each using a CPU core then take this pointer and either destroy it right away, copy and destroy it or use the move semantics and destroy it, in essence emulating the message life-cycle in Blockmon.

Note that this is a test of the C++11 implementation used by Blockmon on the test machine more than anything else, but does serve to illustrate the performance that can be derived from leveraging new technologies in software development.

As shown in figure 4.29, using the move semantics instead of a copy allows savings of up to 200 clock cycles per operation in the case of high concurrency.

4.3.6 Applications

So far we have described the primary mechanisms behind Blockmon and demonstrated its base performance in an experimental setting. However, whether these mechanisms are useful in real applications is a different question. Here, we address this question by examining two applications: a simple monitoring system that keeps per-flow statistics, and an anomaly detector for detecting TCP SYN flooding attacks. We use the first application to illustrate the flexibility of composable measurement systems, and the second to demonstrate more complicated compositions.

4.3.6.1 Heavy Hitter Statistics

A common task in traffic monitoring is collection of per-flow statistics for heavy hitter tracking. This application is useful on its own, and simple enough to implement

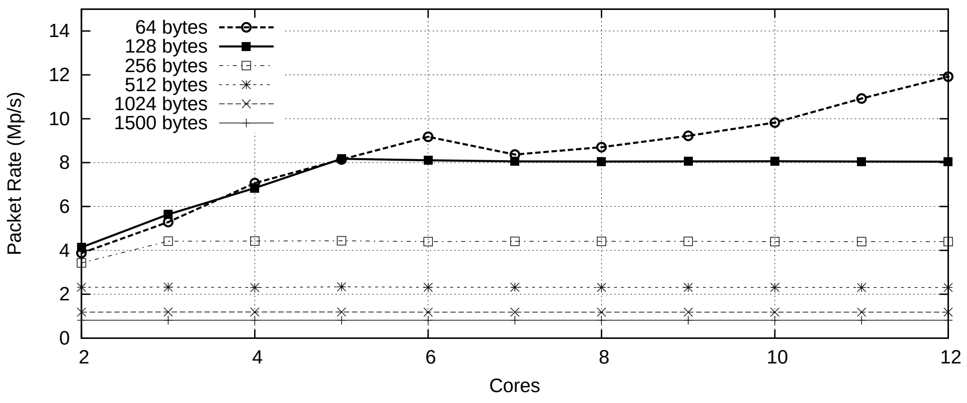


Figure 4.25: Performance for different packet sizes using the parallelized composition and the batching optimization.

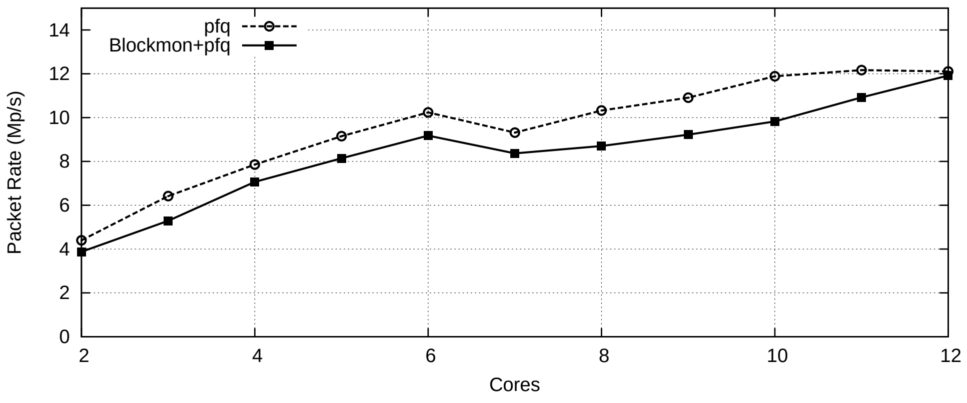


Figure 4.26: Overhead introduced by the Blockmon architecture.

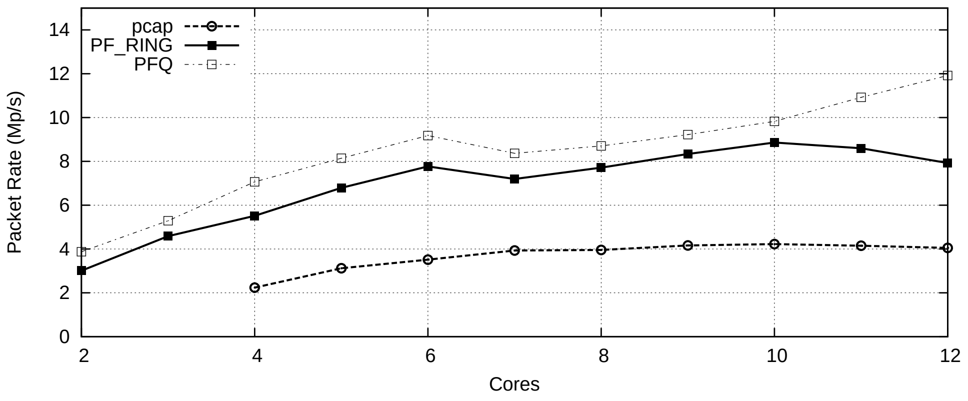


Figure 4.27: Comparison of the various capture blocks provided in Blockmon.

twice, to illustrate the impact of integrating specialized hardware into a composable measurement system. This application is implemented using the two compositions shown in figure 4.30.

The software version uses the *FlowStats* block, which parses a flow's 5-tuple and stores statistics using a simple hash. The statistics are periodically exported to a *HeavyFlowSelector* block, which filters and exports only flows with large packet or byte counts.

The hardware-based composition mimics this, with a few differences. The sniffer is now a wrapper block called *ComboFlowSniffer* which uses INVEA-TECH's combo card and keeps the per-flow statistics. This blocks outputs data from the hardware card in SZE2 format, where SZE2 is a zero-copy API for high-speed generic data transfers. The *SZEToTupleStatistic* converts the SZE2 data format into the flow statistics that *HeavyFlowSelector* expects.

The results of the performance tests are shown in figure 4.31. We used minimum-sized packets, and had 64 of the generated flows ("elephants") account for 30% of the offered traffic rate of 12Mp/s; the rest of the flows were smaller ("mice"). It is worth noting that the 30% figure is an overly pessimistic version of the 80/20 rule [198], where 20% of the flows on the Internet contribute to 80% of its traffic. In our experiments, we use only 64 flows to account for 30% of the traffic to force Blockmon to keep track of more flows. The results show that the application, which is not particularly optimized, is able to process as many as 7.5Mp/s minimum-sized packets while keeping statistics on a per-flow basis.

The hardware-based composition yielded line-rate processing for all packet sizes using a single CPU core (recall that for these tests we had access to a hardware traffic generator, so we did not have the 12Mp/s bottleneck). These results shows that Blockmon's performance can be even further enhanced by using specialized hardware.

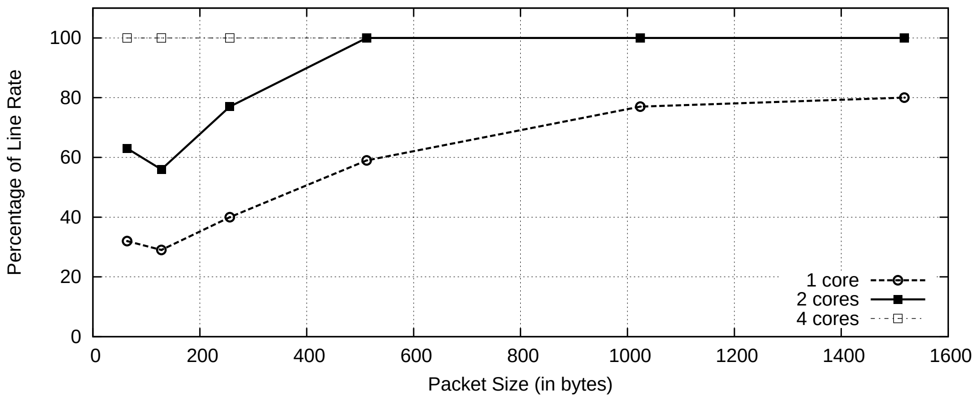


Figure 4.28: Accelerating a Blockmon composition using specialized hardware.

4.3.6.2 SYN Flooding Detection

SYN flooding is a common denial of service attack. It works by sending many TCP SYN packets to a victim, forcing it to keep potentially large amounts of state for half-opened connections.

We used Blockmon to implement an application that detects these attacks and identifies their victims. Our solution leverages the libraries provided with Blockmon, using a Count Min Sketch (CMS) to store the number of TCP SYN packets sent to each IP address, and the multi-channel NP-CUSUM algorithm to watch all the values of the sketch and detect any abrupt changes in the number of TCP SYN packets sent to a particular IP address.

The detection is divided into different blocks brought together by the composition in figure 4.32. Note that for presentational reasons we only show a composition with 2 counters; this number increases in the performance evaluation with the number of CPU cores in use. The blocks involved are:

- **SynSynchronizer**: generates and sends a sketch using the CMS library during initialization to all *SynCounter* blocks, so that the size of the sketch and the hash functions used are uniform for all counters.
- **SynCounter**: uses the CMS library to store the number of TCP SYN packets sent to each IP and periodically sends the computed sketch as a message, reseting it in the process.
- **CmsMerger**: merges the sketches arriving from the counter blocks and exports the merged sketch.
- **SynFloodingDetection**: periodically receives the sketch and uses the CUSUM library to monitor all cells of the sketch and detect abrupt changes. If enough

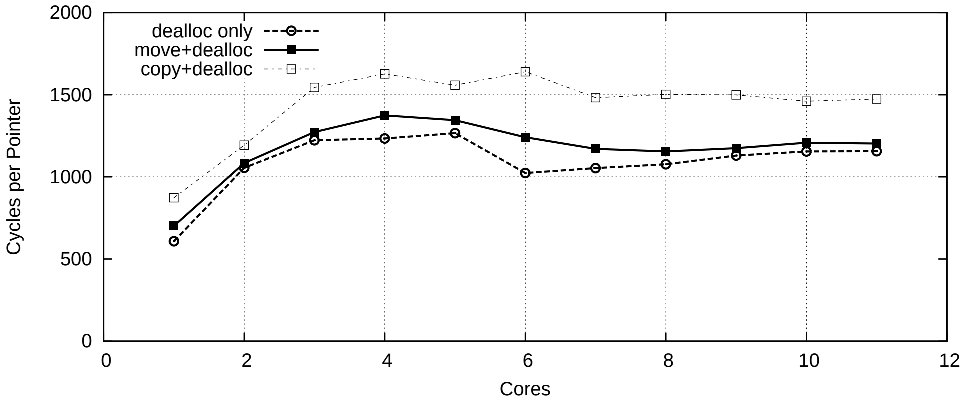


Figure 4.29: Effects of the move optimization.

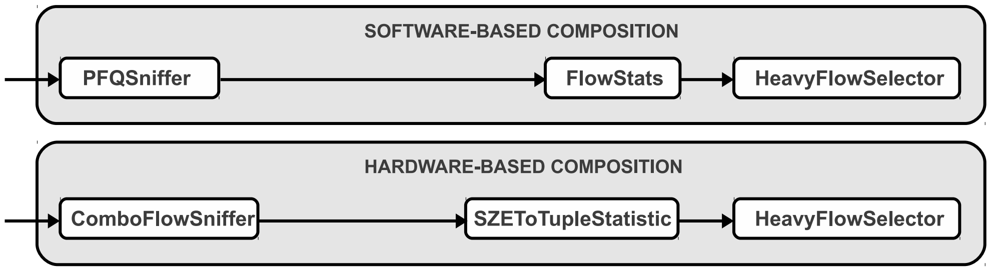


Figure 4.30: Software- and hardware-based compositions for the per-flow statistics application.

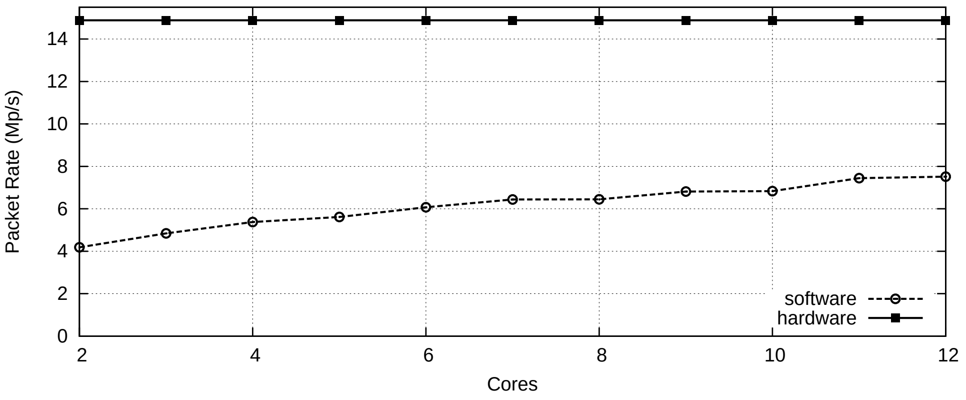


Figure 4.31: Performance of the per-flow statistics application.

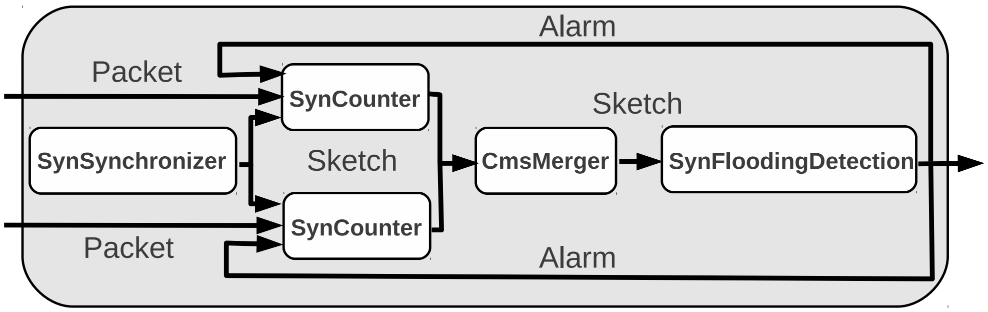


Figure 4.32: SYN flooding detection parallel composition (2 counters).

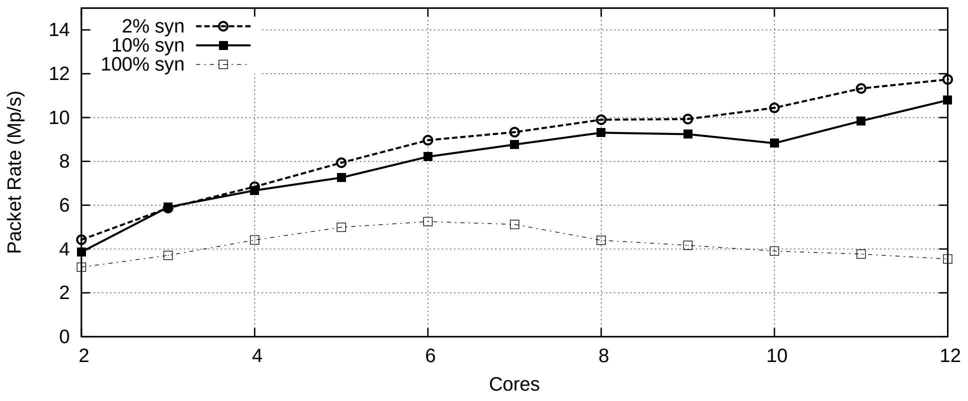


Figure 4.33: Performance of the SYN flood detection application.

abrupt changes have occurred, it sends an alarm with the index of the faulty cells as a message.

Further, the *SynFloodingDetection* block sends the alarm back to the *SynCounter* block; since the sketch is not reversible, *SynFloodingDetection* does not know which IP address is the target of the attack. To remedy this, *SynCounter* keeps a small list of recently seen IP addresses, which it tests when it receives the alarm to see which of them triggered it.

We plot the performance of the TCP SYN flood detection application in figure 4.33. Each of the curves shows a different proportion of TCP SYN packets in the offered traffic. The results are encouraging: even at 10% (more than one million SYN packets per second) Blockmon is able to perform anomaly detection at a rate of approximately 12Mp/s. To test accuracy, we manually instructed our generator to insert a burst of SYN packets at different points during the tests (recall that the detection is based on abrupt changes); each time we did so the *SynFloodingDetection* block reported an anomaly and gave the victim's IP address.

Chapter 5

Scalable coordination and correlation architectures for distributed monitoring systems

In the last years, the scale of the internet cyberthreats has been rapidly growing. Skilled hackers can easily recruit an army of infected hosts (called bots) which can then be used to perform large–scale attacks and other malicious activities. Botnets with hundreds of thousands of hosts have been reported several times in the last few years. A big network domain can therefore expect to be a target of distributed attacks, involving several hosts scattered across the global internet. Therefore, being able to gather and correlate the information gathered by a large set of probes is essential for the future security systems. In a certain sense, as botnets represent a distributed attack infrastructure, a distributed defense infrastructure is needed. The standard approach to distributed monitoring has usually been based on a set of probes exporting raw data (be them sampled packets or Netflow/Ipfix reports) to a central collector which would perform the actual processing. However, such an approach will hardly scale with the constant growth of internet traffic and the increasing need for complex analysis applications. In addition, privacy–preserving legislation prevents indiscriminate export of traffic–related data. For these reasons, a distributed overlay of smart monitoring nodes, which can be able to perform in–network processing and correlation, seems to be the right path to follow while designing the future distributed monitoring systems. In this chapter, we contribute to such a research field with several novel proposals. In section 5.1 we propose an infrastructure for distributed correlation of cross–protocol events. Such an architecture is based around the use of probabilistic data structures for exporting a compressed summary of the observed data. In section 5.2, instead, we propose a distributed coordination infrastructure whose goal is to assign a subset of the traffic to be monitored to each of the probes in an overlay. This allows to avoid duplicate measurements (which would, in turn, bias the monitoring results) and, at the same time, to optimize the utilization of the probes' capabilities. Finally, in section 5.4 we propose an algorithmic solution to the duplicate prevention problem: indeed, we propose a hybrid data structure which implements the functionality of a sketch while automatically discarding duplicates. We show how such a structure can be used to implement a distributed anomaly detection system with no need for a coordination system.

5.1 Crosstalk: A Scalable Cross-Protocol Monitoring System for Anomaly Detection

Monitoring large networks in order to detect such anomalies is inherently difficult for several reasons. First, many of these anomalies require cross-protocol correlation in order to be detected. Botnets, for example, often use several protocols to coordinate activities and to carry out attacks (e.g., IRC for control and SMTP to send out spam) [199][200]. Another example where cross-protocol detection is needed is VoIP, since calls tend to be split into signaling and media traffic, as is the case with SIP and RTP.

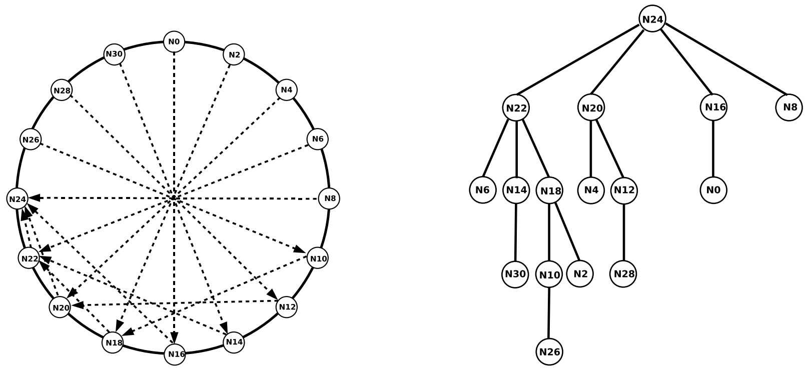
In addition to cross-protocol correlation, monitoring needs to be done in a distributed fashion, since traffic from a particular attack or a mis-configuration may cross different monitoring points in the network. Making matters more difficult is the relentless growth of IP traffic volume, nearly doubling every two years [201]; this growth raises serious scalability issues when designing a system that not only needs to monitor large quantities of traffic in real-time, but also to aggregate results in order to provide network-wide anomaly detection.

In this section we introduce Crosstalk, a scalable architecture that gathers data from a potentially large set of distributed monitoring probes, and performs cross-protocol correlation to detect network anomalies. While previous work has looked into the area of cross-protocol detection [202][203], it has focused on single-point solutions, and so did not scale nor could it correlate attack traffic traversing more than one monitoring point. In [204] the authors implement a distributed system, but its evaluation does not show how it would scale under heavy load and a large number of monitoring probes. SDIMS [205] presents a scalable infrastructure leveraging Distributed Aggregation Trees (DATs), but again does not evaluate its performance under heavy load nor does it look at ways of reducing messaging and data transmission overheads.

As mentioned, several anomalies can be detected using cross-protocol correlation. For the purposes of evaluating Crosstalk's scalability and performance, we pick one point in this application space and focus on SIP-based VoIP attacks.

5.1.1 Crosstalk's Architecture

Crosstalk's architecture consists of three main features that allow it to perform distributed detection in a scalable way: leveraging Distributed Aggregation Trees (DATs), taking advantage of probabilistic data structures (e.g., Bloom filters), and using a novel mechanism called *backtracking* (BT).



(a) Chord fingers for all nodes to node N24.

(b) Same Chord fingers, this time shown as a tree.

Figure 5.1: Example of a Distributed Aggregation Tree built on top of Chord with node N24 as the root.

5.1.1.1 Distributed Aggregation Trees

The simple approach of exporting data from several monitoring probes to a centralized location clearly does not scale. In order to cope with this scalability issue, efforts both in the research and standardization communities have focused on creating tree-based hierarchies, whereby monitoring *probes* export measurements to intermediate nodes called *mediators*. These in turn perform some sort of data reduction operation (e.g., aggregating packet counts) and export the results up the tree hierarchy. In the final step the root, which is a special mediator called a *collector*, stores the aggregated results.

Ideally we would like to have a way of deriving such a tree-based topology dynamically in order to adapt to traffic conditions. This is precisely the goal of DATs, which create this structure on top of a peer-to-peer network such as Chord [206], thus providing the best from both worlds: the scalability (and resilience) of p2p networks with that provided by the aggregation mechanism of a tree structure.

The basic insight behind a DAT is that Chord's fingers already provide a tree structure. In order to illustrate this, figure 5.1(a) shows a regular Chord network with dotted lines representing the path from each node to node N24 (perhaps the responsible node for a particular key); figure 5.1(b) then shows these same connections but this time drawn as a tree. As can be seen, for any given key, Chord naturally builds a tree rooted at the node responsible for that key. In this way, each key has its own DAT, with all the DATs sharing the same peer-to-peer infrastructure. Within each DAT, intermediate nodes (i.e., all nodes except the leaves of the tree can aggregate data as it travels towards the root, thus providing scalability.

5.1.1.2 Probabilistic Data Structures

Clearly nodes in the DAT will need to export information about the monitored data, and this will consume bandwidth. Another consideration for real-time monitoring and detection is being able to perform the cross-protocol correlation quickly. To achieve both of these goals we rely on probabilistic data structures, and more specifically Bloom filters (BFs)[1].

The details of what a Bloom filter represents are application-specific, but generally we use one filter per protocol. For instance, in the case of VoIP attacks presented later in the section, we use one filter for SIP traffic and another one for RTP, with an entry in a filter denoting a SIP identifier such as the SIP dialog id.

This compressed representation of the data consumes little bandwidth when transmitted, and it allows us to perform aggregation and correlation (since they are based on fast bitwise operations) very quickly. Probabilistic data structures do carry a cost in the form of false negatives/positives; next we introduce a mechanism to deal with this and in subsection 5.1.3 we evaluate their impact.

5.1.1.3 Backtracking

While some applications might be content to only receive the summarized data from a DAT's collector, others will use such summarized data as a trigger for retrieving more detailed information (e.g., packet headers) at the monitoring probes, perhaps to determine the cause of the trigger. In addition, Bloom filters carry a low but non-negligible probability of false positives, and so we need a way to verify whether a result is valid or just a false positive.

In order to accomplish these goals we introduce a mechanism called *backtracking*. The idea behind it is simple: when exporting Bloom filters to nodes in the DAT, keep a copy of them locally so that the system can track back to the original probes that monitored the traffic.

Figure 5.2 shows the process in greater detail. Probes P0 and P3 monitor traffic and export data about it in the form of Bloom filters, depicted as a set of squares with each square representing a bit in the filter (note that the figure is simplified for explanatory purposes: normally an entry in the Bloom filter would use up several bits, and more than one Bloom filter would be used to represent the protocols to be correlated). In addition, probes, as well as mediators, keep a local copy of exported Bloom filters, shown in the figure in grey. As the exported filters travel up the tree, mediators perform a bitwise operation to combine the filters, which eventually reach the collector C.

Upon receiving all the combined data, C correlates Bloom filters from different protocols and, depending on the application, triggers a backtracking request to all its immediate mediators, in this case M4 and M5. The request includes the collector's Bloom filter (shown in white), which the mediators use to compare it with their locally stored state by performing a bitwise operation: if the number of set bits in the resulting filter is higher than a user-defined threshold, the backtracking request is propagated to all of the mediator's children; otherwise, no relevant probes exist in this area of the DAT and the backtracking process finishes. Eventually the backtracking message arrives at the probes, in this case P0 and P3. In subsection 5.1.3

[1]While Crosstalk can function with more advanced probabilistic data structures such as sketches, Bloom filters are simpler and provide all the necessary mechanisms.

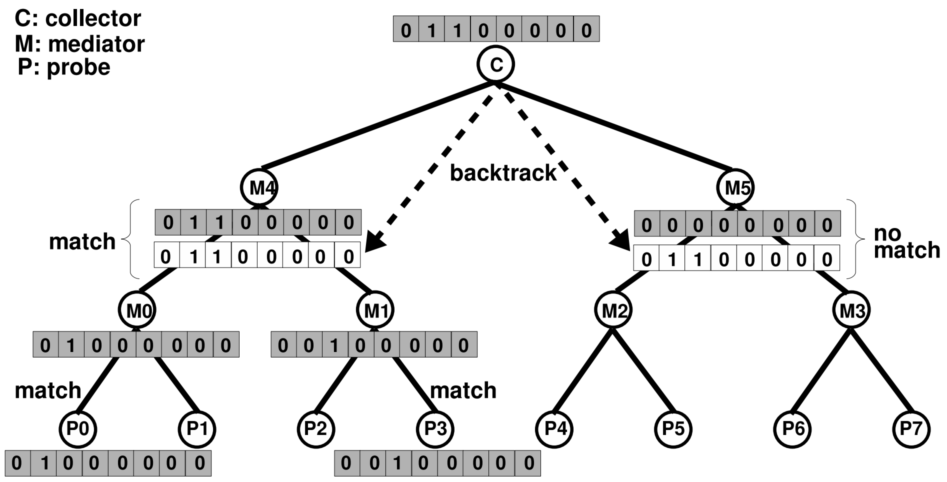


Figure 5.2: Example of backtracking. Each set of squares represents a Bloom filter. A dark Bloom filter represents stored state, while a white one information sent with the backtracking request. Here the matching is done using a bitwise "AND".

we provide an evaluation of the costs associated with this mechanism and show its applicability even in large networks.

5.1.2 Application: VoIP Attack Detection

In order to evaluate the performance of Crosstalk, we implemented an application over it aimed at detecting SIP-based VoIP attacks. Several types of attacks on SIP and its related media protocol, RTP, exist. In [207], for example, the authors describe billing frauds whereby a legitimate host is sent a fake SIP BYE message causing its call to be hijacked so that the attackers are using the operator's resources while the victim is being billed for the communications. The work in [202] describes an attack targeted at fooling the billing system into thinking a call is over by prematurely sending a SIP BYE message while keeping the corresponding RTP media traffic going. Further, SIP communications are also vulnerable to Denial-of-Service attacks such as BYE and CANCEL attacks, as well as call hijacking based on fake REINVITE messages [204].

The common thread among all of these attacks is that the RTP session keeps flowing (at least in one direction) even though the SIP control session has been torn down or redirected. As a result, Crosstalk can be used to detect such attacks by detecting a live RTP flow corresponding to a recently terminated (or redirected) SIP session. After this detection, backtracking can be used to reveal the actual nature of the anomaly, to pinpoint the malicious users, and to discard false positives. We use the remainder of the subsection to describe the application's implementation details, and evaluate its performance in the next subsection.

Our attack detection application works as follows: each of the probes monitors all the ongoing SIP and RTP traffic. The SIP messages are parsed and, for each call, the two end-points of the media traffic are located by examining the SDP data; as for the RTP traffic, the two end-points simply correspond to the source and destination addresses of the messages.

Our method assumes the probes to synchronously and periodically export their probabilistic summaries of the monitored traffic. Thanks to the large time granularity involved in voice traffic (seconds), an offset of tens of milliseconds among the probes' clocks is certainly acceptable; consequently, the probe synchronization requirement can be simply met by using protocols like NTP, with no need for special-purpose hardware.

The monitored data is used to fill two Bloom filters:

- A Bloom filter for keeping track of the end-points of the media traffic corresponding to SIP calls that have been terminated (or redirected) within the last measurement period.

- A Bloom filter for keeping track of the end-points of the RTP sessions that have been terminated (or redirected) within the last measurement period.

Clearly, the hash functions associated with these BFs must be the same so that the RTP and SIP endpoints associated with the same call are hashed into the same bit positions. Once these BFs are created, they are exported to the nearest mediator (i.e., the parent of the probe in the DAT). The mediator then joins all of the SIP BFs and all the RTP BFs received from its sons by performing a bitwise "OR", thus obtaining two summarized BFs that it forwards to its own mediator. Each mediator along the way caches a copy of the last BFs it has sent up the DAT in order to support possible backtracking requests.

The detection of the anomalous behavior is achieved by a node in the DAT performing a bit-wise "XOR" of the RTP and SIP BFs: if two bits in the same position are different, that means that either the data stream or the control stream have not been terminated. In that case, all of the node's children receive a backtracking request which includes the indices of the unmatched bits (i.e., the set bits that appeared in one BF but not the other). Each intermediate node then checks such bits against its cached aggregated BFs, and, if at least one among those is set, it propagates the BT request to its children. Such a procedure is repeated recursively until all the probes which have logged relevant information are reached.

Of course, collisions with other calls in the BFs may prevent the detection of such an event (a false negative); however, as we will show in the following subsections, the system can be dimensioned in order to keep this probability arbitrarily low. On the other hand, the false positive rate is almost negligible for this system: as collisions on the BFs cannot generate false alarms, these events can happen only in very rare cases:

- When, upon detection of an actual anomalous event, the backtracking requests reach some probes which did not log relevant information

- When the terminations of the media and control flows happen so close to the boundary between two measurement intervals that the two events are recorded in different time windows.

In both cases, false alarms are easily spotted: in the former, the post-event analysis (perhaps looking at logs) allows to discard pointless requests, while in the latter it is enough to match two adjacent time windows. We point out that, in case a system cannot tolerate any missed detections, a minor modification to our system allows us to fulfill this requirement: use the RTP BF to record the end-points of the ongoing (instead of those of the terminated) media flows, and detect anomalies by looking for matching bits between the two aggregated BFs by performing a bitwise "AND"; this change comes at the the cost of increased utilization of network resources (larger BFs are in general needed).

5.1.3 Evaluation

In this subsection we provide extensive simulation results to show the performance of Crosstalk, and in particular that of the VoIP attack detection application. Please note that throughout this subsection we use the term report to mean the Bloom filters exported between probes and mediators as a result of the monitoring and aggregation process.

5.1.3.1 Setup

In order to assess the performance of our solution, we evaluated several performance parameters through extensive simulations. In greater detail, we extended the Oversim overlay network simulator [208] by implementing a new application module with Crosstalk's basic functionality and which runs on top of the Chord.

The input to the simulation consists of Call Data Records (CDRs), in order to match the format used by our VoIP data set (a CDR is a short record of a VoIP communication, including fields like caller, callee, and call duration). To have control over their distribution, CDRs are fed to the simulated monitoring probes by a centralized CDR dispatcher module: each node is assigned a given range of the overall hash ID space and each CDR is handed over to the responsible node (based on its source address).

In order to simulate the fact that RTP and SIP traffic for the same call may traverse different paths, two separate copies of the same CDR, representing in turn the RTP and SIP traffic associated with a given call, are assigned to two distinct probes by using two independent hash functions. Such a choice is rather conservative, since in a significant fraction of the real cases, the two traffic streams are likely to follow the same path, but this approach is still useful to show that our system can cope with even this extreme case. It is worth noting that the simulated probes are actually nodes on the DATs, meaning that they can act as probes for one key but as mediators (and even collector) for others simultaneously.

Regarding CDR generation, we took two approaches. First, we generated CDRs randomly by setting the timestamps and the call durations according to a Poisson process (such a simple model has been extensively used in the field of telephone traffic measurement). The purpose here was to be able to effectively tune and change the simulation parameters to show the performance of the system. In the second approach we relied on an extensive data set gathered from a large VoIP operator in order to demonstrate Crosstalk's applicability to a real world scenario. In both cases we modified the CDRs at a certain rate (set as a percentage of the total CDRs) in order to simulate malicious calls.

5.1.3.2 Performance Analysis

In this subsection we present simulation results based on generated CDRs in order to assess the system's performance and scalability. Crosstalk's VoIP application depends on a number of different parameters:

- **Bloom filter size**, which affects several factors such as the missed detection rate, the bandwidth consumed and how much state nodes in the DAT keep.
- **The call rate**, in other words, how much traffic the system needs to monitor, export, and correlate.
- **The measurement interval**, which determines how long the probes keep data locally before exporting (longer intervals result in lower overheads but increase the detection delay).
- **The anomaly rate**, or percentage of malicious calls, which increases the costs associated with backtracking requests.
- **The number of probes**, equal in our case to the number of nodes in the p2p system, affecting the DAT's topology and therefore the messaging overhead, the amount of aggregation, and the detection delay.

Bloom filter size and call rate: For the first experiment we took a look at the first two parameters and their relationship to false negatives and positives. In other words, given a certain call rate, how would an operator deploying Crosstalk dimension the Bloom filter size (which affects things like bandwidth consumption) so that the false negative and positive rates are relatively low? To this end, consider that missed detections (i.e., false negatives) happen when a collision in one BF causes a match with a "true" set bit in the other BF, resulting in the "XOR" matching operation to return 0 (the misdetection). Because the cause is the collision within a BF, all the well-known results about BF performance evaluation and dimensioning apply to our system. In particular, as the number of keys in the BFs equals the call rate λ times the measurement period T, the missed detection (md) probability can be expressed as:

$$P(md) = (1-(1-\frac{1}{M})^{K\lambda T})^K \sim (1-e^{-\frac{K\lambda T}{M}})^K$$

where M stands for the BF size (in bits) and K for the number of hash functions (which is set to an optimal value depending on the other parameters). In order to dimension the BF size for a given missed detection probability, the following inequality

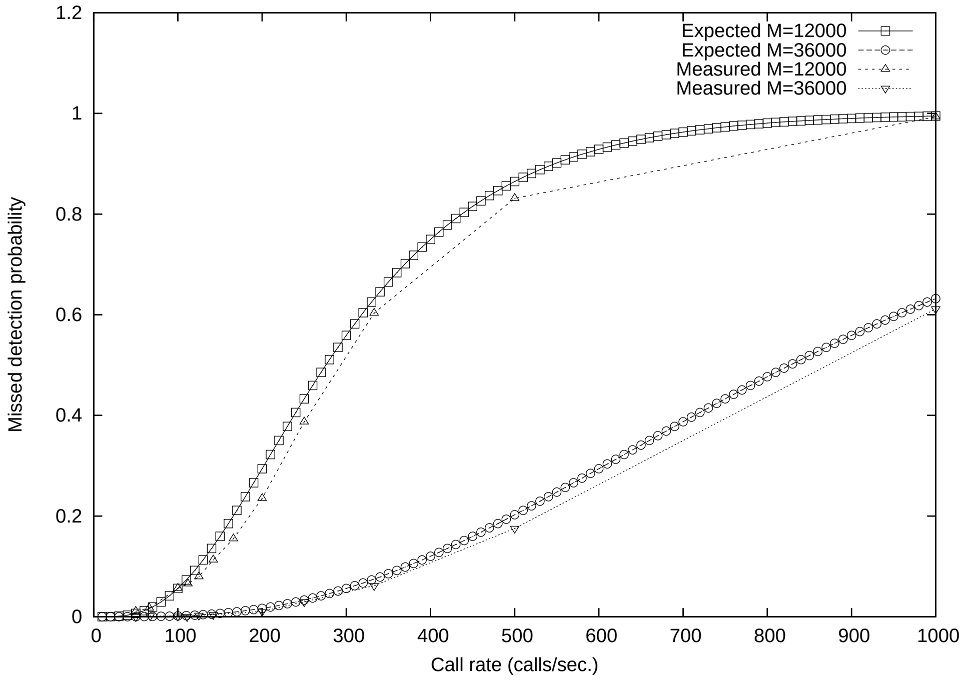


Figure 5.3: Measured and expected missed detection probabilities for different call rates and BF with sizes of 12,000 and 36,000 bits.

can be leveraged:

$$M \geq \lambda T \log_2(e) \log_2(\frac{1}{P(md)})$$

As for the false positives, they are mainly due to backtracking (BT) messages reaching probes which did not log any relevant event. However, since BT requests are triggered only when an actual anomaly is detected, the probability of such events (an anomaly happening and a request reaching a wrong probe) is definitely low. During our simulations we never observed more than a dozen such events, even when generating hundreds of thousands of calls.

In order to verify this model's accuracy, we ran simulations to evaluate the missed detection probability for two different BF sizes and call rates ranging from 10 to 1,000 calls per second, plotting the measured results against the model's expected values (see figure 5.3). As shown, the model can, to a fairly high degree, predict the actual system behavior.

To get a feel for the system's performance we rely on this model and on our CDR database, which shows a peak call rate well below 100 calls per second. Even if we assume that since more and more users are migrating from PSTN to VoIP such a figure will increase in the future by an order of magnitude, our system can handle the resulting traffic volume (1,000 calls/sec.): exporting data every 10 seconds and using 17KB-wide BFs yields a target missed detection probability of 10^{-3}, while using 34KB-wide BFs yields a target missed detection probability of 10^{-6}. For a measuring infrastructure made up of 1,000 probes the *total* reporting traffic adds up to only few MB/sec.

Measurement interval: If the BF size does not vary, a longer measurement period implies a larger number of keys in the BF, and, in turn, an increased missed detection probability. On the other hand, of course, this involves a lower bandwidth consumption, as data summaries are exported less frequently. Depending on the operational constraints, the previously presented mathematical model allows to find out a good trade-off; we do not present more extensive results here due to space constraints.

Anomaly rate: The next parameter we looked at was the anomaly rate, and in particular how it affects the costs related to backtracking (BT). Backtracking is triggered by either a detected anomaly or a false positive. Assuming a well-dimensioned system with a low false positive rate (e.g., less than 1%) and no anomalies, simulation results show about an order of magnitude difference between export messages and BT messages.

Arriving at more precise figures is difficult since the actual number of backtracking messages generated depends on the number of probes which have to be reached by a BT request and on the topology of the tree. Having said that, we ran a simulation to get a feel for the effects of the anomaly rate on the system, and in particular the cost of backtracking (see figure 5.4).

The figure shows that, unless a very unrealistic scenario is assumed (a network where one in ten calls is malicious), the fraction of BT messages is small (usually an order of magnitude smaller) with respect to the number of reports, which proves that the BT mechanism can locate the relevant probes without flooding the DAT with messages. Further, the behavior of the system improves as the number of nodes increases.

These figures can be even further improved by reducing the size of each BT message. Observe that the Bloom filter obtained through a bit-wise "XOR" of the aggregated SIP and RTP reports must have a very limited number of set bits (in fact, the number of such bits should be lower than the number of malicious calls times the number of hash functions), which lends itself to compression. In order to effectively compress such a "sparse" bitmap, it is sufficient to include the indices of the set bits within the BT message: the resulting message size would be roughly some dozens of bytes, which is negligible when compared to the bandwidth consumed by the reporting messages. Even more efficient compression schemes for sparse bitmaps can be adopted: Fastbit [209] is just an example of a technique achieving good compression while still allowing bitwise operations to be performed over the codified data.

Number of nodes: The number of probes does not affect the accuracy of our system (that, in fact, depends on the overall number of monitored calls) but rather the aggregation and backtracking delay and the overall bandwidth consumption. The former depends on the depth of the tree, which, in turn, grows logarithmically with the number of probes. On the other hand, the overall bandwidth consumption due to the report messages grows linearly with the number of probes (each additional probe corresponds to an additional edge on the tree, which, in turn, corresponds to an additional report being transmitted). As for the BT requests, their amount depends on several variables, but we already showed their bandwidth consumption to

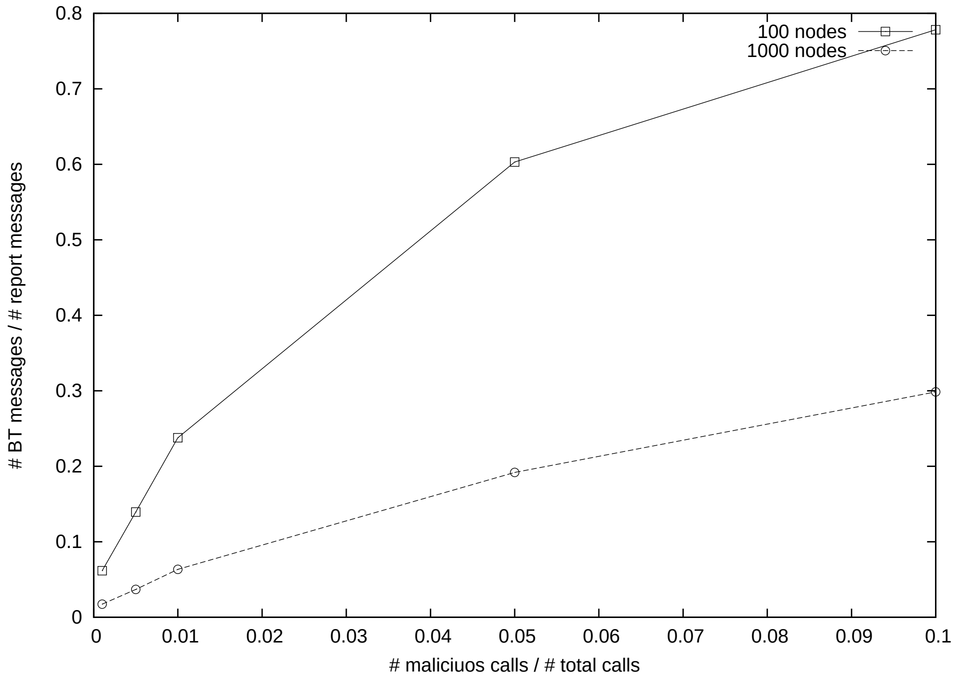


Figure 5.4: Rate between backtracking and report messages in DATs made up of 100 and 1,000 probes with a varying anomaly rate.

calls/sec.	P(md)	P(fp)	no. BT/ no. reports
30	0.0014782	0.0000316	0.2585591
34	0.0014215	0.0000632	0.258106
42	0.0033482	0.0000527	0.2986333
41	0.0016393	0.0000632	0.2970421
33	0.0021536	0.0000738	0.2561671
8	0.0000000	0.0000000	0.1249855
1.3	0.0000000	0.0000000	0.024015
0.5	0.0000000	0.0000000	0.0125396

Figure 5.5: Crosstalk's performance when using real-world data from a large VoIP provider. Each row represents a different 30-minute time sample in our data set.

be negligible with respect to that of the report messages.

5.1.3.3 Real-World Performance

In the previous subsection we looked at how Crosstalk behaves when varying a number of different parameters in simulation. To get a sense of how it would perform in a realistic scenario we replaced the generated CDRs with those of an extensive data set consisting of more than 100 million CDRs from more than 15 million users collected over a period of more than 4 weeks. In order to test our system in different traffic scenarios without having to face prohibitive simulation times, we ran our experiments by using 30 minutes time slots that we sampled out of our complete database. We assumed a system with 1,000 probes, 4KB BFs, a measurement interval of 20 seconds, and a (quite conservative) anomaly rate of 3% (see figure 5.5). The results clearly show that Crosstalk is more than able to cope with this traffic, yielding very low false negative (md) and false positive rates as well as a negligible number of backtracking messages with respect to the number of reports. It is worth noting that while the table lists figures from a few 30-minute time samples in our data set (one per row), we ran simulations for others and obtained similar results.

5.2 DECON: Decentralized Coordination for Large-Scale Flow Monitoring

Monitoring at higher granularities than packets, and in particular at the flow level, certainly alleviates the problem of coping with high traffic volumes. The widespread use of protocols like NetFlow and sFlow is evidence of the fact that monitoring at the flow level provides the necessary data to carry out essential network tasks, while at the same time reducing the load on the devices that gather such data.

While the flow abstraction helps, clearly any sizable network requires several monitoring probes to have different observation points but also in order to scale to the large number of flows that go through it. This situation raises the following question: given a set of monitoring probes and a set of flows going through them, which flows should a probe monitor at any given point in time? Such a mapping of flows to probes should be done with the aim of maximizing the number of flows actually monitored, as well as removing redundancies (for example, preventing a flow to be monitored simultaneously by two or more probes).

In essence this is a coordination problem, and in this section we present DECON, a decentralized coordination system aimed at tackling it. Because the coordination happens in a decentralized manner, DECON scales to large numbers of flows and probes. In addition, the system requires neither topology information nor traffic matrices, as is the case with other approaches. We present extensive simulation results that show that despite having a decentralized coordination mechanism, DECON achieves a high degree of coverage (i.e., the number of flows that are actually monitored) even when faced with a large number of flows, including short-lived ones.

5.3 Related Work

The problem of coordinating flow monitoring tasks among a set of probes was also tackled in CSAMP [210]. Unlike DECON, CSAMP uses a centralized decision point which knows both the routing state and the traffic matrix of the network, and that is in charge of periodically computing the subset of flows each monitoring probe is responsible for. While sharing similar goals as CSAMP's, our system achieves them in a distributed, fault tolerant, and more scalable architecture with no need for detailed information about the network. Another solution in this space [211] suggests that monitoring probes use Bloom filters and a gossip protocol in order to exchange information about which flows they are monitoring, and thus coordinate their activities. While decentralized, this approach suffers from serious scalability problems, since the messaging overhead of a gossip protocol does not scale well with the number of probes nor with the number of flows to be monitored.

In [212] the authors propose a technique for choosing the monitoring points and their associated sampling rates according to optimality criteria. Unfortunately, the approach requires a-priori knowledge of the network routing state and does not address the issue of duplicate measurements (the authors assume that duplicates can be detected at the collector). The work in [213] proposes a double-hash based approach whose purpose is to ensure that the same packets are monitored by all of the probes, in order to provide multi-point measurements. Although this can be also achieved by our scheme, the reverse is not true: a double-hash based schema cannot ensure that every flow is monitored only once, unless the path of each flow is known beforehand.

5.3.1 DECON's Architecture

DECON's architecture is in charge of making decisions about which monitoring probes in the network should monitor which set of flows going through them. The aim is to spread the load across the available resources in order to increase coverage, which is the number of flows actually monitored during a certain time period. Further, DECON achieves this goal in a decentralized way, without the need of calculating traffic matrices nor having knowledge of network topology.

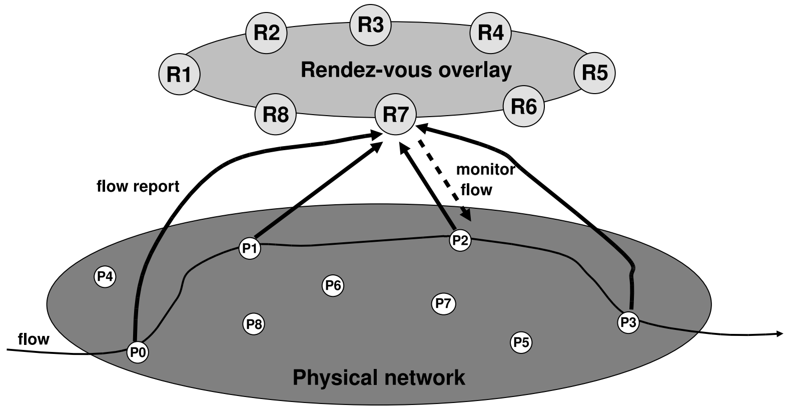


Figure 5.6: DECON's architecture. Monitoring probes (P) send reports about flows to the rendez-vous overlay, which then decides which of the probes seeing a flow should monitor it.

To achieve this, DECON relies on a peer-to-peer network called the *rendez-vous* overlay (see figure 5.6). When a new flow arrives in the network and goes through a set of monitoring probes (P0, P1, P2 and P3 in the figure), each probe computes the flow's hash[1]. Each probe then sends a small report to the node in the overlay responsible for the value resulting from the hash, called a *rendez-vous point* (RP). The RP (R7 in the example) receives messages from all probes seeing a particular flow, and decides which of these (P2) should do the actual monitoring; it then sends messages back communicating the decision. For negative decisions, probes stop monitoring the flow and remove any state associated with it.

Clearly, a number of strategies are possible when deciding which probe should monitor the flow. Perhaps the simplest one is *first-fit*, where the RP assigns the flow to the probe whose message arrives first, a likely less-than-optimal strategy that has the advantage of reducing the decision delay (the time between when a flow is first seen at a probe and the decision message arriving at that probe). A more advanced strategy is *best-fit*, in which probes send a metric in the message reflecting their current load (e.g., the current number of flows being monitored, CPU utilization, etc), and have the RP choose the least-loaded probe as the one that should monitor the flow. This strategy spreads the monitoring load better across the probes, but increases the decision delay, since the RP now has to wait to make sure that all reports from probes seeing a flow have arrived before making a decision. In subsection 5.3.2 we evaluate these two strategies, leaving more advanced strategies as future work.

DECON's decentralized decision process as well as its reliance on a p2p overlay allows it to scale to large networks while being resilient to failure. In addition to this, DECON's coordination mechanism has a couple of other beneficial features. First, the system can easily cope with flows changing their path through the network.

[1]The hash we used is based on the flow 5-tuple <src/dst IP address, src/dst port and protocol ID>, but any other flow definition can be used.

Suppose that in figure 5.6 the path changed so that the flow went through P5 instead of P2. In this case, P5 would send a message to R7 telling it that it has seen a "new" flow. Because R7 keeps state about flows and its previous decisions, it knows that this is not a new flow. As a result, it will send a message to P2 to ensure that it is still seeing the flow. If it is, it may evaluate whether P5 is a better choice (e.g., less loaded) or decide to do nothing, keeping P2 as the "active" probe; if, on the other hand, P2 no longer sees the flow, R7 will evaluate which of P0, P1, P3 and P5 is the best choice to monitor the flow.

The system is also resilient to losses: if a probe completely misses a flow the responsibility of monitoring it will be assigned to another node. In order for a flow to be completely ignored, each of the probes on its path must be unable to monitor it: as we will show in the evaluation subsection, this happens seldom even with high network loads.

The second feature of the system is that, by nature of the decision process, it prevents undesired duplicate monitoring. It may, of course, sometimes be desirable to monitor a flow more than once (for example, to measure performance statistics at various points in the network). One of DECON's strengths is that it can accommodate a number of different decision strategies in order to suit different monitoring needs.

As an additional feature, our system can easily support flow sampling, as Csamp does. As opposed to per-packet sampling, such a sampling technique involves monitoring a given flow with a certain probability. This can be accomodated by DECON: the RP point can, with some probability, decide whether or not the probes detecting a flow will monitor it, thus implementing random sampling.

5.3.1.1 Batch Optimization

In order to reduce messaging overhead, reports can be batched. Since each probe accesses the rendez-vous overlay through a single *ingress node*, it is possible for the probe to bundle these reports into a single report, and send this batch report to the ingress node upon expiration of a timer (the reports consist of very little information, so it is possible to store many of them in a single packet). Upon receival, the node parses the reports and sends each to the responsible RP. This same optimization can be implemented in order to reduce the number of response messages directed to a monitoring probe: the RP sends the response messages to the corresponding ingress points for each reporting probe; the ingress point then can, in turn, bundle the response messages into a single batch response.

As a result of this mechanism, the number of exchanged messages outside the overlay would then depend only on the batching period and no longer on the number of flows in the network. Further, this mechanism keeps most packets related to reports within the overlay, an infrastructure which has to fulfill only the coordination task and that can be easily scaled. Of course, such an optimization may increase the decision delay, as the reports are queued waiting for a batch message to be sent; however, we will show in the evaluation subsection that the overall performance is only marginally affected.

5.3.2 Evaluation

We conducted extensive simulations to show that DECON can scale to a large number of flows. In this subsection we describe the simulation setup, the simulation results, and performance results from a prototypical monitoring probe that show that even commodity-hardware can fulfill DECON's requirements.

5.3.2.1 Simulation Setup

In order to assess the performance of our solution we implemented a special-purpose discrete-event simulator which models all the variables that affect the behavior of our system even under heavy traffic load. We simulated several network topologies composed of hundreds to thousands of nodes; to this end, we leveraged the simple and well-known Barabasi-Albert model [18], which allows to build huge scale-free graphs with a preferential attachment procedure. Even if such a model does not exactly represent all of the topological features of a real network, it nonetheless reproduces a topology where a few hub nodes are crossed by a large number of paths, as is common in real networks.

In order not to bias our results by assuming a particular probe placement strategy, we assumed each node in the network to be a probe. Regarding link delays, we generated them randomly within a range of values that spanned up to ten milliseconds; we chose such a range of values after observing delay statistics published by the Internet2 network observatory (such values usually never exceed a few dozen milliseconds). For the communication between probes and the overlay we used larger latencies of up to 20 milliseconds, since we assumed that reports could cross several links before reaching the overlay. As for the overlay, we assumed the rendez-vous points to be organized in a Chord ring, where the delays for each hop are in the order of a few milliseconds.

We generated flows by picking up a random pair of end-points within the generated topology and by assuming a Pareto-distributed duration (the simplest mathematical model for a heavy tail distribution), with mean values of around 30 seconds.

However, by examining the results of experimental runs performed with different average durations, we found out that such a parameter does not significantly influence the peformance, as long as it is in the order of seconds (i.e. much larger than the network latencies). A study from a few years back confirms this, claiming an average tcp session to be between 12 and 19 seconds long [214].

We made such a choice after analyzing traffic traces published by the Mawi group; such traces were captured on a trans-Pacific line in early 2009, thus representing up-to-date samples of real backbone traffic. As for the number of flows, we once again relied on Internet2 data which had about 9 million flows over a 5 minute time span, or about 30,000 flows/sec. Since we would like our system to scale up to very large topologies, we actually simulated much larger values (hundreds of thousands of flows per second over the whole topology).

5.3.2.2 Simulations

We used the simulator to evaluate several performance parameters of the system. One of the most relevant is the achievable flow coverage, in other words, the percentage of flows that can be monitored with a fixed amount of resources (we assume that each probe can monitor up to a certain limit of flows at the same time). In greater detail, we simulated a network with 300 monitoring probes, each of them capable of monitoring up to 10,000 flows. We evaluated the flow coverage that can be achieved by using our coordination scheme under the two flow assignment strategies mentioned in subsection 5.3.1: first-fit and best-fit. Further, in order to more clearly illustrate DECON's impact, we ran simulations to see what happens when no coordination is used at all; the results for different traffic loads are shown in figure 5.7. It is evident that, while without coordination the number of missed flows grows quickly with the network load, DECON keeps these misses almost constant and significantly lower. In particular, the best-fit strategy, as expected, achieves the best performance when faced with very high flow rates.

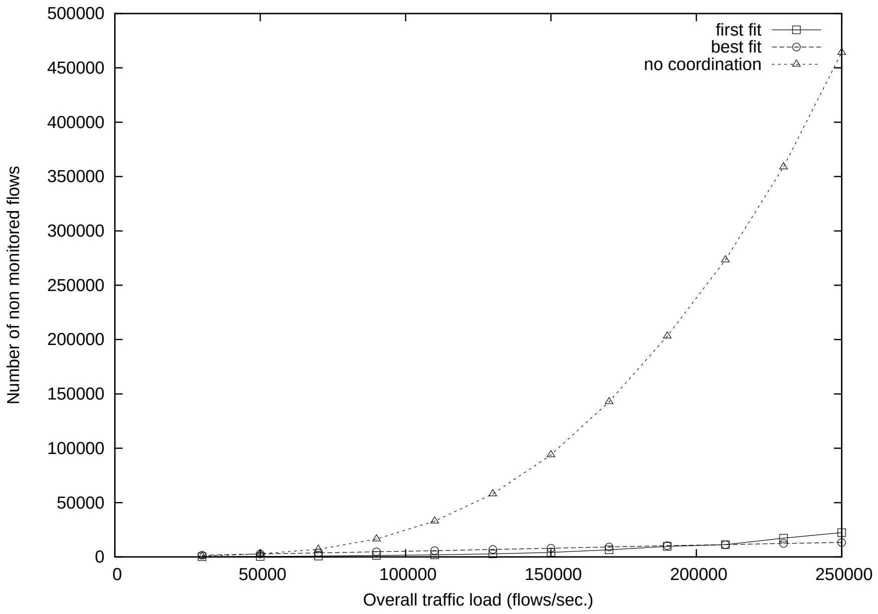


Figure 5.7: Number of total flows actually monitored without coordination and using two different coordination strategies.

Besides improving flow coverage, our solution prevents two or more probes from unnecessarily monitoring the same flow (DECON can of course also allow a flow to be observed at several probes when needed). In figure 5.8 we show the average number of times a single flow is measured when no coordination mechanism is used: even if such a figure improves with higher traffic rates (there are simply not enough resources for duplicate measurements) it is clear that, on average, even under high load, each flow is wastefully monitored more than once.

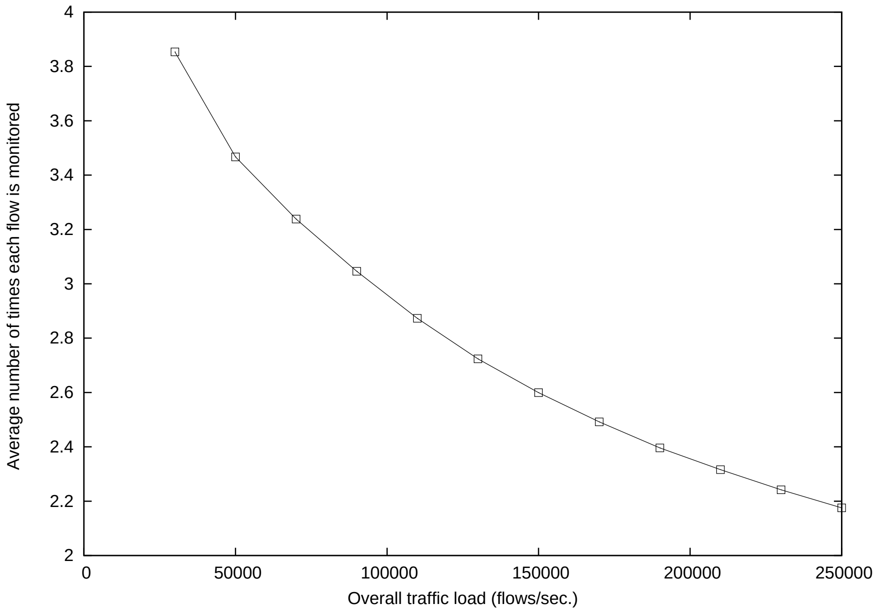


Figure 5.8: Number of times a single flow is monitored without coordination.

We also evaluated the ability of our system to balance the burden of the monitoring activity among all the probes. Load balancing is not trivial to achieve because some nodes in the topology act as hubs, and, without a proper coordination scheme, are likely to be overloaded. Figure 5.9 shows the histogram of the average number of monitored flows for each probe in a scenario with 200 probes, each one able to concurrently monitor up to 10,000 flows and with a rate of 190,000 new flows per second over the overall network. Again, we plot the results achieved by the two different allocation strategies and those obtained with no coordination.

As expected, the best-fit scheme achieves the best balance among all the probes (it has the highest number of probes with a similar number of flows), while, with the first-fit allocation strategy, a small number of the probes (likely the hub nodes) are overloaded. With no coordination scheme, the mean resource occupation is much higher and a large fraction of the monitoring probes is always overloaded.

In order to provide a way of dimensioning our system, we ran a series of simulations without imposing any resource limitation on the probes (in terms of number of flows monitored), measuring how many resources would be needed on the probes in order to monitor all the traffic with no (or negligible) losses. More specifically, we computed the 99-percentile of the number of monitored flows with a varying number of probes (reaching up to 1,500 probes) and with a fixed load of 100,000 new flows/sec. Further, we used a best-fit allocation strategy, since, without buffer limitation, first-fit would simply allocate a flow to the first probe reporting it. The results

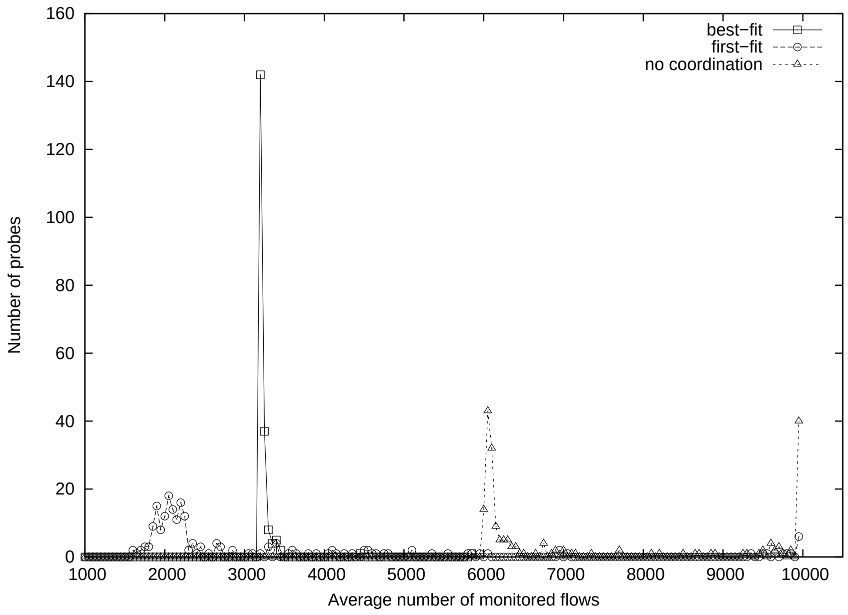


Figure 5.9: Load-balancing histogram showing the number of flows each probe has to monitor with and without coordination.

are plotted in figure 5.10 and show that, by leveraging a large number of measurement probes and a proper coordination scheme, DECON can monitor high traffic volumes while requiring a small amount of resources from each probe. In the next subsection we will show that such a resource constraint can be met by using cheap commodity hardware.

Another important parameter that we evaluated is message overhead (i.e., the number of messages that probes send to the coordination overlay). In particular, we computed the average number of messages per probe when having 200 and 400 probes and with different number of flows; the results are plotted in figure 5.11. As shown, the number of generated reports is well below 10,000/sec even for 180,000 flows. Since each report has a very small payload, this number corresponds to a rate of less than 1 MB/s.

We also extended our simulator in order to support the batching optimization described in the previous subsection. In particular, we tried to evaluate the impact on the overall flow coverage that the additional delays incurred by this scheme had. To this end, we ran several simulations with different traffic loads and different batching periods. In each scenario, we evaluated the ratio between the number of missed flows with batching and the number of missed flows without batching for varying time periods (see figure 5.12). As expected, the performance gets worse with increasing batching periods, as responsiveness is being traded-off against lower overhead. However, we point out that even for fairly large batching periods (0.1 seconds corresponds to 10 messages per second per probe) the loss is relatively small, and this figure only improves with higher numbers of flows.

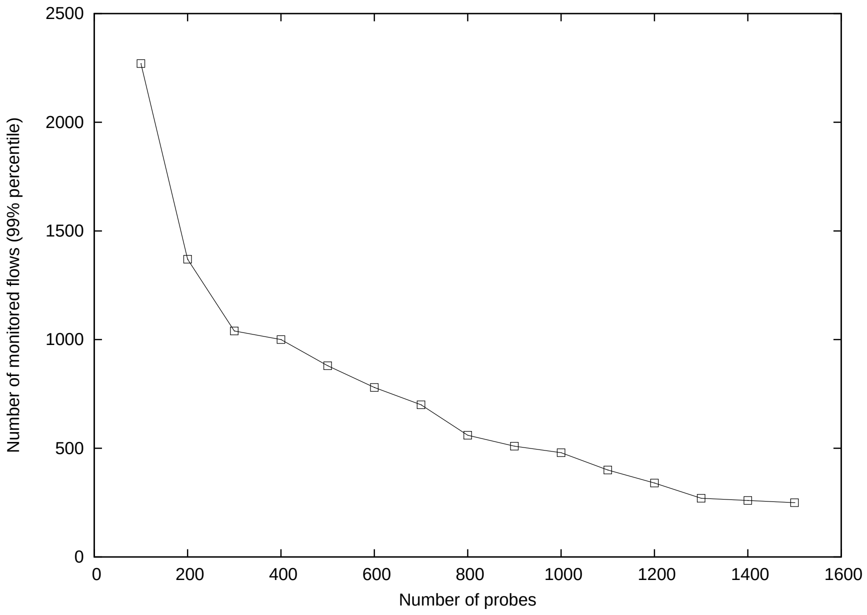


Figure 5.10: 99-percentile of the number of monitored flows per probe for a varying number of probes and a fixed load of 100K flows/sec.

5.3.2.3 Monitoring Probe

As shown in the simulations, DECON puts certain state requirements on monitoring probes, more specifically in terms of how many flows a probe has to keep track of at any given point in time. In order to demonstrate that these are not unreasonable, we built a simple monitoring probe using the Click modular router. Click is based around the concept of *elements*, which are small units that perform different kinds of packet processing such as looking up an entry in a forwarding table, responding to ARP queries, or queueing packets; a Click configuration file then specifies how elements should be connected to each other.

To implement the probe, we created a new Click element called *FlowMon*. The element is based around a hash, and keeps track of all flows that the probe is in charge of, updating simple statistics about them such as packet and byte counts. While the probe has timeout counters for detecting flow expiration and for when a decision takes too long to arrive from a rendez-vous point, we disabled these during this evaluation in order to test the worst-case performance where flows do not expire and the probe is responsible for all flows it sees.

In terms of hardware, we used a Dell 2950 with two Intel Xeon X5355 2.66GHz quad-core processors, 8 GB of main memory and 3 quad-port Intel 82571EB PCI express network cards. In addition, we used Dell 1950s to both generate and count traffic. Since the Dell 2950 acting as the probe had a maximum of 12 network interfaces, we connected three traffic generators and three traffic counters to it, as shown in figure 5.13 (the dell 1950s can generate packets at line rate for all packet sizes out of a

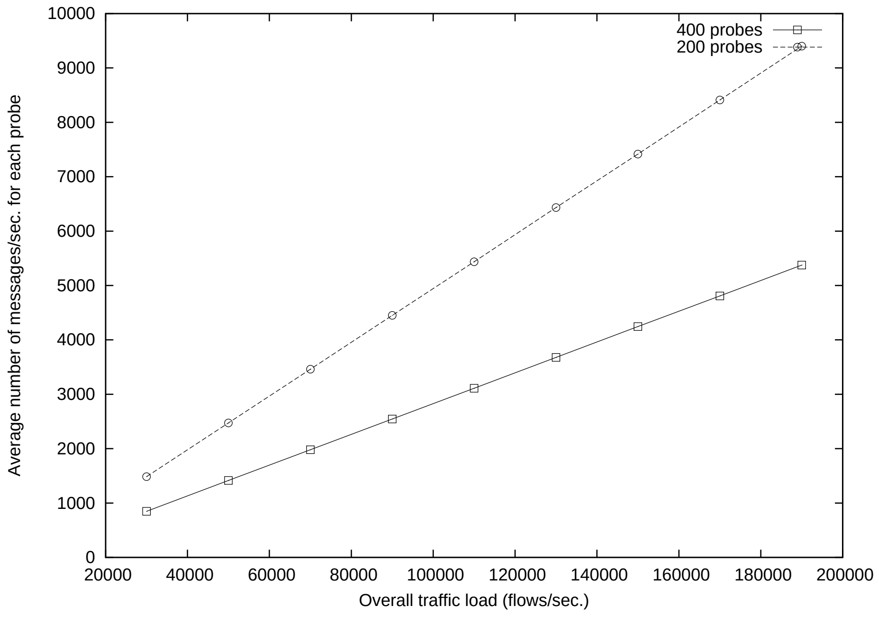


Figure 5.11: Average number of messages per probe.

maximum of two interfaces). The aim was to test the performance of the monitoring probe when faced with a large number of flows of different packet sizes. While our generators (x86 servers running Click) could send packets at line rate for all packet sizes, due to memory limitations each of them could only generate 5,000 flows, for a total of 15,000 flows going through the monitoring probe. With this in place, we measured the probe's throughput for different packet sizes while keeping track of statistics for all of these flows (figure 5.14). As can be seen, even for minimum-sized packets the probe reaches a very reasonable 2.5Gb/s; this figure quickly ramps up to the line rate value of 6Gb/s for 200-byte packets and larger. These results show that the state requirements arising from DECON's coordination (recall from figure 5.10 a maximum of about 2,300 flows going through any one probe) can be met even by inexpensive, off-the-shelf hardware.

5.4 The LogLog Counting Reversible Sketch: a Distributed Architecture for Detecting Anomalies in Backbone Networks

During the last years, many research groups have focused their attention on developing novel detection techniques, able to promptly reveal and identify network attacks, mainly detecting Heavy Changes (HCs) in the traffic volume [215], [216], [217], [218], and [219].

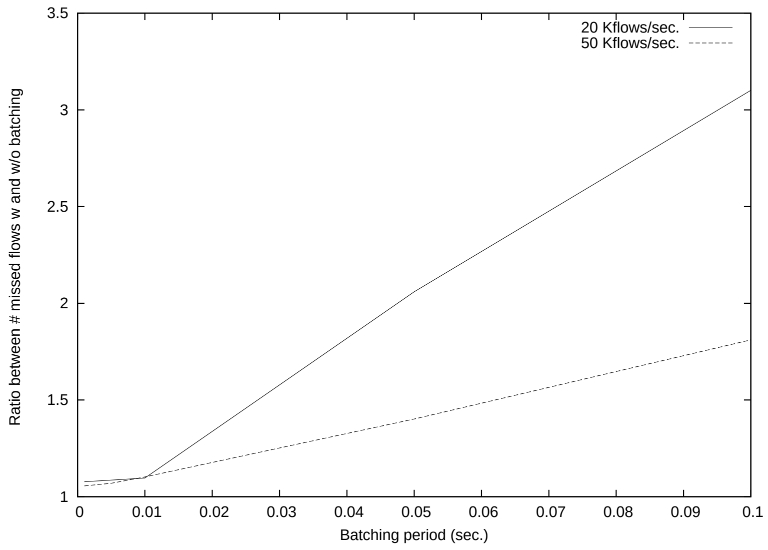


Figure 5.12: Ratio between the number of missed flows with batching and the number of missed flows without batching.

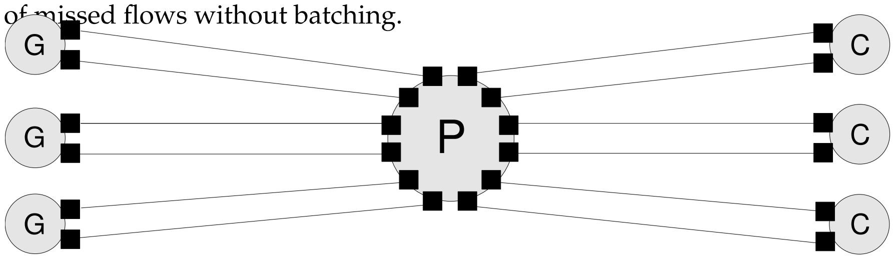


Figure 5.13: Network topology used to test the probe's performance. G stands for generator, P for probe, and C for counter.

Nevertheless, the recent spread of coordinated attacks, such as large-scale stealthy scans, worm outbreaks, and distributed denial-of-service (DDoS) attacks, that occur in multiple networks simultaneously, makes extremely difficult the detection, using isolated intrusion detection systems that only monitor a limited portion of the Internet. Hence, the research efforts are now moving to develop distributed approaches to solve such an issue [220].

For this reason, in this section, we propose a novel distributed architecture for the detection of network anomalies in backbone networks. Such architecture can be seen as a general framework on top of which most anomaly detection techniques can be implemented.

According to our approach, multiple detection probes, distributed in the backbone network, monitor a given portion of the network separately. Such reports are collected and aggregated by an overlay of nodes (named *mediators*) that analyzes the

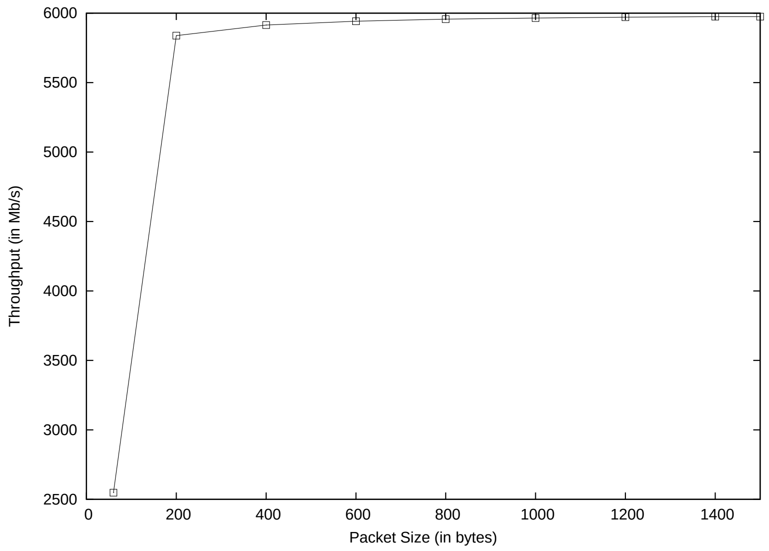


Figure 5.14: Click monitoring probe throughput performance while monitoring 15,000 flows of different packet sizes.

data and generates the alerts. The working principle is similar to that of the system described in section 5.1,

In this work, we consider anomaly detection algorithms that analyze traffic volumes, that means that the data collected by the probes are represented by the estimation of the number of traffic flows observed in a given time bin.

Hence, the first problem to be solved is to provide a reliable estimate of such quantities. This task, that is not trivial when performed over the multi-gigabits links of a backbone network, has been discussed in several previous works, and the use of probabilistic data structure (e.g., sketches) has emerged as a *standard* approach [221].

To effectively solve this estimation step we propose a novel probabilistic data structure, named LogLog Counting Sketch (LLCS), based on the combined use of the LogLog Counting algorithm and the k-ary sketches, as detailed in the following subsections.

In the proposed approach, this task is performed by the distributed probes, that then forward the computed LLCSs to mediators, which are responsible for aggregating them. It is worth noticing that, when aggregating these structures at the mediator level, we have to solve the hard problem of not counting duplicated flows, that are the flows observed by more than a single probe and thus observed in more than a single LLCS.

In the following we demonstrate that our distributed architecture is able to efficiently solve such problem also taking into account some privacy related problem that can arise in a distributed environment. Finally, we also show that not only is our architecture able to detect the anomalies but it can also effectively solve the identification problem (namely identifying the IP addresses responsible for the detected anomaly) by back-propagating the detection information from the mediator to the probes that run an appropriate identification algorithm.

5.4.1 Theoretical Background

In this subsection we present some theoretical background information, necessary to understand the proposed architecture. Note that we focus on the useful details only, referring the reader to the provided references for a complete description of both the LogLog counting algorithm and the reversible sketches.

5.4.1.1 LogLog counting

The LogLog counting is a probabilistic algorithm proposed in [222] as a means for counting the number of distinct elements in a data set.

The problem can be formalized as follows. Given a multi set M produced starting from a discrete universe U, we want to estimate its cardinality, that is the number of distinct elements it comprises.

Like in many similar algorithms, even in this case it can be assumed that a hash function is available for transforming each element of U into sufficiently long binary strings (x) producing a "random" set M with cardinality n. Note that the use of hash functions allows us to obtain strings x with random uniform independent bits.

Given that, let us suppose that the strings are infinitely long, that is $x \in \{0,1\}^\infty$ (this is a convenient abstraction at this stage) and let $\rho(x)$ denote the position of its first 1-bit. We expect that about $n/2^k$, among the distinct elements of M, have a ρ-value equal to k.

Thus, the quantity

$$R(M) = \max_{x \in M} \rho(x) \tag{5.1}$$

can be reasonably a rough approximation of $\log_2 n$.

To improve the estimate of n, we can divide the elements of M into m groups ($M^{(j)}$ with $j = 1, 2, \cdots, m$). Typically, $m = 2^k$ so that we can use the first k bits of x to represent the binary index of the group. For each group $M^{(j)}$ the algorithm computes the parameter $R^{(j)}$, after discarding the first k bits of the strings $x \in M^{(j)}$. Then, the arithmetic mean

$$\frac{1}{m}\sum_{j=1}^{m} R^{(j)} \tag{5.2}$$

can legitimately be expected to approximate $\log_2(n/m)$ plus an additive bias.

Thus, the estimate of n according to the LogLog algorithm is:

$$E = \alpha_m m 2^{\frac{1}{m}\sum_{j=1}^{m} R^{(j)}} \tag{5.3}$$

where α_m is the bias correction factor in the asymptotic limit and can be evaluated as follows.

$$\alpha_m = \left(\Gamma(-1/m)\frac{2^{-1/m}-1}{\log 2}\right)^{-m} \tag{5.4}$$

$$\Gamma(s) = \frac{1}{s}\int_0^\infty e^{-t} t^s \, \mathrm{d}t \tag{5.5}$$

5.4.1.2 Reversible Sketch

The sketch has proven to be useful in many data stream computation applications [223], [224], [221]. Recent work on a variant of the sketch, namely the k-ary sketch, showed how to detect heavy changes in massive data streams with small memory consumption, constant up- date/query complexity, and provably accurate estimation guarantees [225].

However, sketch data structures have a major drawback: they are not reversible. That is, a sketch cannot efficiently report the set of all keys that correspond to a given bucket of the sketch.

To overcome such a limitation, [226] proposes a novel algorithm for efficiently reversing sketches, focusing primarily on the k-ary sketch. The basic idea is to hash intelligently by modifying the input keys and/or hashing functions so as to make possible to recover the keys with certain properties like big changes without sacrificing the detection accuracy.

The basic idea is to modify the update procedure for the k-ary sketch by introducing modular hashing and IP mangling techniques.

The modular hashing works partitioning the n-bit long hash key x into q words of equal length n/q, that are hashed separately using a different hash function, h_{di} $(i = (1, \ldots, q))$. Let us consider that the output of each function is m-bit long. Finally, these outputs are concatenated to form the final hash value (see Figure 5.15).

$$\delta_d(x) = h_{d1}(x)|h_{d2}(x)| \ldots |h_{dq}(x) \tag{5.6}$$

Since the final hash value consists of $q \times m$ bits, associated with the LLCS columns, it results $w = 2^{q \times m}$.

Note that the use of the modular hashing can cause a highly skewed distribution of the hash outputs. Consider, as an example, our case in which IP addresses are used as hash keys. In network traffic streams there are strong spatial localities in the IP addresses since many IP addresses share the same prefix. This means that the first octets (equal in most addresses) will be mapped into the some hash values increasing the collision probability of such addresses.

To effectively resolve this problem, the *IP mangling* technique has to be applied before computing the hash functions. By using such technique the system randomizes, in a reversible way, the input data so as to remove the correlation or spatial locality.

The other key point introduced in [226] is the algorithm for reversing the sketch, given the use of modular hashing and IP mangling. For the sake of brevity, we skip the discussion of this algorithm, referring the reader to [226] for all the details.

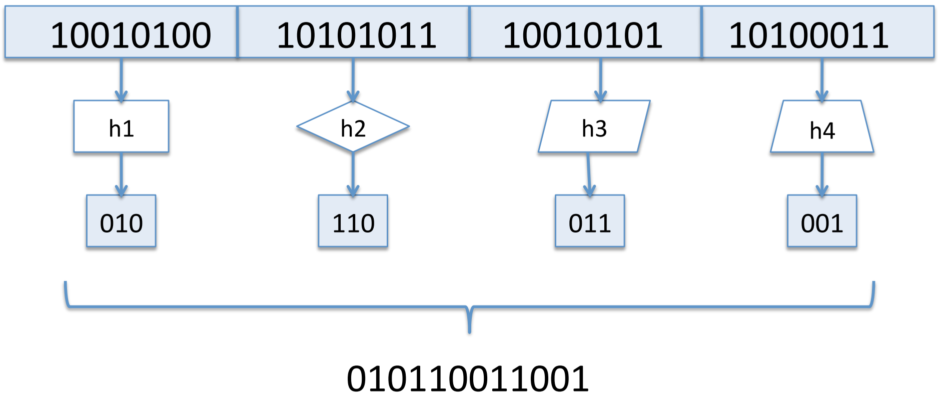


Figure 5.15: Modular Hashing

5.4.2 System Architecture

In this subsection we present a novel collaborative Intrusion Detection System based on a distributed architecture.

According to our approach multiple detection probes, distributed in the network, monitor a given portion of the network separately and report the collected information to a set of mediators, which perform partial aggregation and analysis. Such mediators are part of an aggregation tree, whose root node generates the alerts.

A backtrack mechanism is then used by the mediator to ask the probes for flow identification. Figure 5.16 depicts the proposed architecture; for the sake of simplicity, only one mediator is shown.

5.4.2.1 System Input

The system input consists of data measuring network traffic gone through a given probe. Starting from the observed traffic, they produce a periodical report that contains information related to the traffic measured in a given time bin, and consists of a collection of flow keys: ⟨*IP source address*, *IP destination address*, *source Port*, *destination Port*, *Protocol*⟩.

5.4.2.2 LogLog counting Sketch module

The periodical reports are passed as input to the module responsible for the construction of the LogLog Counting Sketch table (LLCS). This data structure is a three-dimensional array $S_{D\times W\times L}$, where each row d ($d = 1, \dots, D$) is associated with a function δ_d that can assume values in the interval $(1, \dots, W)$ that are associated to the columns of the array. Note that the two-dimensional substructure $S_{D\times W}$ is a *standard* sketch table. The third dimension L has been introduced to integrate the

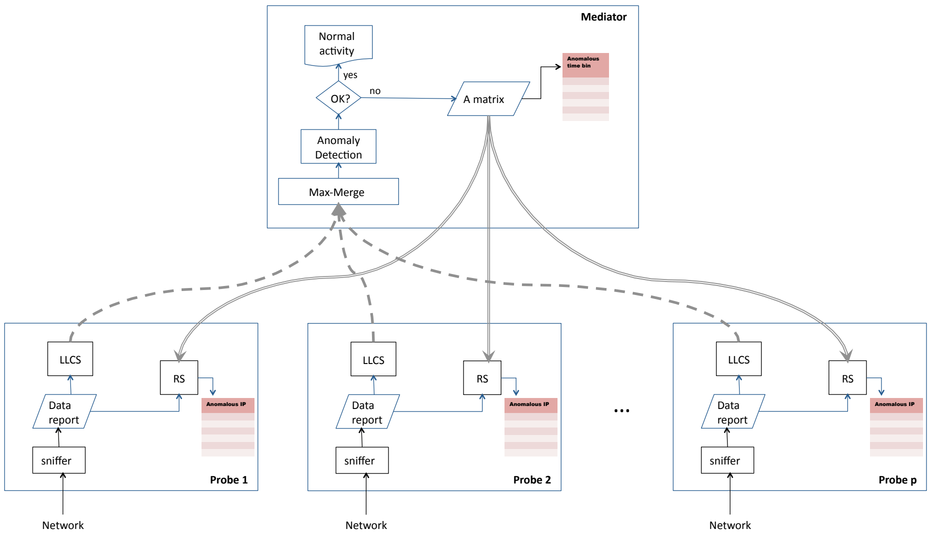


Figure 5.16: System Architecture

LogLog counting algorithm in such sketch table. To this aim, each bucket of the sketch table is associated with another hash function H_{dw} that gives output in the interval $(1, \ldots, L)$, associated with the layer (depth) of the array. In more detail, let us see how the different operations necessary for the LLCS construction are performed. First of all, as far as the the hash functions are concerned, we have chosen to use functions belonging to the 4-universal hash family[1] [227], obtained as:

$$h(x) = \sum_{i=0}^{3} a_i \cdot x^i \ mod\ p\ mod\ W \tag{5.7}$$

where the coefficients a_i are randomly chosen in the set $(0, 1, \ldots, p-1)$ and p is a random prime number (we have considered the Mersenne numbers).

LLCS is update by using *IP mangling* and *modular hashing* techniques, as described in Section 5.4.1.2.

Thus, by applying such techniques, the system "selects" a bucket $S[d][w][\cdot]$ and the associated hash function H_{dw}. Note that, given the structure of the LLCS, each

[1]A class of hash functions $H : (1, \ldots, N) \rightarrow (1, \ldots, W)$ is a *k-universal hash* if for any distinct $x_0, \cdots x_{k-1} \in (1, \ldots, N)$ and any possible $v_0, \cdots v_{k-1} \in (1, \ldots, W)$:

$$Pr_{h \in H} = \{h(x_i) = v_i; \forall i \in (1, \ldots, k)\} = \frac{1}{W^k}$$

bucket contains a LogLog counter.

For all the flows that collide in a given bucket the system computes the flow ID, that is a function of the header fields IP source and destination addresses, source and destination ports, and protocol. Then it computes the hash function H_{dw} of the flow ID and update the LogLog counter according to the algorithm described in Section 5.4.1.1.

Moreover, in parallel to the described LLCS, each probe also constructs a reversible sketch (RS) [226], which will be eventually used in the identification phase. Note that, for making the identification possible, the RS and the LLCS must be realized using the same hash functions.

Once the LLCS has been constructed, it is passed to the mediator that performs a merging of the LLCSs received by the different probes. To allow this operation, all the different probes must share the same hash functions.

It is worth noticing that the idea of building two distinct data structures, instead of a unique reversible version of the LLCS is mainly due to performance considerations and privacy concerns. Indeed, transmitting a reversible version of the LLCS from the probes to the mediator would imply the transmission of the sketch tables together with all the list of keys associated to the single buckets. This, on one hand, would worsen the performance in terms of transmission time and bandwidth usage, and, on the other hand, would imply the probes to disclose "potentially" private information (the keys) to the mediators.

5.4.2.3 Detection Phase

The mediator, responsible for the detection phase, combines the information exported by each probe (related to the same time bin) and constructs a merged LLCS, $M[d][w][l]$.

In more detail, given the different LLCSs ($S^p[d][w][l];\ p = 0, \ldots, P$), constructed by each probe p, in a given time-bin, the mediator computes $M[d][w][l]$ by applying a "max-merge" algorithm:

$$M[d][w][l] = \max_p S^p[d][w][l] \tag{5.8}$$

This technique implicitly solves the problem of not counting duplicated flows (that is the flows observed by more than one single probe). Indeed, given the nature of the LogLog counting algorithm, the resulting aggregated sketch is exactly equivalent to the one that would be constructed in an "ideal" case, in which the mediator would be able to directly observe all the traffic. This also implies that the estimation error, due to the probabilistic nature of the used data structure, is not worsened by the distributed framework, being equivalent to that of a single counter, studied in [222].

At this point the mediator counts the number of distinct flows that collide in the same bucket. This task can be solved by applying Eq. 5.3 to each bucket $M[d][w][\cdot]$.

At this stage, by using a classical detection method (e.g. PCA, wavelet analysis, heavy hitter) the system is able to decide if there are or not anomalous aggregates in a given time bin. The output of this phase is a binary matrix ($A[d][w]$) that contains a "1" if the corresponding bucket is considered anomalous "0", otherwise.

Note that, given the nature of the sketches, each traffic flow is part of several random aggregates (namely D aggregates), corresponding to the D different hash functions. This means that, in practice, any flow will be checked D times to verify if it presents any anomaly (this is done because an anomalous flow could be masked in a given traffic aggregate, while being detectable in another one).

Due to this fact, a voting algorithm is applied to the matrix A. The algorithm simply verifies if at least H rows of A contain at least a bucket set to "1" (H is a tunable parameter). If so the mediator reveals an anomaly, otherwise the matrix A is discarded.

5.4.2.4 Identification phase

In case the mediator reveals some anomalous time bin during the detection phase, it back-propagates the matrix $A[d][w]$ to the probes.

At this point each probe uses the RS, computed for the anomalous time bin, for identifying the IP addresses responsible for the anomalies (see [226] for the algorithm details).

5.4.3 Experimental results

The proposed system has been tested using a publicly available data-set, composed of traffic traces collected in the Abilene/Internet2 Network [228], that is a hybrid optical and packet network used by the U.S. research and education community.

The used traces consist of the traffic measured in one week on nine distinct routers. The traces of each router are organized into 2016 files, each one containing data about five minutes of traffic (Netflow data). To be noted that the last 11 bits of the IP addresses are anonymized for privacy reasons; nevertheless we have more than 220000 distinct IP addresses.

To test the effectiveness of the proposed architecture we have considered three distinct case:

- **case 1**: the traces have been analyzed router by router and the obtained results have been combined all together. This case corresponds to the application of a centralized approach, where each probe analyzes the traffic independently of the others.

- **case 2**: the traces have been processed by the single probes, responsible for the construction of the LLCSs, then combined together by the mediator. This case corresponds to our proposed architecture.

- **case 3**: the traces have been manually pre-processed so as to combine them all together by eliminating the duplicate flows. This case represents an "ideal" scenario (not applicable to a real network), where the mediator is able to observe all the traffic in the network and has been introduced as a performance benchmark. To be noted that, in this case, the only (statistically bounded) error is due to the use of probabilistic structures.

	Case 1	**Case 2**	**Case 3**
# HHs	6107	767	767
# HHs also detected in Case 1	-	765	765
# HHs also detected in Case 2	765	-	767
# HHs also detected in Case 3	765	767	-

Table 5.1: Experimental Results

Since our architecture is quite general, we have applied a simple algorithm for the estimation of the Heavy Hitters (HHs) in the number of flows, instead of a complete anomaly detection algorithm, so as not to bias the results with the performance of the chosen algorithm. It is also important to highlight that the aim of such experimental test is to show the differences among the three presented cases, in terms of number of detected HHs, duplicated HHs, and so on. For this reason, the identification part, realized applying the algorithm presented in [226], has not been considered, as essentially out of the scope of this work.

It is worth noticing that the aim of these tests would be to show that not using a distributed approach (Case 1) leads to detect a huge number of duplicated HHs (those observed by more than a single probe) together with a certain number of HHs, that are, in fact, not significant in the network considered as a whole. In addition, the other main target is also to show that our proposed architecture (Case 2) is able to almost achieve the ideal situation of Case 3.

Regarding the parameters of the system, in this work, we have used as hash key, for the modular hashing, the IP destination address and we partitioned the 32-bit IP addresses into $q = 4$ words of 8 bits. The outputs of the different hash functions are 3-bit long and are concatenated to form a 12-bit long final hash. The resulting sketch is a table of dimension $D = 16$ rows and $W = 2^{12} = 4096$ columns.

As far as the LogLog counting is concerned, the flow ID is given by the simple concatenation of IP source and destination addresses, source and destination ports, and protocol, while $L = 8$. To be noted that this results in a memory occupancy of about 20MB that is a reasonable small amount of memory if considering to process all the traffic going through a backbone network.

Table 5.1 reports the results obtained in the experimental tests. As expected, in Case 1 the system reveals a huge number of HHs (6107): 765 of these are also revealed in the second and third case, while the remaining 5342 correspond to 3570 repeated detection plus 1772 additive HHs that do not correspond to "real" HHs (they represent a significant portion of the traffic traversing a given probe, but not of the whole network traffic).

Regarding Case 2 and 3, we can easily observe that they present exactly the same behavior, detecting the same 767 HHs, demonstrating that the proposed architecture (Case 2) is able to achieve the same performance of the "ideal" case (Case 3).

Conclusions

This thesis has addressed several technical challenges in the broad field of network monitoring and measurements. A set of novel tomographic algorithms have been developed, for inferring both the network topology and the congestion state of the links along a network path. Several novel algorithms and data structures have been proposed, in order to speed up packet processing thanks to an optimal exploitation of the cache hierarchy. The issues involved in designing hybrid systems, made up of both task specific and general purpose components, have been addressed and several novel architectures have been proposed. Also, several novel solutions for leveraging the last–generation commodity platforms in order to speed up network monitoring software have been illustrated. Finally, distributed systems for building an overlay of coordinated monitoring probes have been designed.

References

[1] A. Pasztor and D. Veitch, “The packet size dependence of packet pair like methods,” 2002. [Online]. Available: citeseer.ist.psu.edu/pasztor02packet.html

[2] A. Johnsson, B. Melander, and M. Björkman, “Modeling of packet interactions in dispersion-based network probing schemes,” Tech. Rep., April 2004. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=0706

[3] R. M. Castro, M. J. Coates, and R. D. Nowak, “Likelihood based hierarchical clustering.” [Online]. Available: citeseer.ist.psu.edu/castro04likelihood.html

[4] M. Coates, R. Castro, R. Nowak, M. Gadhiok, R. King, and Y. Tsang, “Maximum likelihood network topology identification from edge-based unicast measurements,” in *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 2002, pp. 11–20.

[5] H. Meng Fu Shi, “Topology discovery on unicast networks: a hierarchical approach based on end-to-end measurements.” [Online]. Available: citeseer.ist.psu.edu/castro04likelihood.html

[6] M. Coates, M. Rabbat, and R. Nowak, “Merging logical topologies using end-to-end measurements,” in *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM Press, 2003, pp. 192–203.

[7] “The ns manual,” 2007. [Online]. Available: http://www.isi.edu/nsnam/ns/doc/

[8] V. Jacobson, “Congestion avoidance and control,” in *ACM SIGCOMM '88*, Stanford, CA, Aug. 1988, pp. 314–329. [Online]. Available: citeseer.ist.psu.edu/jacobson88congestion.html

[9] S. Keshav, “A control-theoretic approach to flow control,” *Proceedings of the conference on Communications architecture & protocols*, pp. 3–15, 1993. [Online]. Available: citeseer.ist.psu.edu/keshav91controltheoretic.html

[10] R. Carter and M. Crovella, “Measuring bottleneck link speed in packet-switched networks,” Boston, MA, USA, Tech. Rep., 1996.

[11] C. Dovrolis, P. Ramanathan, and D. Moore, “What do packet dispersion techniques measure?” in *INFOCOM*, 2001, pp. 905–914. [Online]. Available: citeseer.ist.psu.edu/479183.html

[12] R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi, “Capprobe: a simple and accurate capacity estimation technique,” in *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM Press, 2004, pp. 67–78.

[13] K. Lai and M. Baker, “Nettimer: A tool for measuring bottleneck link bandwidth,” pp. 123–134. [Online]. Available: citeseer.ist.psu.edu/lai01nettimer.html

[14] http://www.cs.ucla.edu/NRL/CapProbe/.

[15] N. Bonelli, S. Giordano, G. Procissi, and R. Secchi, “Brute: A high performance and extensibile traffic generator,” in *Int'l Symposium on Performance of Telecommunication Systems* , 2005.

[16] A. Di Pietro, D. Ficara, S. Giordano, F. Oppedisano, and G. Procissi, “Noise reduction techniques for network topology discovery.” in *Proc. of PIMRC 07*, 2007.

[17] A. Di Pietro, D. Ficara, S. Giordano, F. Oppedisano and G. Procissi, “Pingpair: a lightweight tool for measurement noise free path capacity estimation,” in *Proc. of ICC 08*, 2008.

[18] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, pp. 47–97, January 2002.

[19] D. Ficara, F. Paolucci, L. Valcarenghi, F. Cugini, P. Castoldi, and S. Giordano, “The beacon number problem in a fully distributed topology discovery service,” in *Proc. of GLOBECOM 07*.

[20] A. Medina, A. Lakhina, I. Matta, and J. Byers, “Brite: Universal topology generation from a user”s perspective,” Boston University, Boston, MA, USA, Tech. Rep., 2001.

[21] B. M. Waxman, “Routing of multipoint connections,” *IEEE Journal on Selected Areas in Communications*, vol. 6, no. 9, pp. 1617–1622, December 1988.

[22] P. N. Klein, T. B. Sebastian, and B. B. Kimia, “Shape matching using edit-distance: an implementation,” in *Proc. the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2001.

[23] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[24] P. Zhu and R. Wilson, "A study of graph spectra for comparing graphs," 2005.

[25] M. Rabbat, R. D. Nowak, and M. Coates, "Multiple source, multiple destination network tomography." in *INFOCOM*, 2004.

[26] M. Gunes and K. Sarac, "Analytical ip alias resolution," *Communications, 2006. ICC '06. IEEE International Conference on*, vol. 1, pp. 459–464, June 2006.

[27] R. Govindan and H. Tangmunarunkit, "Heuristics for internet map discovery," in *IEEE INFOCOM 2000*. Tel Aviv, Israel: IEEE, March 2000, pp. 1371–1380.

[28] Y. Breitbart, M. Garofalakis, B. Jai, C. Martin, R. Rastogi, and A. Silberschatz, "Topology discovery in heterogeneous ip networks: the netinventory system," *IEEE/ACM Trans. Netw.*, vol. 12, pp. 401–414, 2004.

[29] A. Shaikh, M. Goyal, A. Greenberg, R. Rajan, and K. K. Ramakrishnan, "An ospf topology server: design and evaluation," *IEEE Journal on Selected Areas in Communications*, 2002.

[30] T. Bu, N. Duffield, F. L. Presti, and D. Towsley, "Network tomography on general topologies," in *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 2002, pp. 21–30.

[31] D. Ficara, F. Paolucci, L. Valcarenghi, F. Cugini, P. Castoldi, and S. Giordano, "The beacon number problem in a fully distributed topology discovery service," in *Proc. of GLOBECOM 07*, Washington DC, USA, 2007, pp. 2591–2596.

[32] R. Govindan and H. Tangmunarunkit, "Heuristics for internet map discovery," in *IEEE INFOCOM 2000*, Tel Aviv, Israel, March.

[33] B. H. r, D. Plummer, D. Moore, , and k. Claffy, "Topology discovery by active probing," in *SAINT-W '02: Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops*. Washington, DC, USA: IEEE Computer Society, 2002, p. 90.

[34] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "Network topology generators: Degreebased vs. structural," in *ACM SIGCOMM 2002*, 2002.

[35] M.-F. Shih and A. Hero, "Unicast-based inference of network link delay distributions using mixed finite mixture models," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, vol. 2, may 2002, pp. II–1305 –II–1308.

[36] ——, "Unicast inference of network link delay distributions from edge measurements," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, vol. 6, 2001, pp. 3421 –3424 vol.6.

[37] F. L. Presti, N. G. Duffield, J. Horowitz, and D. Towsley, "Multicast-based inference of network-internal delay distributions," *IEEE/ACM Trans. Netw.*, vol. 10, pp. 761–775, December 2002. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.805026

[38] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, vol. 39, no. 1, pp. 1–38, 1977.

[39] N. G. Duffield and F. L. Presti, "Network tomography from measured end-to-end delay covariance," *IEEE/ACM Trans. Netw.*, vol. 12, pp. 978–992, December 2004. [Online]. Available: http://dx.doi.org/10.1109/TNET.2004.838612

[40] Y. Tsang, M. Yildiz, P. Barford, and R. Nowak, "Network radar: tomography from round trip time measurements," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, ser. IMC '04. New York, NY, USA: ACM, 2004, pp. 175–180. [Online]. Available: http://doi.acm.org/10.1145/1028788.1028809

[41] "http://www.caida.org/tools/utilities/others/pathchar/."

[42] G. V. I. C. G. Simon, P. Hga, "A flexible tomography approach for queueing delay distribution inference," in *Proceedings of Internet Performance, Simulation, Monitoring and Measurement (IPS-MoMe 2005), p39-48, 14-15 March 2005, Warsaw, Poland*, 2005.

[43] M. Coates and R. Nowak, "Network tomography for internal delay estimation," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, vol. 6, 2001, pp. 3409 –3412 vol.6.

[44] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with conte xt," in *Proc. of CCS '03*. ACM, pp. 262–271.

[45] *Snort: Lightweight Intrusion Detection for Networks, http://www.snort.org/.*

[46] *Bro: A system for Detecting Network Intruders in Real Time, http://bro-ids.org/.*

[47] W. Eatherton and J. Williams, *An encoded version of reg-ex database from cisco systems provided for research purposes*.

[48] S. Kumar, S. Dharmapurikar, F. Yu, P. C. y, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for dee p packet inspection," in *Proc. of SIGCOMM '06*. ACM, pp. 339–350.

[49] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. of ANCS '07*. ACM, pp. 155–164.

[50] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. of CoNEXT '07*. ACM, 2007, pp. 1–12.

[51] R. Smith, C. Estan, and S. Jha, "Xfas: Fast and compact signature matching," University of Wisconsin, Madison, Tech. Rep., August 2007.

[52] ——, "Xfa: Faster signature matching with extended automata," in *IEEE Symposium on Security and Privacy*, May 2008.

[53] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection withextended finite automata," in *SIGCOMM '08*.

[54] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[55] B. Commentz-Walter, "A string matching algorithm fast on the average," in *Proc. of ICALP '79*. Springer-Verlag, pp. 118–132.

[56] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," University of Arizona, Tech. Rep. TR-94-17.

[57] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. of INFOCOM 2004*, pp. 333–340.

[58] W. Eatherton, Z. Dittia, and G. Varghese, "Tree bitmap: Hardware/software ip lookups with incremental updates," 2004. [Online]. Available: citeseer.ist.psu.edu/dittia02tree.html

[59] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. of ANCS '06*, pp. 93–102.

[60] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[61] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. of ANCS '07*, 2007, pp. 145–154.

[62] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging." in *Proc. of INFOCOM 2007*, May 2007.

[63] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. of ISCA'06*, June 2006.

[64] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. of ANCS '06*. ACM, pp. 81–92.

[65] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers Inc., 2004.

[66] *Intel Network Processors, www.intel.com/design/network/products/npfamily/.*

[67] *Michela Becchi, regex tool, http://regex.wustl.edu/.*

[68] J. E. Hopcroft and J. D. Ullman, *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman, 1990.

[69] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *ANCS '08*, 2008.

[70] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[71] G. Khanna, I. Laguna, F. Arshad, and S. Bagchi, "Stateful detection in high throughput distributed systems," in *IEEE SRDS 2007*.

[72] E. Ketcha Ngassam, D. G. Kourie, and B. W. Watson, "On implementation and performance of table-driven DFA-based string processors," in *Proceedings of the Prague Stringology Conference '06*, 2006.

[73] E. K. Ngassam, B. W. Watson, and D. G. Kourie, "Hardcoding finite state automata processing," in *SAICSIT '03*, 2003.

[74] K. Ngassam, "Towards cache optimization in finite automata implementations," Ph.D. dissertation, University of Pretoria, South Africa, 2007.

[75] *http://www.circlemud.org/ jelson/software/tcpflow/*.

[76] *http://www.intel.com/design/network/products/ npfamily/ixp2800.htm*.

[77] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[78] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, 2005. [Online]. Available: http://www.internetmathematics.org/volumes/1/4/Broder.pdf

[79] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 254–265, 1998.

[80] D. Ficara, S. Giordano, G. Procissi, and F. io Vitucci, "Multilayer compressed counting bloom filters," in *Proceedings of the 27th Conference on Computer Communications, INFOCOM '08*, 2008.

[81] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Counting bloom filters for pattern matching and anti-evasion at the wire speed," *Network, IEEE*, vol. 23, no. 1, pp. 30–35, January-February 2009.

[82] M. Mitzenmacher, "Compressed bloom filters," in *PODC '01: Proc. of the twentieth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2001, pp. 144–150.

[83] A. Kirsch and M. Mitzenmacher, "Distance-sensitive bloom filters," in *ALENEX '06: Proc. of Algorithm Engineering and Experiments*, 2006.

[84] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic bloom filters," in *Proc. of INFOCOM 2006. 25th IEEE International Conference on Computer Communications.*, vol. 1, 2006.

[85] A. Kumar, J. J. Xu, L. Li, and J. Wang, "Space-code bloom filter for efficient traffic flow measurement," in *Proc. of IMC '03*. New York, NY, USA: ACM Press, 2003, pp. 167–172.

[86] S. Cohen and Y. Matias, "Spectral bloom filters," in *SIGMOD '03: Proc. of the 2003 ACM SIGMOD international conferenceon Management of data*. New York, NY, USA: ACM Press, 2003, pp. 241–252.

[87] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J. L. Larriba-Pey, "Dynamic count filters," *SIGMOD Rec.*, vol. 35, no. 1, 2006.

[88] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *LNCS 4168, 14th Annual European Symposium on Algorithms*, 2006, pp. 684–695.

[89] N. Hua, H. Zhao, B. Lin, and J. Xu, "Rank-indexed hashing: A compact construction of bloom filters and variants," in *Proc. of ICNP'08*. IEEE, 2008, pp. 73–82.

[90] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: an aid to networkprocessing," in *Proc. of SIGCOMM '05*. New York, NY, USA: ACM, 2005, pp. 181–192.

[91] *http://www.intel.com/design/network/products/npfamily/ixp2350.htm*.

[92] *Intel® IXP2800 Hardware reference manual*.

[93] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *Proceedings of ACM SIGCOMM'05*, 2005.

[94] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "Hexa: Compact data structures for faster packer processing," in *Proc. of ICNP 07*, 2007.

[95] D. Ficara, S. Giordano, S. Kumar, and B. Lynch, "Divide and discriminate: Algorithm for fast and deterministich hash lookups," in *ANCS '09: Proc. of the ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2009.

[96] B. Chazelle, J. Kilian, R. Rubinfeld, A. T. al, and O. Boy, "The bloomier filter: An efficient data structure for static support lookup tables," in *Proc. of the Fifteenth Annual ACM-SIAM Symposium on Discrete Al gorithms (SODA)*, 2004, pp. 30–39.

[97] P. Hall, "On representatives of subsets," *J. London Math. Soc.*, vol. 10, pp. 26–30, 1936.

[98] R. Motwani, "Average-case analysis of algorithms for matchings and related problems," *Journal of the ACM*, vol. 41, pp. 1329–1356, 1994.

[99] J. Hopcroft and R. Karp, "An n5/2 algorithm for maximum matchings in bipartite graphs," *SIAM Journal on Computing*, vol. 2, 1973.

[100] M. Mitchell, "An introduction to genetic algorithms," 1996.

[101] D. Ficara, "Accelerating traffic classification and measurements on network processors," Ph.D. dissertation, University of Pisa, 2010.

[102] *http://protocols.netlab.uky.edu/ esp/pktgen/*.

[103] R. Bolla, R. Bruschi, M. Canini, and M. Repetto, *A High Performance IP Traffic Generation Tool Based On The Intel IXP2400 Network Processor*, ser. Distributed Cooperative Laboratories: Networking, Instrumentation, and Measurements. Springer Berlin Heidelberg, 2006, pp. 127–142.

[104] *http://netgroup-serv.iet.unipi.it/brute/*.

[105] A. Abdo, H. Awad, S. Paredes, and T. J. Hall, "Oc-48 configurable ip traffic generator with dwdm capability," in *Proc. of the Canadian Conference on Electrical and Computer Engineering*, May 2006, pp. 1842 – 1845.

[106] *Intel® IXP2400/2800 Developer's Tool reference manual*.

[107] *Intel Corporation, 21555 Non-Transparent PCI-to-PCI Bridge User's manual*.

[108] *Linux Device Drivers, Third Edition, http://lwn.net/Kernel/LDD3/*.

[109] *http://www.spirent.com*.

[110] K. Park and W. Willinger, *Self-Similar Network Traffic: An Overview*. Wiley Interscience, 1999.

[111] "tcpdump/libpcap." [Online]. Available: http://www.tcpdump.org/

[112] "Wireshark protocol analyzer (was ethereal)." [Online]. Available: http://www.wireshark.org

[113] "Ntop network traffic probe." [Online]. Available: http://www.ntop.org

[114] P. Wood, "libpcap-mmap." [Online]. Available: http://public.lanl.gov/cpw

[115] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997. [Online]. Available: citeseer.ist.psu.edu/article/mogul95eliminating.html

[116] L. Deri, "Improving passive packet capture:beyond device polling." [Online]. Available: citeseer.ist.psu.edu/695645.html

[117] L. Deri, “Passively monitoring networks at gigabit speeds using commodity hardware and open source software,” in *Proceedings of PAM*, 2003.

[118] *A Distributed Architecture for IP Traffic Analysis*, 2007.

[119] *Real-Time IP Flow Measurement Tool with Scalable Architecture*, 2004.

[120] *Distributed Flow Monitoring Tool Using Network Processor*. Washington, DC, USA: IEEE Computer Society, 2007.

[121] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, “An active splitter architecture for intrusion detection and prevention,” *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 1, p. 31, 2006.

[122] T. Wolf, R. Ramaswamy, S. Bunga, and N. Yang, “An architecture for distributed real-time passive network measurement,” in *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 335–344.

[123] D. Ficara, S. Giordano, F. Rossi, and F. Vitucci, “Refine: the reconfigurable packet filtering on network processors,” *International Journal of Communication Systems*, 2008.

[124] *http://www.mvista.com/*.

[125] *Design of a Multi-Dimensional Packet Classifier for Network Processors*, 2006.

[126] *http://www.spirentcom.com/analysis/technology.cfm?az-c=pl&media=7&ws=325&ss=101*.

[127] R. Gusella, “Characterizing the variability of arrival processes with indexes of dispersion,” vol. 9, no. 2, pp. 203–211, Feb. 1991.

[128] H. Heffes and D. Lucantoni, “A markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance,” vol. 4, no. 6, pp. 856–868, Sep. 1986.

[129] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and k claffy, “The architecture of coralreef: An internet traffic monitoring software suite,” in *In PAM*, 2001.

[130] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, “An architecture for large scale internet measurement,” *IEEE Communication Magazine*, vol. 36, no. 8, pp. 48–54, 1998.

[131] CISCO, “Parallel express forwarding in the cisco 10000 edge service router.” [Online]. Available: http://whitepapers.zdnet.co.uk/0,1000000651,260007268p-39000421q,00.htm

[132] Cisco, “Cisco systems netflow services export version 9,” 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3954.txt

[133] "Ip flow information export (ipfix)." [Online]. Available: http://datatracker.ietf.org/wg/ipfix/charter/

[134] "Endace." [Online]. Available: www.endace.com.

[135] "Palo alto networks." [Online]. Available: www.paloaltonetworks.com

[136] G. Bianchi, E. Boschi, F. Gaudino, E. A. Koutsoloukas, G. V. Lioudakis, S. Rao, F. Ricciato, C. Schmoll, and F. Strohmeier, "Privacy-preserving network monitoring: Challenges and solutions," in *17th ICT Mobile & Wireless Communications Summit 2008*, 2008.

[137] FP7-PRISM, "Deliverable d3.1.1 : State of the art on data protection algorithms for monitoring systems," Tech. Rep., 2008. [Online]. Available: http://fp7-prism.eu/images/upload/Deliverables/fp7-prism-wp3.1-d3.1.1-final.pdf

[138] A. Hintz, "Fingerprinting websites using traffic analysis," in *Workshop on Privacy Enhancing Technologies*, 2002.

[139] G. Bissias, M. Liberatore, D. Jensen, and B. Levine, "Privacy vulnerabilities in encrypted http streams," in *Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, G. Danezis and D. Martin, Eds. Springer Berlin / Heidelberg, 2006, vol. 3856, pp. 1–11.

[140] W. Yurcik, C. Woolam, G. Hellings, L. Khan, and B. Thuraisingham, "Privacy/analysis tradeoffs in sharing anonymized packet traces: Single-field case," in *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 237–244.

[141] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, pp. 29–38, 2004.

[142] "Fp7 prism." [Online]. Available: http://fp7-prism.eu/

[143] FP7-PRISM, "Deliverable d2.2.2: Detailed system architecture specification," Tech. Rep., 2010. [Online]. Available: http://telscom.ch/wp-content/uploads/Prism/FP7-PRISM-WP2.2-D2.2.2.pdf.

[144] G. V. Lioudakis, F. Gogoulos, A. Antonakopoulou, A. S. Mousas, I. S. Venieris, and D. I. Kaklamani, "An access control approach for privacy-preserving passive network monitoring," in *Proc. Int. Conf. for Internet Technology and Secured Transactions ICITST 2009*, 2009, pp. 1–8.

[145] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Counting bloom filters for pattern matching and anti- evasion at the wire speed," *Netwrk. Mag. of Global Internetwkg.*, vol. 23, no. 1, pp. 30–35, 2009.

[146] G. Bianchi, E. Boschi, S. Teofili, and B. Trammell, "Measurement data reduction through variation rate metering," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[147] C. Sun, C. Hu, Y. Tang, and B. Liu, “More accurate and fast syn flood detection,” in *Proc. 18th Internatonal Conf. Computer Communications and Networks ICCCN 2009*, 2009, pp. 1–6.

[148] *The Open Flow Switch Consortium, www.openflow.org*.

[149] *NetFPGA Official Web Site, www.netfpga.org*.

[150] M. Casado, M. Freeman, J. Pettit, N. McKeown, and S. Shenker, “Ethane: Taking control of the enterprise,” *Sigcomm Computer Communication Review*, 2007.

[151] J. Naous, D. Erickson, A. Covington, G. A. ler, and N. McKeown, “Implementing an openflow switch on the netfpga platform,” in *ACM ANCS*, 2008.

[152] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, “A clean slate 4d approach to network control and management,” *Sigcomm Computer Communication Review*, 2005.

[153] P. Molinero-Fernandez and N. McKeown, “Tcp switching: exposing circuits to ip,” in *IEEE Micro*, 2002.

[154] E. Kohler, R. Morris, B. Chen, J. Jahnotti, and M. F. Kasshoek, “The click modular router,” *ACM Transaction on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[155] G. Antichi, A. D. Pietro, D. Ficara, S. o Giordano, G. Procissi, and F. Vitucci, “On the use of compressed dfas for packet classification on netfpga,” in *IEEE CAMAD*, 2010.

[156] G. Antichi, A. D. Pietro, D. Ficara, S. G. iordano, G. Procissi, and F. Vitucci, “A prefix-distribution adaptive scheme for routing lookup acceleration,” in *Proc. of GLOBECOM ’09*. IEEE, 2007, pp. 1–12.

[157] S. Han, K. Jang, K. Park, and S. Moon, “Packetshader: a gpu-accelerated software router,” in *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 195–206. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851207

[158] L. Rizzo, “http://info.iet.unipi.it/ luigi/netmap/.”

[159] “http://www.ntop.org/products/pf_ring/dna/,” 2011.

[160] “http://www.ntop.org/pf_ring/building-a-10-gbit-traffic-generator-using-pf_ring-and-ostinato/,” 2011.

[161] *http://caia.swin.edu.au/genius/tools/kute/*.

[162] *http://rude.sourceforge.net/*.

[163] A. Botta, A. Dainotti, and A. Pescape, “Multi-protocol and multi-platform traffic generation and measurement,” in *Proc. of INFOCOM 2007 DEMO Session*, May 2007.

[164] S. Avallone, A. Pescape, and G. Ventre, "Analysis and experimentation of internet traffic generator," in *Proc. of New2an 2004, International Conference on Next Generation Teletraffic and Wired/Wireless Advanced Networking*, February 2004.

[165] "http://www.grid.unina.it/software/itg/."

[166] "http://code.google.com/p/ostinato/."

[167] "http://code.google.com/p/ostinato/issues/detail?id=39," 2011.

[168] "http://lxr.linux.no/linux+v3.1.6/net/core/pktgen.c."

[169] A. Botta, A. Dainotti, and A. Pescapé, "Do you trust your software-based traffic generator?" *Comm. Mag.*, vol. 48, pp. 158–165, September 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1866991.1867012

[170] M. Paredes-Farrera, M. Fleury, and M. Ghanbari, "Precision and accuracy of network traffic generators for packet-by-packet traffic analysis," in *2nd International IEEE/Create-Net Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 2006.

[171] T. N. R. Group, "Pfq," "http://netgroup.iet.unipi.it/software/pfq/", July 2011.

[172] L. Deri, "ncap: wire-speed packet capture and transmission," in *End-to-End Monitoring Techniques and Services on 2005*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 47–55. [Online]. Available: http://portal.acm.org/citation.cfm?id=1251986.1253236

[173]

[174] l. M. m. o. l. Phil Woods, "http://public.lanl.gov/cpw/."

[175] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *IMC 2010*, 2010, pp. 218–224.

[176] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou, "Forwarding path architectures for multicore software routers," in *Proc. of PRESTO '10*. New York, NY, USA: ACM, 2010, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/1921151.1921155

[177] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "Flexible high performance traffic generation on commodity multi-core platforms," in *To appear in Traffic Monitoring and Analysis (TMA 2012) Workshop*, 2012.

[178] N.Bonelli, A. D. Pietro, S. Giordano, and G. Procissi, "Packet capturing on parallel architectures," in *IEEE workshop on Measurements and Networking*, 2011.

[179] G. Iannaccone, "Fast prototyping of network data mining applications," in *Passive and Active Measurement Conference 2006*, Adelaide, Australia, mar 2006.

[180] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, "Towards high performance virtual routers on commodity hardware," in *Proceedings of ACM CoNEXT 2008*, Madrid, Spain, December 2008.

[181] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of USENIX SOSP 2009*, Big Sky, MT, USA, October 2009.

[182] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: towards programmable network measurement," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 97–108, 2007.

[183] F. Fusco, F. Huici, L. Deri, S. Niccolini, and T. Ewald, "Enabling high-speed and extensible real-time communications monitoring," in *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 343–350.

[184] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, "Comparing and improving current packet capturing solutions based on commodity hardware," in *Proceedings of the 10th annual conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 206–217. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879168

[185] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of the 10th annual conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 218–224. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879169

[186] Endace, "Endace DAG 9.2X2," http://www.endace.com/dag-9.2x2-packet-capture-card.html, Auckland, New Zealand, mar. 2010.

[187] Napatech, "Napatech NT20E2," http://www.napatech.com/products/capture_adapters/2x10g_pcie__nt20e2.html, Soeborg, Denmark, apr. 2010.

[188] C.-W. Chang, A. Gerber, B. Lin, S. Sen, and O. Spatscheck, "Network DVR: A programmable framework for application-aware trace collection," in *Proceedings of the Passive and Active Measurement Conference (PAM) 2010*, Zürich, Switzerland, mar 2010.

[189] Invea-Tech, "COMBOv2 FPGA Boards," http://www.invea-tech.com/products-and-services/combo-fpga-boards, Brno, Czech Republic, 2009.

[190] B. Trammell and E. Boschi, "An introduction to ip flow information export," *IEEE Communications Magazine*, vol. 49, no. 4, Apr. 2011.

[191] C. Inacio and B. Trammell, "YAF: Yet Another Flowmeter," in *Proceedings of the 24th USENIX Large Installation System Administration Conference (LISA '10)*, San Jose, California, nov 2010, pp. 107–118.

[192] S. McCanne and V. Jacobson, “The BSD packet filter: a new architecture for user-level packet capture,” in *Proceedings of the USENIX Winter 1993 Conference*. San Diego, California: USENIX Association, 1993.

[193] 1024cores, “1024cores,” ”http://www.1024cores.net/”, July 2011.

[194] S. Han, K. Jang, K. Park, and S. Moon, “Packetshader: a gpu-accelerated software router,” in *Proceedings of ACM SIGCOMM 2010*, New Delhi, India, September 2010.

[195] ISO/IEC JTC1/SC22/WG21, “Working Draft, Standard for Programming Language C++,” http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, February 2011.

[196] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[197] Intel, “Receive side scaling on Intel Network Adapters,” ”http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm”, July 2011.

[198] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang, “Sift: A simple algorithm for tracking elephant flows, and taking advantage of power laws,” in *43rd Allerton Conference on Communication, Control and Computing*, 2005.

[199] G. Gu, R. Perdisci, J. Zhang, and W. Lee, “Botminer: clustering analysis of network traffic for protocol- and str ucture-independent botnet detection,” in *SS'08: Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 139–154.

[200] W. T. Strayer, D. Lapsley, R. Walsh, and C. Livadas, “Botnet detection based on network behavior,” in *Botnet Detection: Countering the Largest Security Threat*, W. Lee, C. Wang, and D. Dagon, Eds. Springer-Verlag, 2007.

[201] Cisco Systems, “Cisco Visual Networking Index: Forecast and Methodology,” ”http://www.cisco.com”, June 2011.

[202] Y.-S. Wu, S. Bagchi, S. Garg, N. Singh, and T. Tsai, “Scidive: A stateful and cross protocol intrusion detection architecture for voice-over-ip environments,” in *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004, p. 433.

[203] B. Barry and A. Chan, “Towards intelligent cross protocol intrusion detection in the next generation networks based on protocol anomaly detection,” in *The 9th International Conference on Advanced Communication Technology*, 2007, pp. 1505–1510.

[204] Y.-S. Wu, V. Apte, S. Bagchi, S. Garg, and N. Singh, “Intrusion detection in voice over ip environments,” *International Journal of Information Security*. [Online]. Available: http://dx.doi.org/10.1007/s10207-008-0071-0

[205] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," in *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2004, pp. 379–390.

[206] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 149–160.

[207] R. Zhang, X. Wang, X. Yang, and X. Jiang, "Billing attacks on sip-based voip systems," in *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–8.

[208] I. Baumgart, B. Heep, and S. Krause, "OverSim: A Flexible Overlay Network Simulation Framework," in *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, May 2007, pp. 79–84.

[209] "Fastbit: An efficient compressed bitmap index technology." [Online]. Available: http://sdm.lbl.gov/fastbit/

[210] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen, "CSAMP: a system for network-wide flow monitoring," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 233–246.

[211] M. Sharma and J. Byers, "Scalable Coordination Techniques for Distributed Network Monitoring," *Passive and Active Measurement Conference*, 2005.

[212] G. R. Cantieni, G. Iannaccone, C. Barakat, C. Diot, and P. Thiran, "Reformulating the monitor placement problem: Optimal network-wide sampling," in *In Proc. of CoNeXT*, 2006.

[213] R. Serral-Gracia, P. Barlet-Ros, and J. Domingo-Pascual, "Distributed sampling for on-line sla assessment," in *Local and Metropolitan Area Networks, 2008. LANMAN 2008. 16th IEEE Workshop on*, Sept. 2008, pp. 55–60.

[214] B. Reynolds and D. Ghosal, "Secure ip telephony using multi-layered protection," in *In Proc. of NDSS*, 2003.

[215] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *In Internet Measurement Workshop*, 2002, pp. 71–82.

[216] J. D. Brutlag, "Aberrant behavior detection in time series for network monitoring," in *Proceedings of the 14th USENIX conference on System administration*. Berkeley, CA, USA: USENIX Association, 2000, pp. 139–146. [Online]. Available: http://portal.acm.org/citation.cfm?id=1045502.1045530

[217] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *In ACM SIGCOMM*, 2004, pp. 219–230.

[218] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan, "Network anomography," in *In IMC*, 2005.

[219] M. Thottan and C. Ji, "Anomaly detection in ip network," in *IEEE Trans. Signal Processing*, vol. 51, 2003, pp. 2191–2204.

[220] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Computers Security*, vol. 29, no. 1, pp. 124–140, 2010.

[221] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, pp. 182–209, September 1985. [Online]. Available: http://portal.acm.org/citation.cfm?id=5212.5215

[222] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *In ESA*, 2003, pp. 605–617.

[223] G. Cormode and S. Muthukrishnan, "Holistic udafs at streaming speeds," in *In SIGMOD*, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.2257

[224] ——, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58 – 75, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/B6WH3-4BM8Y1G-1/2/71b7980bb85b570bc57ee73f8afcd62f

[225] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM Press, 2003, pp. 234–247. [Online]. Available: http://dx.doi.org/10.1145/948205.948236

[226] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible sketches for efficient and accurate change detection over network data streams," in *Proceedings of the ACM SIGCOMM conference on Internet Measurement*, ser. IMC '04. New York, NY, USA: ACM, 2004, pp. 207–212. [Online]. Available: http://doi.acm.org/10.1145/1028788.1028814

[227] M. Thorup and Y. Zhang, "Tabulation based 4-universal hashing with applications to second moment estimation," in *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms (SODA)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004, pp. 615–624.

[228] "The Internet2 Network," http://www.internet2.edu/network/.