Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2016

João Pedro
Martins de Carvalho

Sistema de Inspeção Superficial de Pacotes de Alto Desempenho para Identificação de Tráfego

High Performance Shallow Packet Inspection System for Traffic Identification

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2016

João Pedro
Martins de Carvalho

Sistema de Inspeção Superficial de Pacotes de Alto Desempenho para Identificação de Tráfego

High Performance Shallow Packet Inspection System for Traffic Identification

"There is nothing noble in being superior to your fellow man; true nobility is being superior to your former self..."

— Ernest Hemingway

João Pedro Martins de Carvalho

Sistema de Inspeção Superficial de Pacotes de Alto Desempenho para Identificação de Tráfego

High Performance Shallow Packet Inspection System for Traffic Identification

Dissertação apresentada na Universidade de Aveiro para cumprimento dos requisitos necessários á obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Professor Doutor Paulo Jorge Salvador Serra Ferreira, Professor Auxiliar do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro.

o júri / the jury

presidente / president

Professor Doutor Rui Luís Andrade Aguiar
Professor Catedrático da Universidade de Aveiro

vogais / examiners committee

Professor Doutor Pedro Miguel Alves Brandão
Professor Auxiliar da Universidade do Porto - Faculdade de Ciências (arguente)

Professor Doutor Paulo Jorge Salvador Serra Ferreira
Professor Auxiliar da Universidade de Aveiro (orientador)

Dedicatória /agradecimentos

Dedico este trabalho aos meus pais, Afonso e Lucinia, aos quais muito agradeço, e sem os quais não estaria a culminar este passo que, bem sei, os enche de orgulho. Dedico-o também à Isabel pelo seu incansável companheirismo e apoio durante fases criticas do ano. Não posso também esquecer a minha avó Ester e por extensão, toda a família e amigos.

Em primeiro lugar, gostaria de agradecer ao meu orientador. O Professor Doutor Paulo Salvador, pela disponibilidade demonstrada, pelas boas discussões na busca da melhor solução, assim como pela cedência do código da sua biblioteca de "wavelets" a qual permitiu, efectuar algumas experiências, que resultaram no enriquecimento do trabalho. Quero, de seguida, agradecer a todos os colegas e, essencialmente amigos que fui criando ao longo deste percurso, pelas noites mal dormidas e pela partilha de "dores" inerentes ao dia-a-dia.
Por fim, gostaria de agradecer aos meus amigos que estiveram mais próximos de mim nestes últimos meses, casos como o do Bernando, David e Manuel, assim como a todos os meus colegas da "Team IT" pela boa disposição e partilhas técnicas, sempre importantes.

Palavras chave

Captura de tráfego, Classificação de tráfego, Inspeção de Pacotes, alto rendimento

Resumo

A evolução e crescimento da *Internet* tem levado a uma crescente preocupaçao tendo em vista a alocação dinâmica de recursos em redes de grande dimensão, assim como uma adopção sem precedente de políticas de segurança baseadas em classificação de tráfego. Este fenómeno desencadeou a criação de mecanismos de inspeção profunda de pacotes onde se assiste a um acesso transversal, que assenta na obtenção de sequências de *bytes* específicas, presentes no *Payload* de cada pacote, o que levanta uma série de limitações técnicas, éticas e potencialmente legais. Com a crescente necessidade de desenvolvimento de mecanismos de inspeção menos invasivos e mais eficientes em termos de velocidade e potencialmente gestão de memória, a comunidade científica começou a trabalhar em outros tipos de abordagem ao problema.

Nesta dissertação, propomos um sistema de classificação de fluxos de tráfego que assenta em *Shallow packet inspection*. Tendo em conta as ultimas previsões e dados estatísticos atuais, que estimam que cerca de 90% de todo tráfego na *Internet*, seja do tipo vídeo nos próximos anos, decidimos dedicar especial atenção sobre esse tipo específico. Para isso, procedemos à recolha de informação não sensível, com a qual efetuamos um estudo estatístico baseado em estatísticas de baixo nível. Os resultados obtidos nesse estudo, foram analisados de um ponto de vista comportamental, por forma a alcançar uma prova de conceito na extração de regras coerentes que permitam diferenciar tipos de tráfego independentes. Por fim, estudamos, concebemos e testamos um paradigma de organização de fluxos de forma eficiente.

O sistema foi testado e avaliado recorrendo a testes de inundação por pacotes, seguidos da medição e avaliação dos resultados em termos de tempo de processamento, assim como, ao uso de memória principal.

Keywords Packet Capture, Traffic Classification, Packet inspection, High Performance

Abstract

The evolution and growth of the Internet has led to a growing preoccupation regarding dynamic allocation of resources in large networks, as well as to an unprecedented growing adoption of security policies based on traffic classification. This phenomenon triggered the creation of deep inspection mechanisms for packets where we can see a cross-access that is based on the retrieval of specific strings present in packet's Payload. This event raises a number of technical, ethical, and potentially legal limitations. With the increasing need to develop less invasive and more efficient inspection mechanisms, in terms of processing speed and potentially, memory management, the scientific community began working in other types of approaches to solve the problem.

In this dissertation, we propose a traffic flow classification system based on Shallow packet inspection. Given the latest forecasts and current statistical data, which estimates that about 90 % of all traffic will be video in the next few years, we have decided to devote special attention to this specific type. For this, we proceeded to collect non-sensitive information, with which we perform a statistical study based on low-level statistics. The results obtained from this study were analysed from a behavioural point of view, in order to reach the extraction of coherent rules that allow the differentiation of independent types of traffic. Finally, we studied, conceived and test an efficient flow organisation paradigm.

The system has been tested and evaluated using packet flood tests. Following to the measurement and examination of results in terms of processing times as well as the use of main memory.

Contents

List of Figures

List of Tables

Listings

Glossary

API Application Programming Interface. 7

BPF Berkeley Packet Filtering. 10

BST Binary Search Tree. 31

COS Class-of-Service. 16

DAG Data Acquisition and Generation. 9

DMA Dynamic Memory Allocation. 28

DPDK Data Plane Development Kit. 7

DPI Deep Packet Inspection. 13

ECPA Electronic Communications Privacy Act. 2

FD File Descriptor. 22

FPGA Field programmable gate array. 18

FT Fourier Transform. 51

FTP File Transfer Protocol. 14

HTTP Hypertext Transfer Protocol. 14

IANA Internet Assigned Numbers Authority. 14

ISP Internet Service Provider. 1

KNN k-Nearest Neighbors. 18

LDA Linear Discriminant Analysis. 17

LSH Locality Sensitive Hashing. 18

NIC Network Interface Card. 7

NN Nearest Neighbors. 17

OS Operating System. 9

OSN Online Social Network. 1

QoS Quality-of-Service. 1

SPI Shallow Packet Inspection. 3, 14

SSH Secure Shell. 14

VOIP Voice over IP. 1, 14

WAND Network Research Group. 8

WT Wavelet Transform. 51

XML EXtensible Markup Language. 44

Chapter 1

Introduction

The Internet, as we know it, has been changing throughout the years as we witness a constant growth, to becoming the most widespread communication grid worldwide, today it has evolved to the most powerful platform for accessing and spreading of information and services. Actually, nowadays people can watch videos on-line, share an unlimited number of contents, access Voice over IP (VOIP) services, watch television broadcasts, and many many more. However, networks nowadays are increasingly being used for many areas of business, such as, finance, research, military services, and e-commerce. Considering this fact, many corporations rely on time critical applications where a connectivity problem or a security breach can cost huge amounts of money. Otherwise, good network solutions, represent as we tend to see, an increasing amount of revenues each year. With this in mind, despite the great revolution taking place at the moment in terms of great features and platforms, all combined increases the number of vulnerabilities and security breaches, that need to be addressed as soon as possible. As expected, many studies were, and are still being conducted addressing these problems [15] [16] [17].

As the Internet keeps on growing, it gets accessible to an increasingly number of clients that demand satisfaction, by making their money, worth for each service that they contract. Basically, when talking about customer satisfaction, the problem of granting Quality-of-Service (QoS) arises in a critical fashion. Considering the issue, it is of vital importance to look at the problem from a large scale point of view. Therefore, to manage a huge network, such as, the one from an Internet Service Provider (ISP) or from a big corporation, one should understand the applications and the traffic they generate. As we know, Online Social Network (OSN), work with the integration of a wide variety of applications and services that are implementing new ways to communicate. In a study conducted a few years back by Nazir et al. [18] the authors tried to measure the impact of OSN applications as they concluded that those had significant impacts in terms of latency, resulting in direct consequences for the user experience. Being aware of this situation, made even worse by the explosion in number of users, we can understand that mapping traffic to their origin application is vital in huge networks to provide and balance the network's resources for each case, or even to get information in terms of time scale. Considering the last remark, it is particularly important for the creation of user profiles to infer future network loads and dynamically assign resources, preventing saturation or waste.

In conclusion, we can deduce that security will definitely benefit from an accurate mapping, in the sense that flows generating suspicious signatures can be more easily tracked and taken

down to protect users and platforms, or even, to prevent monetary losses that can reach tens of millions of dollars [19]. Addressing the classification problem, several works have been developed using different kinds of methodologies or approaches, always trying to adapt to the evolution of the Internet. Most approaches are based in deep inspection of packet contents and or statistical analysis of traffic patterns [20]. Classification techniques, can be divided into two main branches of investigation as we will see in next chapter. Those being, real time and non-real time approaches. In the first case, the classifier tries to cope with the demands, extracting information and reaching conclusions at the pace that events occur, it is definitely faster, however more error prone. On the other hand, non-real time relies on capturing and later processing scheme, perhaps with more accuracy but most of the times slower.

One problem commonly associated with this field of research, a part from the technical problems, that are going to be discussed afterwards, is that, behind most techniques, there are other kind of challenges that arise from deep inspection of packets, such as, ethical flaws, legality concerns, along with potential surveillance overlap, provoked by promiscuity between private sectors and governmental power. Looking carefully at the situation, these kind of techniques definitely changed Internet governance in a radical manner. In a study conducted by Mueller and Asghari [21], they discussed the problem of deep inspection usage by ISP for Bit Torrent traffic throttling in Canada and America. In the end, they have reached a clear conclusion, that there was a "causal relationship" between deep inspection and disruptive Internet regulation, giving ISP more power than they probably should have. This topic is of vital importance, as it changes the way people live and interact with each other, blurring the boundaries of what is private and should not be tracked, and what might be of, for instance, state security importance [22]. As we might expect, sophisticated network management, (in most cases packet inspection awareness) is common process these days. However, the legal consequences in terms of the liabilities, whether civil or criminal, of the Service Provider in connection with the type of management used have been a latent problem. A very famous example of this fact, has taken place in 2007, when Embarq authorized a third party (NebuAd) to collect certain information from their network relating to the websites visited by some customers in order to allow NebuAd to target these customers with specific advertisements. The information was collected using an Ultra Transparent Device installed on Embarq's network allowing the customers information to be sent to NebuAd servers. The information provided, enabled NebuAd to identify users based on an assigned number, but not to identify the customers themselves. After the discovery, one of these customers filed a law suit in federal court arguing that this procedure violated the Electronic Communications Privacy Act (ECPA). In the end, Embarq won the law suit claiming that the access was only made to data to which it has access in the ordinary course of business providing Internet service to its clients, fading way from ECPA regulation infringement but opening, door to future use of other types of information, including private [23].

1.1 Motivation

Traffic classification has been a target field of research for quite a long time, with several works being developed throughout the years. Deep packet inspection based techniques suffer from some disadvantages, such as the introduction of great bottleneck on capture, reassemble and analysis of traffic flows, with the increase of detailed analysis representing more impact on the system. Thus, to keep up with the demands, ISP need to invest huge amounts of money to mitigate performance degradation.

With that in mind, the urge for novel methods of packet capture, efficient packet handling, along with the need of new approaches for flow classification to cope with, network's complexity, and legal limitations have been pointed out in other works [11]. Another interesting fact that has been studied is the recent paradigm change in terms of the type of traffic flowing in modern networks, and it can be divided into two main branches. The first is the continuous growth in usage of encryption technologies as depicted in Figures, 1.2, 1.3, 1.4, is, indeed, a case against deep inspection of packets. The second is the reported huge increase in video content which is expected to reach 90% of the total traffic, in the next few years [24] [25]. This results in network stress, and nowadays, peaks have already been registered on around 60% of the total traffic in North America, with the advent of pay per view services, such as Netflix, as can be seen in Figure 1.1. Thus, the motivation for this work was the urge and the increasingly necessity of an answer to the limitations of DPI for traffic analysis and classification, exploring other types of approach such as, Shallow Packet Inspection (SPI). Which, relies on the observation of application's behaviour. On this work, we have put special focus on video traffic, without the need of disruptive packet analysis, along with efficiency and wide range of benchmarking testing.

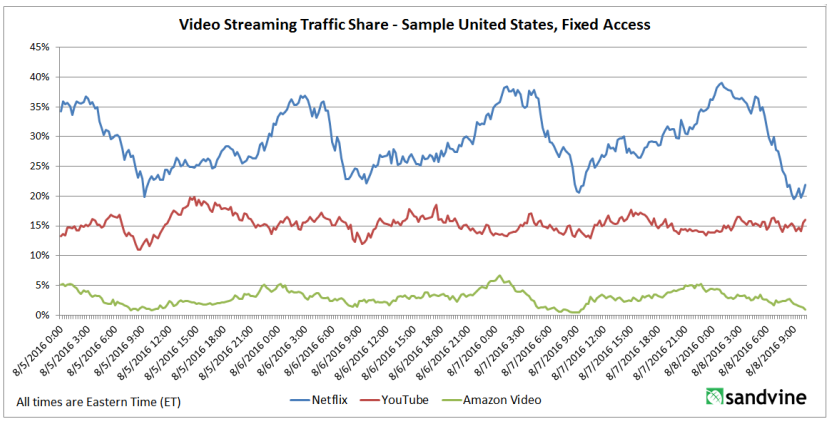


Figure 1.1: North America video services percentage, extracted from [1]

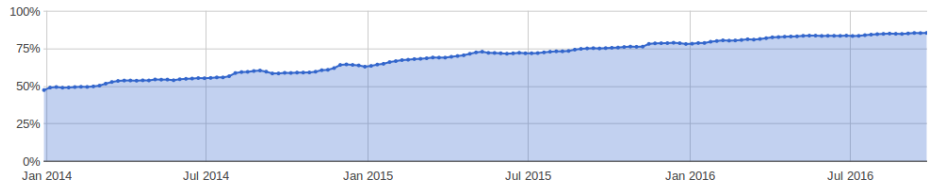


Figure 1.2: Evolution in the adoption of encrypted connections in Google taken from [2]

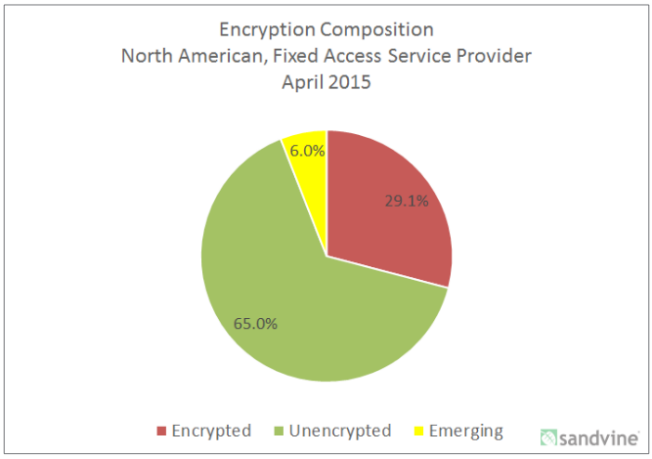


Figure 1.3: Information of North America encrypted traffic in 2015 provided by [1]

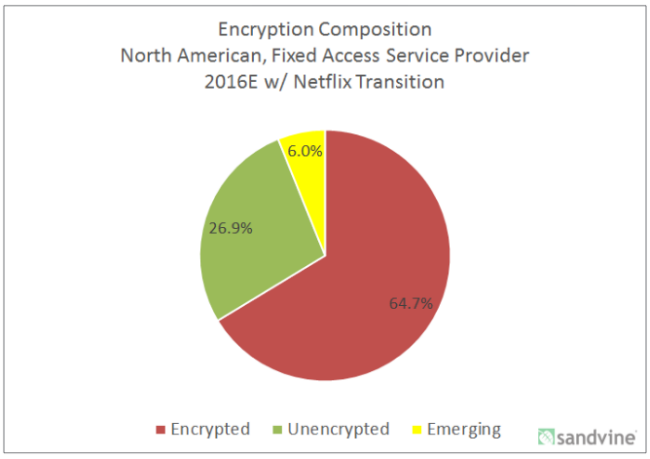


Figure 1.4: Information of North America encrypted traffic in 2016 provided by [1]

1.2 Objectives

In such a diverse area of studies, there are a wide range of challenges to tackle. Therefore, in this dissertation we want to address the following range of objectives. The first one is the investigation of a state of the art technologies, regarding packet capturing and data structures frameworks/libraries, along with a bibliographic revision on classification methodologies, including their different types. The second, will be the implementation of a high speed and a controlled memory system, for efficient packet capturing, and handling. Lastly, we want to study flow dynamics exploring simple indicators on packet flows, such as: Packet counts, packet lengths and inter-arrival sampling. We expect, to explore low level statistics, along with, clustering algorithms to group flows and also apply wavelet analysis. Using the information extracted from those statistics, we aim to propose rules for basic traffic classification with special focus on video.

1.3 Dissertation's Structure

This dissertation is is structured as follows:

In Chapter 2 we are going to make a revision of the state of the art, presenting the most relevant work in this field, and consequently for this dissertation. It will comprise the revision on: Packet capturing libraries, Frameworks for flow accommodation and organisation; Approaches to packet inspection, and finally classification approaches.

Chapter 3 will present the implemented solution with exhaustive description of all its components and engineering options followed during the development. After this, we will give room to a performance evaluation, measuring the performance of the packet handling.

Chapter 4 will present the results obtained on the classification algorithm, finishing with other performance evaluation on classification time per flow. In the final chapter 5 we will reflect and evaluate the outcome of this dissertation's work, along with the analysis of the actual limitations, finishing with final remarks over the future work that might be developed.

Chapter 2

State of the art

This Chapter, aims to give an overview on the work that has been made so far on the topics addressed by this dissertation. Taking that into consideration, we will present the state of the art on, data organization libraries, packet capturing frameworks and libraries, packet inspection approaches and techniques. Finally we will end up with traffic classification techniques.

2.1 Packet sniffing

Packet sniffing as defined by Gandhi et. al. [26] is a technique which passively or actively captures packets travelling through a network. Typically, depending on the Network Interface Card (NIC), where it is installed, it will perform network monitoring and packet analysis. In this section, we have proceeded to packet capturing frameworks and libraries preliminary investigations. On this field, developers are always striving for efficiency, flexibility and standardization of the Application Programming Interface (API). Taking these premises as a starting point we have found a series of different tools and approaches to packet sniffing and network monitoring.

2.1.1 Libraries and Frameworks

During the research we have came across, a few libraries and frameworks that have been developed in order to fulfill rather specific needs trying to overcome weaknesses of others.

Data Plane Development Kit (DPDK)
: This framework developed by Intel, has become open source recently, which eventually originated a significantly growth in terms of its community. DPDK comprises four main modules [3] that implement different features at the hardware level to provide fast packet capturing and processing. Besides this, it implements a rich Application Programming Interface (API) featuring device managing abstractions, memory management functions, locking premises, along with other data structures and algorithms.
Even though, it supports a wide range of Network Interface Card (NIC), it relies on a specific installation process, and demands a rather advanced set of configurations in order to get the best usage out of the hardware.
As can be seen, in Figure 2.1, it shows the interactions amongst different modules on high-performance packet processing.

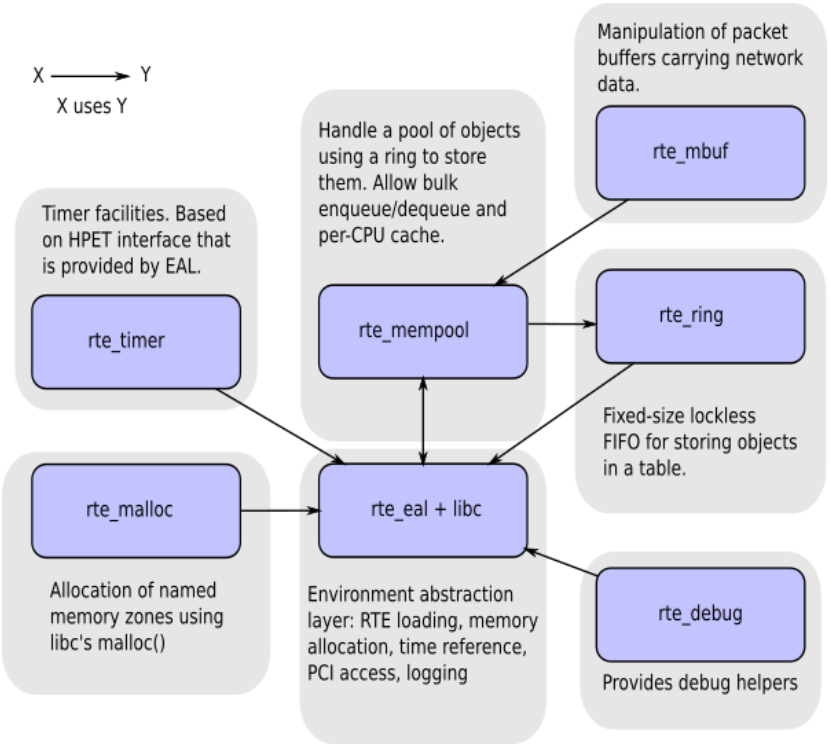


Figure 2.1: Core Components Interactions, extracted from [3]

DPDK has been designed for performance, hence, its architecture denotes a few critical development options that are worth mentioning:

1. Polling Mode
   Polling mode is a very distinctive characteristic, as it permits constant interrogations to the hardware [27] at a small cycle footprint, we can say it implements a busy-wait architecture efficiently.
2. Kernel Bypass
   It permits skipping the kernel network stack processing, which consistently delivers a much better performance, relegating interrupts delays.[27]
3. Multi Core Distribution
   DPDK is also capable of distributing work at processor core level, which might represent a boost in efficiency, considering a producers-consumers approach. [28]

LIBTRACE

Libtrace was developed at Network Research Group (WAND), it aims to provide better performance with an improved API to permit a better programming experience. Recently, the library has been growing in terms of popularity, with a great quantity of projects being developed using it [29] [30].
Amongst its main features we have [31]:

1. API
   Its API has been developed aiming consistency and simpleness, in fact, a programmer writing libtrace program can rely on header and protocol handling by the

library itself. Indeed, as stated, a program can be expressed in 40% fewer lines than libpcap's (one of its main rivals) approach.

2. Versatility
   Libtrace is very versatile in the way it treats formats, as it can support a wide variety, and can also read or write from NIC's and Endace Data Acquisition and Generation (DAG).
3. Compression
   It supports reading and writing compressed formats, namely gzip an bzip2, doing it in an efficient way avoiding pipes to specific applications.
4. Performance
   Apart from a threaded I/O approach, libtrace has used a copy avoidance method to scape from eventual onerous packet copies when unnecessary.

NETMAP

Netmap is a framework that started being developed at University of Pisa, Italy which main purpose is to enable Gbit captures in commodity hardware. The basic novelty behind it is a memory mapping approach 2.2, which allows that a NIC can be forced to send information to the data structures implemented by netmap instead of additionally sending those packets through the Operating System (OS) networking stack. The work behind this feature is done recurring to basic system calls that provide the initializations needed for the communication to occur and are used eventually for simple updates and data validation. Taking that in consideration it allows a significant speed up when comparing to Os networking stack [32].

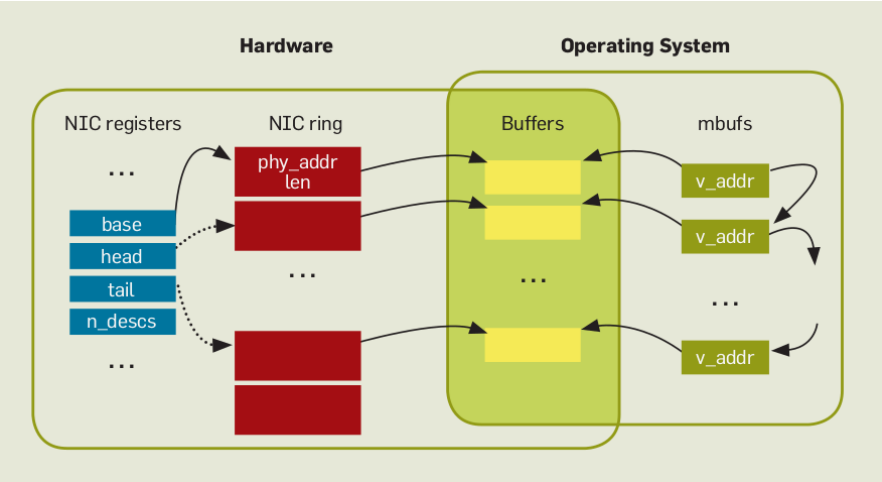


Figure 2.2: Memory mapping in Netmap diagram extracted from [4]

PCAP

Pcap is a very well known, open source library, which was started as a project in Berkeley Labs. It soon got recognition and gained a very supportive community with countless projects using it [33] [34]. It is a piece of standard technology which main objective was to create an horizontal abstraction to every NIC on the market, avoiding individual

packet capturing modules, per application [35].
Considering main features, we can outline four of them [36]:

1. Support
   Libpcap is one of the oldest out there it comes with good documentation and relies on a very wide community support.
2. Versatility
   It possesses the ability to capture from a wide range of devices.
3. Consistency
   As the environment changes the API remains the same, on each OS.
4. Filtering
   It implements Berkeley Packet Filtering (BPF) approach directly at kernel level 2.3 in order to achieve better performance

Libpcap, as said before, constitutes a reference technology even nowadays, with significant works being developed to improve its performance taking advantage of other techniques at kernel level to reduce the packets processing time at that level [36].

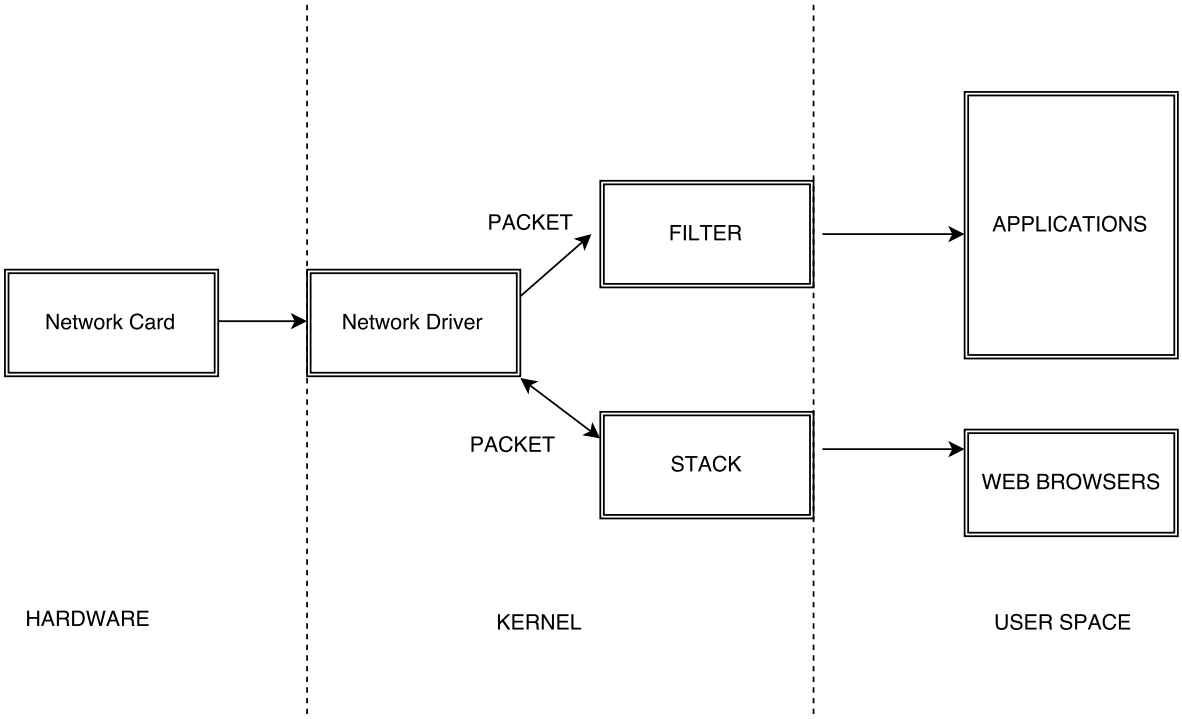


Figure 2.3: Pcap design scheme

Throughout our research, other solutions such as, PF_Ring and Sniffer10G have been investigated, although, we opted to let them out of the comparison because they are very specific solutions and in most cases proprietary software.

2.2 Data Structures Libraries

In this section we are going to describe some existent libraries used for data organisation, which mainly comprise collections of data structures for general use. In this scenario it is of particular importance because of its relevance in the overall performance of the system.

1. Tommyds

   Tommyds is a collection of data structures developed by Andrea Mazzoleni designed for high performance [10]. It comprises a set of structures and algorithms written in C, such as:

   (a) Lists
   The Lists are implemented using a double linked approach, with a special feature that differs from classic implementations, which is the usage of a single pointer to represent the list itself, reducing the memory footprint. Another important feature is the possibility of insertion in the end and at the start of the list. Allowing equal elements keeping its insertion order.

   (b) Arrays
   The Tommy Array implementation is a very interesting implementation based on segments of exponential growing size, this fact allows exponential growing without reallocation. The basic algorithm behind is based on an allocation scheme, where the pre-allocated memory is not reallocated, but rather, a new segment is allocated, preventing heap fragmentation, and resulting in constant element addressing.

   (c) Tries
   A Trie is a binary tree that has keys associated with each of its leaves [9]. Tommyds implements, two types of tries in a slightly different way. First of all, tommyds tries are general in definition, which means that the number of leaves per node is arbitrary. Finally the implementation comes in two flavours, an in place trie, where the objects are not stored in order, and there is no need for an external allocator; And the second flavour a cache optimised trie, where an external allocator is needed and the elements are stored in order.

   (d) Hash Tables
   Concerning Hash tables, in tommyds the implementations come in three flavours. The first one is a fixed size table and respectively a more naive implementation, resulting in a performance degradation after 0.75 of load factor. Another implementation is an hash table with dynamic growing, allowing the resolution of the load factor problem, although it is stated by the developer himself, that this implementation should not be used in real time systems, as it proceeds to very costly re-size operations (100 ms to 1 million entries) and clearly fragments the heap in reallocations. The last implementation appears to solve the limitations evidenced by the other two, it is based on a linear chained hashing algorithm, that we are going to explore later in this document.

2. Uthash
   Uthash is a specific implementation of an hash table that is particularly used to store C structures as keys. It was developed by Troy D. Hanson and is presently supported by Arthur O'Dwyer [37]. This piece of software is not a library but rather a single header file, as it doesn't have any library code to link against. It is limited in terms of features yet supports the basic operations over the data structure. It allows the programmer to test from a wide range of hashing algorithms and is implemented using macros allowing automatic inlining. One of its main disadvantages is the fact that in every entry has to be embedded an explicit handler which forces an overhead of 56 bytes in a 64 bit system.

3. Judy Arrays
   Judy Arrays is a C library developed by Doug Baskins [38] that provides technology that implements a sparse dynamic array. Judy arrays are declared simply by using a null pointer, and consumes memory only when it is populated. Judy's key benefits are scalability, high performance, and memory efficiency. A Judy array is extensible and can scale up, bounded only by machine memory which, in certain scenarios might be a disadvantage. Judy is designed with an unbounded approach, the size of a Judy array is not pre-allocated but grows and shrinks dynamically. It combines scalability with ease of use. Its API is accessed with simple insert, retrieve, and delete operations. Tuning and configuring are not required (in fact not even possible), additionally, sort, search, count and sequential access capabilities are built in as well. Judy can be used whenever dynamically sized arrays, associative arrays or a simple-to-use interface is required. Judy was designed to replace common data structures, such as arrays, sparse arrays, hash tables, B-trees, binary trees, linear lists, skiplists, other sort and search algorithms, and counting functions.

4. Sparsehash
   Google Sparsehash project was released in 2005 and offered two different hash table implementations [39]. The Densehash developed having in mind speed and Sparsehash optimized for space. The Sparsehash implementation, is arguably considered one of the most space efficient hash tables available out there, requiring only two bits of overhead per entry. Given that fact, just saying whether something exists or not costs only a bit, which is a very interesting approach. The sparse implementation being indeed the best solution for space conservation, it can still perform only 3 times slower than most libraries.

2.3 Packet Inspection

2.3.1 Deep Packet Inspection (DPI)

Deep Packet Inspection (DPI) is a well known surveillance technique, widely used, based on a complete and multilevel packet analysis 2.4, it does not only inspects the header but also its payload contents [40].

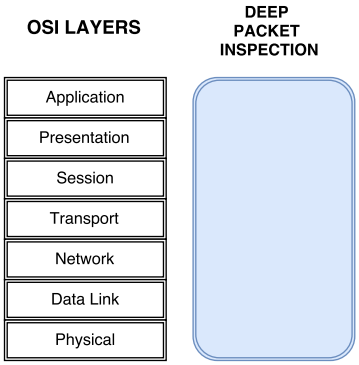


Figure 2.4: DPI multi level inspection

Regarding DPI, several works have been done over the years. Most times, commercial solutions are using this technique to provide accurate predictions. They are characterized and rely on signature, extraction and comparison [41], [42], [43]. Thus they are commercial solutions, literature suggests that its claims cannot be correctly validated because of their black box behaviour, in the sense that, their tests were performed using private datasets and its code is not open source, hence, not available freely to the public [44]. Subsequently, we will make a small introduction to some open source solutions that have been proving to be a valid solution with high hit rates.
In fact, one of those tools was implemented in University of Waikato and it is called libprotoident, it is part of a project written in C to fulfill the requirements for lightweight payload processing. One of the few definitions of lightweight DPI is proposed here [45] as the authors claim to accomplish leveled results using only 4 bytes of packet payload instead of full payload processing. Considering what was just stated, the authors managed to achieve acceptance from ISPs by using traces provided by them, as they agreed, that client's privacy wouldn't be put in danger while accessing such a small portion of the payload. The library uses essentially a payload analysis approach, even though, it uses rules based on port classification and statistics information for accuracy purposes [45].

Another solution, developed by ClearFoundation, is called l7-filter. It constitutes one of the few exceptions found that, even being a commercial solution it is open source code, and officially supported by the company itself. It is used in Linux's Netfilter to identify a wide range of protocols, complementing its functions with IP match approach and port rules [46]. This software is available in 2 versions, one present at kernel space and other at user space. Considering the last one, the basic algorithm works the following way: Firstly it spawns 2 threads, in which, one is responsible to receive updates from the kernel, keeping track of new connections, while the other creates a queue instance responsible by holding the packets for processing, it is worth mentioning that a packet will only be processed if the first thread has received the new connection event by that time, which in UDP flows will lead to the first

packet of each flow being ignored. In fact, l7-filter is very customizable and it can overcome some limitations such as the one mentioned, using extensions [47].

Finally, there is other solution that is often mentioned in literature called OpenDpi, it started being an open source project supported by Ipoque, which was developed alongside with PACE (their commercial version cited before), removing the support for encrypted traffic as well as optimizations. Eventually, the corporation dropped its development and now the project is considered close, originating one interesting fork called ndpi [44]. This technology is particularly interesting because of its extra features aside from DPI pattern matching, it uses behavioural and some sort of statistical analysis.

2.3.2 Shallow Packet Inspection (SPI)

In the urge to answer to the limitations of DPI, the scientific community has stepped up developing a different approach for traffic analysis and classification which is called Shallow Packet Inspection (SPI). Its main novelty is based on the fact that it doesn't rely on full or partial payload inspection but rather on observable or measured features that depend of specific application's behaviour. As an example we may consider, packet's inter arrival time and the variation of packet sizes during communications [48]. There are a few approaches in the literature regarding what we might consider SPI, although, it lacks in dedicated tools using this concept extensively. The techniques used will be discussed further in the following subsection.

2.4 Traffic Classification Techniques

In this section will be discussing on DPI and SPI techniques used nowadays, some of them already mentioned above with further explanation missing.

2.4.1 Port-Based Approaches

Port numbers are distributed in three well defined groups. Those groups were defined by Internet Assigned Numbers Authority (IANA) and the ranges are distributed as such, Well Known Ports span from 0 to 1023, corresponding to reserved ports for special usages; Registered Ports vary from 1024 to 49151; Lastly, Dynamic and/or Private Ports going from 49152 to 65535 [49].

One of the most primitive classification methods would rely on the assumption that specific applications had specific ports to bind, making them much easier to track and map [50]. There are several examples of that, such as, port 80 being bounded to Hypertext Transfer Protocol (HTTP) traffic, port 22 being commonly used for Secure Shell (SSH) and ports 20-21 to File Transfer Protocol (FTP). In fact, during early stages of the Internet some work was developed considering that principle achieving accurate results, one of the most cited works is present here[12].

Regarding this technique, even though it has reached accurate results in the past, nowadays, with the advent of distributed contents, P2P technology and Voice over IP (VOIP), this approach has revealed to be very inefficient and not suited for traffic classification on its own, as most protocols changed to arbitrary port binding, and/or using well known ports of other applications in order to disguise behaviours [50].

In a study conducted by Madhukar and Williamson [51], it has been used a two year trace from the University of Calgary and the conclusion extracted was that a port based approach cannot be used on its own, reliably, since between 30% and 70% of the total trace has come up unclassified. However, the results are very notable, here [12] the researchers defend that the integration of port-based approaches might be helpful in certain scenarios, such as, in conjunction with packet sizes and TCP header flags.

2.4.2 Payload-Based Classification Approaches

This approach to traffic classification is the basis for DPI. As mentioned before, it is considered to be the most reliable technique out there, and it is based on the belief that most protocols transport certain consistent strings inside its payload which, in fact, can allow the possibility to distinguish them clearly. Technically, those strings, as a convention, are called digital signatures.

Considering P2P traffic, it has been regarded in literature as one of the most difficult types of traffic for classification [52]. In one of the first works on the field [53], the researchers have proposed signatures at application level for an efficient identification of P2P traffic. The authors have proceeded to the research analyzing data and packet traces from different P2P clients in order to obtain the application-level signatures, which were then used to develop filters that could efficiently track P2P traffic even at high-speed network links. Researchers have still reached very accurate results, with a very narrow border of false positives and false negatives. Even thought the results were promising, it had a major weakness, as the solution required previous knowledge of every single application in order to come up with the corresponding signatures. This spotted flaw cut the approach short, as it prevented an automatic adaptation to new applications down the road.

On one of the most important legacy works on the subject [13] the researchers addressed several reports that stated significant decrease in P2P file-sharing traffic activity. Their protocol, started by measuring traffic from P2P protocols and, using reverse-engineering, analyzed these protocols for identifying digital signatures in the payload, like the ones shown below 2.1. Classification heuristics were proposed to make sure whether a specific trace was generated by a P2P protocol or not. Also, including a mix approach with the analysis of ports defined as "known P2P ports", in which that case the flow would be tagged as P2P. To confirm their initial supposition, the authors compared each packet's payload against the obtained signatures, which allowed them to determine the exact P2P protocol. In conclusion, their findings contradicted the previous reports claiming a decrease on the volume of P2P traffic.

Protocol	String	Transport Protocol
eDonkey2000	0xe3, 0xc5	A TCP/UDP
Gnutella	"GNUT" / "GIVE" / "GND"	TCP/UDP
MP2P	GO!!, MD5, SIZ0x20	TCP
SSH	"SSH"	TCP
IRC	"USERHOST"	TCP

Table 2.1: List of signatures adapted from [12] and [13]

In a recent work, presented early this year [5] Middlebox services were used as exam-

ple taking in consideration nowadays challenges for deep packet inspection in cloud-based Web applications and the fact that they typically look for known patterns that might appear anywhere in the payload. Subsequently they addressed the problem of most software for this purpose starting to become a serious bottleneck in systems as string pattern matching approaches stagnated while software packet processing technologies have paved the way with more efficiency. During the course of their investigation, they proposed an efficient multi-pattern string matching algorithm, on witch they claim to have achieved a significant reduction in terms of memory accesses and cache misses by using "small and cache-friendly data structures" 2.5, minimizing also, sequential data dependencies. The researchers have delivered impressive results as they claim to have out scored commercial network appliances up to 3.6 times and also, comparing typical Web application firewalls delivering 57-160% improvement in performance.

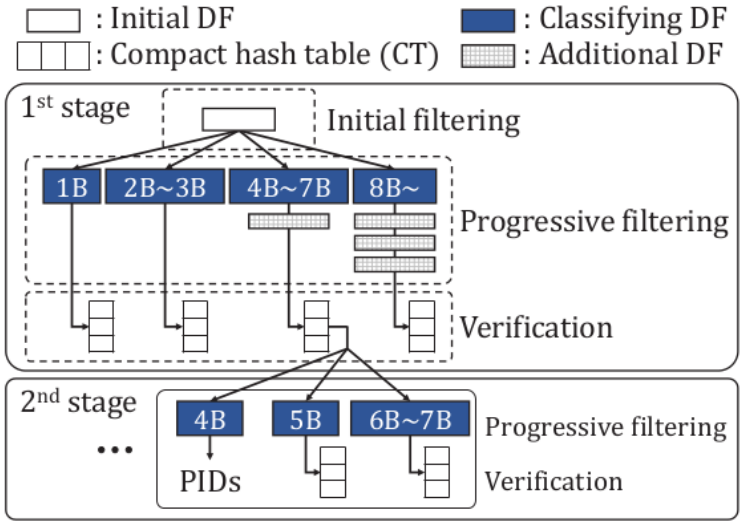


Figure 2.5: Accelerating String Matching approach extrcted from [5]

2.4.3 Statistical Classification Approaches

Studies regarding statistic characteristics of traffic flows, are based on the fact that, typically, most applications produce different behaviours, thus generating distinguishable traffic patterns, which eventually enabled the identification of the underlying protocols. This approach has been taken in high regard in terms of its feasibility and accuracy.

In a revolutionary and widely cited work [54], the authors presented an approach for mapping captured traffic to its Class-of-Service (COS) and described its foundations and associated challenges, creating a solution for measurement based classification for QoS purposes. The researchers stated that the signatures generated were not very useful for application awareness but could rather perform exceptionally well, measuring how applications were actually used: interactively or for bulk-data transport. The work focused on four generic classes:

1. Interactive: This class would comprise all the traffic which would result of user requests and responses, to perform real-time interactions with other system;

2. Bulk data transfer: Traffic used to perform big data volume transfers without any real-time necessity;

3. Streaming: Typically multimedia traffic with pivotal real-time constraints;

4. Transactional: Typical small number of request-response pairs;

Considering those four classes, several features were extracted from the captures at different levels. After that, several characteristics vectors were constructed. Then proceeding to the classification using Linear Discriminant Analysis (LDA) and Nearest Neighbors (NN). The quality and accuracy assurance were attested by large traffic traces.
In a recent interesting work using this classification paradigm, a methodology was presented for the identification of application-level traffic based on the construction of statistic signatures using payload size, transmission order, and direction of the first N packets in the flow and was proposed in [14]. In this work, the authors developed a pipeline to proceed to the signature generation, and then developed other one to do the traffic classification itself. They started by converting the flows into packet size distribution vectors and then dividing the flows by process, after this step, flows were grouped by the distance similarity of the flow vectors generated before. After this process, the groups go under optimization right before the signature is generated. The second pipeline, works as proof of concept, basically it generates flows of packets that are continuously captured. Then those flows are converted to vectors and they are matched against the Ground-truth of signatures previously generated.

Their results were very impressive, depicting precision over 99.9% for the majority of the applications studied as can be seen in table 2.2, with the exceptions of Skype and Kartrider, fact explained by the existence of plenty flows with only one or two packets resulting in erroneous classifications by other application signatures.

Application	Precision
Dropbox	99.99%
uTorrent	99.97%
Nateon	99.99%
Skype	97.90%
Kartrider	97.60%

Table 2.2: Precision by application adapted from [14]

2.5 Real-Time vs Non Real-Time Classification Approaches

Real-Time traffic classification is a task of great importance for many network management decisions and operations: Knowing in real-time the applications behind the generated traffic can be extremely valuable in ISP environments to optimize network tasks, provide constraints direct linked to Qos, prevent network saturation, proceed to adequate resource management, which can lead to significant money saves, as well as detect anomalous behaviours or even attacks. Even thought this is a critical task, achieving such ability is not easy. Summing all the factors including nowadays complexity of networks and applications, along with the actual state of legal restrictions, preventing complete packet analysis. It all contributes to hinder accurate real-time traffic classification.

2.5.1 Real-Time Classification Approaches

In an innovative work published in late 2010 [55], the authors proposed a Field programmable gate array (FPGA) based approach to boost the speed of the identification on multimedia applications, with the compromise of keeping high accuracy. Well known multimedia applications were studied such as, Skype and Instant Messaging. Then using algorithms such as, k-Nearest Neighbors (KNN) and Locality Sensitive Hashing (LSH) they have shown that the approach could be deployed in high bandwith links. Regarding that deploy they have achieved an accuracy level higher than 99%.

Addressing a very important topic on the issue of timely P2P traffic classification, specifically focused on its well renowned client BitTorrent, the following work [56], proposed a machine learning approach for its fast classification. The research group responsible for the work, designed a group of discriminators for better BitTorrent differentiation:

1. Minimum payload : The appearance of small packets, typically, in the range of 5 up to 17 bytes might indicate bittorrent traffic flow;

2. Small Packet Ratio: Since in bittorrent there are two types of messages: peer updates and information exchange. This discriminator is then defined as the ratio of small packets within total number of packets in the flow;

3. Large Packet Ratio: Ratio of large packets within a total number of packets within a flow;

4. Smaller Payload Standard Deviation: Data transfers consist of bi-directional flows. For each direction the standard deviation of the TCP packet size is calculated. The smallest value is used.

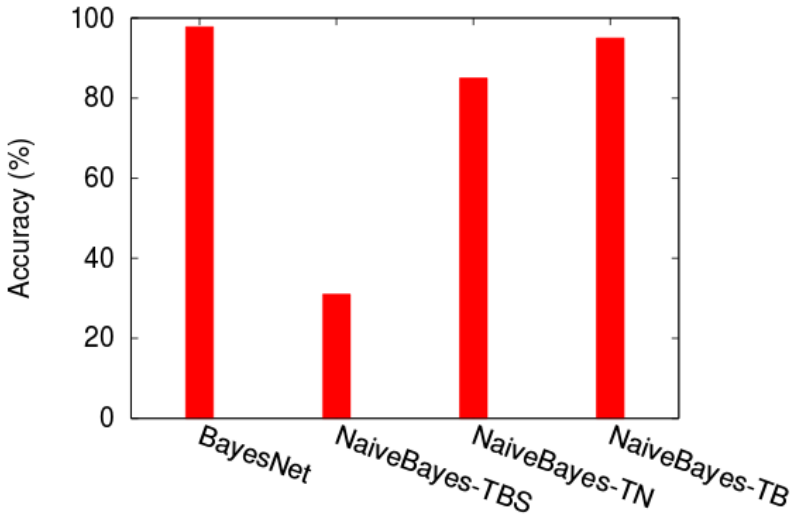


Figure 2.6: Accuracy with different algorithms used from [6]

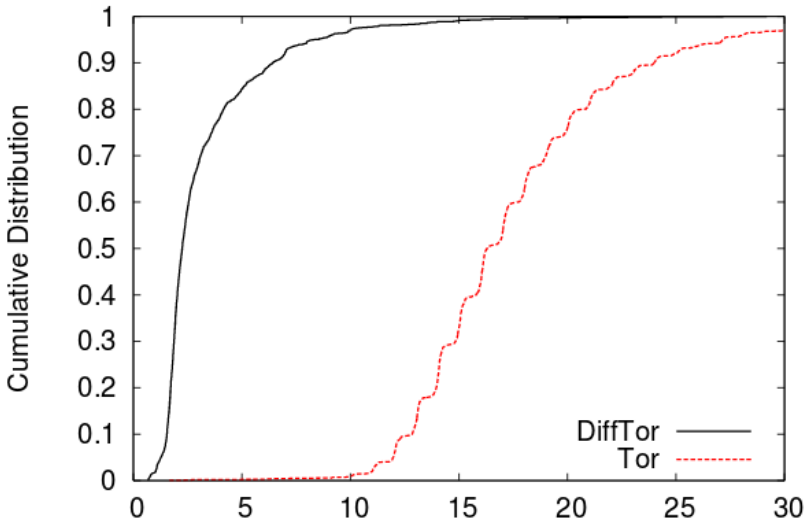


Figure 2.7: DiffTor performance extracted from [6]

In a very interesting work published in early 2012 [6], the authors studied the Tor architecture an anonymity-preserving network, and tried to address one of its main weaknesses, which is related with performance degradation. Their approach was based on real-time traffic classification to deal with congestion and low relay-client ratio. The authors explored the issue with the definition of classes of service. They have decided to follow that path to address the problem related with QoS. Considering that, most of the traffic passing through Tor's network is mainly interactive browsing traffic, a significant volume of bulk download still benefits from the same Qos policy. To overcome this, they proposed a real-time application of a machine-learning-based approach called DiffTor, to classify encrypted traffic within Tor circuits, using that information to assign proper QoS policies to each class. The work has been well succeded. They have defined 3 classes of traffic, Streaming, Bulk and Interactive, with overall good results on some well known machine learning algorithms explored as we can see in image 2.6. As a result, they have shown impressive benchmarks on TOR's performance when compared to the regular implementation as it is depicted in image 2.7.

Chapter 3

Architecture/Implementation

This Chapter will present a detailed description of the architecture of the developed solution. First we will describe the packet capturing library and structure. Moving forward we will be presenting and explaining the information retrieval scheme. Furthermore, we will be describing and exploring the data structures holding the information and why they were chosen. Considering those data structures, we will discuss their usage in terms of performance and physical limits.

3.1 Generic Conceptual Architecture

In this section we aim to give a generic understanding of the conceptual architecture as can be seen in image 3.1 Throughout the following chapter we will break down on each module and try to explain and exemplify its functions.

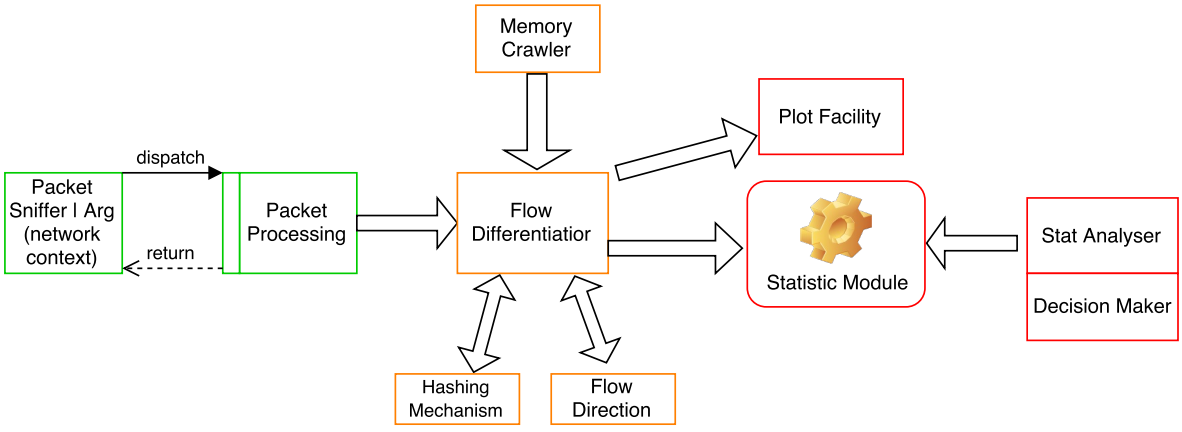


Figure 3.1: Generic Architecture

The first two modules with a green border represent the very well known scheme used by libpcap for packet capturing and processing. The dispatch function is called each time a packet is popped out of the NIC's queue. One of the parameters of the dispatch function is its target function. The Target function is then used for packet processing. Therefore and, as we can see, this process is clearly wrapped in a callback scheme. After, the packet is received, the information needed is extracted and is aggregated within a flow group, responsibility given

to "Flow Differentiator" module. This module is rather large as it comprises all the major operations, and possesses a data structure able to separate flows and respond quickly to basic operations over it, contributing to mitigate a possible bottleneck on data aggregation. The "Hashing Mechanism" is used as an auxiliary tool to the data structure intrinsic operation. When working on flow statistics we use the "Flow Direction" module as a calculation tool to define if each packet belongs to upload or download stream, and "Statistics Module" which works on per flow statistics. "Plot Facility" module works as an auxiliary instrument to visualise the flows behaviour throughout the time as it generates plots from several relevant indicators."Stat Analyser" module work, to process statistic results and is based on the rules defined on "Decision Maker" takes decisions for classification. Finally, as can be seen above, the application's life span is potentially eternal as it keeps on capturing packets indefinitely and autonomously. The main consequence of that behaviour is the need of a custom written "Memory Crawler" module, which eventually, will examine all the memory dynamically reserved and free some memory chunks as they reach a certain time-out of inactivity or the system starts running out of memory.

3.2 Packet Capture

As discussed previously in chapter 2, we have investigated a few libraries for creating an abstraction with the NIC to provide packet capturing facilities. When looking for a library/framework we have thought about a few requirements that it should meet. Considering all the possibilities presented, we have selected libpcap for the job. We based our decision on its simpleness, good documentation, great support by its community, its standard API and, perhaps its main feature, versatility across different compilers and systems. Presenting these kind of advantages the option came up straight, even considering others, perhaps more efficient speed-wise, it turned out to be the best suited option.

The code for this dissertation has been developed as said before, using mainly libpcap library and C programming language. The language has been chosen considering the characteristics where it excels: Its maturity, versatility, and mainly, its high efficiency even across systems and compilers.

Speaking about packet capturing we have to understand the library's organisation scheme, and the general layout of a libpcap sniffer so we can understand how things work. The following steps will give a proper explanation to it:

1. The first step is also the basic parameter of the application, it consists of the specification of the interface where we intend to do the capture on.

2. The second step is the one where we activate pcap by opening a session of capture in the interface we passed before, actually the process is quite similar to the one of opening a file for reading or writing, which means that, underneath, this operation creates a File Descriptor (FD) which can be managed later.

3. The third step is denominated as "compilation", this process is used to make pcap aware of the type of filter we want to apply for our application. Pcap follows BPF approach and considering the scheme presented before in 2.3 our interest was on the selection comprising all the packets from IP protocol version four and six as follows "ip or ip6".

4. Finally, we tell pcap to start its execution loop. In this step pcap will, in our case, behave as an infinite loop and for every packet captured it will send it to the proper callback responsible for its processing.

The process behind packet processing has been studied and tested in different ways for efficiency purpose. As we already know, libpcap organization demands a callback scheme. Therefore one of the main questions lays on the fact that the capture loop gets blocked on its associated callback during the time it consumes for processing the packet contents.

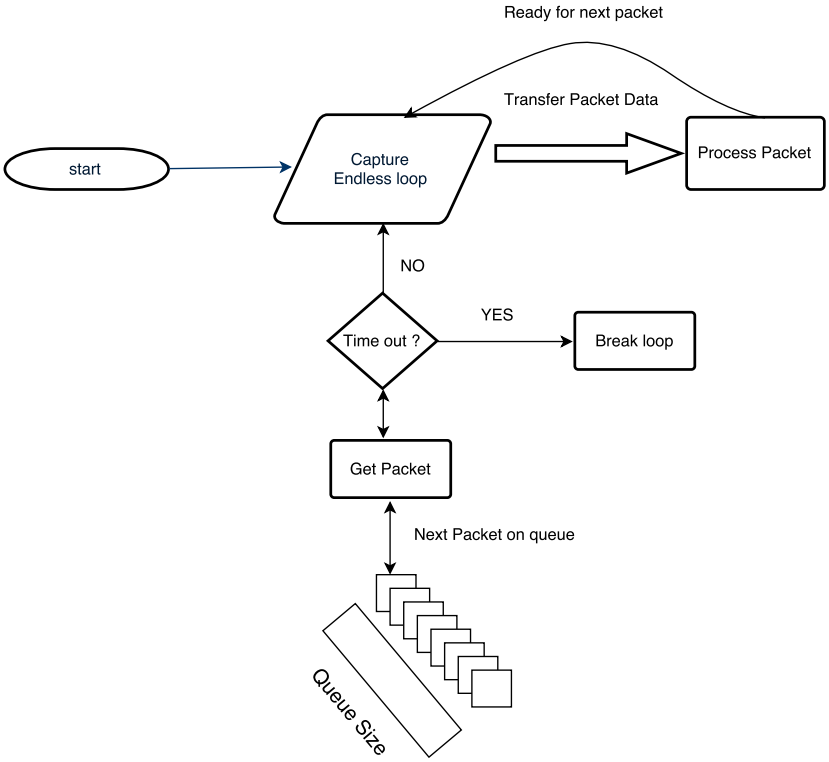


Figure 3.2: Packet Capture scheme

Considering the image above 3.2 we can easily understand why the process might be problematic. Taking in consideration that the NIC's queue size is always a system dependable variable, we surely know that it has its limits. Therefore the anomalous behavior we intend to address is the potential increasing of packet drop rate as the packet rate of incoming packets also increases. This issue is particularly important even in a rather small company with a few machines generating traffic.

Regarding the problem, three possibilities were tested. The first one was the most simple and rather naive approach, we basically delegated all packet processing expenses upon the callback. The results were disappointing but expected, we witnessed a slow behaviour and increasing packet drop during bursts and an acceptable behavior during small packet rates. The second test was based on a uncontrolled addition of parallelism. We defined it uncontrolled, as the approach consisted of launching a new thread for each packet captured. In fact, this time the results were better on low packet rates, which can be justified by the addition of parallelism to the equation, freeing the callback from potential heavy processing,

and delegating the task on the brand new launched thread. However, we resolved to proceed and run two small stress tests, to figure out the application's performance under the generation of a few hundred packets per second and afterwards a few thousand packets per second with random packet size between 100 and 1500 bytes. The results were very disappointing for the two tests, in the first one, we started to witness a slowdown in packet processing and mostly on the overall machine's performance with periods of memory congestion with a high number of simultaneous resource allocations and deallocations, and on top of that, we have to consider constant context commutations. In the second one, we faced a rather dramatic result as the system yielded lack of resources for the number of requests at hand, resulting in the program unexpected end.

At this point we ended up considering two options. The first would be the introduction of controlled parallelism and the second would be a totally different approach using a framework for event processing based on fd's such as one of the most renowned frameworks out there [57]. A very important work [58] discussed whether the usage of event based approach would deliver better performance over the usage of threads for high-concurrency servers. Their conclusions in the end of the study came back in favor of threads, as they considered that most arguments against thread based processing were a matter of specific thread implementations, also considering threads a simpler and a more "natural" programming style.

Considering the above we have decided to take the path of controlled concurrency. To achieve our objective we have decided to use a Thread Pool 3.3.

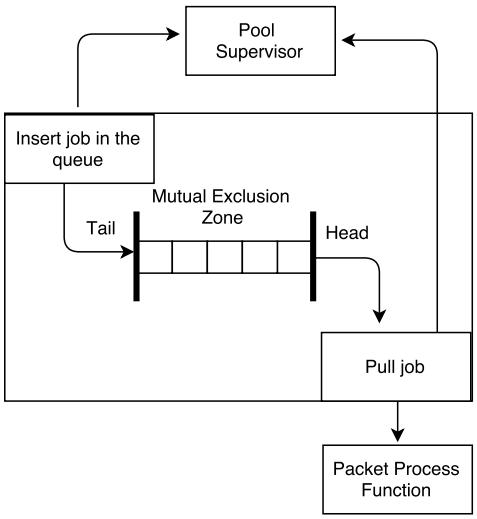


Figure 3.3: Thread Pool use scheme

We call this approach controlled concurrency in the sense that it enables the possibility to create a more hardware suited solution, with pre-allocation of resources and intrinsic reusability of them, avoiding constant memory demands for launching and disposable of threads, resulting in a better overall performance. The implementation idea behind this concept is rather simple. At the beginning, the internal structures are initialised and a n sized queue is generated to hold all the jobs inserted. Clearly, in a highly concurrent environment the jobs must be inserted and taken off the queue for processing one at a time, for that to happen the supervisor handles the complexity behind the process. It checks for queue consistency and manages all thread states during the run. The use of a thread pool has been the turning

point for the problems previously at stake, as it brought more stability during runs. Perhaps, the only downside about it, is that, it is not a size fits all solution, as the number of threads must be tested and refined for any particular machine.

In our specific case, and considering our table of specs 3.1. After 5 consecutive runs, we have calculated the mean of the values and we have obtained the following results depicted in table 3.2.

Characteristic	Value
OS	Linux Mint 64bit
Compiler	GCC version 5.4.0
Thread model	Posix
Kernel	4.4.0-45-generic
Processor	Intel Core I5
Disk	SSD 128 GB
RAM	8GB

Table 3.1: Machine's Specs

Pckt rate (s)	100	1000	10000	100000
1 Thd per pckt	0.0311±0.015 ms	0.0735±0.017 ms	-	-
ThPool/64	0.0159±0.010 ms	0.0163±0.013 ms	1.0962±0.0917 ms	2.9317±0.085 ms
ThPool/128	0.0171±0.016 ms	0.0214±0.016 ms	2.5034±0.070 ms	3.4706±0.082 ms
ThPool/256	0.0192±0.020 ms	0.0324±0.028 ms	3.3273±0.083 ms	4.3195±0.091 ms
ThPool/512	0.0234±0.031 ms	0.0337±0.032 ms	3.9939±0.093 ms	5.3643±0.153 ms
ThPool/1024	0.0234±0.032 ms	0.0509±0.041 ms	5.1003±0.145 ms	6.6256±0.185 ms

Table 3.2: Results of the simulation

From the observation of the table we have witnessed very interesting results concerning the efficiency of each solution. The naive solution has shown, as expected, to perform reasonably well at low packet rates, falling clearly short after a few thousand packets per second. Ending in a program crash right before tens of thousands of packets per second, due to the facts explained previously. Apparently, because our test machine is an ordinary laptop with commodity hardware, its parallelism capacities are rather limited. Even though the results, for number of threads higher than 128 might be a little bit misleading or even confusing because of unexpected hops in times obtained. The solution with 64 initialised threads stands out from the others as the best fit in our architecture.

3.3 Information Retrieval and Data Organization

The next step for the solution considers information extraction. At this point, we had to make a decision on the information that we should be taken out of each packet. As said previously, we wanted to build an approach on which privacy, efficiency and simpleness could be gathered together. Therefore, since all the packets are treated equally by libpcap we have analysed both versions of IP headers along with the upper headers for transport protocols.

Taking in consideration, essentially memory efficiency purposes we have taken only the first 60 bytes out of each packet. This decision was supported by some important characteristics of packet's header organization that we have already assumed from our previous analysis. As we proceeded to the analysis of packet headers, we have understood that we only needed enough information comprised in Layer 2, IP and, finally, some essential information from transport layer(TCP/UDP) headers, as a way to cope with our organization and classification demands, and still preserving users privacy. The 60 byte mark is intrinsically linked with IPv6 support. In IPv6, the header is fixed at 40 bytes, and extension headers are added when needed. Extension headers, along with headers of higher-layer protocols such as TCP or UDP, are chained together with the IPv6 header to form a cascade of headers, sizing 20 bytes, as can be seen in Figures 3.4, 3.5.

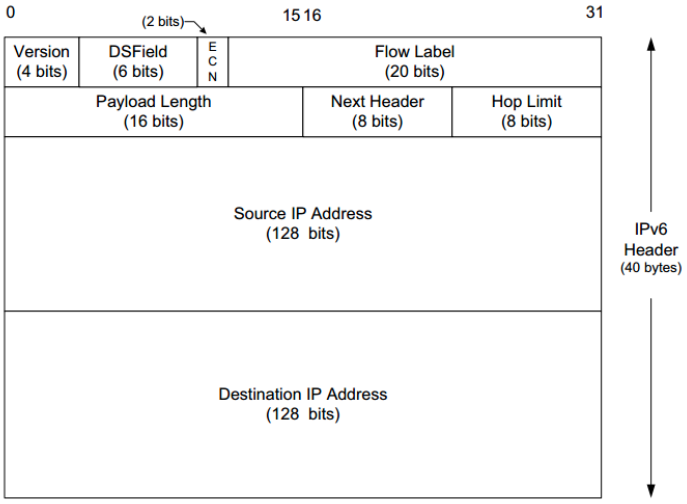


Figure 3.4: IPv6 header organization extracted from [7]

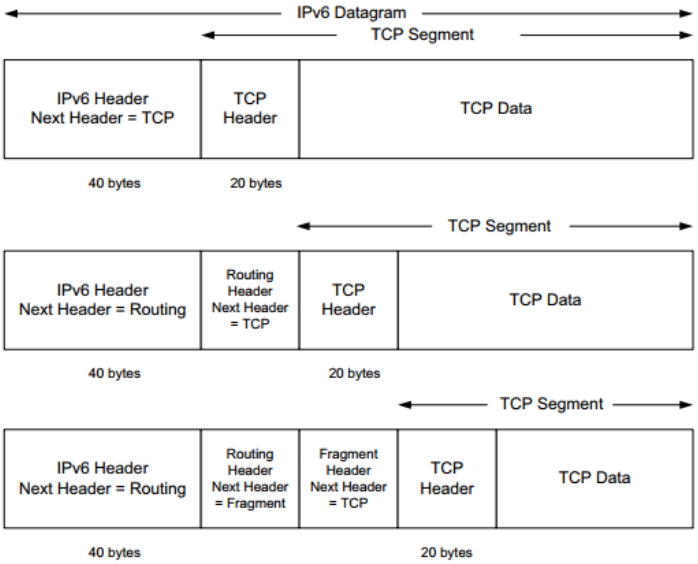


Figure 3.5: IPv6 extension header organization used from [7]

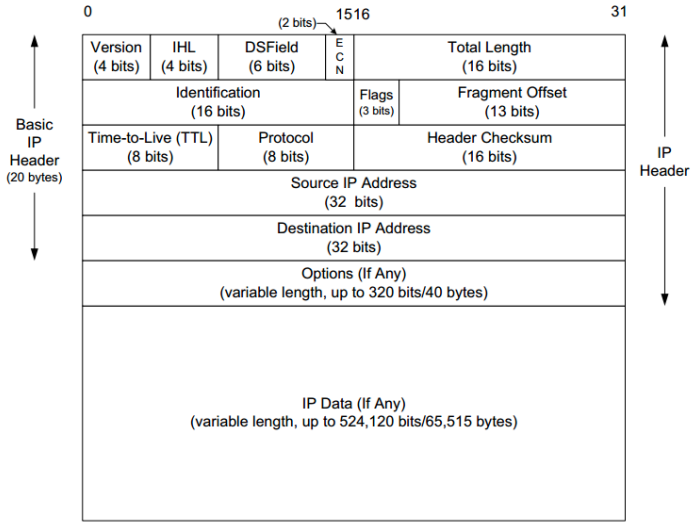


Figure 3.6: IPv4 header organization [7]

To accomplish our objective, we have proceeded for each packet, to the division of its information in two main classes. As a consequence, since our method is flow based, we have used the most widely used definition of flow, which is the famous fifth-tuple comprised of:

ψ=<Destination IP address, Source IP address, Destination port, Source port, Protocol>

As we can see, most of our information retrieval process was practically done after analysing the packet's header only, which in the "worst" case (IPv6) will only cost us forty bytes of memory allocations. The ψ described above works as our basic structure, it defines a unique identifier to catalog packets per flow, that means, all Upload and Download directions mixed up together.

Actually, after the process of defining what a flow is, we went for other information that could be useful for the type of study that we wanted to conduct, we thought at this structure as post identification step, where we extracted packet's specific information for the statistic analysis we decided to make. Considering that, and since part of our analysis is based in time related calculations, we figure out that a time stamp would be one key, therefore, we have used the time stamp provided by libpcap. This kind of value is a data type that comprises two fields and it works by providing one field for elapsed time in seconds, and other that goes up to microseconds resolution witch we thought would be a fairly good resolution considering the range of time we wanted to address. Afterwards, we extracted a value that, as literature suggests could be used as a good argument for traffic analyses which was packet size. Finally, we decided to have an idea over the operations within a TCP flow so we extracted information on the flags for connection creation and ending. Those, were the only information extracted from each packet, upon witch we based our next processing and decisions.

3.3.1 Note on Memory Usage

When talking about low level programming, such as programming with C language, there are several details that need to be taken care of, one of those is memory usage. In this

sub-segment, we are going to discuss some problems and C specifics, that were taken in consideration during the implementation, such as, dynamic memory allocation and structures mapping in memory. The subsequent discussion, is intrinsically machine dependant, therefore we will assume the previous table of specs 3.1, where the solution was developed and tested.

Dynamic memory Allocation

Dynamic Memory Allocation (DMA) is common place for C programmers, in the sense that it is an extremely used resource, enabling the programmer to provide data size at run time. However, even being aware of its extreme importance is crucial to understand its weaknesses. As expected, we use this technique extensively. Thus the drawbacks of using DMA in our specific context, are mainly two:

1. Seeking Bottleneck

   Today's standard allocators (used in our code) such as malloc, calloc and free, rely their performance in multi-threaded environments to locking implementations (shipped with pthread option on compile time). Thus they are implemented in a blocking fashion, deteriorating performance. This problem is constant during the application life time, being specially relevant as packet bursts need to be processed. As explained previously, libpcap, passes arguments to its associated callback, those are, a pointer to a region of memory where the actual packet contents are stored and the size of the packet in bytes. It's the programmer's responsibility to assemble (create data structure able to be processed) the information to process and reserve the necessary memory. Inside each thread, the actual packets are processed, and the information, described previously, is extracted and another two dma are performed. In our context there are, potentially, thousands of, malloc free pairs per second.

2. Memory Fragmentation

   Memory fragmentation is a very well known issue that concerns low level memory management. It happens when a memory request is issued to a memory system and even being able to fulfill it there is no contiguous block enough to store it. To simplify, the Percentage of fragmentation in memory is defined like this:

   $$Fragmentation = 1 - \frac{(Greatest\_Block\_of\_memory\_available)}{(Total\_Memory\_Available)} * 100$$

   As explained above, there are a great number of malloc, free pairs, among structures with different sizes. Actually, in most regular applications this wouldn't be much of problem because of their ephemeral life time. Although, in an application with a potential "eternal" life time, and heavy memory usage, this might be a performance deteriorating factor. In image 3.7 we can see a clear example of the way that might happen.

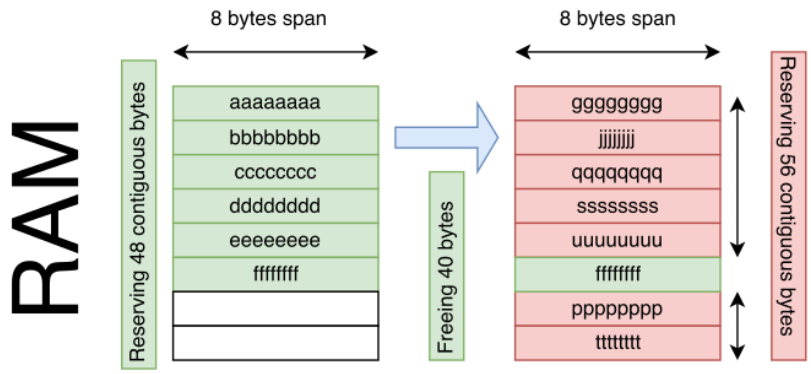


Figure 3.7: Fragmentation example

Usage of other solution [59], have been tested, without any apparent benefits in our machine. Moreover, in a work about memory allocators [60] it has been shown it highly fragments memory under less core architectures. In other very important study [61] the authors discuss whether a group of memory allocators would perform well under multi-threaded environments or not. They have clearly concluded that glibc's implementation would perform well in commodity hardware, typically two or four core architectures, even under largely threaded environments. Thus, in this version we opted to use the standard implementation provided by GNU, because it has been fully tested, and has been proved to also have good score in a very important study about memory fragmentation [62]. Finally, even admitting that might be a better solution, we think this is a topic of extreme importance and should be extensively addressed in the future and tailored to any specific use or architecture.

Structures Mapping in memory

In our scenario, it is of extreme importance to understand how data types are mapped in memory, and to take decisions to ensure robustness to our solution. Thinking about that, we went on to study glibc's structure memory layout and to calculate each individual type size. This especially important because of low level structure comparison that we use extensively. Once again, when talking about low level management, we have to take in consideration every particular machine, therefore the study has been performed considering the specs described in table 3.1. Before we get to the bottom of the question, we have to understand that, almost every C compiler works with alignment constraints to provide fast access to data stored in memory, as expected it works as far as basic data types are concerned. The convention in C compilers at least for x86 and Arm architectures is that all basic data types are self aligned, with no padding needed. Self alignment is a key aspect as it permits to generate single-instruction fetches, avoiding jumps within the memory which contributes to unwanted latency. In general, a struct instance in C will have the alignment of its widest scalar member. Most compilers will behave that way as a technique to easily ensure that all the members are self-aligned for fast fetch. Another important fact is that in C, the address of a struct type will always be the address of its first member, thus avoiding leading padding. In listing 3.1 is depicted our final layout of data types sizes and distribution in memory.

Listing 3.1: Code Mapping for the Flow Id's

```
/*IPv4 Flow id*/
typedef struct flow_id_v4        /* 24 bytes */
{
    int padding;                 /* padding 4 bytes */
    int  protocol;               /* protocol 4 bytes */
    int  port_src;               /* source port 4 bytes */
    int  port_dest;              /* destination port 4 bytes */
    struct in_addr  ip_src;      /* source ip 4 bytes */
    struct in_addr  ip_dest;     /* destination ip 4 bytes */
} flow_id_v4;

/*IPv6 Flow id*/
typedef struct flow_id_v6        /* 48 bytes */
{
    int padding;                 /* padding 4 bytes */
    int  protocol;               /* protocol 4 bytes */
    int  port_src;               /* source port 4 bytes */
    int  port_dest;              /* destination port 4 bytes */
    struct in6_addr  ip_src;     /* source ip 16 bytes */
    struct in6_addr  ip_dest;    /* destination ip 16 bytes */
} flow_id_v6;
```

Our first approach was written ignoring the rules described previously. Thus, the results generated by our program were undefined. That behaviour is visually explained in image 3.8, in green we can see the distribution of the actual elements in memory and in red we have a representation of the hole generated by that distribution. In C, compilers tend to fill the holes within structs applying random binary garbage. For a matter of searching within a structure, that wouldn't be much of a problem, as we can access all the elements individually, knowing exactly its boundaries. However, we impose the padding ourselves, filling the holes with "zeros" to permit low level (complete blocks of memory) comparisons to be successful without being affected by any kind of random garbage.

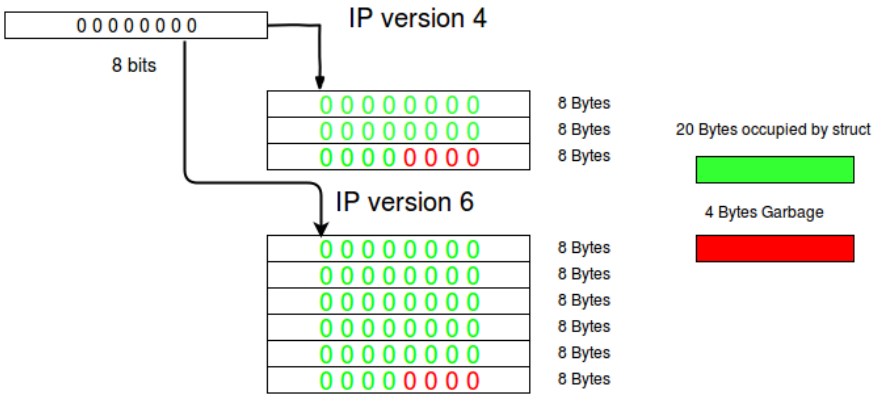


Figure 3.8: Data Alignment Memory View

3.3.2 Data Storage and Organisation Process

At this point after the information is extracted from the packet, it's time to understand how information is organised, and the relevance of our choices concerning data structures. First of all, considering our previous premise of speed over memory efficiency, we have proceeded to a conceptual study of this compromise to make sure we found the best fit for our needs, while choosing a data structure to hold a table of flows. When choosing a data structure for a certain need, we designed a simple example architecture to verify our basic work flow per packet, depicted in image 3.9.

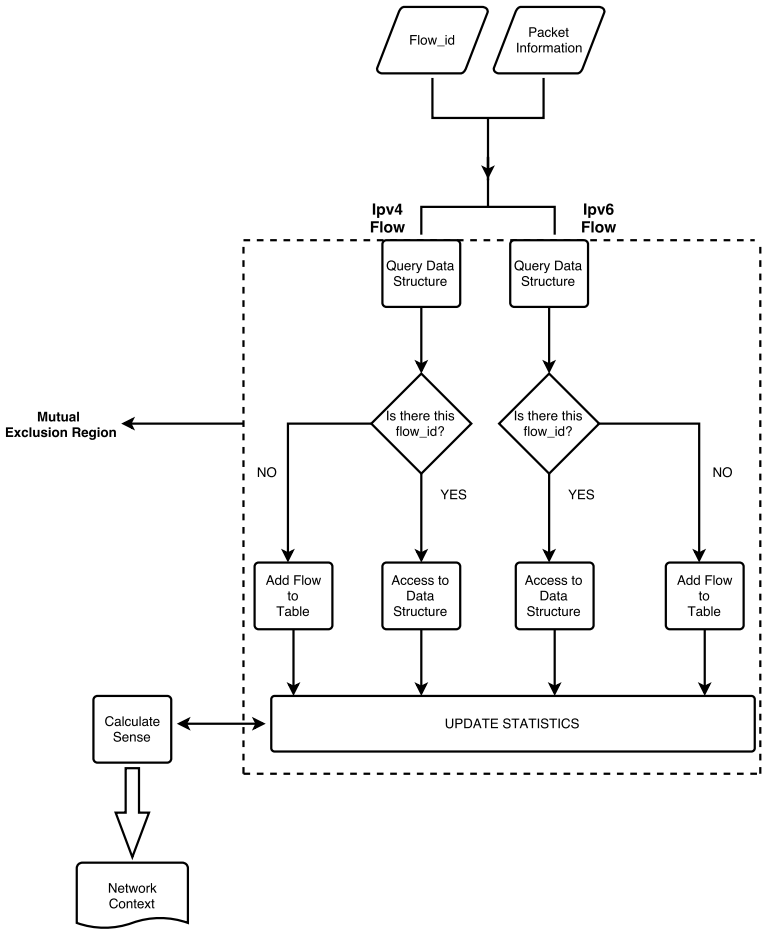


Figure 3.9: Basic Work Flow for the differentiation process

As the scheme clearly presents, for each packet processed we have, always, one query to the data structure, to evaluate the existence, or not, of the current flow. In that sense, this design characteristic, points a rather crucial feature for our data structure to support. It is the capacity to perform quick lookups and data accesses, in order to avoid being the bottleneck for the system. When a problem with potential great amounts of data, demanding quick lookups arises, one automatically have to look for a search solution based on adapted Symbol Tables or a Binary Search Tree (BST).

A Symbol Table is a data structure holding items with keys associated that supports two basic operations: insertion of a new item, and returning of a previously inserted item given its key [9]. We can clearly state that this kind of structure works in the same sense as a dictionary, indexing a word to its content (synonym) ordered alphabetically. As expected, such a limited range of operations over this kind of structures makes it unusable in real life contexts. Since this kind of solutions is of extreme importance, it has been extensively studied throughout the years and a set of operations have been suggested to this abstract data type, such as:

1. Create Symbol Table
2. Insert a new key item pair
3. Search for an item given the key
4. Delete a specified item
5. Symbol Table sort facility (traversal in key order)
6. Destroy table

The first approach to this kind of structure is based on Key-Indexed Search. Perhaps, the simplest and straight forward implementation assumes small positive integer values as keys. The solution's code is as follows: initialising all the entries of arr[] to "Nullref", then proceeding to insert each k value into its appropriate self addressed position arr[k], and obviously, perform a search by accessing arr[k] itself. In this naive implementation we leave to the user the responsibility of handling repeated keys. This first step is considered to be a point of departure to all the more sophisticated symbol table implementations.

It opens doors to a very important, yet simple property that states that : "If a key values are positive integers less than M and items have distinct keys, the the symbol table data type can be implemented with key-indexed arrays of items such that insert, search, and delete operations require constant time; and initialise, select, and sort require time proportional to M, whenever any of the operations are performed on an N-item table" [9].

Another approach, more suited for general key values from a large range for them to be used as index, is based on an implementation that stores the items in order inside an array. When a new item comes to insertion, all the entries behave just as an Insertion Sort, by moving larger objects over one position to accommodate the newcomer without losing order. Moreover, since the array holding the structure is in order, both select, and sort are of trivial implementation. This last approach, as expected allows the possibility to overcome the previous limitation on repeated keys [9].

A different procedure called Binary Search, is based on a divide-and-conquer paradigm where we divide the set of items in two separated sets and then, from there, decide where to go. The obvious way to perform the division operation is to keep the holding array sorted and then use indexes to delimit the part of the array being worked on. From this basis an important property can be deducted as such : "Binary search never uses more than $\lfloor logN \rfloor + 1$ comparisons for a search (hit or miss)" [9].

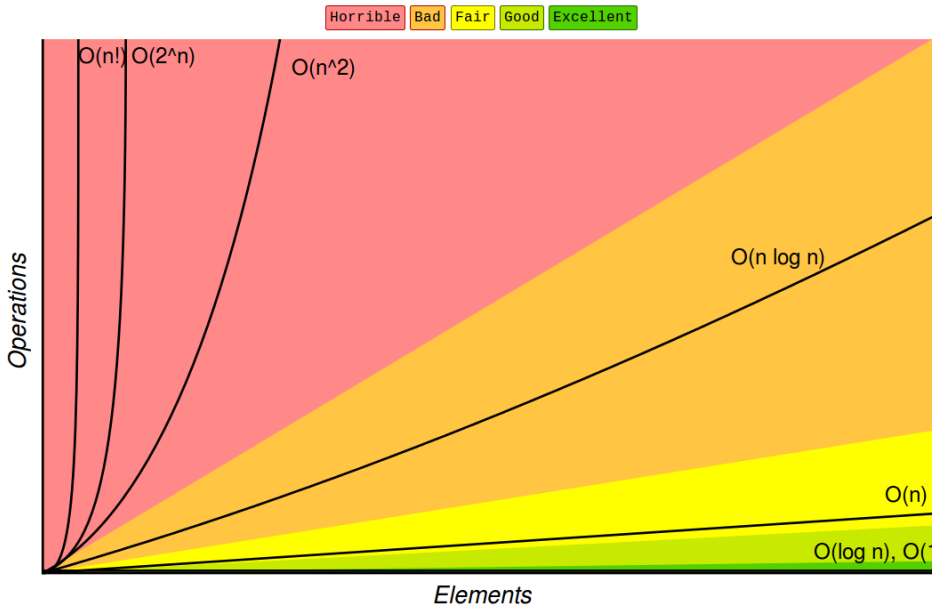


Figure 3.10: Algorithmic Complexity Mapping [8]

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Figure 3.11: Data Structures Operation Complexity [9] [8]

Data Structure Efficiency Analysis

Taking in consideration the definitions and properties explored in the previous subsection. Naturally the table 3.11 and its respective graphical mapping present in image 3.10 covering the space and time Big-O complexities of common data structures used in computer science, were the conclusion extracted for some data structure that we find more relevant. As far as speed is concerned, we aim to achieve constant time operations in the majority scenarios, specially in search operations.

Evaluating table 3.11 we can conclude that the major property we strive for, is only achieved, at least in the best and average cases, by a hash table, which gives the dictionary abstraction for our case, in flow differentiation. There is still a problem that we have to address, which is the worst case scenario, where it results in a worse linear complexity that, we strive to avoid. This behaviour might occur, and a hash table can degenerate in a common array, because of a problem that is intrinsically related with the hash calculation step. As we know, this data structure, works converting keys into direct table addresses, allowing the constant time operations depicted in table 3.11. The second step of a hashing algorithm is to take a decision on how to handle the case when two different keys hash to the same address. There are two main methods to resolve collisions:

1. Separate Chaining

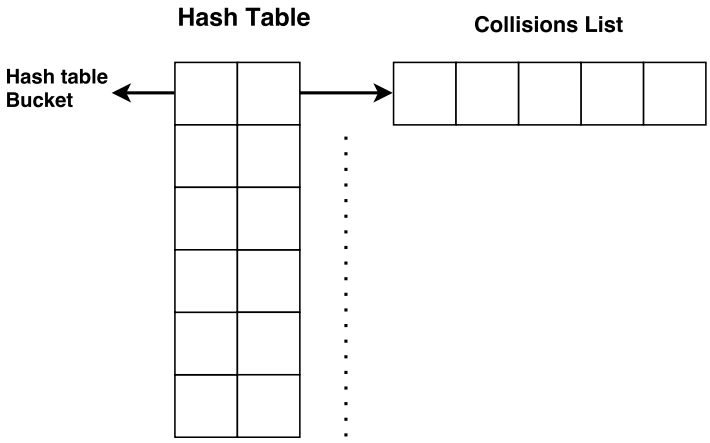


Figure 3.12: Graphical representation of hash table using separate chaining

This method works by creating a pointer to a chain (typically an array or linked list) associated with every bucket (address), to deal with different keys hashing to the same number, therefore originating the same storing address.

This method is commonly called *separate chaining*, because colliding items, are chained together in each respective list. An example of this is depicted in image 3.12. As Sedgewick [9] refers in its analisys of the issue, hash tables are a great classic example of "time-space trade-off" as we use more memory in exchange for speed. In separate chaining this fact always happens, wastage of more memory with list pointers for every bucket, but regulating speed with short length lists. In the case of separate chaining approach, with M lists and N keys, is very probable (typically with probability around 1) that the number of keys laying on each list is a small constant factor of N/M. A classical probabilistic analysis shows that the probability of each list having k elements is as such:

$$\binom{n}{k}\left(\frac{1}{M}\right)^{k}\left(1-\frac{1}{M}\right)^{N-k}$$

Assuming that choosing any k elements out of the total N presented. Those k hash to the same address with probability of $\frac{1}{M}$ and the others N-k hash to independent ones, with probability of the opposite event 1-$\frac{1}{M}$. Considering $\alpha = \frac{N}{M}$ we rewrite the expression as:

$$\binom{n}{k}\left(\frac{\alpha}{N}\right)^k\left(1-\frac{\alpha}{M}\right)^{N-k}$$

Making use of Poisson approximation analysis we retrieve that, it is, less than:

$$\frac{\alpha^k e^{-\alpha}}{k!}$$

From this result we can see that for practical ranges of the parameters, the probability of hashing to any list with a significant bigger length is rather low. From this analysis, we can easily conclude that separate-chaining should be used with confidence if its hashing function is strong and returns values approximately random [9].

2. Probing

This method is based on the principle that we can predict, in the beginning, how many elements our table will need to hold, so one can over dimension it in order to fill the holes up in case of collisions of conflicting keys. Such methods are commonly denominated open-addressing hashing [9]. The simplest method on this field is linear probing. Basically, when there is a collision, then we just check the next position for availability, such operation is typically called a probe. This method, is recognised by the evaluation of three distinct outcomes of a probe: If the table address possesses an item whose key matches the search key, it results in a search hit; if the position is empty, then we have a search miss; Finally if the position holds a key that does not match the search key, we just keep on probing to the next position until we reach the bottom of the table, after that event, we wrap back to the beginning of the table proceeding the same process until the search key or an empty space are found. In the end, if we want to proceed to an insert after this process, we should insert the item in a address that stopped the process itself. This method's efficiency is directly connected to the ratio $\alpha = \frac{N}{M}$, even being similar to the ratio discussed in separate-chaining, this time, the relation is interpreted as a fraction of the total table length with the number of occupied buckets, which is commonly called load factor [9]. Using this paradigm we expect a low load factor to permit empty spot finding with just a few probes. Otherwise, for a almost fully table (load factor nearly 1) a search could require an impracticable number of probes, even resulting in a infinite loop, if the table is completely full. In an analysis of speed performance, we should be aware of the cluster concept. A cluster on this regard, is the way items are stored in contiguous blocks of memory, allowing taking advantage of the reference locality principle. The average behaviour of linear probing depends of the way elements cluster together. Considering a scenario where a table is half full (M = 2N): In a best case scenario, items would be distributed, for instance, in odd table addresses, while the respective contiguous even addresses would be empty. However in the worst case, we would consider the same number of items, but this time with a totally different distribution, therefore, the first half of the table would be completely full while the rest would completely empty. Taking that in mind, the average length for

any cluster, regardless the case is $\frac{N}{2N} = \frac{1}{2}$, while for an unsuccessful search the average number of probes is 1. In the best case we have the following relation:

$$\frac{(0+1+0+1+0+1+...)}{2N} = \frac{1}{2}$$

For the worst case analysed we have:

$$\frac{(N+(N-1)+(N-2)+...)}{2N} \approx \frac{N}{4}$$

Evaluating the previous relations, generalising them, we find that the average number of probes defined for an unsuccessful search is directly proportional to the clusters lengths squared. From the analysis just made, we can conclude that, given a specific table scenario we can quickly calculate the average burden originated by unsuccessful searches in that table. Although, the clusters being formed and distributed based on a dynamic algorithmic process makes extremely difficult to characterise it analytically. In conclusion, Sedgewick [9] defines that when collisions are resolved using linear probing methods the average number of probes required in a hash table of size M containing $N = \alpha M$ keys is approximately:

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad \textit{and} \qquad \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

Another data structure that deserves our attention, is a variation of the traditional binary search trees, this specific implementation allows the usage of the key's bits to position the elements within a tree in order, this approach is called a trie. The basic idea behind this type of "tree" is to store keys only at the tree's leaves level. The concept appears as solid alternative in the field of search algorithms and search optimized structures. Tries are used commonly within fixed number of bits per key, assuming that they are all distinct, which is a rather decisive feature, inclusively being used in important academic projects and applications [63] [64] [65].

In a trie, keys are kept in the leaves (node with no children) of a binary tree. As a principle this is a very different, and actually a good idea, allowing the bits of the key guide the search, while keeping the invariant order criteria at each node that all keys whose current bit is 0 should fall in the left sub-tree all keys whose current bit is 1 should fall in the right sub-tree [9]. The main difference between a BST and a trie is very well depicted on the form that a search is performed. As said before, each key is stored in a leaf, following the path of its leading bits from the root to that specific leaf it works in the same sense that it follows a prefix along the way. Null links in node that are not leaves correspond to search fail as the specific bit pattern doesn't exist within the tree. Therefore, to proceed in a search a key within a trie , we just branch according to its bit, avoiding comparison of internal nodes [9]. Another peculiarity of tries is related with the main characteristic just explained, which is that, for any set of keys, there is a unique distribution resulting in a unique tree. It is dependent on the key set itself and also dependent on the order in which we insert the keys. All this properties lead to another distinctive property related with its complexity which is that, for any insertion of a random key in a trie built with N random and distinct bitstrings

requires on average log(N) bit comparisons. The worst case being bounded by the number of bits presented in the key.

The analysis, must be done based on the assumption of the distinctive keys, in this case we use (infinite) sequence of bits. On average it all comes to the following probabilistic argument. The probability that each of the N keys in a random trie differ from a random search key in at least one of the leading t bits is

$$\left(1 - \frac{1}{2^t}\right)^N$$

Calculating the opposite probability, which gives us the probability of one of the keys in the trie matching all the leading t bits, it appear as:

$$1 - \left(1 - \frac{1}{2^t}\right)^N$$

From fundamental probabilistic analysis, the sum for $t \geq 0$ of the probabilities of a random variable is t is the average value of that random variable, it is represented by the following:

$$\sum_{t \geq 0} \left(1 - \left(1 - \frac{1}{2^t}\right)^N\right)$$

As form of interpret the outcomes of the analysis we can use the elementary approximation $\left(1 - \frac{1}{x}\right)^x \sim e^-1$, we deduct that the average cost of search is given by approximately by:

$$\sum_{t \geq 0} \left(1 - e^{\frac{-N}{2^t}}\right)$$

The conclusion extracted from the relation above is that, the summand is very close to 1 for approximately log(N) terms with 2^t way smaller than N; it is very close to 0 for all the terms with 2^t way bigger than N, being somewhere between 1 and 0 for the few terms respecting $2^t \approx N$. Therefore the majority is always bounded to log(N) [9].

Data Structure Library Choice

After the analysis and the choice of the kind of data structure we were interested in use, we went on to evaluate the best options on the subject of data structure libraries written in C. To proceed to that evaluation, we have divided the decision process in two main criteria branches. First of all, we have used, programmer convenience, which is rather subjective and is exclusively dependent on the developer itself. From other side, we have used analytical indicators such as, memory and processing speed benchmarks along with previous studies from independent users.

On the developer side (which was the least important parameter) we were interested in find an implementation that could offer a well written interface, active support and development, along with comprehensive documentation. From this perspective, the winner by far came up to be TommyDs library, as it provided an active development spirit, a well written implementation released under an open source license, good documentation, adding it up with good examples. Another great asset in our point of view is the vast work on testing and benchmarking of the code and on the release of that particular code as way clearing the field for others testing. Taking that in consideration, we have used the author's work on

benchmarking to make up our mind on what library to use. He has performed a series of tests with different scopes, trying to measure the impact of different sequences of instructions on several competitor libraries. The most interesting scenario was simulating real world problem with random sequences. The main idea consisted in storing a set of N pointers to objects and searching them using integer keys. The point of comparing pointers instead of mapping integers to integers is that, mapping pointers to objects makes them also dereferenced, to simulate the object access, resulting in additional cache misses. This scheme was particularly important, as we follow the same paradigm during the coding stage. The performed tests were:

1. **Insert test** - Inserting all the objects starting from an empty container.

2. **Hit test** - Finding with success all the objects and dereference them.

3. **Change test** - Find and remove one object and reinsert it with a different key, repeating for all objects.

For the test all the objects were pre-allocated on Heap memory, being ignored all the times related with operations over memory such as, allocations and deallocations. The objects were identified and stored using **unique** integer keys. The key domain used was dense, and defined by the set of N even numbers starting at 0x80000000 and going until 0x80000000+2*N. The use of even numbers allows to have missing keys inside the domain for the Miss test. Being used in that test a domain defined by the set of N odd numbers starting at 0x80000000+1 until 0x80000000+2*N+1. The choice behind the 0x80000000 based can be explained as the necessity to mislead the eventual effects of implementations treating zero based numbers in a special way. The hashing process was just used on the hash table containers using the tommy_inthash_u32() function. For all the rest being inserted all the keys directly without previous hashing. In order to get advantage of our random scenario, we have analysed the results from that context, being all the keys inserted in a random order.

As we can see in image 3.13 TommyDs stand out as the best option in the random Change test. As expected, the hash table implementations came out with the best performance. The event can be justified because of its randomness on access which in an environment with no collisions, is a decisive advantage. One interesting point about this test is the little vertical leap occurred at around 100 k objects. This fact is expected and indeed, can be explained by the system's cache reaching its limits, starting to get the information from main memory instead of using the cache. In hit test depicted in image 3.14 we can see that hash table implementation came up as the best solution with tries coming in second and binary search trees in third. Once again Tommyds implementation got the best results with a close result by google's implementation. Finally, the memory test, depicted in image 3.15 has shown a rather different conclusion when compared with the previous experiences. In this particular case, we can clearly witness the space speed trade-off, where implementations based in tree and trie structures got generally a better score on memory footprint than hash tables. Perhaps, another interesting fact worth mentioning is that trees and tries have a more stable result, most of the times, representing a similar burden in memory as the straing lines show. On the other hand, hash tables perform generally worse and also show peculiar memory spikes and, in same cases approximately periodic in tommyds implementation.

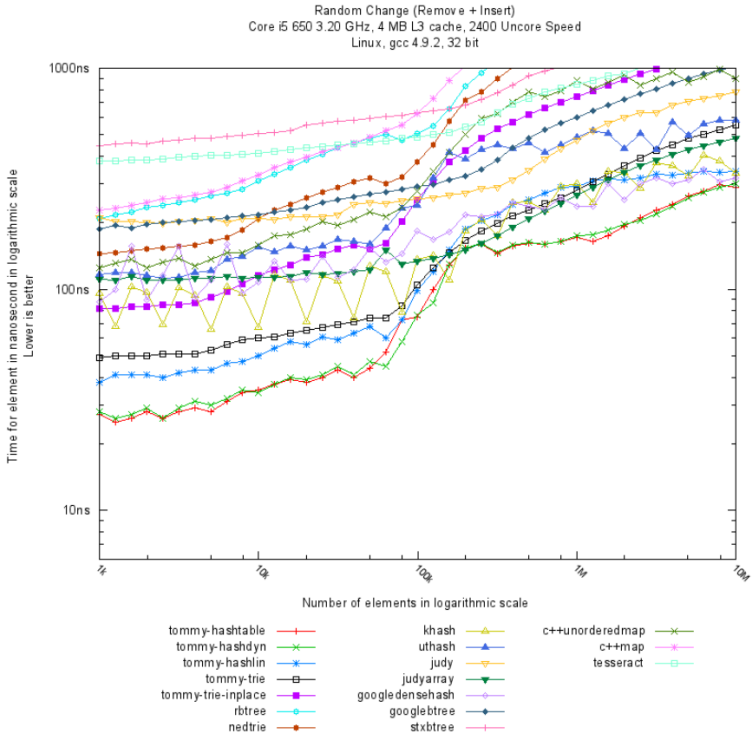


Figure 3.13: Results of the change test benchmark extracted from [10]

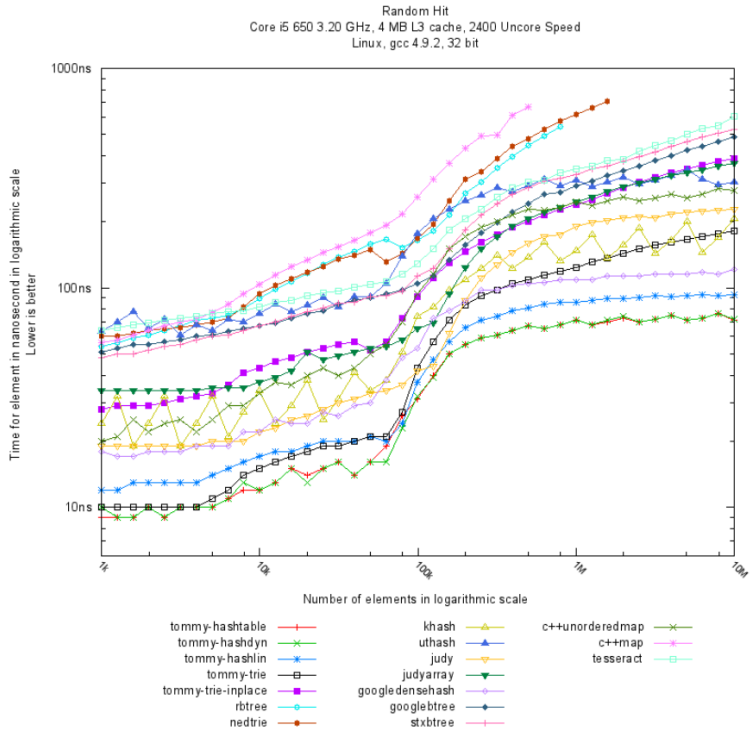


Figure 3.14: Results of the hit test benchmark extracted from [10]

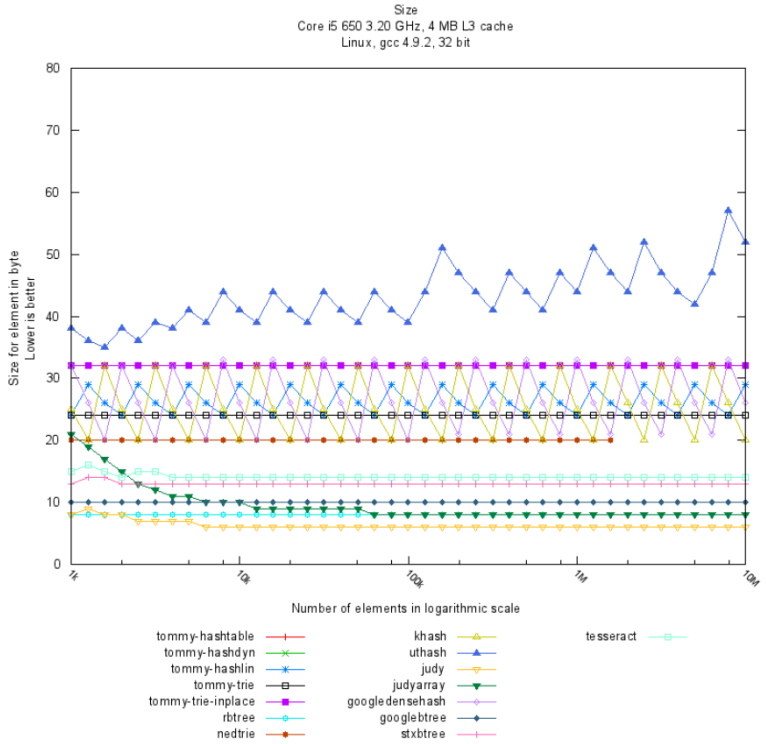


Figure 3.15: Results of the memory test benchmark extracted from [10]

From the evaluation performed we have taken the decision of choosing Tommyds library for all the facts previously exposed. The next step was the one where we tried to choose from all the provided solutions which one would be the best fit for our specific use case. From that point of view, we have evaluated two of the most interesting data structures in library, tommy hashlin and tommy trie. Tommy trie uses a custom memory allocator, as the author states that malloc function on itself cannot keep up with demands of a cache optimized solution such as tommy trie. This structure is a standard implementation that stores elements in a specific order defined by the comparison function passed by the programmer in advance. It possesses an interesting feature related with the, as we call it, ramification factor, which is the number of branches per node, implying that more branches is directly connected with more speed, but at the same time, it implies more memory foot print per node, in this particular experience we have used a the value by default which is 8 to to exactly fit a typical cache line of 64 bytes. Tommy hashlin is one of the three implementations provided in the library, we have chosen to test it over the other two because of its real time friendliness and capacity to perform well under potentially big time spans as we want it to do. It basically works by starting with the minimal size of 16 buckets, doubling its when it reaches a load factor greater than 0.5 halving its size with a load factor lower than 0.125. The progressive resizing, by using the linear hashing algorithm has been proved and suggested in important bibliography [66] as being simple and efficient technique for applications, where the cardinality of the key set is not known in advance, implying a good behaviour in real-time and interactive environments making insert and delete operations take approximately the same time. In the resizing step another good idea was implemented, for that, was used a dynamic array that supports access to not contiguous segments of memory. This way, only allocating additional table segments on

the heap, without freeing the previous table, and then not increasing the heap fragmentation, which might influence in a long running time span.

In image 4.3 we can see the results obtained in a simple benchmark respecting the characteristics previously mentioned in table 3.1. In this first one, we wanted to measure the behaviour of each of the data structures when receiving a burst of forward order flows. The benchmark worked by simply measuring time at the beginning of the operations and subtracting the time at the final of the program. These results were obtained by calculating the mean of ten consecutive runs of the benchmark. Evaluating the results we can see that both of the structures responded very well and levelled its performance during most of the experience, occurring a little leap at the four million elements. This time, the second experiment was performed integrating the previous one, but now, measuring the time to search by a random flow within the complete set of flows, the results are depicted in image 3.17. Once again, the results were very tight, with a, perhaps, surprising slight advantage for the trie over the hash table. We believe this fact can be explained by the forward order of insertion previously mentioned. Tries tend to outscore hash tables on this specific scenario of ordered insertion, as a result of a well balanced tree scheme.

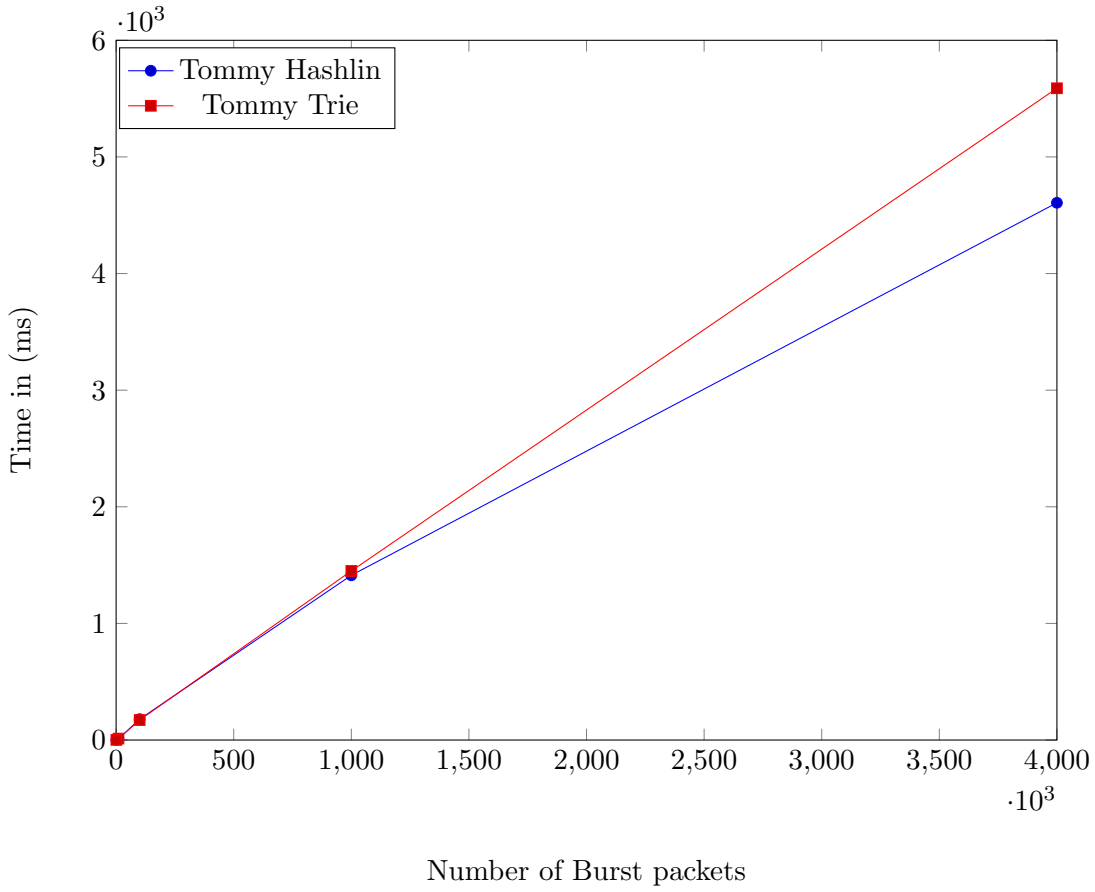


Figure 3.16: Results of the burst insertion benchmark

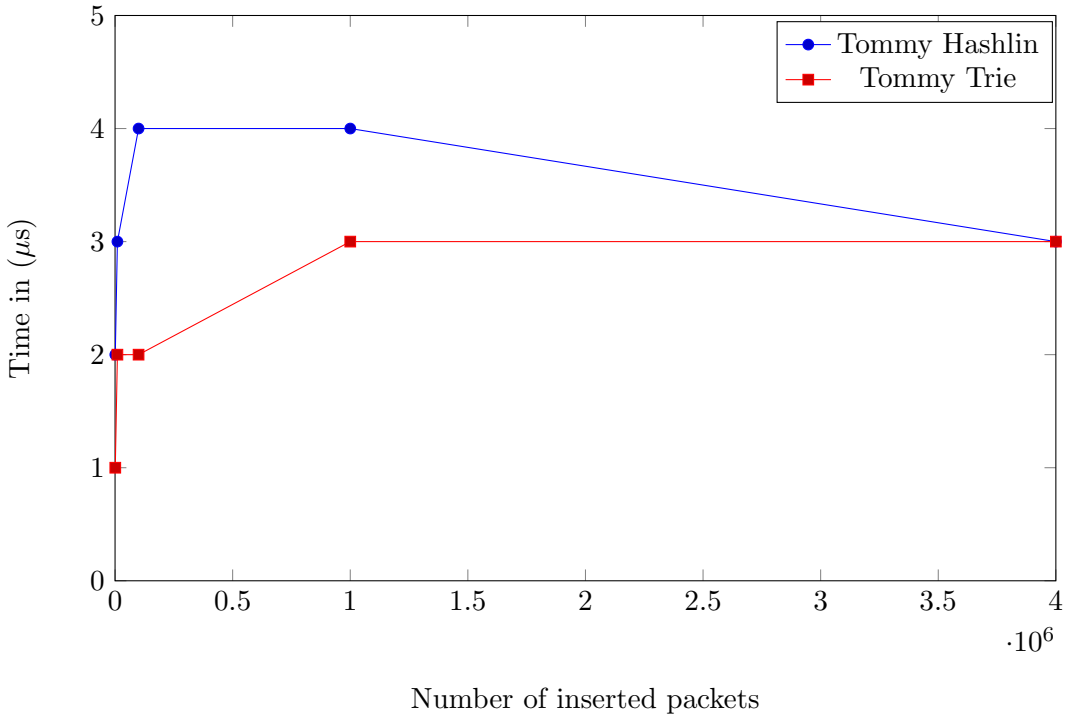


Figure 3.17: Results of the random single look up

From the previous analysis, and being aware of such a levelled score, we have opted for the hash table for the implementation of our solution. The choice was made after a risk calculation was taken into the equation. Tries rely on the uniqueness of all its elements to perform in a correct manner. Tommyds's trie implementation require the element's indexing process to have an unique integer associated with each element on the insertion. Therefore, the solutions adopted would forcibly have to be a calculated hash. At the 4 million mark the signal of collisions has not been felt but as the structures grow to, perhaps, tens of millions of elements, without a collision management capacity, eventual collisions might degenerate in erroneous behaviour, destroying the solutions capabilities.

In listing 3.2 we have the code layout, used in the insertion of a new element into our hash map. The structure is constituted by 6 fields, being divided in 2 distinct areas. The first area can be defined has data structure's specific use. While the second area is directly connected with our working process. When speaking about library specific fields, one should pay attention, first of all, to the node variable, which is used internally by hashlin to perform the hashing and the posterior storage. One interesting and rather unexpected field is undoubtedly the list_node, this variable is directly linked with one of tommyds's limitations. That limitation is related with the absence of a declared and explicit definition of an iterator, to perform linear searches over hash maps or trees. Furthermore, as we are going to see later, this feature is of critical importance for us throughout the life span of our application. Therefore, our way to turn the problem around was accomplished by simultaneously adding the node to the hash map, but right after, associating it to a list that, possesses iterating as one of its intrinsic characteristics. Now it is time to understand the relevance of each the remain variables, that belong to our specific work. First of all, we have the flowid, which has been identified previously in section 3.3, it constitutes the corner stone of our model allowing

to distinguish among all the flows passing by. The next two fields, are time related, the timeofcreation variable, as the name indicates, is used to keep track of the time a flow started to be track down, in practise, it matches the time a flow is added to the hash map. The second time field is called lastinserted, it is highly updated field, being used to keep track of the time when the last packet belonging to that specific flow has been processed and got into the statistics. Finally, the last field is used to connect our organisational architecture with an array of statistics that we are going to explore further in the next section.

Listing 3.2: Node inserted inside the hashmap

```
/**
 * Structure to be inserted in the data structure hashmap IPv4
 */
typedef struct objectv4
{
    tommy_node node;
    tommy_node list_node;
    struct flow_id_v4 *flowid;
    struct timeval timeofcreation;
    struct timeval lastinserted;
    struct samplestatistic *stat;
} objectv4;

/**
 * Structure to be inserted in the data structure hashmap IPv6
 */
typedef struct objectv6
{
    tommy_node node;
    tommy_node list_node;
    struct flow_id_v6 *flowid;
    struct timeval timeofcreation;
    struct timeval lastinserted;
    struct samplestatistic *stat;
} objectv6;
```

3.4 Statistical Work

In this section we want to provide insights on the statistic calculations that have been performed.

Heading back to image 3.9 where is depicted the basic work flow during packet differentiation, we can clearly see that the information extracted from the packet is divided in two main objects. The first, is the well known tuple defining the flow unique identification, and the rest being the information required for statistic calculations called "Packet information". At this point, the thread responsible for that job, enters a mutual exclusion region, questions the flow table and the algorithm progresses in one of two ways. If the flow is not already indexed, structures depicted in listing 3.2 are created, and all its dynamic fields are created and properly initialized, then the object is inserted in the flow table along with a reference to the list previously referred for posterior iteration. Right after, the flow indexing process, its information and a pointer to the flowid itself is sent to a function where the statistics are updated. The second way is similar to the first but this time the flowid already exists and

the object retrieved from the flow table is passed to the update statistic function as well. For the statistics to work we have had to modulate and develop a way to distinguish whether a specific flow would be considered download or upload. For that to happen we have used the simplest rule, for convenience and also for efficiency. As a rule, we have assumed that for the solution to work, we need to give the program a network context. Therefore, using that network information we have defined following rule: All the traffic coming from an IP inside our known network is considered upload as it is going to somewhere else outside, as a consequence the opposite is considered download traffic. For that to happen, we have created the network context file depicted in listing 3.3. This is a simple EXtensible Markup Language (XML) file that has to be passed as a parameter at the beginning of the execution, in which we write the information of the current network context we are monitoring on. The format of the file itself is very simple, and quite self explanatory. It basically works based on tags attributes rather than creating a mesh of nodes with nested values in it. For each line, we have to provide the version of the IP protocol we want to consider, followed by the subnet mask, finishing with the proper IP address. The contents of the file are loaded in memory, converted in its binary form, and then divided in two distinct lists, one for IPv4 and other for IPv6. Finally, from that point on, for all the packets captured the sense to which those belong to, is calculated by our "netutilities" module. The process is constituted by three basic steps:

1. Iterate over the list containing pairs (network address, subnet mask);
2. Until the end of the list or until reach a verdict, apply bit-wise AND to (Source IP, net mask) pairs;
3. Compare the previous result with network address for equality.

Listing 3.3: Supported format for Network context

```
<?xml version="1.0" encoding="UTF-8"?>
<networks>
    <netaddr versions ="4" mask ="24" addr = "123.34.45.0"/>
    <netaddr versions ="4" mask ="24" addr = "193.136.93.0"/>
    <netaddr versions ="6" mask ="64" addr = "2001:4131:A213:00D0::"/>
    <netaddr versions ="6" mask ="64" addr = "2002:4130:1213:0020::"/>
</networks>
```

As stated before, our main intention was to perform packet inspection followed by a statistical study based on low level statistics. To fulfil that objective we have opted to use sampling of counters in order to have a good notion about what was going on per slot of time.

In diagram 3.18 we have distinct picture of the structures involved in the process and how they interact among each other. In practise and, as can be seen, in listing 3.2, the "Ds_Object" has in its definition a pointer to a data structure called "samplestatistic" which is also depicted in listing 3.4. As most of the times in the C programming language, that pointer variable holds the address to the first position of an array of the same type. Internally, and concerning this specific variable, all the memory requested to it, is allocated dynamically

and zeroed as a way of force counts at zero avoiding possible problems. The "samplestatistic" object is defined to hold values for the counters extracted during time samples. In order to achieve the sampling itself, we used a simple algorithm to decide in which position of the array a certain packet information should be going in, or what information should be updated or not. To solve the problem we needed a pair of timestamps, a spectrum of monitoring and lastly, a dividing constant called Δ to position our counters at the right position within the spectrum.

$$position = \left\lfloor \frac{timeofcreation - currenttime}{\Delta} \right\rfloor$$

From this simple formula results that the field "timeofcreation" in object "Ds_Object" is of extreme importance as way to determine the time when the flow has arrived and the counting has started. The, "currenttime" field comes as way to know the moment when a certain packet was captured, and finally the Δ is used to transform the previous time stamp difference in a mapping to our sampling array.

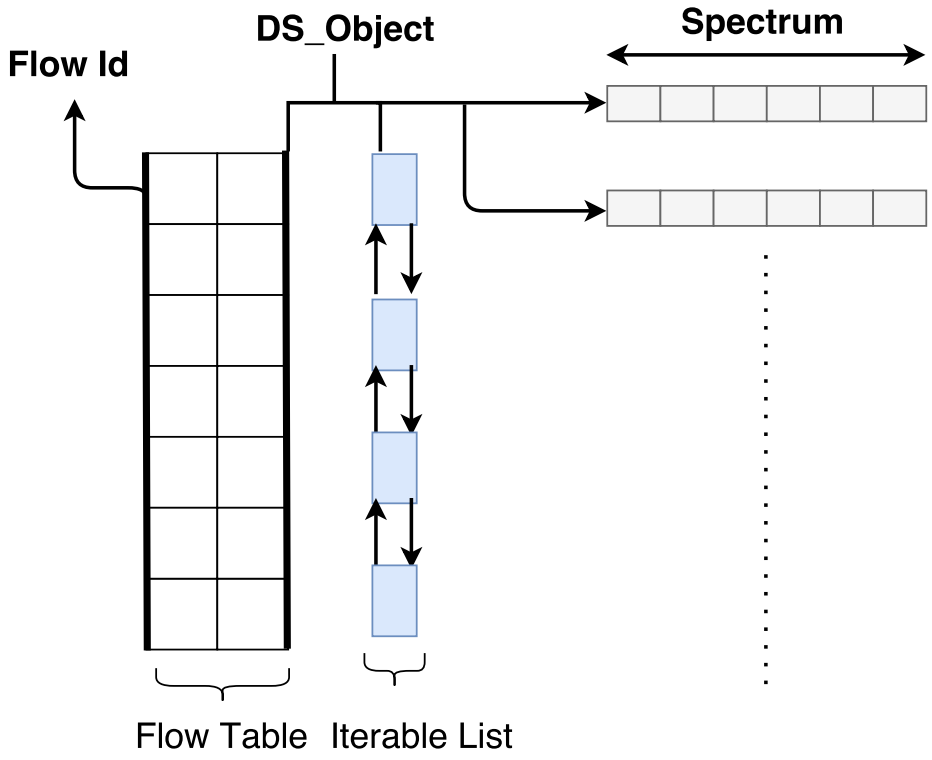


Figure 3.18: Diagram with interactions among structures

Listing 3.4: Object holding the statistics for the sampling

```
/**
 * Structure to hold the connection between the data structure and the data
     for statistical analysis
 */
typedef struct samplestatistic
{
    int npacksUp;        // Number of packets going upwards (upload)
    int npacksDown;      // Number of packets coming downwards (download)
    int avgsizeup;       // Average size of the packets going upwards
    int avgsizedown;     // Average size of the packets going downwards
    int pcksizesumup;    // Sum of the size from the packets going upwards
    int pcksizesumdown; // Sum of the size from the packets going downwards
    int synup;           // Number of "SYN" packets going upwards;
    int syndown;         // Number of "SYN" packets going downwards;
    int resetup;         // Number of "RESET" packets going upwards;
    int resetdown;       // Number of "RESET" packets going downwards;

}samplestatistic;
```

3.5 Memory Crawler

In most long lasting (potential eternal) applications or with a potential huge memory demand. One of the most important topics to discuss is how to make a rigorous and judicious memory management as way of avoiding system overload or great pitfalls in terms of overall system performance.

In our specific case, perhaps, we have a conjunction of the two previous motives. Most of the memory allocations are requested to the system dynamically, therefore the amount of memory requested and the time that the requests will come is completely unknown and dependable of the environment.

For all the reasons previously mentioned a solution to address these problems is particularly important, and must obey three main premises:

1. Periodically and independent action;
2. Mutual exclusive;
3. Reactive in congestion.

Taking in consideration the previous premises we have come up with the conceptual design depicted in image 3.19. For the crawler to work we have created notions called, crawler cycle and entry time out. The crawler cycle defines the period in seconds on which the crawler must be woken up to perform its tasks, the entry time out is used as reference to inform the crawler on the time of inactivity of each entry, so it can decide if the memory needs to be freed or not. Both parameters must be passed as arguments in the beginning of the program.

The mutual exclusion is granted not by the crawler itself but by the context in which it interacts. This means that, the crawler must compete with all the threads trying to perform operations over the data structure, this resulting in a wait by "packet threads" while the crawler performs its actions and doesn't frees the lock. Other important characteristic is

related with the last item on the list above. Under certain conditions, when the system starts to run out of memory, and starts to export blocks of main memory to the disk, performing swapping; One possible symptom is the failure of the memory allocation functions, such as, malloc and calloc. When this happens, the callback function, responsible to allocate memory when packet information is received from the NIC and queue the packet for future processing, has a mechanism to communicate with the crawler, start it up and force it to free memory that might be still being used.

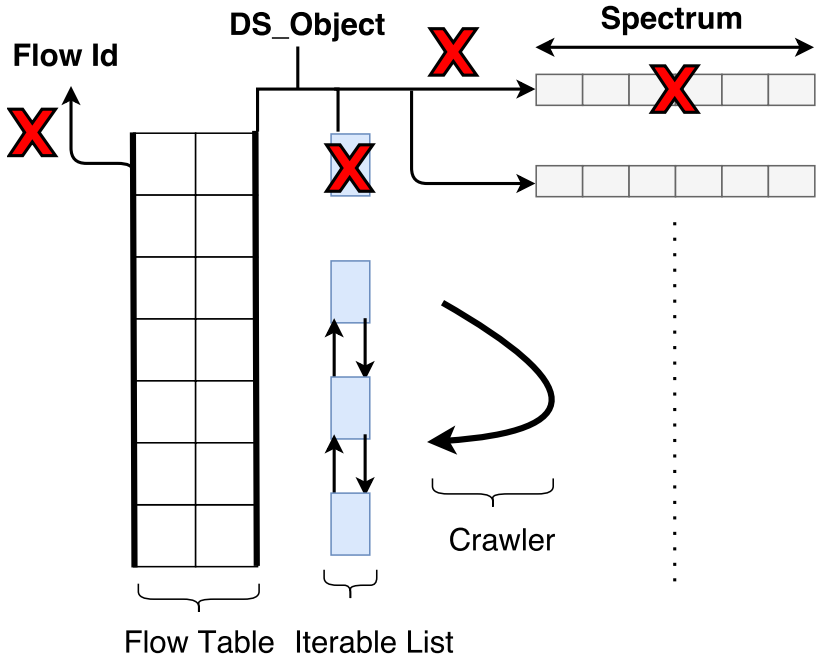


Figure 3.19: Diagram depicting how memory is freed

The crawler, has been implemented in a very simple way, the basic concept relies on a single thread that is created at the beginning, and then intercalating periods of activity with periods of sleep, dependable on the crawler cycle defined. As can be seen in image 3.19 the crawler life cycle is executed following this steps:

1. Compete for the lock to access data structure;
2. Iterate over the iterable list;
3. For each node compare the "lastinserted" field with the entry timeout;
4. Considering the result, decide to free memory or not;
5. In the end, of both IPV4 and IPV6 iteration get back to sleep.

The memory liberation steps are performed in the opposite way of the creation in order to avoid hierarchical failure of referenced memory at liberation process.

Chapter 4

Classification Modules

In this chapter we aim to discuss our classification module along with other developed module, where we perform some superficial experimental evaluation on sampled Internet traffic in order to visualise patterns among flows. We will be starting with a description of mathematical tools used in each module for the analysis of the frequency components of signals and time-series. Finally, we will give some insights on the basic algorithms used for flow classification. Finally, we intend to give an overview over the results obtained during the last steps of this work. Those results are going to be specially focused on the analysis of the classification modules and their efficiency in our architecture, as other results were explained in previous sections.

4.1 Silence Analysis

Our classification module has been developed using very simple statistical and counting methods. As described in chapter 3, we have used time sampling as a way to perform the job of counting the packets per time slot on each flow. Taking that development option into consideration, we have understood its implications by, mainly, proceeding to packet captures in different places with different bandwidth availability, and sooner, a clear bandwidth influence was noticed in terms of packet count per bin. Thus, from that clear characteristic, we have thought of a method inspired in inter-arrival counting analysis. Actually, from that set of observations, we have concluded that, more important than the amount of packets coming in or going out in a specific time frame, other, more important parameter would be the silence count, or the absence of packets in a specific time bin.

Using the principles and ideas just described, we went on to develop a simple, proof of concept, module to test the system integration and, as a consequence, the success rate of each classification. Before the implementation itself, we have decided to take on a simple and self contained scenario where we could test the algorithms behind the concept. That self contained scenario was chosen and is based on three specific traffic patterns that we found accessible and important to try the classification. As so, taking in consideration our previous remarks in chapter 1, about video traffic growth, "youtube video","youtube live streaming" and, finally, standard browsing traffic (originated by regular page visiting) were the types of traffic chosen for the analysis. For that purpose, a set of ten characteristic flows, corresponding to each type of traffic, were generated. Those flows were captured, processed and returned the sampling vector requested, which has been saved in text files. As a matter of convenience, the sampling

files were processed using, a Python script that has been written for that purpose.

At this point, we have come to the definition of silence in sampling, as a matter of simplicity, and considering that we have used bins at the magnitude of a second, we have considered the following relation: $silence = bincount < 10$.

The Python script, worked as a ground truth builder, from which, we have generated the statistic indicators depicted in XML format in listing 4.1.

Listing 4.1: Supported format for types of traffic.

```
<?xml version="1.0" encoding="UTF-8"?>
<trafficcenters>
    <center traffictype ="Video" mean ="70.5" variance = "17.13"/>
    <center traffictype ="Browsing" mean ="104.38" variance = "23.48"/>
    <center traffictype ="Youtube Live" mean ="36.14" variance = "14.40"/>
</trafficcenters>
```

Right after this process, we have started to think of a way to make use of the information calculated, n order to proceed to a classification. The analysis of listing 4.1 has suggested us the approach to a "Cartesian Coordinate System", as (mean,variance) would represent a point in a two dimension axis system. Therefore, using that idea we went on to write an algorithm on those basis, using, once again, purely C. The sort of, mental mapping that we have come up with is depicted in image 4.1. In practical terms, the wider points represent the position of the three applications that we want to classify, while the centre point represents the flow needing classification.

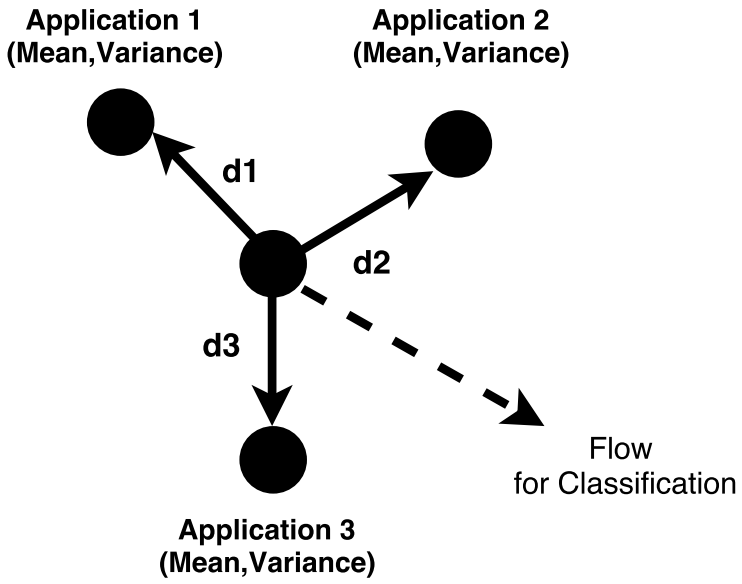


Figure 4.1: Distance diagram

The algorithm for each flow, uses the sampling array calculated previously, counts the number of silences on that specific flow, and then performs the calculation of distance between each point and the generated point, using Euclidean distance, in the way it is defined. If $u = (x_1, y_1)$ and $v = (x_2, y_2)$ are two points on the plane, their *Euclidean distance* is given

by:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{4.1}$$

Geometrically, it's the length of the segment joining u and v, and also the norm of the difference vector. If $a = (x_1, x_2, \ldots, x_n)$ and $b = (y_1, y_2, \ldots, y_n)$, then formula 4.1 can be generalized to ^{n} by defining the Euclidean distance from a to b as

$$d(a, b) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}. \tag{4.2}$$

The last step on the classification algorithm is used to extract probabilities of classification for each flow, performing a normalization in the following manner.

$$Pds = \frac{\frac{1}{ds}}{\sum_{i=0}^{3}(\frac{1}{di})}. \tag{4.3}$$

Finally, the process returns an array of probabilities, in which we only consider success on probabilities higher than 50%.

4.2 Multi-Scale Analysis Tools

Another approach that we have tested is based in signal processing tools as a way to explore the spectral sampling already calculated. This curiosity was inspired by a very good Phd work on traffic classification, where the author exposes some mathematical tools to traffic patterns. [11]. As mentioned in several bibliography [11],[67],[68] Fourier Transform (FT) and Wavelet Transform (WT) are widely used in signal processing and compressing. Being each one of them widely used from a few years now.

4.2.1 Fourier Transform

As Mallat states the "indisputable hegemony of the Fourier transform" with its reliance on a time-invariant operators[68] FT's are the most widely used technique for analyzing the frequency spectrum of a stochastic process, doing it, by decomposing it into complex exponential functions possessing each one of them different frequencies[67]. Formerly being represented by:

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t}. \tag{4.4}$$

Where ω refers to the frequency of the sinusoid. Since the support of the sinusoid is not localized, FTs have a poor time resolution and are only suitable for the analysis of stationary signals, i.e, signals with the same frequency component in the whole analysis spectrum. Consequently, FTs are unable to provide time-frequency representation, where the different frequency components of a non-stationary process are depicted together with the time-intervals where they occur. Therefore, time-varying or transient signals require other analysis tools.[68][67]

4.2.2 Wavelet Transform

Taking in consideration that, in many cases, time-frequency representation restrictions might be respected by the analyzed data and an accurate decomposition can be achieved. However, Internet traffic which is very well known to be non-stationary presenting potential drastic variations in either time and frequency. WTs, by assuming non-stationarity, are able to provide a time-frequency representation of a signal and are widely applied in many different areas. In fact, the usefulness of wavelets can be phrased as "the ability to match that intuition with mathematical rigor" [69].

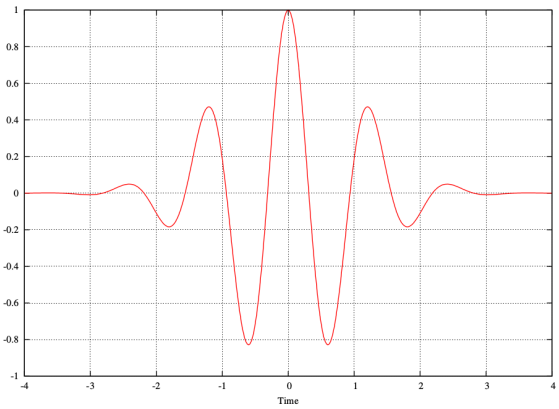


Figure 4.2: Morlet Wavelet example [11]

Wavelets are mathematical functions that are used to divide a given signal into its different frequency components. They were introduced in 1980 by J. Morlet to perform signal decomposition and approximation, and consist of a short duration wave-like oscillation with a limited amplitude, occurring during a short period of time which gives it a good time and frequency resolution. Wavelets enable the analysis of each one of the signal components in an appropriate scale and present several advantages over other signal analysis techniques, such as Fourier Transforms. Since wavelets, as said before, are a good instrument that allows the analysis of any process in both time and frequency domains. Using the mother wavelet ψ, and the set of child functions, denominated the wavelet daughters, one can map the original time series into a function of its daughters τ and s WT's are able to provide us with a representation of that series in the time and frequency domains. A Wavelet Scalogram can be defined as the normalized energy $\hat{E}_x(\tau, s)$ over all possible translations (set T) in all analyzed scales (set S), and is computed as such [11]:

$$\hat{E}_x(\tau, s) = 100 \frac{\left|\Psi_x^{\psi}(\tau, s)\right|^2}{\sum_{\tau'} \in T \sum_{s'} \in S \left|\Psi_x^{\psi}(\tau', s')\right|^2} \tag{4.5}$$

Our approach regarding this tool was a simply exploring in search for interesting results. Our basic algorithm depends of a very small and efficient library, written by Professor Paulo Salvador. With which we calculate the "ground truth" wavelet and the unclassified flow wavelet proceeding to the subtraction of both waves, analyzing the result looking for similarities.

4.3 Silence Module Results

In this sub section, we are going to provide the results for the proof of concept experiences that had been performed concerning the classification steps.

The results are going to be displayed in terms of time per classification and in the case of the Euclidean distance Method, we are going to provide insights on the classification accuracy study. In each one of the plots presented, we expect to give a proper context to each experience and provide comments about the results obtained.

Right now, we are going to start by analysing the results concerning time per classification. The experience depicted in plot 4.3 was performed respecting the following protocol :

1. Select ten packet traces for each one of the types evaluated;
2. Process each one of them consecutively;
3. Repeat the process 100 times;
4. Repeat the previous two steps for ten different spectrum lengths;
5. Calculate mean and variance for each one of the cases.

For this experience the results have been interesting in terms of the continuity regarding the mean, as its value, does not seem to be varying that much. We can clearly see some ups and downs along the graph but we can deduct an approximation for linear complexity. One peculiar fact has been indeed the big numbers in terms of standard deviation. For a matter of readability we have divided the presented values in $\frac{1}{4}\sigma$. In conclusion this approach has come up in average, as a very reliable approach, even thought its values can vary greatly.

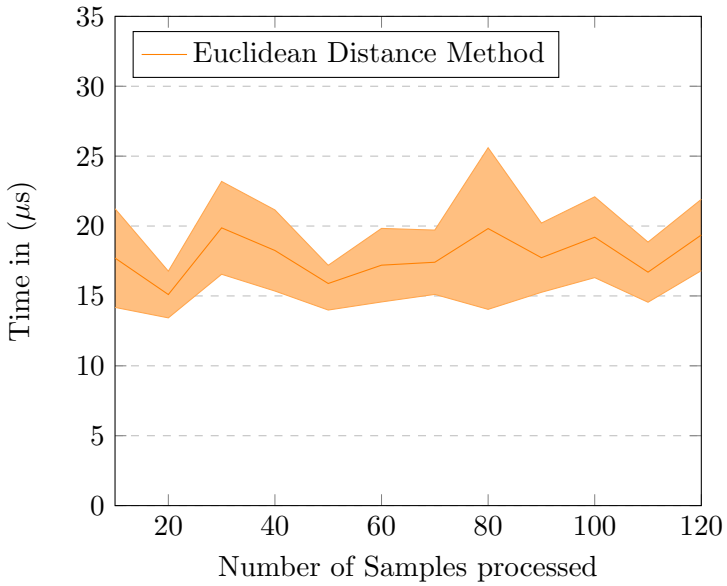


Figure 4.3: Plot for the Euclidean Distance simulation for different number of elements.

The experience performed and depicted in image 4.4 followed a protocol as follows:

1. Another selection of 10 traces per type of packet;
2. Usage of the previous ground truth for classification;
3. Run the acquired traces against the ground truth;
4. Repeat the process for five different spectrum lengths, recalculating centres each time;

From the graphic observation, we can clearly see, rather positive results, and one interesting phenomena that happens with Video and Streaming lines both having an opposite behaviour, forming a hole in the middle resulting of their crossing. The results related with Browsing stand up from the the others because of its perfect score in classification, the simpler explanation is the fact that we have created a proof of concept system with only three types of traffic in which, browsing stands up by difference from the others. Considering, video and live streaming, their classification score is very interesting and can be justified because of their similarity in the beginning of each flow. Live Streaming tries to keep packets rhythm every time, while "normal" video starts strong, and showing some periods of silence as time goes by.

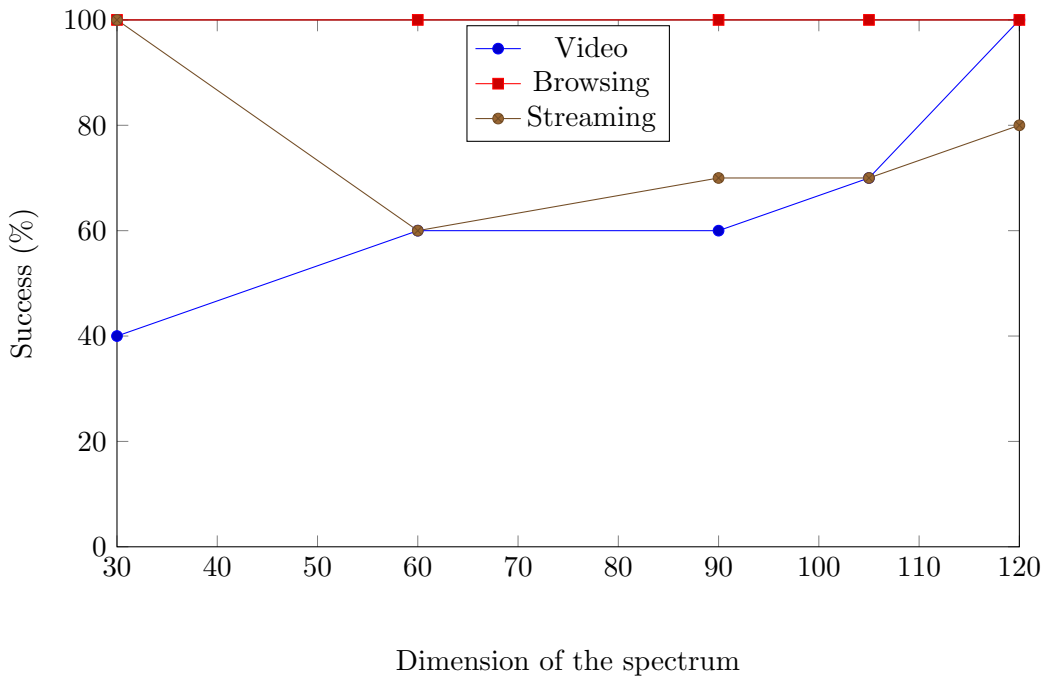


Figure 4.4: Results of the classification for different data lengths

4.4 Wavelet Module Results

Considering the wavelet calculation approach, we have understood that, this type of calculations tend to be very complex and take time that can go up to two orders magnitude when compared to the Euclidean Distance approach. This time, we have made the exactly same thing by cutting the standard deviation by a quarter for the purpose of readability. Analyzing the results we can clearly state that wavelet calculations tend to vary greatly, being hard to predict, even though the results came up better when compared to the Euclidean distance method.

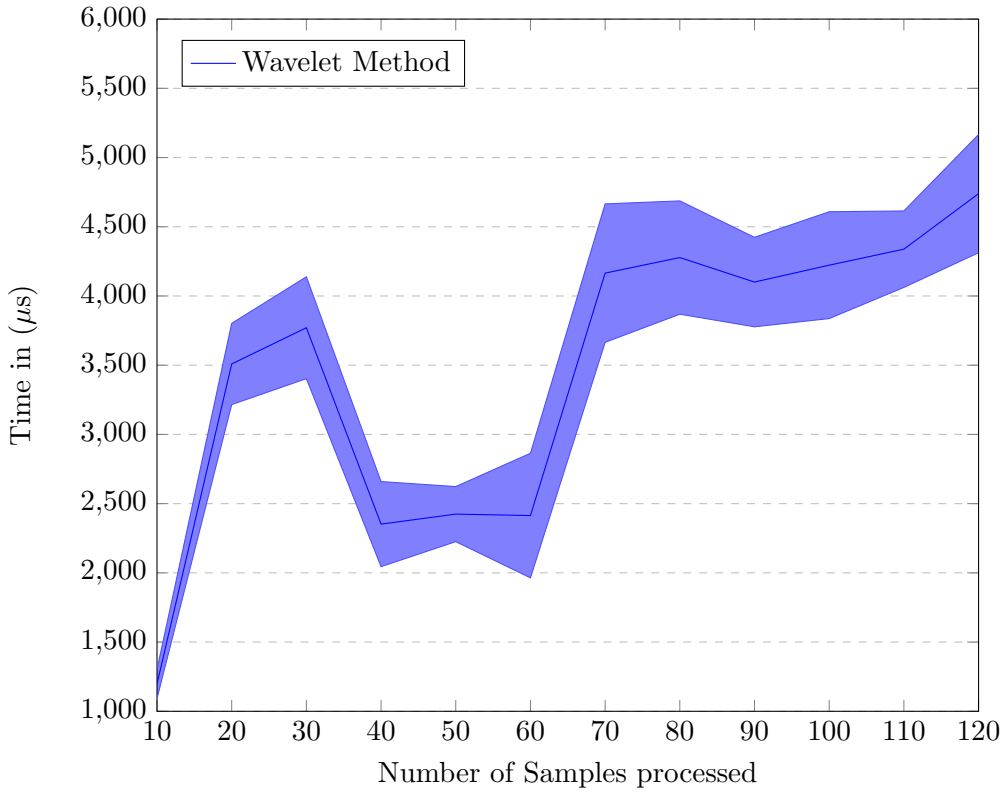
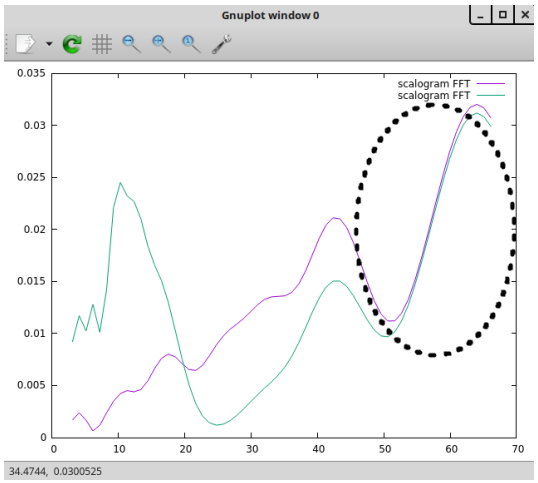
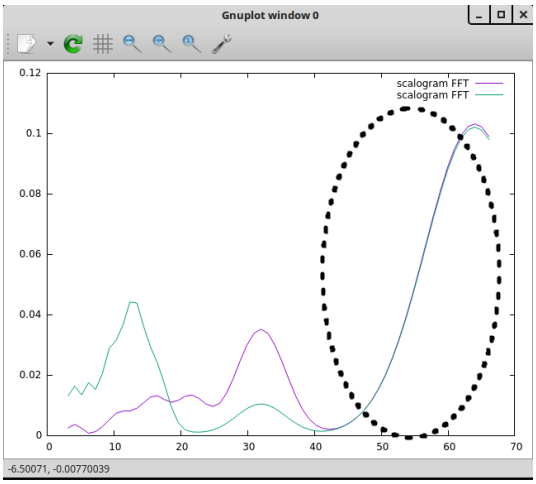


Figure 4.5: Plot for the Wavelet simulation for different number of elements.

The experiment performed with wavelets returned very interesting results, that were useful to corroborate the previous work that served as an inspiration to this dissertation [11]. As referenced in the last chapter, wavelets appear has a great mathematical tool to detect events with the similar energy levels (y axis) along the the frequency space (x axis). The experience performed, depicted in figure 4.6 was conducted as a way to figure out its potential. Both signals have been obtained by simple browsing in the web reading articles in an on-line newspaper. The signaled zone in both pictures, show the characteristics shared by each signal in any particular frequencies. Actually, the results related with the direct analysis of the plot are clear enough to see the common properties, as both signals tend to approach each other in energy values and in their monotony behaviour.



(a) Scalogram calculation with 120 s data



(b) Scalogram calculation with 60 s data

Figure 4.6: Traffic behaviour depicted in wavelets

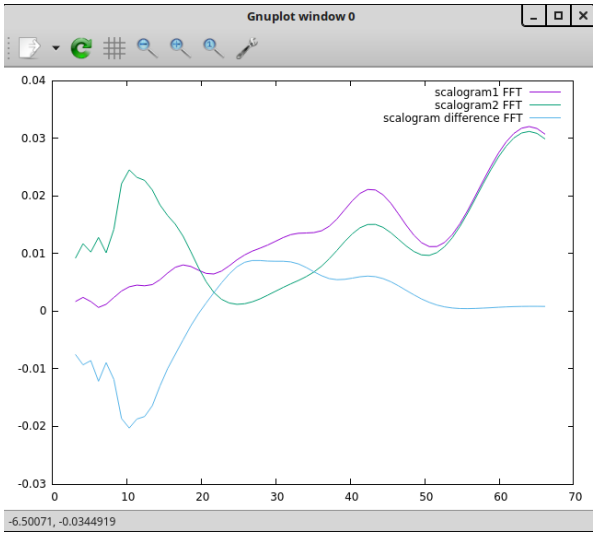


Figure 4.7: Scalogram Subtraction

Our approach to identify this type of events have been a basic subtraction of waves. By performing the subtraction we get a result similar to a straight line around the zero level of energy, as can be seen in image 4.7. As explained before, we didn't go further than this but we believe this is worth of further investigations.

Chapter 5

Conclusion and Future Work

The emergence of the Internet as the greatest communication platform that comprises a great number of services, applications and devices, triggered the need to perform traffic analysis and user profiling. Several approaches were referred to in Chapter 2. Despite their validity and scientific importance, the increasing capacity of the networks, and the increasing complexity of Internet applications, together with disguised traffic, have made the classification task much harder. The outcome of this dissertation proposed another kind of approach to monitoring, processing of packet's information and traffic classification, using basic low level statistics. We were able to distinguish a small set of different applications, considering their specific traffic. Concerning the packet monitoring, the performance has been measured and a few data structures have been benchmarked for efficiency in terms of processing time and main memory usage. We have evaluated the packet processing viability, and we believe to have found an acceptable point of compromise between performance, resources allocation and traffic prediction accuracy. Regarding traffic classification, we have been able to formulate simple rules for traffic classification, giving a small contribution on that subject. The overall system has been tested and benchmarked, depicting stability during consecutive runs, even during stress tests, generating flows with tens of thousands of packets at different packet rates.

Considering that the main objectives of this dissertation were accomplished, there are a few limitations linked with the solution, and there is a lot of potential to grow the concept and add new features. Addressing the actual limitations, perhaps, its main flaw is an intrinsic limitation of our approach, in the sense that a real-time approach was neglected in order to obtain better traffic distinction. Another limitation would be the lack of testing under "real life" conditions, for instance, in a production server. Another limitation is definitely the number and the variety of applications supported. We believe that the work developed in the scope of this dissertation has been a good point for further investigations on the area.

Taking these into consideration and talking about future work, one idea would be, the implementation of more complex rules that could eventually support different types of traffic, even using the information that is not being used nowadays. To complement this idea, the implementation of indexes (that started in our work but did not follow through) would, perhaps, be a good way to go. A more profound study of real-time approaches to design a system tailored for that paradigm, but also to provide better classifications earlier. Another challenge would be the usage of Data mining and Machine Learning to work on low level statistics and provide relative intelligence to the application, allowing autonomous reactions

to suspicious traffic. Finally, in human interaction perspective, another good ways to go would be related with delivery of structured reports, important and valuable resource, to network managers, or even the develop of alarming systems, to give proper insights on the fly, to understand what is going on in the network, in a wider time frame.

Bibliography

[1] sandvine Intelligent Broadband Networks. Global internet phenomena spotlight: Encrypted internet traffic. Website, 10 2016. last checked: 14.10.2016.

[2] Google. Transparency report. Website, 10 2016. `https://www.google.com/transparencyreport/https/` last checked: 14.10.2016.

[3] Intel Corporation. Intel dpdk, data plane development kit. Website, 9 2016. `http://dpdk.org/` last checked: 30.09.2016.

[4] Luigi Rizzo. Revisiting network i/o apis: The netmap framework. *Commun. ACM*, 55(3):45–51, March 2012.

[5] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han. Dfc: accelerating string pattern matching for network applications. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 551–565, 2016.

[6] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. Enhancing tor's performance using real-time traffic classification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 73–84. ACM, 2012.

[7] Shichao An. Shichao notes. Website, 11 2016. `https://notes.shichao.io/tcpv1/ch5/` last checked: 30.11.2016.

[8] bigocheatsheet Contributors. bigocheatsheet. Website, 11 2016. `http://bigocheatsheet.com/` last checked: 4.11.2016.

[9] Robert Sedgewick. *Algorithms in C*. Peter S. Gordon, 3rd edition, 1998.

[10] Andrea Mazzoleni. Tommy data structures. Website, 10 2016. `http://www.tommyds.it` last checked: 31.10.2016.

[11] Eduardo Rocha. *Methodologies for Traffic Profiling in Communication Networks*. PhD thesis, University of Aveiro, 2011.

[12] David Moore, Ken Keys, Ryan Koga, Edouard Lagache, and Kimberly C Claffy. The coralreef software suite as a tool for system and network administrators. In *Proceedings of the 15th USENIX conference on System administration*, pages 133–144. USENIX Association, 2001.

[13] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: Multilevel traffic classification in the dark. *SIGCOMM Comput. Commun. Rev.*, 35(4):229–240, August 2005.

[14] Hyun-Min An, Myung-Sup Kim, and Jae-Hyun Ham. Application traffic classification using statistic signature. In *APNOMS*, pages 1–6, 2013.

[15] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. Key challenges in cloud computing: Enabling the future internet of services. *IEEE Internet Computing*, 17(4):18–25, July 2013.

[16] Wayne Jansen, Timothy Grance, et al. Guidelines on security and privacy in public cloud computing. *NIST special publication*, 800(144):10–11, 2011.

[17] Dimitrios Zissis and Dimitrios Lekkas. Addressing cloud computing security issues. *Future Generation Computer Systems*, 28(3):583 – 592, 2012.

[18] Atif Nazir, Saqib Raza, Dhruv Gupta, Chen-Nee Chuah, and Balachander Krishnamurthy. Network level footprints of facebook applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 63–75, New York, NY, USA, 2009. ACM.

[19] Riley Walters. Cyber attacks on us companies in 2014. *Heritage Foundation Issue Brief*, (4289), 2014.

[20] Michael Finsterbusch, Chris Richter, Eduardo Rocha, Jean-Alexander Muller, and Klaus Hanssgen. A survey of payload-based traffic classification approaches. *IEEE Communications Surveys & Tutorials*, 16(2):1135–1156, 2014.

[21] Milton L Mueller and Hadi Asghari. Deep packet inspection and bandwidth management: Battles over bittorrent in canada and the united states. *Telecommunications Policy*, 36(6):462–475, 2012.

[22] Milton Mueller and Andreas Kuehn. Einstein on the breach: Surveillance technology, cybersecurity and organizational change. In *12th Workshop on the Economics of Information Security (WEIS 2013), Georgetown University, Washington, DC June*, pages 11–12, 2013.

[23] Barbara Miller Steve Augustino. Court rules for isp in deep packet inspection lawsuit. Website, 11 2013. last checked: 19.11.2016.

[24] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan. Analyzing the potential benefits of cdn augmentation strategies for internet video workloads. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 43–56, New York, NY, USA, 2013. ACM.

[25] Xi Liu, Florin Dobrian, Henry Milner, Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. A case for a coordinated internet video control plane. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 359–370, New York, NY, USA, 2012. ACM.

[26] Charu Gandhi, Gaurav Suri, Rishi P Golyan, Pupul Saxena, and Bhavya K Saxena. Packet sniffer–a comparative study. *International Journal of Computer Networks and Communications Security*, 2(5):179–187, 2014.

[27] M. A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal. Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration. In *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*, pages 74–78, Nov 2015.

[28] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[29] Diógenes Luis B de Freitas, Kaio RS Barbosa, and Eduardo Feitosa. Mydnsdump: Uma ferramenta para mediçao do tráfego dns.

[30] Iain R Learmonth, Brian Trammell, Mirja Kuhlewind, and Gorry Fairhurst. Pathspider: A tool for active measurement of path transparency. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 62–64. ACM, 2016.

[31] Shane Alcock, Perry Lorier, and Richard Nelson. Libtrace: A packet capture and analysis library. *SIGCOMM Comput. Commun. Rev.*, 42(2):42–48, March 2012.

[32] Sebastian Gallenmuller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. Comparison of frameworks for high-performance packet io. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 29–38. IEEE, 2015.

[33] Tcpdump and libpcap. Tcpdump and libpcap. Website, 10 2016. `http://www.tcpdump.org/` last checked: 4.10.2016.

[34] Wireshark Foundation. wireshark. Website, 10 2016. `https://www.wireshark.org/` last checked: 4.10.2016.

[35] Luis Martin Garcia. Programming with libpcap-sniffing the network from our own application. *Hakin9-Computer Security Magazine*, pages 2–2008, 2008.

[36] Luca Deri et al. Improving passive packet capture: Beyond device polling.

[37] Troy D. Hanson. Uthash. Website, 11 2016. `http://troydhanson.github.io/uthash` last checked: 1.11.2016.

[38] Doug Baskins. Judy array. Website, 11 2016. `http://judy.sourceforge.net/index.html` last checked: 1.11.2016.

[39] Google. Sparsehash. Website, 11 2016. =https://github.com/sparsehash/sparsehash last checked: 2.11.2016.

[40] Ajay Chaudhary and Anjali Sardana. Software Based Implementation Methodologies for Deep Packet Inspection. *2011 International Conference on Information Science and Applications*, pages 1–10, 2011.

[41] QOSMOS. Classification and metadata engine. Website, 10 2016. http://www.qosmos.com/products/deep-packet-inspection-engine/ last checked: 7.10.2016.

[42] Cisco. Next generation nbar. Website, 10 2016. http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/network-based-application-recognition-nbar/qa_c67-697963.html last checked: 7.10.2016.

[43] PACE. Pace 2. Website, 10 2016. https://www.ipoque.com/products/pace last checked: 7.10.2016.

[44] Tomasz Bujlow, Valentín Carela-Español, and Pere Barlet-Ros. Independent comparison of popular {DPI} tools for traffic classification. *Computer Networks*, 76:75 – 89, 2015.

[45] Shane Alcock and Richard Nelson. Libprotoident: traffic classification using lightweight packet inspection. *WAND Network Research Group, Tech. Rep*, 2012.

[46] PACE. Pace 2. Website, 10 2016. last checked: 7.10.2016.

[47] Chaofan Shen and Leijun Huang. On detection accuracy of l7-filter and opendpi. In *2012 Third International Conference on Networking and Distributed Computing*, pages 119–123. IEEE, 2012.

[48] Cristina Rottondi and Giacomo Verticale. Using packet interarrival times for internet traffic classification. In *2011 IEEE Third Latin-American Conference on Communications*, pages 1–6. IEEE, 2011.

[49] IANA. Service name and transport protocol port number registry. Website, 10 2016. last checked: 11.10.2016.

[50] Hyunchul Kim, KC Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos, and KiYoung Lee. Internet traffic classification demystified: Myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 11:1–11:12, New York, NY, USA, 2008. ACM.

[51] A. Madhukar and C. Williamson. A longitudinal study of p2p traffic classification. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 179–188, Sept 2006.

[52] Min Zhang, Wolfgang John, KC Claffy, and Nevil Brownlee. State of the art in traffic classification: A research review. In *PAM Student Workshop*, pages 3–4, 2009.

[53] Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th international conference on World Wide Web*, pages 512–521. ACM, 2004.

[54] Matthew Roughan, Subhabrata Sen, Oliver Spatscheck, and Nick Duffield. Class-of-service mapping for qos: A statistical signature-based approach to ip traffic classification. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 135–148, New York, NY, USA, 2004. ACM.

[55] W. Jiang and M. Gokhale. Real-time classification of multimedia traffic using fpga. In *2010 International Conference on Field Programmable Logic and Applications*, pages 56–63, Aug 2010.

[56] J. But, P. Branch, and T. Le. Rapid identification of bittorrent traffic. In *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, pages 536–543, Oct 2010.

[57] Niels Provos and Nick Mathewson. libevent—an event notification library, 2003.

[58] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, pages 19–24, 2003.

[59] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[60] Taís Borges Ferreira, Rivalino Matias Jr, and Autran Macêdo. Análise de desempenho de alocadores de memória de código aberto: Estudo de caso em aplicações middleware.

[61] Chuck Lever and David Boreham. Malloc() performance in a multithreaded linux environment. 2000.

[62] Aniruddha Bohra and Eran Gabber. Are mallocs free of fragmentation? In *USENIX Annual Technical Conference, FREENIX Track*, pages 105–117, 2001.

[63] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, April 2002.

[64] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, Mar 2001.

[65] S. Nilsson and G. Karlsson. Ip-address lookup using lc-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, Jun 1999.

[66] Per-Ake Larson. Dynamic hash tables. *Commun. ACM*, 31(4):446–457, April 1988.

[67] Pedro A. Morettin. *From Fourier to Wavelet Analysis of Time Series*, pages 111–122. Physica-Verlag HD, Heidelberg, 1996.

[68] Stéphane Mallat. *A wavelet tour of signal processing*. Academic press, 1999.

[69] Esse Connell. Wavelet transforms and signal processing. Paper.