University of Pisa and Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

DPI over commodity hardware: implementation of a scalable framework using FastFlow

Candidate

Daniele De Sensi

Supervisors

Prof. Marco Danelutto
Dr. Luca Deri

Academic Year 2011/2012

Alla mia famiglia

"Breathe, breathe in the air
Don't be afraid to care"

PINK FLOYD

Ringraziamenti

A pochi giorni dalla fine di questo percorso, non posso fare a meno di pensare a tutte le persone che mi sono state accanto in questi anni e che mi hanno aiutato, ognuno a suo modo, a raggiungere questo traguardo.

Innanzitutto esprimo la mia profonda gratitudine al Prof. Marco Danelutto per la disponibilità e la fiducia che mi ha sempre dimostrato. Il suo supporto, formativo e morale, mi ha permesso di svolgere al meglio il mio lavoro di tesi.

Ringrazio inoltre il Dott. Luca Deri e il Dott. Massimo Torquati per l'aiuto fornitomi durante lo svolgimento della tesi e per essere sempre stati disponibili a risolvere i miei dubbi.

Vorrei poi ringraziare LIST S.p.A., per avermi dato la possibilità di approfondire le mie conoscenze, grazie alla borsa di studio sul tema "*Nuove architetture hardware-software ad alte prestazioni*", conferitami durante il periodo compreso tra Febbraio e Luglio 2012.

Il ringraziamento più sentito lo rivolgo a mio padre, mia madre, Fabri e Davide. Senza la loro guida tutto questo non sarebbe stato possibile. Il loro supporto e la loro pazienza mi hanno sempre accompagnato in tutti questi anni. Inoltre ringrazio gli zii, i nonni, i cugini e tutti coloro che mi hanno sempre fatto sentire la loro vicinanza anche se a chilometri di distanza.

Un ringraziamento speciale va a Mari, che mi è stata accanto sin da quando ho iniziato questo cammino. La sua presenza e il suo appoggio sono stati per me di fondamentale importanza.

Ringrazio poi Andre, Cica e Picci, che hanno condiviso con me questi mesi di tesi. Tra una battuta e l'altra ci siamo sorretti a vicenda e siamo riusciti a sorridere anche di fronte a situazioni sfavorevoli.

Un affettuoso ringraziamento a coloro che ho conosciuto in questi ultimi due anni e con i quali ho passato alcuni dei momenti più piacevoli di questo percorso universitario: Alessio, Bob, Davide, Ema, FraPac, Frenci, Gian, Luigi, Simo, Sina, Tixi e Tudor.

Grazie a Bice, Francesca, Nicola, Roberta, Simone e Stefano, per le serate

passate insieme e per l'affetto che mi hanno sempre dimostrato.

Un pensiero sentito va infine a Maria e Gianluca per la loro sincera amicizia.

A loro tutti va il mio sincero ringraziamento.

Daniele

a89bc677da53e61abf3153299ecb935a

Contents

List of acronyms

IP	Internet Protocol
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
P2P	Peer to Peer
VoIP	Voice over IP
ISP	Internet Service Providers
DPI	Deep Packet Inspection
NI	Network Intelligence
MTU	Maximum Transmission Unit
FPGA	Field Programmable Gate Array
CAM	Content Addressable Memories
IPS	Intrusion Prevention System
IDS	Intrusion Detection System
SPSC	Single Producer Single Consumer
NIC	Network Interface Controller
FIFO	First In First Out
OSI	Open Systems Interconnection
URL	Uniform Resource Locator
GPU	Graphics Processing Unit
MPI	Message Parsing Interface
OpenMP	Open Multiprocessing
CUDA	Compute Unified Device Architecture
OpenCL	Open Computing Language
STL	Standard Template Library

List of Figures

Introduction

In the last years we assisted to a large increase of the number of applications running on top of IP networks. Consequently has increased the need to implement very efficient network monitoring solutions that can manage these high data rates.

Network administrators always deal with the problem of classifying and analyzing the type of traffic is traveling on their networks. This can be functional to multiple purposes, like:

Network security We assisted recently to an increase in the complexity of the applications which run on top of IP networks. Consequently, we have seen a shift from so-called "*network-level*" attacks, which target the network the data is transported on (e.g. Denial of Service), to content-based threats which exploit applications vulnerabilities and require sophisticated levels of intelligence to be detected. Many of these threats are designed to bypass traditional firewalls systems and often they cannot be detected by antivirus scanners [2]. Accordingly, it is no more sufficient to have only a software solution on the client side but we also need to run some controls on the network itself. These types of controls needs to identify the application protocol carried inside the packet and possibly to analyze its content in order to detect a potential threat.

Quality of Service and Traffic shaping Another situation in which protocol classification may be useful is when network administrators may want to limit the transmission of packets that might degrade overall network performance [3, 4]. For example, the streaming of large amounts of data for an extended period of time (like the one performed by Peer to Peer applications) may degrade the other users' experience on the network. Consequently, to improve the quality of service, traffic classification engines may identify the problematic applications and then reduce their priority level. Alternatively, it could be used in the opposite way, ensuring an higher priority to sensitive applications like Voice Over IP (VoIP) or video streaming.

Data leak prevention The analysis of the data traveling over an enterprise network may be useful to detect potential data breach or ex-filtration transmissions and to prevent and block them. In data leakage incidents, sensitive data is disclosed to unauthorized personnel either by malicious intent or inadvertent mistake. These data may include: private or company information, intellectual property, financial or patient information, credit-card data, and other information depending on the business and the industry [5]. This is a very felt issue since the leakage of sensitive data can lead an organization to face with criminal lawsuits [6] which could then cause the bankrupt of the company or takeovers by larger companies. Moreover, leakage of personal data may involve every citizen, since their data are stored electronically by hospitals, universities and other public organizations.

Network access control Traffic classification and inspection mechanisms may be used by network administrators or Internet Service Providers (ISPs) to ensure that their acceptable use policy is enforced, allowing the access to the network only to some kind of traffic. This can be used, for example, for parental control [7] or to limit the access to the network in order to improve employees productivity [8].

For all these tasks, identification of the application protocol (e.g. HTTP, SMTP, SKYPE, etc...) is required and, in some cases, extraction and processing of the information contained in its payload is needed.

However, in order to be able to identify the protocol, is no more sufficient to only look at TCP/UDP ports because protocols often run on ports different from the assigned ones.

To tackle these problems, in the recent years, Deep Packet Inspection (DPI) technology has emerged. Differently from classic monitoring solutions [9] that classify traffic and collect information only based on the <Source IP, Destination IP, Source Port, Destination Port, L4 Protocol> tuple, DPI inspects the entire payload to identify the exact application protocol.

In most cases, protocol identification is mainly achieved by comparing the packet payload with some well-known protocols patterns or, if the protocol is encrypted and we accept to have a reduction in the accuracy, by using some statistical knowledge about the protocol [10, 11, 12].

However, considering the current networks rates, this kind of processing is not suitable for offline analysis, which would require to store the packets traces for further elaboration. Therefore, we need to manage the incoming packets as soon they arrive, without relying on the possibility to store the packets that we are not able to manage for later processing.

Moreover, some protocols like HTTP are so extensively used [13] that we may want to analyze the carried content to classify it in more sub protocols (e.g. Facebook, Gmail, Netflix and others can all be viewed as HTTP sub protocols). Furthermore, as we anticipated, some network management tasks require to extract protocol content and metadata. For this reason, Network Intelligence (NI) technology has been recently proposed [14]. This kind of analysis is built on DPI concepts and capabilities and extends them with the possibility to extract and process content and metadata carried by the protocol.

However, since the network could reorder the messages, to correctly extract the data carried by the protocol, some applications may require to the NI engine to manage the expensive task of IP defragmentation and TCP stream reordering and reassembly. In general, this is not an easy task and needs to be carefully designed since it may be vulnerable to some exploits [15, 16].

For these reasons, and considering the high bandwidth of the current networks, we need highly efficient solutions capable of processing millions of packets per second.

This kind of processing is in many cases implemented, at least in part, through dedicated hardware [17, 18, 19]. However, full software solutions may often be more appealing because they are typically more economical and have, in general, the capability to react faster to protocols evolution and changes.

Furthermore, with the shift from single core to multicore processing elements, it should be possible, in principle, to implement high speed DPI on non-dedicated hardware and, at the same time, to provide performances comparable to those of special purpose solutions.

However, common existing DPI software solutions [20, 21, 22, 23] don't take advantage of the underlying multicore architecture, providing only the possibility to process the packets sequentially. Therefore, when multicores have to be exploited, they demand to the application programmer the complicated and error-prone task of parallelization and fine tuning of the application. Furthermore, many DPI research works that can be found in literature [24, 25, 26, 27] and which exploit multicore architectures are often characterized by a poor scalability, due to the overhead required for synchronization or to the load unbalance among the used cores.

Moreover, in order to be able to manage high data rates, some of these solutions don't inspect the entire payload, using "lightweight" approaches which analyze only few bytes of the packet and are, in general, less accurate than solutions which adopt full packet inspection. Furthermore, these solutions are limited to protocol identification, without providing any mechanism to locate, extract and process the data and metadata contained inside the packets, which are basic requirements for a NI engine.

> The target of this thesis is to explore the possibility to apply *structured parallel programming* theory to support the implementation of an high bandwidth streaming application as the one just described. Using these concepts, we would like to design, realize and validate a DPI and NI framework capable of managing current networks rates using commodity multicore hardware.

Accordingly we will possibly analyze the entire payload in order to reach an high accuracy when identifying the protocol and, at the same time, to give to the application programmer the possibility to specify which protocol metadata to extract and how to process them. However, considering that in general the processing of the extracted data may be computationally expensive, we need to design our framework in such a way that is possible to distribute it over the available processing elements.

> We will show that, using *structured parallel programming* concepts and with an accurate design, we are able to do stateful packet inspection over current network rates using commodity hardware and to achieve a good speedup and results comparable to those obtained by dedicated hardware.

In order to be able to reach our goals, we implemented our framework using the *FastFlow* library [28, 1] which, thanks to its low latency communication mechanisms, allowed us to write an efficient and scalable DPI framework.

Our proof-of-concept framework supports at the moment some of the most common protocols (such as HTTP, POP3, IMAP, DHCP, DNS, MDNS and few others). Moreover, the framework has been designed in such a way that it could be easily extended with new protocols with limited changes in the code.

The rest of the thesis is structured in this way:

- In Chapter 1 we will analyze the context in which this thesis is located, briefly describing some of the techniques commonly used to perform DPI and showing why our work differs from already existing works. Moreover, we will give the definition of *structured parallel programming*, exposing the main features provided by *FastFlow*, the library we used to implement these concepts.

- We will then describe in Chapter 2 the global design of the framework and how, using the concepts introduced in Chapter 1, it efficiently exploits the underlying multicore architecture.

- In Chapter 3 we will discuss the main features of the framework, its internal structure and a sketch of the API offered to the user. We will then describe how we split its parts among the different execution modules, avoiding any type of unnecessary synchronization mechanism in such a way that they can be completely independent from each other.

  Moreover, to validate the framework, we present an application implemented on top of it, which scans all the HTTP messages traveling on the network searching for some specified patterns.

- In Chapter 4 some experimental results will be presented and analyzed, validating our framework and showing that it allows to write scalable DPI applications exploiting the processing power providing by the underlying multiprocessor architecture.

- In Chapter 5 conclusions will be drawn and the limits of the framework will be analyzed. Moreover, we will propose some ideas for possible features which could be added in future to our work.

Chapter 1

Thesis context, related work and tools

In this chapter we will introduce the context of this thesis, analyzing the related work and introducing the concept of *structured parallel programming*, which characterizes our thesis with respect to other existing works. We will then describe *FastFlow*, the library we used to apply these concepts inside the framework.

1.1 Protocol classification

The increasing number of applications running on top of IP networks is making more and more urgent the need to implement very efficient tools for an accurate protocol classification. The ability to identify and classify the packets according to the protocol they carry may be useful for different purposes.

As an example, as far as network security is concerned, in the recent years we have seen a shift from so-called "*network-level*" attacks, which target the network they are transported on (e.g. Denial of Service), to content-based threats which exploit applications vulnerabilities and require sophisticated levels of intelligence to be detected. For some of these threats, it is no more sufficient to have only a software solution on the client side but we also need to run some controls on the network itself [2]. These types of controls needs to identify the application protocol carried inside the packet and possibly to analyze its content in order to detect a potential threat.

For this kind of applications, the accuracy is extremely important, as wrong assumptions on what is happening on the network could lead to nasty effects.

Traffic classification solutions may be roughly divided into two main categories:

Flow based In this case the packets are grouped in flows. A flow is defined as a set of packets with the same <Source IP address, Destination IP address, Source port, Destination port, Transport protocol identifier>. These flows are bidirectional, therefore packets with the key: <w.x.y.z, a.b.c.d, r, s, t> belong to the same flow of the packets with key: <a.b.c.d, w.x.y.z, s, r, t>.

The recognition of the packet is done by using both the information carried by the current packet and the accumulated information about the flow. However this requires to store some kind of data about the previous packets received for the flow (e.g. received bytes, state of the TCP connection and others).

Packet based In this case each packet is analyzed independently from the others and there is no need to store any information inside the classification engine.

Moreover, traffic classification could also be divided according to the type of mechanisms used to identify the protocol carried inside the packets.

Port based This is one of the simplest and most used techniques. It simply tries to classify the protocol according to the ports used by the application. However, this approach exhibits a low accuracy [29, 30]. Indeed, many applications often use ports different from the standard ones or they use dynamic ports which are not known in advance.

Statistical The solutions based on this technique try to identify the protocol using statistical knowledge about the distribution of the packet length or by analyzing the packets interarrival times. Some of these solutions use machine learning techniques to train the engine with previously captured and classified traffic traces to characterize the statistical properties of the specific protocol.

The advantage of this approach is that it doesn't need to look the packet payload at all. Despite this can be very useful in presence of encrypted protocols, it may not be the best solution for non encrypted protocols since it provides a lower accuracy with respect to that provided by payload based techniques.

Payload based This class of solutions try to identify the protocol by searching inside the payload for well known protocol signatures or by analyzing its content and correlating it with that of the other packets belonging to the same flow. Accordingly, in some cases we may need to maintain some information between successive received packets of the flow.

Protocol signatures may be both generated by hand or by using machine learning techniques [31], which automatically extract application signatures from IP traffic payload content.

Since payload based technique can analyze the entire payload, it is the most accurate technique and the only one which allows the extraction and processing of the content and metadata carried by the protocol. However, when metadata extraction is performed or if a very high accuracy is needed, TCP/IP normalization may be required by the application. TCP/IP normalization aims at solving the problems relative to IP fragmentation (i.e. when an IP datagram is larger than the Maximum Transmission Unit (MTU) of the outgoing link and hence it is divided in two or more fragments) and TCP reassembly (e.g. when an application message doesn't fit in a single IP datagram and hence it is split across different TCP segments) [32].

Moreover, in some cases IP fragmentation and TCP segmentation may be used by an attacker to evade the DPI engine [33, 34]. As an efficient alternative to TCP/IP normalization, some existing works [35, 36] try to identify this kind of misbehaving situations and to divert them to a slow path engine, which reassemble such flows. Conversely, the other segmented flows are managed using faster techniques which can match the signatures also in presence of fragmentation or segmentation. However, these techniques are not sufficient when the application explicitly requires to process the data contained inside the protocol payload in the same order they are sent.

1.1.1 Related work

We will now describe some of the existing works in the field of protocol classification, analyzing both hardware and software solutions and presenting their peculiar characteristics.

1.1.1.1 Hardware based solutions

Many hardware solutions exist which perform, at least in part, the steps required for protocol classification.

One of the most common cases, is to use dedicated hardware to search for patterns inside the packets both by using Field Programmable Gate Arrays

(FPGAs) [17, 37, 38] or by using Content Addressable Memories (CAMs) [39, 18, 19, 40]. These solution may both use exact matching approaches or accept a certain rate of false positive matching by using, for example, bloom filters.

The patterns searched inside the packet may both be application signatures used to identify the application protocol, or may be patterns identifying security threats, similarly to what is done by Intrusion Detection/Prevention Systems (IPS/IDS) [41, 42].

However, these solutions only perform stateless pattern matching, without any knowledge of the structure of the packet or of the relationship between them. This kind of knowledge should be provided in a real environment, since its absence could lead to the impossibility to find these patterns.

Let us consider for example the scenario depicted in figure 1.1, where the string DIVISIONBELL is split in two different TCP segment.

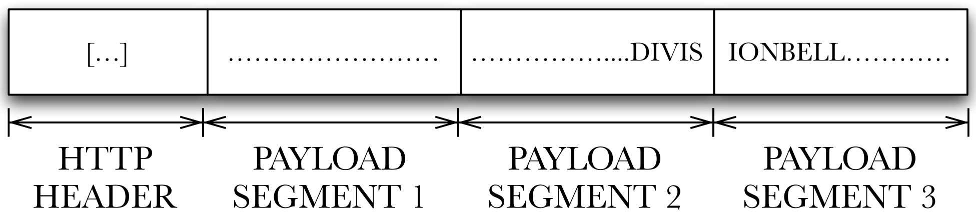


Figure 1.1: Example of a pattern split between two TCP segment

In that case, if TCP analysis and normalization is not provided, the two segments would be analyzed separately and thus the string would not be found by the pattern matching engine. Moreover, this could also happen when these solutions are used to identify the application protocol by signature matching, thus leading to the impossibility to identify some application flows. This is the reason why pattern matching alone is not sufficient and some kind of process and knowledge about the characteristics of the flow is required.

1.1.1.2 Software based solutions

nDPI This is a well known, open source DPI library which supports more than 100 protocols [23]. It has been forked from OpenDPI [20] and optimizes and extends it with new protocols. An inspector is associated to each supported protocol and each of them analyzes the entire packet payload searching for characteristics or signatures which allows it to identify the carried protocol. However, it lacks of support for IP and TCP normalization and doesn't provide any possibility to specify which data to extract once that the protocol has been identified. Moreover, it doesn't have any support for multiprocessor architectures, demanding to the application programmer the difficult and error prone task of parallelizing his application.

Libprotoident In this work the concept of "Lightweight Packet Inspection" (LPI) is proposed [22]. Instead of analyzing the entire payload, LPI tries to identify the application protocol by simply looking to the first four bytes of the payload. However, many applications use HTTP to carry their data and they in principle could be classified by looking to the `Content-Type` or `User-Agent` fields. Consequently, the lack of the remaining part of the payload makes impossible to analyze the HTTP header contents and sub classify HTTP traffic accordingly. Therefore, when such level of traffic classification is required, this approach is not sufficient. Moreover, also in this case, the library provides only the possibility to process the packets sequentially.

L7-filter This is a packet classifier which use regular expression matching on the application layer data to determine what protocols are being used [21]. However, also in this case, it doesn't provide any way to specify how to extract and process the data carried by the protocol once that the packet has been classified. Moreover, as shown in [43, 44], due to simple regular expression matching, its accuracy is lower with respect to more precise approaches as the one adopted by *nDPI* and *libprotoident*.

Solutions with multicore support Software solutions which exploit network processors or commodity multicore architectures can be found in literature. We will now describe some of them together with the choices that have been taken for their parallelization.

- In [24] a software solution which exploit a Cavium network processor is proposed. The main idea behind this solution is to distribute the packets among different threads by means of a shared queue. However, the access to the queue is protected by lock mechanisms and, as also stated by their developers, this is the reason why they are not able to achieve a good scalability.
- In [27] the different steps performed by the DPI engine are profiled and then distributed among the available cores, trying to keep the work balanced. However this is not a general approach and, if executed on different machine, it would require to do again from scratch profiling, design and implementation of, possibly, a different partitioning among

the activity. Moreover, despite lock free data structures are used, also in this case scalability problems are experienced, even using a relatively low number of cores. In this particular case, this is due to the unbalancing between the parts of the application executed by the different cores.

- In [25] different strategies to schedule the packets among threads are analyzed. These strategies take into account factors like cache affinity and load balancing. Anyway, also considered the best proposed strategy, this solution still suffers a limited scalability.

- Also in [26] a distribution of the work among different threads running on separate cores is proposed. However, also this solution is characterized by serious scalability problems caused by the overhead due to synchronization among the threads.

1.2 Structured parallel programming

As discussed above, existing software solutions often do not provide mechanisms to efficiently exploit the current multiprocessor architectures and, when such possibility is provided, they suffer from scalability problems. Indeed, writing an efficient parallel application is not only matter of adding some threads and force the correctness through mutual exclusion mechanisms, but it needs an accurate design in order to achieve the required efficiency.

For example, according to *libprotoident* creators[1]: *"Determined that adding threading to* libprotoident *was completely not beneficial - in fact, it ended up running much slower than before. This seems to be mainly due to the rules being so simple. There was no performance gain to compensate for the overhead of locking mutexes and synching threads that was introduced."*. Moreover, this is something that have also been experienced by other software solutions [24, 27, 25, 26]. As we will see in section 4.4.3, in some cases this implies a low scalability also when a relatively low number of cores is used. Besides the overhead due to synchronization, this can often be caused by poor design choices which will lead to load unbalancing among the used cores.

Moreover, these solutions provide only the possibility to identify the application protocol, without offering any facility to extract and process the data carried by the protocol once that it has been identified.

[1]http://www.wand.net.nz/content/weekly-report-21102011

For this reason in this thesis we would like to propose and implement a novel approach which, thanks to an accurate design, can take advantage from the underlying architecture and, at the same time, doesn't renounce to the precision of a stateful *payload based* classification approach. Moreover, differently from existing solutions, we would like to provide to the application programmer the possibility to specify the data to be extracted from the packet once that the protocol has been identified.

In order to reach this target, we applied to our work the concepts of *structured parallel programming* methodology [45].

Any application can, in general, be viewed as a set of concurrent activities cooperating to reach a common objective. However, developing an efficient parallel application is often a difficult task. Indeed, the user doesn't have only to deal with the algorithm details, but he needs to also take care of the setup of the concurrent modules in which the application is divided, to map and schedule them on the architecture on which the application will run and to implement correct and efficient techniques to let them communicate. All these activities require a big and error prone programming effort.

When structured parallel programming methodology is used, it's possible to hide a part of this complexity and to let it be managed by the library or the programming language used to describe the application [46, 47]. In general, starting from the sequential description of the application, the user can individuate a graph of activities which models it and which can express the same application as a composition of parallel modules.

Considered that the same concurrent application can, in principle, be modeled by many parallel modules compositions, different performance metrics can be used to evaluate a specific graph of activities and to compare it with alternative solutions. In our specific case, where the application works on a stream of packets, we are interested in the following performance metrics:

Bandwidth B Defined as the average number of packets per second that the DPI framework is able to process.

Service Time T_S Defined as the average time interval between the beginning of the executions of two consecutive stream elements. It is the inverse of the bandwidth, therefore it can be expressed as $T_S = \frac{1}{B}$.

Latency L Defined as the average time needed to complete the processing over a single packet. In our case this metric is particularly important when, before forwarding the packet, the application needs to wait for the result of its processing to decide, for example, if it matches or not the filtering rules. Indeed, if an high latency is required by the processing of the packet, we could introduce a not acceptable delay in the forwarding of the packet over the network.

However, to avoid this problem, the packet could be passed to the framework and then immediately forwarded without waiting for the result. In this way, also if the first packets of the flow are forwarded independently from the processing result, the filtering would be still applied to the remaining part of the flow. Anyway, also in this case, we would like to have a low latency in such a way that the decision is taken before the termination of the flow.

Nevertheless, in all the other cases, the only metric to be take into consideration should be the bandwidth (and then the service time).

Furthermore, we are interested in the average **Interarrival Time** of the packets to the application T_A. This quantity is the inverse of the rate of packets arriving to the framework. If we are able to structure the framework so that it has a service time $T_S \leq T_A$, then the framework will be able to process all the received packets.

Another important metric we are interested in is the **Speedup**. For a computation composed by n modules it is defined as

$$Speedup(n) = \frac{B_{seq}}{B_n}$$

where B_{seq} is the bandwidth of the sequential framework while B_n is the bandwidth of the parallel framework when n modules are activated. Ideally, we should have $B_n = \frac{B_{seq}}{n}$ and then $Speedup(n) = n$. However this is different for what will really happen because, in the parallel solution we will have, in addition to the latencies of the sequential version, some other latencies caused by factors like communication latencies and memory contentions.

For these reasons, in general we will have $Speedup(n) \leq n$. However, as we will see in Chapter 4, there are some cases in which we could have $Speedup(n) > n$. This could happen when, for example, the solution composed by n modules exploits a better spatial cache locality as a consequence of the reduction of the size of the working set.

Among all the possible graphs that can be used to represent an application, many real application can be described using some recurrent schemes called *skeletons* [45, 48]. Skeletons are programming patterns which allow to model typical forms of parallelism exploitation, expressed in parametric form. They have a precise semantic and are characterized by a specific *cost model*, which can be used to evaluate their performance metrics. Moreover, they can be composed and nested together to represent more complicated parallelism forms.

Skeletons can work over single data elements or over a so called "stream" of tasks. A stream is a possibly infinite sequence of values of the same type. For example, in our case, we can consider each packet arriving to the framework as an element of the stream.

Depending on the type of parallelism modelled, we can classify them in the following categories:

Stream parallel These skeletons exploit the parallelism among computations relative to independent tasks appearing on the input stream of the program.

Data parallel Are those exploiting parallelism in the computation of different sub tasks derived from the same input task.

As we will show in Chapter 2, for our framework we are interested in two particular *stream parallel* skeletons: *farm* and *pipeline*.

1.2.1 Pipeline

This skeleton can be used when the application computes a function $F(x)$ which can be expressed as a composition of n functions:

$$F(x) = F_N(F_{N-1}(\dots F_2(F_1(x))\dots))$$

where x is the received task. In this case, the corresponding graph of activities (shown in figure 1.2) is made by one module (in this case called *stage*) for each computed function.

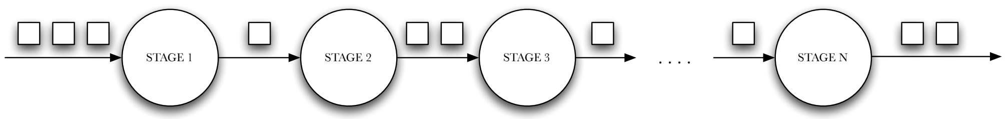


Figure 1.2: Pipeline skeleton

If the function computed by the *i-th* stage has latency L_i, then the service time of the pipeline will be

$$T_S = \max_{1 \le i \le N}(L_i + L_{com_i}) \tag{1.1}$$

where L_{com_i} is the time spent by the *i-th* stage to read the task from the input channel and to send it over the output channel. In general, if an appropriate support is present, L_{com_i} could be overlapped with the computation latency L_i. However, this is not our case because the framework we used doesn't have any support for this kind of overlapping and only allows to execute the communications sequentially with the computation. For this reason, from now on we will always consider that the communication will never overlap with the computation.

The pipeline latency instead is:

$$L = \sum_{i=1}^{N} (L_i + L_{com_i}) \tag{1.2}$$

Therefore, since the sequential application is characterized by:

$$L = T_S = \sum_{i=1}^{N} L_i \tag{1.3}$$

in general, using the pipeline, we decrease the service time of the application while the latency is increased.

1.2.2 Farm

Is a paradigm based on functional replication, and which can be represented by the graph depicted in figure 1.3. It is composed by: a function replicated into a set of modules $\{W_1, W_2, W_3 \ldots, W_N\}$ called *workers*, an *emitter E* which schedules the tasks to the workers and a *collector C* which collects the produced results. Therefore, in general, the function will be applied at the same time by different workers over different tasks of the input stream.

Different scheduling strategies may be used by the *emitter*. The most common are:

On-demand Using this strategy is possible to maintain a good balancing among the different workers. It could be implemented, for example, by allowing each worker to communicate to the emitter when it finished to process the task and is thus ready to accept a new one.

Round robin This is a simple strategy which sends the task circularly to the available workers. Often it is implemented in such a way that, if the communication channel towards a worker is full, the task is sent to the first worker with at least one available slot in the channel.

However, any other scheduling function different from the proposed ones can be used. We will compare the effect of these different strategies in Chapter 4.

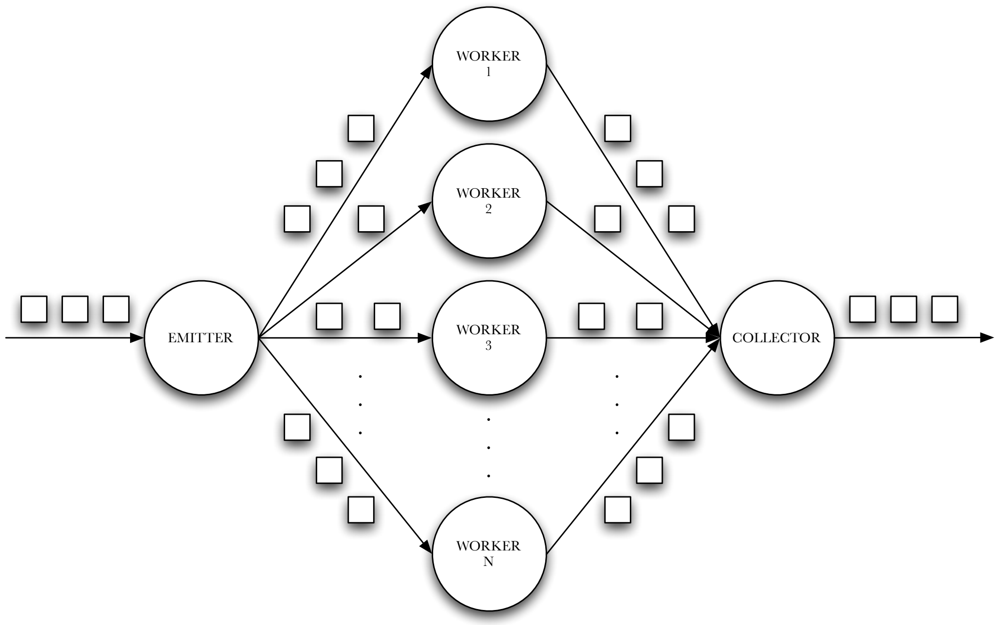


Figure 1.3: Farm skeleton

As we can see from the figure, the farm could be viewed as a 3-stage pipeline. Therefore, if the load of the workers is balanced (i.e. the probability that an input stream element is sent to any worker is $\frac{1}{N}$), the service time of the farm will be:

$$T_S = \max(L_E + L_{com}, \frac{L_W + L_{com}}{N}, L_C + L_{com}) \tag{1.4}$$

with L_E latency of the emitter, L_W the latency of a generic worker and L_C the latency of the collector.

The latency of the farm instead is:

$$L = L_E + L_W + L_C + (3 \times L_{com}) \tag{1.5}$$

Accordingly, also in this case the service time may be reduced while the latency is increased.

Eventually, we would like to find the optimal parallelism degree, i.e. the number of workers $\bar{n}$ such that, if this number of workers is used, the farm has a service time equal to the interarrival time T_A and it is therefore able to manage the whole incoming bandwidth.

Using equation 1.4 we have that $\bar{n}$, the *optimal* number of workers to be used in the farm, is equal to:

$$\bar{n} = \frac{L_W + L_{com}}{\max(T_A, L_E + L_{com}, L_C + L_{com})} \tag{1.6}$$

However, according to *structured parallel programming* concepts, if N is the number of available computational nodes (i.e. of machine's cores in our case) and if $\bar{n} > N$, then we need to restructure the computational graph such that it is composed by a number of nodes equal to N. Indeed, using this methodology, a reduction in the number of the modules is more effective than a multiprogrammed execution of more of them on a single physical node.

This is always true in our case because the modules are running for all the application lifetime processing data received from the input channels. For example, considering the case in which the modules are threads executed over multicore architecture. In this case, if more of them run on the same core, they would interfere and invalidate cache data between each other. The only exception to this may be when a core has more than a single context (i.e. the so-called *hardware multithreading*. However, also in that case, the effectiveness of running multiple threads over different context of the same core depends from the specific case.

Accordingly, if the optimal number of workers is $\bar{n} > N$, we need to reduce them to $n = N - 2$, because 2 cores are reserved to emitter and collector nodes.

In general, if a composition of more skeletons is used, we can use an heuristic [49] and reduce the parallelism degree of each of them multiplying their degree by a factor

$$\alpha = \frac{N - ps}{n_{\Sigma} - ps} \tag{1.7}$$

where N is the number of available physical nodes, n_{Σ} is the amount of nodes of the graph before the reduction and ps is the number of "service module" (i.e. emitters and collectors).

We will see in Chapter 2 how these concepts have been applied to this thesis in order to efficiently structure the framework accordingly to the incoming bandwidth and to the number of nodes provided by the underlying multicores architecture.

1.2.3 Considerations about average values

When referred to our specific case, it is very difficult to consider average execution times. Indeed, the time spent to process a network packet can be influenced by many different factors like:

- Presence of IP tunneling.
- IP fragmentation.
- Specific transport protocol.
- Packet belonging or not to a flow for which we already identified the application protocol.
- Flow length.
- Out of order TCP segments.
- Specific application protocol.
- Payload length.
- Variance in the execution times of the callbacks specified by the user.

The same reasoning can also be applied considering the average interarrival times. Indeed, in a real network, the utilized bandwidth changes many times during the day. Therefore, if the framework is dimensioned for an average value of the bandwidth, sometimes it will be overdimensioned while other times it will be underdimensioned and, consequently, not able to manage all the traffic passing over the network.

Accordingly, we can still try to compute an average and accepting that in some cases it could be very different from the real situation. Otherwise, we could consider all the latencies as upper bounds on real latencies and thus dimensioning the framework to always be able to manage all the network traffic. In this case however, we have to accept that there will be moments when the resources will be underutilized.

Alternatively, our framework could be modified in such a way that it can adapt, time by time, to the real situation of the network, thus dynamically adding or removing computational nodes to avoid resources underutilization. This is considered as possible "future work", however.

1.2.4 Related work

Many libraries and programming languages which implement the concepts of *structured parallel programming* over shared memory or distributed memory architectures exist. Since it was a prerequisite of the thesis, in our framework we used *FastFlow* [28, 1], which we will describe in detail in section 1.3.

We now present some of the most recent works in this field, describing their main characteristics and showing the reasons *FastFlow* is more appropriate for this kind of high bandwidth applications.

1.2.4.1 Muesli

Muesli [50, 51] is a C++ skeleton library which uses OpenMP [52] and MPI [53] to target shared and distributed memory architectures and combinations of the two. It implements the most commonly used task parallel and data parallel skeletons as C++ template classes. Furthermore, Muesli offers the possibility to nest these skeletons to create more complex execution graphs by using the two tier model introduced by the Pisa Parallel Programming Language (P3L) [47]. Basically, the computation can be structured using task parallel skeletons where each node of the skeleton can internally be implemented as a data parallel skeleton. Concerning the data parallel skeletons, Muesli provides distributed data structures for arrays, matrices and sparse matrices and it has been recently extended to support Graphics Processing Unit (GPU) clusters [54].

However it would not be suitable for our framework since, as shown in [55, 56], OpenMP does not perform as well as FastFlow for fine grained streaming application

1.2.4.2 SkeTo

SkeTo [57] is a parallel skeleton library written in C++ with MPI. Although in some intermediate versions it supported also stream parallel skeletons, it mainly provides data parallel skeletons which can operate on data structures like arrays, matrices, sparse matrices and binary trees. For each data structure, the library consists of two C++ classes; one provides the definition of parallel data structure, and the other provides the parallel skeletons. Differently from most frameworks, where the application is entirely build around skeletons, SkeTo provides the possibility to to use their skeletons by means of library calls performed by the sequential program. Currently, except the one provided by MPI, it provides no explicit support for multicore architectures. Additional work have been done over domain specific strategies [58] and optimizations for the construction of data structures [59].

Since we designed our framework to use stream parallel skeletons (section 2.2), SkeTo would not be suitable for our purposes because it provides only data parallel skeletons.

1.2.4.3 SkePu

SkePU [60] is a C++ template library which provides a simple interface for mainly specifying data parallel skeletons computations over GPUs architectures using CUDA [61] and OpenCL [62]. However, the interface is general enough and SkePU provides also an OpenMP based implementation for all the proposed skeletons. In addition to the skeleton templates, SkePU also includes an implementation of a vector with an interface similar to the one of the C++ Standard Template Library (STL) and which hides the complexity of GPU memory management. The skeletons in SkePU are represented by objects and contain member functions representing each of the different implementations, CUDA, OpenCL and OpenMP. If the skeleton is called with `operator()`, the library decides which one to use depending on what is available.

However, since its multicore support is built on top of OpenMP, it would not be suitable to manage the fine grained computations performed by our framework.

1.3 FastFlow

The framework we used to implement *structured parallel programming* concepts inside our framework is *FastFlow* [28, 1]. FastFlow is a `C++` framework targeting both shared memory and distributed memory architectures and which we already shown in a previous work [63] to be capable to manage similar kind of monitoring applications.

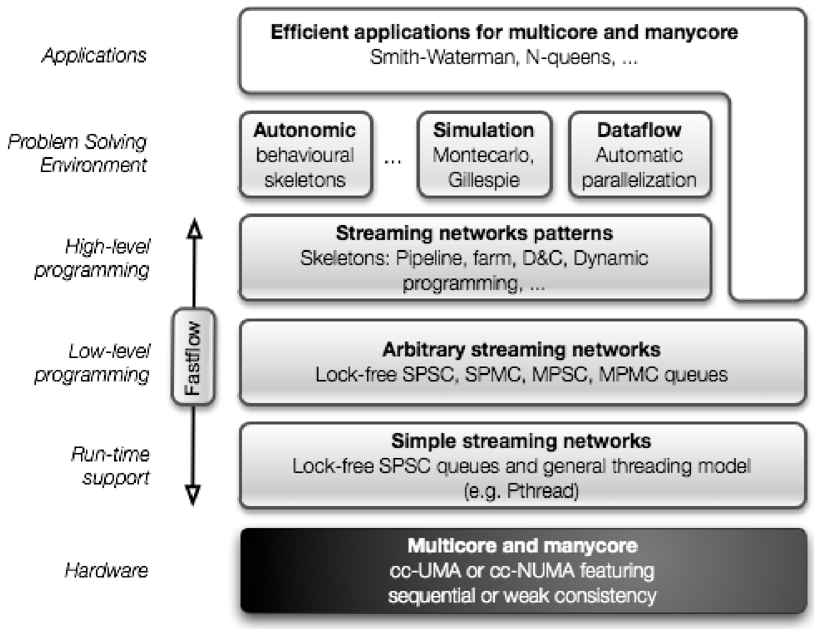

Applications
Efficient applications for multicore and manycore
Smith-Waterman, N-queens, ...
Problem Solving Environment
Autonomic behavioural skeletons
...
Simulation Montecarlo, Gillespie
Dataflow Automatic parallelization
High-level programming
Streaming networks patterns
Skeletons: Pipeline, farm, D&C, Dynamic programming, ...
Low-level programming
Fastflow
Arbitrary streaming networks
Lock-free SPSC, SPMC, MPSC, MPMC queues
Run-time support
Simple streaming networks
Lock-free SPSC queues and general threading model (e.g. Pthread)
Hardware
Multicore and manycore
cc-UMA or cc-NUMA featuring sequential or weak consistency

Figure 1.4: FastFlow architecture (taken from [1])

It is composed by several layers which abstract the underlying architecture. The abstraction provided by these layers (figure 1.4) is twofold: to simplify the programming process offering high-level constructs for *data parallel* and *stream parallel* skeletons creation and, at the same time, to give the possibility to fine tune the applications using the mechanisms provided by the lower layers.

At the very base level we found both bounded [64] and unbounded [65] *single producer, single consumer* (SPSC) queues which can be used as communication channels between threads. These queues are characterized by the total absence of *lock* mechanisms and have been realized taking inspiration from *wait-free* protocols described by Lamport in [66] and from FastForward queues outlined in [67].

Furthermore, FastFlow gives the possibility to define the code to be executed in the different computational module. This can be done by defining a class which extend `ff::ff_node` and implementing the virtual function `svc`. In example 1.1 we can see the definition of a simple module which takes an integer from the input channel, increments it and sends it on the output channel.

These nodes can then be easily linked together using the SPSC queues as communication channels in order to create arbitrary computation graphs similar to those depicted 1.5.

Listing 1.1: Node creation

```
#include <ff/node.hpp>
using namespace ff;

class ComputationalNode: public ff_node {
public:
    ComputationalNode(int max_task):ntask(max_task){};

    void* svc(void* task){
        int* real_task=(int*) task;
        ++(*real_task);
        return (void*) real_task;
    }
private:
    int ntask;
};
```

Alternatively, is possible to directly use some implemented and optimized *stream parallel* and *data parallel* skeletons or nesting among them. In this case the user simply specify the nodes and the skeleton to be used and the

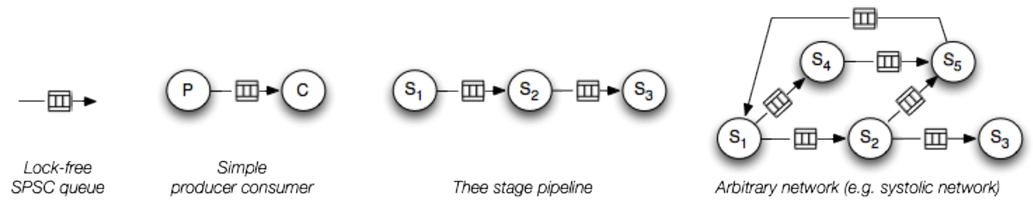


Figure 1.5: Examples of computational graphs (taken from [1])

library will link the nodes together to form the requested computation graph.

Furthermore, FastFlow provides some ways to customize the behaviour of these skeletons, for example by specifying the strategy to be used to distribute tasks in the emitter of the farm. Anyway, if this should still not be sufficient for a specific purpose, the user can use mechanisms provided by the lower layers or can easily extend the library by defining new skeletons.

In listing 1.2 we can see how, using few lines of code, it is possible to parallelize an already existing application. In this example, we define the application as a pipeline composed by two stages. The first of these stages (lines 6-12) reads the packets from the network (line 9) and send them over the communication channel towards the second stage (line 10). On the other hand, the second stage (lines 14-21) process the packets received from the channel (line 18) and indicates to FastFlow that it is ready to receive another task (line 19).

However, instead of processing the packets sequentially, we can decide to process them in parallel by using a farm with four workers (lines 29-32). In this way, the received packets will be transparently scheduled by FastFlow to the different workers. Eventually, we start the execution (line 37) and we wait for its end (line 39).

Although this is a small example, we can already see the advantages of this approach over classic techniques as `pthread` calls, which would require the user to deal with the additional and error prone task of threads synchronization.

Listing 1.2: Definition of the structure of the application

```
1 #include <ff/farm.hpp>
2 #define NWORKERS 4
3
4 using namespace ff;
5
6 class PacketsReader: public ff_node{
7 public:
    void* svc(void* task){
        char* packet=read_packet(network_interface);
        return (void*) packet;
    }
}

class PacketsAnalyzer: public ff_node{
public:
    void* svc(void* task){
        char* packet=(char*) task;
        //Process packet
        return GO_ON;
    }
};

int main(){
    PacketsReader reader;
    std::vector<ff_node*> w;
    ff_farm<> farm;
    ff_pipeline pipeline;

    for(int i=0; i<NWORKERS; ++i){
        w.push_back(new PacketsAnalyzer);
    }
    farm.add_workers(w);

    pipeline.add_stage(&reader);
    pipeline.add_stage(&farm);

    pipeline.run();
    //Do something else
    pipeline.wait();
    return 1;
}
```

We will see in Chapter 2 how the theory presented in this chapter will be used in the design of our framework and how the tools we introduced will be used to implement it in Chapter 3.

Chapter 2

Architectural design

In this chapter we will see how we designed our work in order to reach our the goals stated in the introduction.

2.1 Framework design

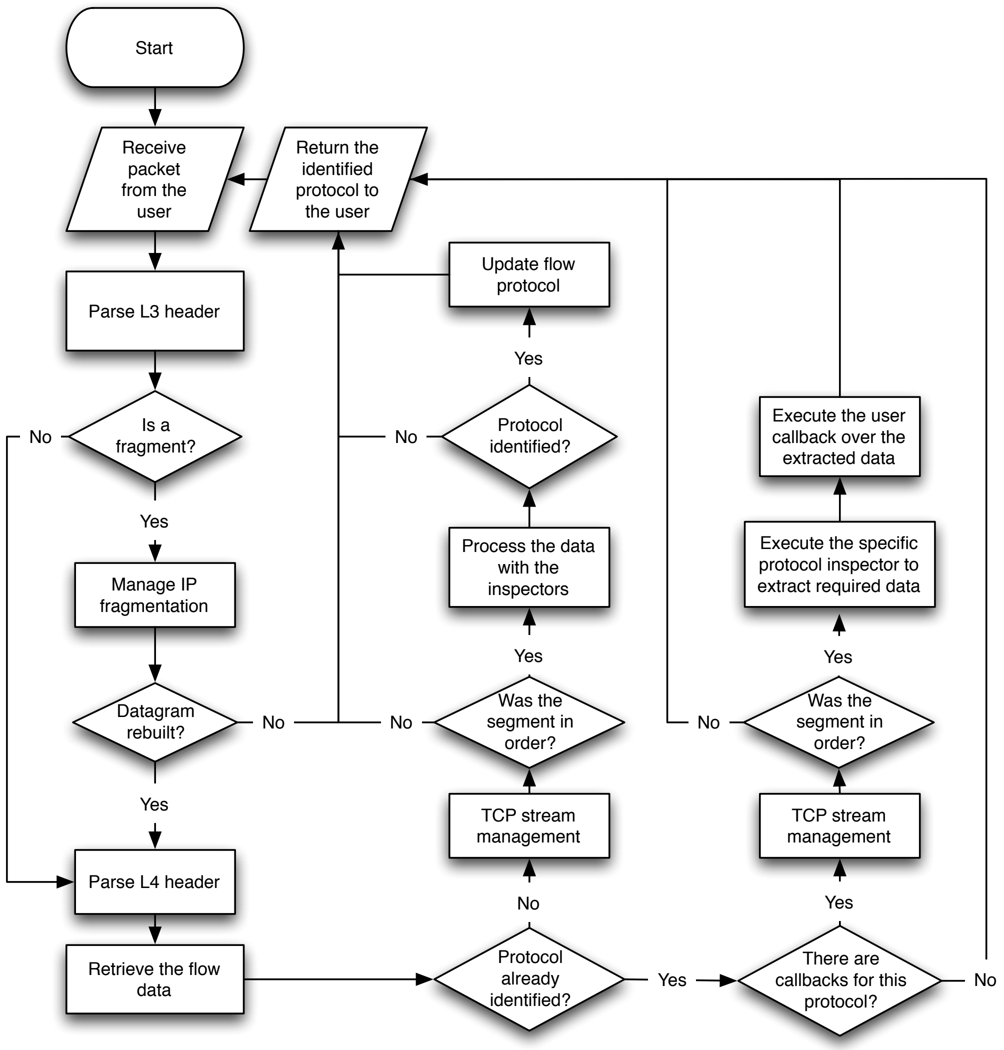
First of all, we assume that the application is able to provide to the framework a stream of IPv4/IPv6 datagrams. This decision has been taken in order to be completely independent from the specific technology that the application uses to capture the packets from the network, thus allowing the framework to be used with different kinds of networks or packets reading mechanisms.

This decision have been taken because current networks are often characterized by a bandwidth in the order of tens of Gigabits per second and, by using traditional communication subsystems, this packets rate will not be completely delivered to the applications [68, 69]. This is mainly due to heavy kernel involvement and extra data copies in the communication path, which increase the latency needed to read a packet from the network. Consequently, leaving to the application programmer the choice of the way in which the packets will be captured, we can exploit possibly available dedicated technologies.

For example, during the past years, different solutions have been proposed and implemented to solve these problems both using specialized Network Interface Controllers (NICs) as Infiniband [70] and Myrinet [71] or by using standard NICs with an appropriate software support [72, 73].

In both cases, zero copy and kernel bypass techniques [74, 75] are used to reduce the latency of the data transfers from the NIC to the application memory and to deliver the whole network bandwidth to the application [76, 77].

In order to provide a sketch of the framework structure, we will now describe the operations that should be performed when a TCP segment is received[1] and which are depicted in the flow diagram in figure 2.1.



Start
Receive packet from the user
Return the identified protocol to the user
Update flow protocol
Parse L3 header
Yes
No
Is a fragment?
No
Protocol identified?
Execute the user callback over the extracted data
Yes
Process the data with the inspectors
Execute the specific protocol inspector to extract required data
Manage IP fragmentation
Yes
Yes
Datagram rebuilt?
No
No
Was the segment in order?
No
Was the segment in order?
Yes
TCP stream management
TCP stream management
Parse L4 header
No
Yes
Retrieve the flow data
Protocol already identified?
Yes
There are callbacks for this protocol?
No

Figure 2.1: Flow diagram of the framework

When a packet containing a TCP segment is passed to the framework, it needs to:

1. Parse the network header and extract the real source and destination

[1]For UDP datagrams the process is the same except for TCP stream management.

addresses if *IP-in-IP* tunnels are present. Then, if the received datagram is a fragment of a bigger datagram, the framework will store the fragment for subsequent reassembly (IP normalization).

Hence, after parsing the network header, if we have a non-fragmented datagram, we parse the transport header in order to get source and destination transport ports.

Therefore, at this point, we have the entire 5-tuple key (composed by <Source IP address, Destination IP address, Source port, Destination port, Transport protocol identifier>) which characterizes the bidirectional flow to which the packet belongs.

2. In order to be able to track the progress of the protocol, we need to store for each flow some data about the previous packets we received for that flow. Notice that this doesn't mean that we need to store all the previous packets belonging to the flow but only a constant size structure containing some elements that will be updated when a new packet arrives (e.g. last TCP sequence number seen or current status of HTTP parser). The only exception to this is when an out of order TCP segment arrives. In this case indeed we need to store the entire segment for future reordering and processing (TCP normalization).

   These data are stored into an internal data structure and are maintained in it for all the flow lifetime. For the moment, we will just say that it has been implemented using an hash table with collision lists. The reasons behind this choice, together with the details about how it have been designed and implemented can be found in Chapter 3.

   When we didn't receive packets for the flow for a certain amount of time or when the TCP connection is terminated, these elements will be deleted.

   Therefore, using the flow key, the framework is able to do a lookup and to retrieve these elements.

3. After we obtained the flow data, if the application protocol has not yet been identified, the framework needs to manage the TCP stream and, if the segment is out of order, to store it for future reassembly. If the segment is in the proper order, the protocols *inspectors* will be executed over the application data. Each inspector, using previously collected elements about the flow and analyzing the current packet will try to infer if the flow carries data of its specific application protocol.

   If, instead, the protocol was already identified and if the application which uses the framework specified some callbacks for that protocol, we manage the TCP stream and we execute the specific protocol inspector in order to extract the required elements from the payload and to process them using the callback specified by the application.

Otherwise, if no callbacks for this protocol were defined, we simply return the previously identified protocol.

2.2 Parallel structure design

In this section we will present some of the possible solutions that could be adopted, together with the reasons which drove our choice towards the solution we decided to implement for our framework.

The first possibility we analyzed, is to use a farm where each worker executes, over a different received packet, all the steps described in the previous section. However the workers, for each received packet, should access to the shared hash table where the flows are stored. Considered that these flows can be modified, this would require to access the table (or at least each collision list) in a mutual exclusive way. Consequently, this solution would not scale with the number of workers. Moreover, we can't assume that the packets will be processed by the workers in the same order in which they arrived. Therefore, also if the TCP segments belonging to a certain TCP stream will be received in order by the DPI framework, it could be processed out of order by the workers. As we will show in section 4.3, this can have a not negligible impact on the overall performances.

However, analyzing the diagram in figure 2.1, we can see that the operations on different application flows are independent and could be then executed at the same time over distinct flows. Anyway, is in general unfeasible to have a module of the concurrent graph for each flow. For this reason, we decided to assign groups of flows to different modules that will process them in parallel exploiting the multicore hardware available.

Accordingly, we decided to structure the framework as a farm, with the flow table partitioned among a set of workers, as shown in figure 2.2 (collector not shown). In this way, the worker i will access only to the flows in the range $[low_i,\ high_i[$ without any need of mutual exclusion mechanisms or synchronization with any other thread.

Anyhow, we should take care of the way in which the packets are distributed to the workers. If *round robin* or *on demand* scheduling strategies are used, we must consider that a worker could receive packets that it can't manage and which should be forwarded to some other worker. To solve this problem we could let each worker communicate with all the other workers of the farm in such a way it can forward the packets not directed to itself to

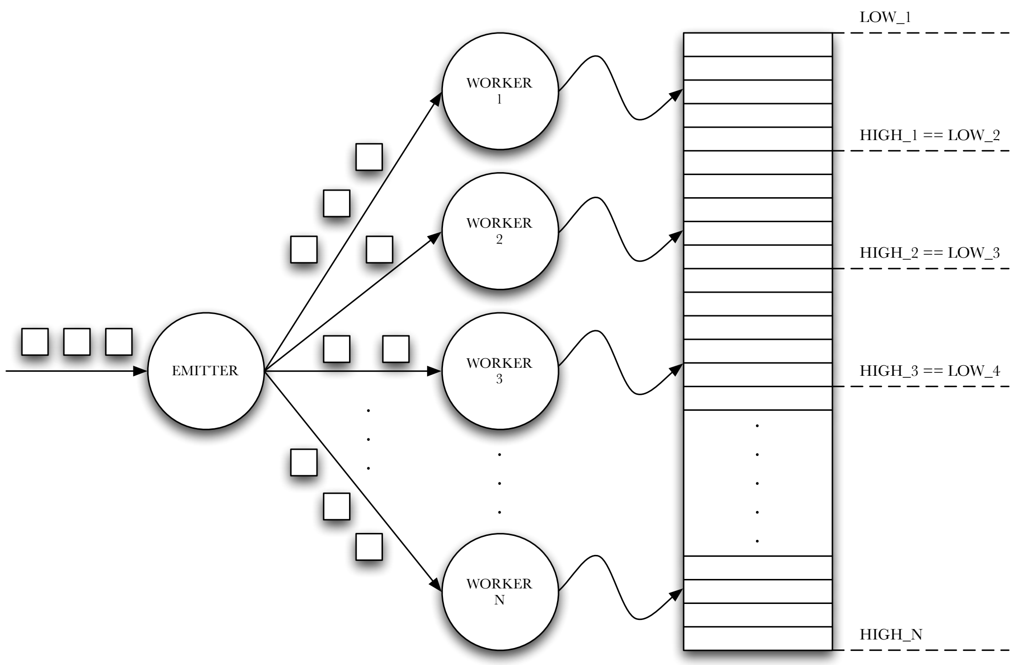


Figure 2.2: Structure of the framework with flow table partitioned among the set of workers (collector not shown)

the corresponding worker. However, also in this case we can't ensure that the packets belonging to the same flow will be processed in the same order they arrived.

For these reasons, we decided to provide the emitter with a scheduling function which distribute to each worker only the packets belonging to the flows contained in its partition. Consequently, the emitter needs first to extract the key of the flow and then, as in the sequential case, to determine the bucket of the table where the flow can be found. Once that the bucket has been found, the emitter can easily derive the partition to which the it belongs and, therefore, can forward the packet to the correct worker.

Moreover, since the communication channels are First In First Out (FIFO) queues, the order of the packets belonging to the same flow is preserved, avoiding thus to reorder data which arrives already ordered to the framework.

It's important to point out that, aside from the communication channels, the different nodes don't share any data structure among each other, allowing thus to advance in their execution without any need of synchronization.

Let's now analyze the service times of the different nodes of the graph.

- The emitter will have a service time

  $$T_E = L_{read} + L_{L3\_L4} + L_{hash} + L_{com} \tag{2.1}$$

  where L_{read} is the average latency of the *reading callback*, L_{L3_L4} is the average latency for network and transport headers parsing and L_{hash} is the latency of the flow hash function.

- The worker will have a service time

  $$T_W = L_{table} + L_{L7} + L_{cb} + L_{com} \tag{2.2}$$

  where L_{table} is the latency required to access the table, L_{L7} is the latency of the protocols inspectors and L_{cb} is the latency of the callbacks that the user specified over the protocols metadata.

- The collector has a service time

  $$T_C = L_{proc} + L_{com} \tag{2.3}$$

  with L_{proc} latency of the *processing callback.*

We should now decide the optimal amount of workers $\bar{n}$ to activate for a given interarrival time T_A. Using equation 1.6 we have:

$$\bar{n} = \frac{T_W}{max(T_A, T_E, T_C)} \tag{2.4}$$

Therefore, the entire rate of packets arriving to the framework can be managed only if $T_A > L_{proc} + L_{com}$ and $T_A > L_{read} + L_{L3\_L4} + L_{hash} + L_{com}$.

These conditions strongly depend on the specific callbacks specified by the user. However, even considering $L_{read} = 0$, the emitter could still be a bottleneck for the application. Indeed, the latency spent to parse the network and transport headers and to apply the hash function is not negligible and may often be greater than the interarrival time.

If this case is verified, we need to find a way to reduce the service time of the emitter. This can be done by replacing the emitter with a farm where each worker executes the steps that were previously executed by the emitter. Consequently, the resulting graph is a pipeline where each stage is a farm, as shown in figure 2.3.

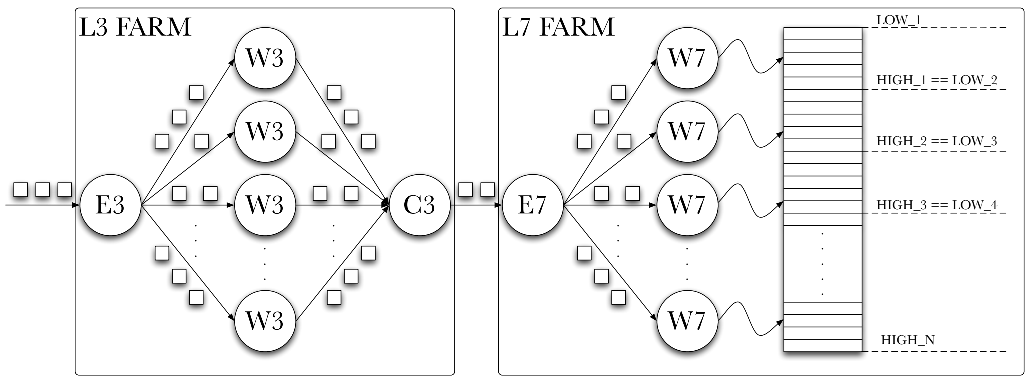


Figure 2.3: Structure of the framework when the emitter is a bottleneck (the collector of the second farm is not shown)

From now on, since the first farm of the pipeline take care of the processing of the network layer (level 3 in the Open Systems Interconnection (OSI) model) and of the transport layer (lever 4), we will refer to it as L3 farm. For similar reasons, we will refer to the second farm as L7 farm.

For L3 farm, any scheduling strategy could be used. However, we would like to have a scheduling strategy which maintains the order of the packets in such a way that they exit from the L3 farm in the same order they arrived. In this way, if the packets belonging to the same flow were already ordered, they will arrive in the same order to L7 farm and thus to the protocols inspectors. Consequently, the framework can avoid the overhead of TCP reordering when it is not really needed. In section 3.7 we will see how this has been implemented and in section 4.3 we will analyze the performance gain obtained by using an order preserving scheduling strategy.

Using this solution, the different nodes will have the service times shown in the following table:

Node	Service Times
L3 EMITTER	$T_{E3} = L_{read} + L_{com}$
L3 WORKER	$T_{W3} = L_{L3\_L4} + L_{hash} + L_{com}$
L3 COLLECTOR	$T_{C3} = L_{com}$
L7 EMITTER	$T_{E7} = L_{com}$
L7 WORKER	$T_{W7} = L_{table} + L_{L7} + L_{cb} + L_{com}$
L7 COLLECTOR	$T_{C7} = L_{proc} + L_{com}$

Consequently, defined $\bar{n}_1$ as the optimal number of workers for L3 farm and $\bar{n}_2$ the optimal number of workers for L7 farm, we will have:

$$\bar{n}_1 = \frac{T_{W3}}{max(T_A, T_{E3}, T_{C3})} \tag{2.5}$$

$$\bar{n}_2 = \frac{T_{W7}}{max(T_A, T_{E7}, T_{C7})} \tag{2.6}$$

However, also in this case there should be cases in which the ideal service time could not be reached because an emitter or a collector is a bottleneck.

For example, let's consider the case in which we have $T_A < T_{E3} = L_{read} + L_{com}$. In this case, in order to reduce the impact of the communication latency, we could calculate the optimal communication grain g and thus send and process the packets in blocks of size g. Accordingly, we need to find the value g such that $g \times T_A = (g \times L_{read}) + L_{com}$.

Therefore, we have:

$$g = \frac{L_{com}}{T_A - L_E} \tag{2.7}$$

It's important to notice that this equation is meaningful only when $T_A > L_E$. Indeed, in all the other cases the bottleneck cannot be removed by increasing the communication grain because $T_A < L_E$ and consequently, for each granularity g:

$$g \times T_A < (g \times L_{read}) + L_{com}$$

If more than one bottleneck is present, we can evaluate all the values of g and taking the maximum among them to ensure that these bottlenecks are removed.

However, we need to take into account that each task that the L7 emitter receives from the L3 farm can contain packets directed to different L7 workers. For this reason the emitter of the L7 farm, when a task is received, needs to redistribute the packets in temporary tasks and, when one of them has been completely filled, to send it to the corresponding L7 worker. Since the redistribution introduces an additional latency in the L7 emitter, its service time becomes: $T_{E7} = T_{redistribute}(g) + L_{com}$. Consequently, there are cases in which the L7 emitter could be a bottleneck. However, as we will see in Chapter 4, this will happen only in a limited number of cases.

On the other hand, concerning the L3 collector, it will never be a bottleneck because its service time will be always be lesser than the service time of the L3 emitter. Moreover, since the service time of the L3 emitter consists only in the reading of the packet, it can be consider as a lower bound on the service time of the framework.

We will see in Chapter 3 how these concepts have been applied for the implementation of the framework.

Chapter 3

Implementation

We will now describe some of the most important details concerning the implementation of the framework, relying on the design described in Chapter 2.

Concerning the sequential implementation, we will first analyze the interface provided to the application programmer, describing the choices we made to make it as much flexible as possible, in order to satisfy different application requirements and scenarios. Then we will analyze and motivate the main choices we made to implement the steps needed to perform a correct protocol identification. We start describing the processing of the network and transport headers, analyzing in detail the structures used to perform a functional and efficient IP defragmentation. After that, we describe how the management of the flows have been handled by our framework, describing some alternative solutions which will be then evaluated in Chapter 4. Furthermore, we will present the TCP reordering techniques we used and the interaction of the framework with the protocol inspectors. Eventually, we describe the possibility we provide to extract and process specific data and metadata of the protocol, once it has been identified.

Concerning the parallel structure, we will analyze how the concepts described in Chapter 2 have been implemented in this thesis. Also in this case different solutions have been proposed and then evaluated in Chapter 4. It's important to point out that FastFlow allowed us to prototype and test these different alternative solutions with relatively low programming effort. This has been possible because FastFlow provides an higher abstraction level with respect to that provided by directly using `pthreads`, which otherwise would have required much more effort due to the complicated and error prone task of thread synchronizations.

3.1 Application interface

The framework has been written in `C` and it can be used in two different ways:

Stateful mode Targeted for applications which don't have a concept of "flow" and therefore don't store any information about the previously received packets. In this case, the framework will keep an internal table with the information about each individual application flow.

Stateless mode This mode is designed for applications which already store some kind of data about the application flows (e.g. packets and bytes received). In this case, the elements needed by the framework can be stored into the application table. Therefore, the application should be modified in order to keep, with its own flow data, also a pointer to the flow elements created and managed by the library. In this case, however, the retrieval of the data must be entirely managed by the application.

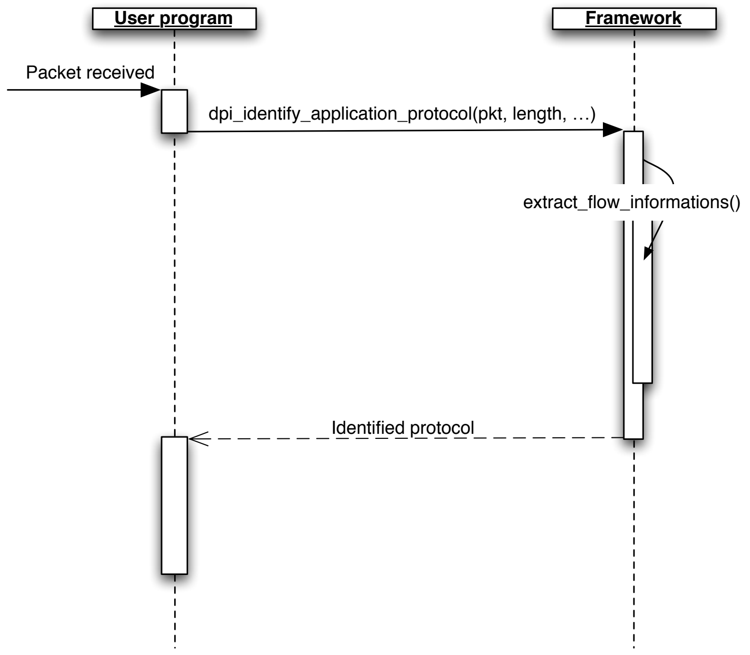

User program
Framework
Packet received
dpi_identify_application_protocol(pkt, length, ...)
extract_flow_informations()
Identified protocol

Figure 3.1: Stateful interaction

The two interaction modes are depicted in figures 3.1 and 3.2. We can see that in the stateful interaction the application simply provides to the framework the packet and then gets the identified protocol. Conversely, in the stateless mode, the parsing of the network and transport headers is decoupled from the identification of the protocol. Indeed, the application needs first invoke the framework to parse the network and transport headers and to get the flow key. Then, using this key, retrieves the flow data from its own table. Using the flow data, it can then call the framework to identify the protocol.

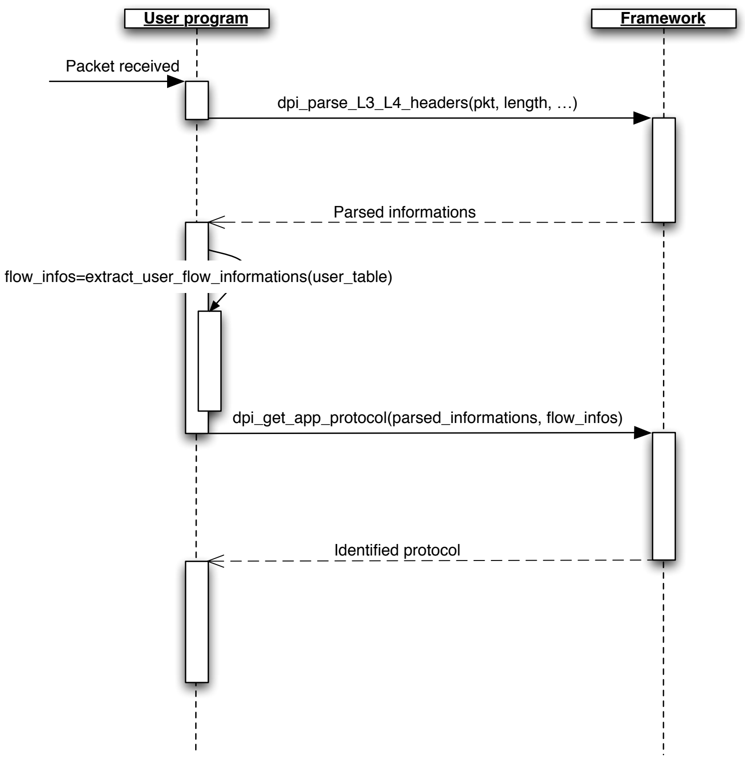

User program
Framework
Packet received
dpi_parse_L3_L4_headers(pkt, length, ...)
Parsed informations
flow_infos=extract_user_flow_informations(user_table)
dpi_get_app_protocol(parsed_informations, flow_infos)
Identified protocol

Figure 3.2: Stateless interaction

Almost all the calls we provided need an handle to the framework. This handle can be created with a specific framework call which will take care of creating and initializing its internal structures. By default, the framework activates all the protocols inspectors. Anyway, when is needed, the application can decide at runtime to activate only some protocols inspectors. This is particularly useful in *firewall-like* applications when the set of protocols to which the application is interested can dynamically change during its execution.

IP fragmentation and TCP reassembly supports are also activated by default. It is possible to disable them if needed, for example when they are provided by some other parts of the application or by the specific packet reading technique used[1].

The framework has an internal clock that is updated each time a packet is passed by the application. It is responsibility of the application to pass to the framework, together with the packet, the time it has been captured (it is sufficient to have a resolution of one second). In this way, the programmer can decide which mechanism is more appropriate or feasible for its application. Indeed, when managing millions of packets per second, adding a timestamp to each packet using classic calls as `gettimeofday` or `time` may result in a bottleneck.

As an alternative, if available, the application programmer could decide to use hardware timestamping provided by the network card or to use some other software level mechanisms (for example, to use the clock cycles counter). Anyway, since this task is left to the programmer, he can use the more appropriate mechanism with respect to the context in which the application is running.

3.2 Network and transport headers processing

First of all, the framework checks if the datagram is an IPv4 or an IPv6 datagram. After that, it starts parsing the header and, in case of presence of tunnels, they are unwrapped and the framework will consider as actual source and destination of the flow those found in the inner datagram. Up to now, we implemented the support for 4in4, 4in6, 6in4 and 6in6 tunnels[2] and all the combinations among them.

After parsing the network header, in case the transport protocol is TCP or UDP, the framework extracts the transport ports, finds the offset at which the application data start and build the flow key.

However, if the packet we received was carrying a fragment of a bigger datagram, before extracting the transport header elements we need to reconstruct the original datagram. Let's now see how this has been implemented in the framework.

[1]For example when the application, instead of reading "raw" data, decides to read from operating system sockets.

[2]An outer IP header is added before the original IP header. The outer IP addresses identify the "endpoints" of the tunnel while the inner IP header addresses identify the original sender and recipient of the datagram [78].

3.2.1 IP fragmentation support

We briefly recall that, when an IPv4 datagram is larger than the MTU of the outgoing link, it will be split in different fragments. After that, an IPv4 header will be attached to each fragment and they will be sent separately over the network (figure 3.3). However, to be able to reconstruct the original datagram, all its fragments will be marked with the same identification number. Moreover, each fragment is characterized by a specific fragment offset, which represents the point in which this fragment must be put inside the reconstructed datagram. This means that if a fragment has an offset x and a length of l bytes (considering only the payload), it carries the bytes that in the original datagram were in the range $[x, x + l[$[3]. From now on we will generically refer to this range as `[Offset, End[` range. Eventually, the last fragment of the datagram can be identified because it is the only one with a specific flag in the IP header (the MF flag) set to zero.

For IPv6 the fragmentation process is slightly different[4]. Anyway, as the reassembly process performed by our framework is similar, we will describe only the one for IPv4.

To be able to reconstruct the original datagram, we need to store its fragments into a set and then, when all the fragments have been received, to put their content together and in the correct order. At this point the framework can analyze the content of the original datagram as if it has never been fragmented. Furthermore, for each datagram we are reconstructing, we need to store also a timer and its original IPv4 header. The timer is needed as, if the datagram is not reconstructed in a certain amount of time, all the pending data that we have about the datagram will be removed from the library [79].

In principle, the fragments will not overlap between each other. However, an attacker could send overlapping fragments to try to exploit vulnerabilities in the defragmentation algorithm [15]. This is the reason why, when we insert a new fragment we need to check that it doesn't overlap with the fragments already present in the set and, if overlaps are present, we need to store only the part of the fragment that we didn't received yet.

Therefore, when a fragment is received, the framework needs to:

1. Obtain the set of fragments that we already received for this datagram.

[3]Actually the offset is expressed in 8-bytes blocks. However, to simplify the exposition, we will consider the offset as expressing the exact byte where the fragment starts.

[4]Differently from IPv4 it can be performed only by end nodes, and involves IPv6 optional *fragmentation header*. Moreover, some of the optional headers present in each fragment must be copied in the final datagram

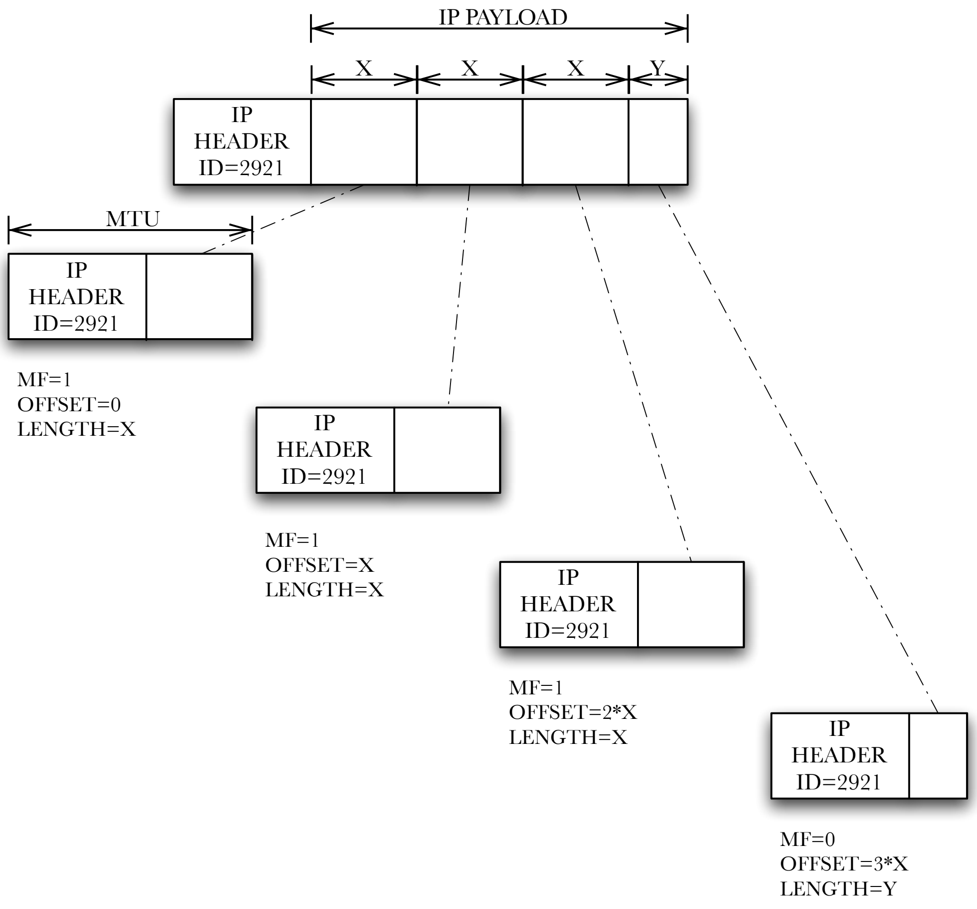


Figure 3.3: IP fragmentation process

2. Check if the fragment we are inserting overlaps with any existing fragment by comparing the `[Offset, End[` ranges.

3. Insert the non overlapping bytes into the set.

To avoid to check all the fragments present in the set, we decided to implement the set as a linked list, sorted according to the `[Offset, End[` range (figure 3.4). In this way, we will check only the fragments that have bytes belonging to the `[Offset, End[` range of the fragment that we are inserting.

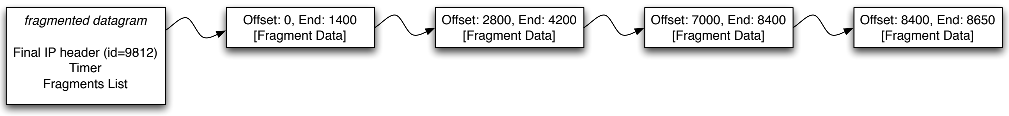


Figure 3.4: List of IPv4 fragments

Anyway we still didn't say how, given a datagram, the framework retrieves the list of its fragments. It's important to point out that we cannot use for this purpose the same structure that we use to keep track of the other flow data. Indeed, at this point, we still don't know to which flow the packet belongs because the transport header could have been split in different IP datagrams.

For this reason, we need a separate data structure. Moreover, we would like to organize this structure in such a way that we can easily know the amount of memory that the framework uses for each IPv4 source and, if a predefined threshold is exceeded, we can delete its oldest outstanding fragmented datagrams. Accordingly, for each source, we have a structure containing the sum of its used memory and the list of its fragmented datagrams (figure 3.5).

Eventually, we need to organize these sources in such a way that when a datagram is received, we can easily retrieve the fragmented datagrams generated by that specific IPv4 source. For this purpose, we decided to store the set of sources into a simple hash table where collisions are resolved through separate chaining (figure 3.6).

In conclusion, putting all together, when a fragment is received the framework will:

1. Check if there are expired fragmented datagrams.

2. Apply an hash function over the IPv4 source address to get the bucket of the hash table where the data about this specific source are stored.

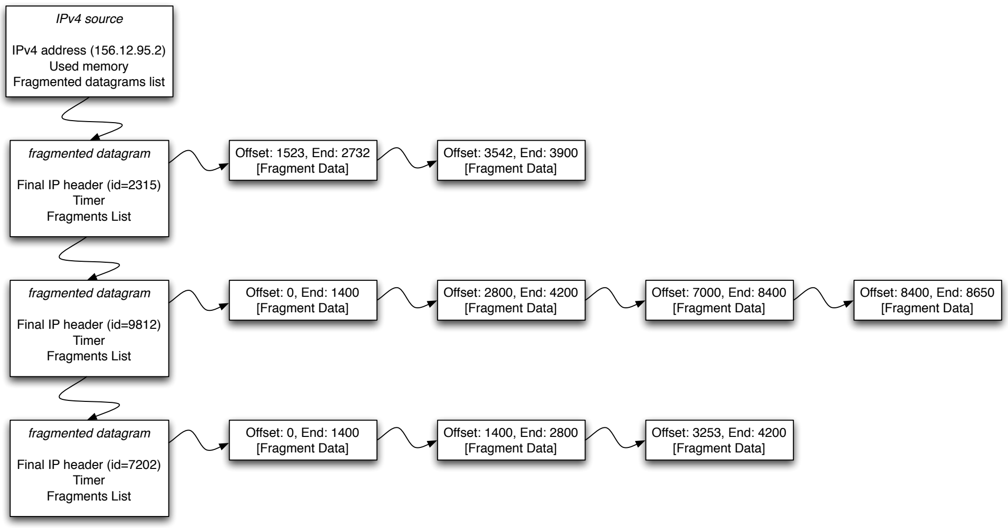


Figure 3.5: List of IPv4 fragmented datagrams

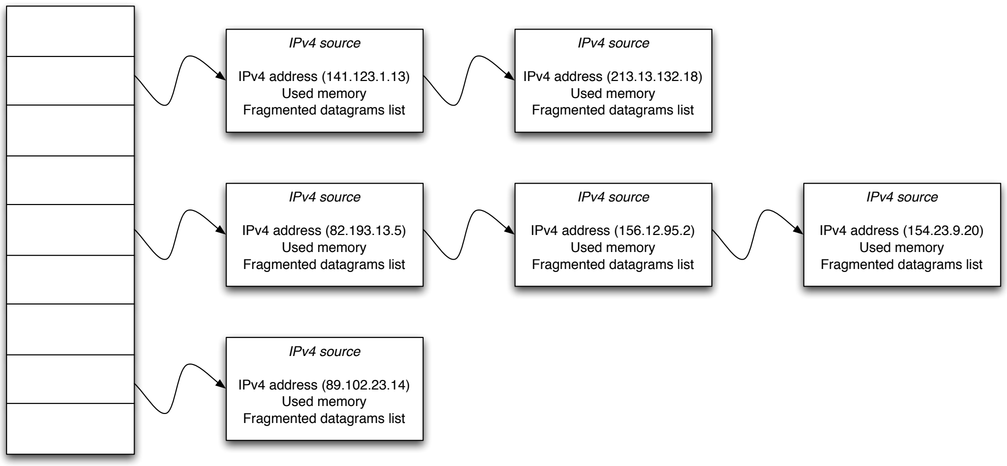

IPv4 source
IPv4 address (141.123.1.13)
Used memory
Fragmented datagrams list
IPv4 source
IPv4 address (213.13.132.18)
Used memory
Fragmented datagrams list
IPv4 source
IPv4 address (82.193.13.5)
Used memory
Fragmented datagrams list
IPv4 source
IPv4 address (156.12.95.2)
Used memory
Fragmented datagrams list
IPv4 source
IPv4 address (154.23.9.20)
Used memory
Fragmented datagrams list
IPv4 source
IPv4 address (89.102.23.14)
Used memory
Fragmented datagrams list

Figure 3.6: Hash table containing the IPv4 sources which have outstanding fragmented datagrams. For sake of simplicity, the list of outstanding fragmented datagrams is not shown

3. Scan the collision list to find the IPv4 source.

4. Scan the list of its outstanding datagrams.

5. Scan the fragments list and put the fragment in the correct position.

To implement point 1, and to avoid to check all the datagrams, the framework keeps a list of timers (one per fragmented datagram). When the first fragment of a datagram is received, we insert a new timer into the head of the list. In this way, the list is automatically kept sorted from the newest to the oldest received fragmented datagram. If some datagrams are expired, they will be located in the last positions of the list. Therefore, when the framework needs to check if there are expired datagrams, it will start checking from the tail of the list avoiding, in general, to scan the it entirely.

3.3 Flow data management

As we anticipated before, if the framework is used in *stateful* mode, it is responsible of storing and retrieving flow data. These data include:

- The application protocol, if already identified.
- Data that the application wants to associate to the flow (we will see in section 3.5 how this is used).
- TCP tracking information, including the segments that have been received out of order.
- Protocol specific tracking data (for example, the state of the HTTP parser).

The interaction of the framework with these data is depicted by the underlined functions in algorithm 1.

Since this structure should be accessed using the flow key, we decided to implement it as an hash table. Accordingly, if we have f flows, a table of size s and we have an uniform hash function, we will have an average search complexity of $O(\frac{f}{s})$.

We decided to manage the collisions using separate chaining by means of double linked lists. Consequently, the creation of a new flow will always have $O(1)$ complexity. The same holds true for the deletion of a flow as a result of a TCP connection termination. In this case indeed it will be done only after that the flow have already been found.

Algorithm 1 Flow management in our framework

```
Require: A packet $p$ belonging to a flow with key $k$ has been received
  $f \Leftarrow$ Find($k$);
  if $f$ == NULL then
    $f \Leftarrow$ CreateFlowData($p$);
  end if
  $protocol \Leftarrow$ IdentifyApplicationProtocol($p$, $f$);
  Update($f$, $protocol$);
  if $f$ is a TCP flow and if the connection terminated then
    Delete($f$);
  end if
```

We now introduce the mechanism we used to check time by time if there are expired flows in the table. Each flow, when updated[5], is marked with the timestamp of the last received packet and, when we want to know if a flow is expired, we simply check that the difference between the current time and the timestamp of the flow doesn't exceed a predefined threshold. However, we would like to avoid to check all the flows each time that a new packet arrive. Taking into account that the framework clock has a resolution of one second, this check will be done at most one time per second.

Moreover, we would like to avoid to scan all the flows and to only check the oldest flows in the table. In order to do this, we keep the collision lists sorted from the most recently updated to the less recently updated flow. This can be done by simply moving a flow to the head of its collision list when a packet for that flow is received. In this way, for each collision list, we start executing the check from its tail and, being the list sorted, we remove all the traversed flows, up to the point when the difference between the current timestamp and the timestamp of the flow that we are analyzing is less than the inactivity threshold. We will analyze in section 4.2 the impact of this technique on the overall framework bandwidth.

It's important to notice that, keeping the collisions lists sorted, we also have the advantage to have the most active flows in the first positions of the list, thus reducing the search time for the flows for which we are presumably receiving more packets.

3.3.1 Hash functions analysis

At this point, we have to find a good hash function, in order to keep an uniform distribution of the flows over the table. For this purpose, we proposed,

[5]For TCP flows this is done only if the received TCP segment is in order

implemented and analyzed four different hash functions:

Simple hash This function simply sums together the five fields of the flow key.

FNV1a-32 hash The "Fowler/Noll/Vo" (or FNV) hash functions are designed to be fast while maintaining a low collision rate. Particularly, we used the alternate algorithm for 32 bits keys shown in algorithm 2.

Algorithm 2 FNV1A-32 hash function

```
function FNV(data, data_length)
    hash ⇐ OFFSET_BASIS
    for $i = 1$ to data_length do
        hash ⇐ hash $\oplus$ data[i]
        hash ⇐ hash $\times$ FNV_PRIME
    end for
    return hash
end function
```

More details on how OFFSET_BASIS and FNV_PRIME are chosen are described in [80].

BKDR hash This hash function comes from Brian Kernighan and Dennis Ritchie's book "The C Programming Language" [81] and is described in algorithm 3.

Algorithm 3 BKDR hash function

```
function BKDR(data, data_length)
    seed ⇐ 131
    hash ⇐ 0
    for $i = 1$ to data_length do
        hash ⇐ (hash $\times$ seed) + data[i]
    end for
    return hash & 0x7FFFFFFF
end function
```

Murmur3 hash We will not show here its pseudocode because is more complex than the previous functions (details about the algorithm can be found in [82]). However, it's important to say that, with respect to the other presented algorithms, MURMUR3 hashes are influenced by a seed that can be randomly chosen in order to make the function more robust against possible attacks. On the other hand, in FNV and BKDR the seed can only be chosen from a predefined small set of seeds which satisfy some particular properties.

In section 4.1 we will do a comparison between these hash functions, comparing them according to the uniformity of the distribution, to the execution time and to the real impact on the DPI framework.

3.4 Protocol identification

After that the framework parsed the packet and retrieved the data about the previous packets belonging to the same flow, it can proceed and try to identify the application protocol, if not yet identified.

When a flow is created, it is in the NOT DETERMINED state. When a new packet for that flow is received, the flow can remain in the NOT DETERMINED state, it can move to the IDENTIFIED state or it can move to the UNKNOWN state.

A flow remains in the NOT DETERMINED if the framework still judges possible to classify the packet in one of two or protocols and it needs more packets to identify the exact protocol of the flow. To avoid to keep continuously analyzing packets in presence of this kind of ambiguity, the application can specify the maximum amount of packets to analyze for each flow. When this amount is exceeded, the framework moves the flow in the UNKNOWN state and will no more try to identify the protocol.

Alternatively, if after one of more packets the framework identified the protocol, then the flow is moved in the IDENTIFIED state. In this case, if no callbacks are present (section 3.5), when a new packet for this flow is received the framework will simply return the already identified protocol without executing any additional operation on the packet[6].

3.4.1 TCP stream management

TCP is a reliable and stream oriented protocol. For this reason, the applications that use TCP as transport protocol assume that their data will be received in the same order in which they are sent. Therefore, considered the best effort delivery provided by IP networks, if we want to correctly identify the protocol, we need first to put the data in the same order in which

[6]Actually, if the packet belongs to a TCP flow, the framework will still check if the segment carries the FIN flag in such a way that the connection can be correctly closed and the data about the flow removed.

they were sent. It's important to notice that this is not a problem for UDP based protocol as, in that case, when reliability and reordering are needed, they are implemented through application specific mechanisms and so in the framework they are demanded to the specific protocol inspector.

Accordingly, before executing the protocols inspectors on the packet payload, if the received packet contains a TCP segment, the framework needs to update the information on the TCP connection of the flow and, if the segment is out of order, to store it for further reordering.

The connection information stored by the framework includes the expected sequence numbers in both directions, a list of out of order segments and information about connection establishment and termination (for example, if one of the two endpoints sent the FIN segment).

It's important to notice that, at this point, these elements are already available to the framework because they were stored together with the other flows data that we retrieved in the previous step.

We can distinguish two different types of TCP flows: those who start while the framework is working and those who already started before the framework was started.

In the former case is easier to start tracking the connection because the framework can follow the three-way handshake. Therefore, it knows which are the sequence numbers that both endpoints will use and, consequently, it can identify the out of order segments.

The latter case is more complex because, if the first packet received for a flow isn't the first segment of the three-way handshake, then the framework cannot determine if the segment is out of order or not. In this case it needs more packets in order to be able to evince the current state of the connection. Basically, the framework will consider all the received segments as out of order and, instead of storing them for future reordering, it will discard these packets. When the highest acknowledgment number seen in one direction of the connection coincides with the expected sequence number in the other connection direction (and vice versa) the framework knows the current state of both endpoints and it can start to behave as if he has just seen the three-way handshake.

From this point, we can start managing the out of order TCP segments, using techniques similar to those we used for IP defragmentation.

First of all, for each out of order segment we store its content, its sequence number and its length. Then, the segment is inserted into a linked list containing the other out of order segments, checking for possible overlaps with those already present and, in case, inserting only the non overlapping bytes.

When a segment is received in order, if it fills an hole in the stream (i.e. if the first byte of the first out of order segment in the list follows the last byte of the received segment), the framework compacts the segments and passes them to the inspectors for protocol identification.

3.4.2 Protocols inspectors

After the framework obtained ordered data, it will invoke the protocols inspectors on that data, trying to identify the protocol of the flow.

Up to now our prototype framework implementation supports the following protocols: HTTP (based on the open source parser in [83]), DNS, MDNS, DHCP, DHCPV6, SMTP, POP3, BGP and NTP.

Each of these protocols has its own inspector parsing the application payload to try to understand if the protocol carried by the flow matches. Each inspector has the following input parameters:

- The handle of the framework, that can be used by the inspector to get information about the current state of the framework. For example, when the application programmer specifies some callbacks to be executed when specific protocol metadata are found in the packet, the pointers to these callbacks are stored in the handle of the framework. Accordingly, when the inspector finds such data, it can call the corresponding callback.
- A pointer to the parsed packet.
- A pointer to the application data.
- The length of the application data.
- Information about the the flow, used by the inspector to keep track of the current state of the protocol.

When an inspector is called, it can return three different responses: MATCHES, DOESN'T MATCHES or MORE DATA NEEDED. The latter is returned in case the inspector needs more data to decide if the protocol matches or not. For example, consider the case in which the HTTP inspector is analyzing a TCP flow which started when the framework was still not running. In this case, it could have started analyzing the flow in the middle of a file transfer and, consequently, the first packets that it receives for that flow contains HTTP *body* segments. Therefore, the inspector can't determine if the flow carries HTTP data by simply looking to the first packets of the flow. For this reason it could return a MORE DATA NEEDED response and wait for other packets before deciding.

The responses given by the inspector are in a strict relationship with the possible states of a flow. For each flow the framework keeps a set of possible matching protocols. Each time that an inspector returns a DOESN'T MATCHES response, the corresponding protocol is removed from the set. On the contrary, if a MATCHES response is returned, then the protocol has been identified, the flow will pass from the NOT DETERMINED state to the IDENTIFIED state, and no other inspectors will be ever called for this flow. If, after the maximum number of trials specified by the application, the set still contains two or more protocols or if it is empty (i.e. if all the inspectors returned a DOESN'T MATCHES response), then the framework will move the flow from the NOT DETERMINED state to the UNKNOWN state. This maximum number of trials can be indicated by a specific call of the framework and can be changed during its execution. Since this set is maintained for all the flow lifetime, for the packets successive to the first one, only the inspectors that returned a MORE DATA NEEDED response will be invoked.

The order in which the inspectors is invoked is not the same for all the flows. Indeed, if the source or destination port of the flow is a well known port, the first inspector to be invoked will be the one corresponding to the protocol which usually runs on that specific port. For example, if one of the ports of the flow is port 80, then the first inspector to be invoked will be the HTTP inspector. Then, if it doesn't return a MATCHES response, the framework will invoke the other inspectors.

Algorithm 4 DHCP inspector

```
if (payload_length >= 244 AND
   (source_port == 67 AND dest_port == 68) OR
   (source_port == 68 AND dest_port == 67)) AND
   payload[236] == 0x63 AND
   payload[237] == 0x82 AND
   payload[238] == 0x53 AND
   payload[239] == 0x63 AND
   payload[240] == 0x35 AND
   payload[241] == 0x01) then
     return PROTOCOL_MATCHES
else
     return DOESN'T_MATCHES
end if
```

In algorithm 4 we can see an example of a simple protocol inspector for the DHCP protocol.

As we can see, it tries to determine if the flow matches by simply looking to some bytes, in specific positions, of the received packet. In the DHCP case these bytes will be always the same independently from the DHCP message. Therefore, after only one packet, the inspector can determine if that packet belongs to a DHCP flow or not. Anyway, for more complicated protocols the identification process turn out to be not so simple and could require many packets to correctly identify the protocol.

The framework can be easily extended with new protocols with few isolated addition to the code. If the programmer wants, he can implement a new inspector by following these simple steps:

1. Give to the protocol the next available numeric identifier.

2. Create a new inspector, by implementing a `C` function with the previously described signature and semantic.

3. If the inspector needs to store information about the application flow, add an appropriate structure in the flow data description.

4. Add the inspector to the set of inspectors which will be called by the framework. This can be done by inserting a pointer to the corresponding function into an appropriate array.

5. If the protocol usually run on one or more predefined ports, specify the association between the ports and the protocol identifier.

6. Recompile the framework.

At this point, when the framework will be executed, it will use this new inspector as any other inspectors already provided by it.

3.5 Callbacks mechanisms

The last point to describe, is about how the application can specify which actions the framework should execute on the data or metadata of the packet once that its protocol has been identified.

For example, let us consider the case in which the application programmer wants to write a monitoring application which extracts the `Host` header from all the HTTP packets and, if it its value is contained in a blacklist, to block the connection.

For this purpose, the framework provides to the application the possibility to define protocol and metadata specific callbacks. For the moment this possibility has been provided only for HTTP protocol and for its different types of metadata. Anyway, this can be extended to other protocols extending the inspector through a process similar to the one described for the creation of a new inspector.

Let us now see the possibilities provided by the library for the parsing and processing of HTTP metadata. First of all, the application can specify the callbacks by using an appropriate function which requires as parameters:

- The handle of the framework.
- A pointer to some data that will be accessible from the callbacks. For example, considering the HTTP host filtering application, this could be a pointer to the Uniform Resource Locator (URL) blacklist.
- A data structure containing one or more callbacks. Using this parameter the application can specify:
  - The callback to be used on the HTTP *request-URI*.
  - The names of the HTTP *header fields* that the application wants to inspect, along with a callback for each specified *field*. Since the *field* value could be split in more than one TCP segment, the framework will store the different parts in which it is divided and will invoke the callback only when the entire *field* has been reconstructed.
  - A callback to be called when the HTTP *header* has been completely received and processed. This type of callback has been provided because the HTTP *header* may be split in more TCP segments. Therefore, it is needed in order to distinguish the case in which a specific *header field* callback has been not called because the corresponding *field* was not present, from the case in which it has not been called because the header is still not completely received and therefore the *field* may be present in the successive segments.
  - A callback to be called on the *body* of the HTTP message. Like the *header*, also the *body* could be split in different TCP segment. Anyway in this case, instead of waiting to receive and store the entire body, the framework invokes the callback on each body chunk as soon it arrives. This decision has been taken because in case of large files transfer it may be not feasible to store the entire *body* in main memory before invoking the callback. Moreover, in some cases the application might take its decision by analyzing the first bytes of the *body* or might not be willing to wait for its entire reception.

The arrival of the last chunk is signaled through an appropriate callback parameter.

Each callback may have different parameters and must be specified as a `C` function. For example, the HTTP *body* callback has the following parameters, which are provided by the framework and which can be accessed from the code specified inside the callback:

- Information about the HTTP message. It can be used, for example, to check if the message is an HTTP *request* or *response* or to know the `http` *version*.
- A pointer to a chunk of the HTTP *body*.
- The length of the chunk.
- A flag which indicates if this is the last chunk of the HTTP *body*.
- Information extracted from the network and transport headers of the packet (e.g. IP addresses and TCP or UDP ports).
- A pointer to some *flow application data*. This data is stored by the framework, together with the other data about the flow and can be used to keep track of accumulated knowledge about the flow between successive received packets or between successive invoked callbacks. Moreover, this pointer will be returned by the protocol identification call of the framework, together with the identified protocol. In this way the data collected by the callbacks about the flow can be communicated to the rest of the application.

  Furthermore, the application can specify an additional callback to be invoked by the framework on the *flow application data* when the flow expires. Indeed, while the other information about the flow are allocated, managed and deallocated by the framework, *flow application data* is created and managed by the application and thus it should be the application to decide how to manage this data once the flow is terminated.
- A pointer to the global HTTP application data. Differently from the previous parameter, this is a global structures that can be accessed by any HTTP callback, independently from the specific flow.

In listing 3.1 we can see how the application can specify and provide to the framework a simple callback which checks if the HTTP flow is carrying a FLASH video stream.

Listing 3.1: HTTP callback example

```c
/**
 * Definition of the callback function with the parameters
 * previously specified.
 **/

#define FLASH_CONTENT_FOUND 0x01

void callback(dpi_http_message_information_t* http_info,
              const u_char* content_type_string,
              u_int32_t content_type_string_length,
              dpi_pkt_infos_t* L3_L4_parsed_information,
              void** app_flow_data,
              void* app_data){

    if((*flow_specific_app_data==NULL) &&
       (strncmp(content_type_string,
                "video/x-flv",
                content_type_string_length)==0)){
        /**
         * Simply to indicate to the main a
         * match has been found.
         **/
        *app_flow_data=FLASH_CONTENT_FOUND;
    }

}

struct packet_information{
    char* pkt; /** Packet **/
    uint len; /** Length **/
    uint ts; /** Timestamp **/
};

int main(int argc, char** argv){
    dpi_http_header_field_callback* cb[1]={&callback};
    /**
     * Indicates to the library that the callback
     * is associated to "Content-Type" header field.
     **/
    const char* ct[1]={"Content-Type"};

    dpi_http_callbacks_t callbacks={
        .header_url_callback=NULL, .header_names=ct,
        .num_header_types=1, .header_types_callbacks=cb,
        .header_completion_callback=NULL,
        .http_body_callback=NULL};

    dpi_library_state_t *state=dpi_init_stateful();
    dpi_http_activate_callbacks(state, &callbacks, NULL);

    struct packet_information packet;
    dpi_identification_result_t r;

    while(true){
        packet=receive_packet();

        r=dpi_stateful_identify_application_protocol(
            state, packet.pkt, packet.len, packet.ts);

        if(r.status>=0 &&
           r.protocol==HTTP &&
           r.app_flow_data==FLASH_CONTENT_FOUND){
            printf("Flash stream found.\n");
        }
    }

    dpi_terminate(state);
}
```

The struct returned by the protocol identification function contains:

- The status of the processing. When an error occurs (e.g. if the packet is truncated) it will be less than zero. If it is greater or equal than zero, the processing succeeded and this field provides additional information about the processed packet (e.g. if it was a fragment or if it was the TCP segment which terminated the connection).

- The application protocol if it has been already identified, otherwise UNKNOWN protocol is returned.

- The *application flow data*, used by the callbacks to communicate to the rest of the application the result of the processing over the flow to which the packet belongs.

3.6 Demo application

A demo application has been implemented to show the potentialities of the framework and to test the developed code into a real environment.

The target of this application is to scan all the HTTP traffic and to search for well known threats signatures, similarly to what is done by Intrusion Detection and Prevention Systems (IDS/IPS) [41, 42]. As starting point, we used the open source application available in [84] and, with few changes, we modified it in order to read packets from a *pcap* file and to pass them to the framework. It's worth noting that, from a functional point of view, reading the packets from a file or from the network doesn't make any difference. Indeed, the framework has been developed in such a way that it can be independent from the specific technology used to read the packets and it only requires, for each packet, its length and a pointer to a memory area containing it.

Actually, to avoid that the packet reading turns out to be a bottleneck for the application, we first load all the packets in the main memory and then we pass to the framework, one after the other, the pointers to the packets. In this way, we can test the application under the highest load conditions possible on that target architecture.

After that the packets have been loaded, the application reads the set of virus signatures from a text file and insert them into a trie. The signatures check is done inside an HTTP callback similar to the one described in listing 3.1. This callback is executed on all the packets which contain a chunk of an HTTP *body* and, using a modified version of the Aho/Corasick pattern matching algorithm [85], searches if any of the viruses signatures loaded in the trie is present inside the packet. In order to allow a correct recognition of the signature, the framework must be run with TCP reordering enabled. Moreover, the algorithm is written in such a way that when the next *body* chunk is received, it starts the check from the point where it was left in the previous chunk.

It's important to notice that, from the programmer point of view, all the packet processing, TCP reordering, HTTP parsing and data extraction comes for free and it only needs few calls to the framework. In this way, the programmer can focus on what to do with the data instead of how to extract and manage the needed information from the traffic on the network.

As a further example, by changing a single line in the application code it could be modified in order to search patterns in other parts of the HTTP packet. For example, to implement an application which checks if an host is trying to connect to a blacklisted HTTP URL, the programmer needs to indicate that the same callback must be executed on the `Host` *field* instead of the HTTP *body.*

This application has been validated by executing it over a *pcap* file containing the HTTP transfer of an infected file and by checking that it is successfully recognized by the application.

3.7 Parallel structure

We will now describe how we modified the framework in order to exploit the underlying multiprocessor architecture using the concepts introduced in the previous chapter.

3.7.1 Interface with the application

Differently from the sequential version, in this case the execution control is completely managed by the framework. Therefore, instead of calling the packet processing function directly, the application must specify a callback to let the framework read the packets from the network (*reading callback*) and another one to let it process the results of the identification process (*processing callback*). Moreover, it can specify a pointer to some *callback data* (e.g. a network socket) to be accessed from both the callbacks.

This model of interaction is depicted in figure 3.7, where the dashed boxes represent the callbacks specified by the application.

The *reading callback* has in input the *callback data* and returns to the framework a structure containing: a pointer to the packet, its length, the current time and an arbitrary pointer (*application pointer*). This can be used to associate some application information to the packet and therefore to the corresponding result.

For example, if the application reads the packets from multiple network interfaces, and the actions to take on the processing result changes depending on the interface from which the packet has been received, it may be used to associate the interface to the packet. In this way, when the result of the processing is obtained, the *processing callback* knows the interface from which the packet was received and therefore, the specific action to take on it.

It's important to point out that this pointer was not needed in the sequential framework because the model of interaction with the application was different.

If the returned packet pointer is equal to NULL, then it will be interpreted as an indication that the processing is finished and that the application wants to terminate the framework. This could be useful in case the application

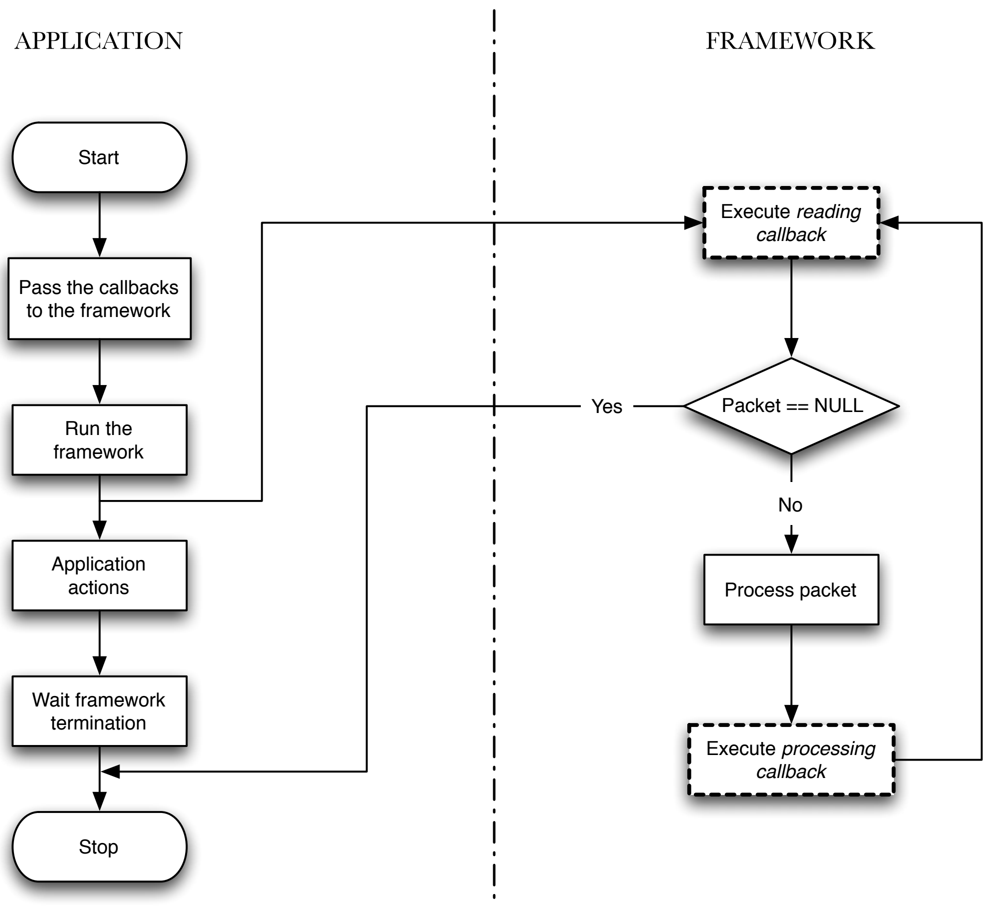

APPLICATION
FRAMEWORK
Start
Pass the callbacks to the framework
Run the framework
Application actions
Wait framework termination
Stop
Execute *reading* callback
Packet == NULL
Yes
No
Process packet
Execute *processing* callback

Figure 3.7: Interaction between the application and the parallel framework

received a termination signal from another process and want to provoke a "gentle termination".

The *processing callback* has in input: the *callback data*, the *application pointer* specified by the *reading callback* and the processing result, containing the same information's that were returned by the protocol identification function of the sequential framework.

After specifying the callbacks, the application can run the framework and, concurrently, keep doing some other work. For example, if the application behaves like a firewall, after starting the framework it could wait for changes in protocols filtering policies and, consequently, invoke over the framework the functions to enable or disable the corresponding inspectors.

Eventually, when the application has no more actions to do, it can synchronously wait for the framework termination.

3.7.2 Implementation details

From the point of view of the implementation, we used the `farm` and `pipeline` constructs provided by FastFlow to implement the concepts described in Chapter 2. FastFlow needs that the task that will travel over the computation graph are void pointers. However, to avoid to dynamically allocate and deallocate each of these task, we provided a feedback channel from the collector to the emitter of the farm in such a way that the tasks, when arrive to the end of the graph, can be sent back to the emitter to be reused again.

Regarding the parallelization of the emitter, one of the problems which may arise is that, in general, it could change the order of the received packets. Consequently, we could incur in the cost of TCP normalization also in the cases in which we originally received an ordered stream. To avoid this, we provided the possibility to define the L3 farm as an *order preserving* farm. This have been implemented by simply changing the scheduling strategy used by the emitter (using a variant of *round robin*) and the collection strategy used by the collector.

Therefore, with this small modification in emitter and collector strategies, the packets exit from L3 farm in the same order in which they entered. However, as we will see later, this will not lead in all the cases to an increase of the overall performances. Indeed, there are cases in which is preferable to incur in the cost of TCP reordering instead of blocking the L3 emitter or the L3 collector over the communication channels.

Moreover, we also provided the possibility to apply an *on demand* scheduling strategy for the L3 farm. We will analyze in section 4.3 the impact of these different strategies.

Another important consideration to be done is about the management of IP defragmentation in the cases in which the processing of the network and transport headers is managed by the L3 farm. In this case indeed each worker may receive a fragment and therefore may need to access the table containing all the outstanding fragmented datagrams. Since it is shared among the different L3 workers, they need to access it in mutual exclusion. For this reason, we decided to protect the access to the datagram fragments table by using *spin locks*. It's important to notice that, since the amount of fragmented traffic is usually below the 1% of the total [86], this would not considerably harm the final performances of the framework. However, as a future work, this could be further optimized in order to reduce the critical section length.

Moreover, we should consider now the way in which the hash table has been split among the L7 workers. In principle, since each partition is independent, we could use n distinct hash tables of size $\frac{s}{n}$, where s is the size of the global table and n is the number of L7 workers. However, we decided to have only one table divided among the workers by means of two indexes for each worker. In this way, the worker i will only access the flows in the buckets of the table included in the bounds $[low_i, high_i[$.

The reason behind this decision can be found considering some of the possible future developments for this work. Indeed, having a global and partitioned structure it should be possible, in principle, to dynamically change the sizes of the partitions during the execution of the framework by simply modifying the bounds of each individual worker. This could be useful for two main purposes:

Dynamic workers reconfiguration This can be useful when we want to dynamically change the parallelism degree according to the current input bandwidth. Therefore, if the number of workers is increased or decreased, we need consequently to split again the table among the new number of workers.

Dynamic flow distribution This could be used in the cases in which the workers are unbalanced. Indeed, also if the hash functions we proposed present a good uniformity of the flow key, not all the flows require the same processing cost. This can depend from different factors like: protocol of the flow, its length or the average size of its packets. Accordingly, when the workers are unbalanced, the flows could be redistributed in a different way to try to mitigate this effect.

The last thing to point out is the way in which the callbacks that the application defines over protocol data and metadata are invoked. Since they must be called after the extraction of the flow information from the hash table, they will be invoked by the L7 workers. Such that the packets belonging to the same flow will be processed sequentially, the data about the flow can be accessed by the callback without any need of synchronization. However, if the callback needs to access some data which is common to all the flows, then the application must consider that it could be accessed concurrently by multiple threads of the framework.

Chapter 4

Experimental results

In this chapter we assess the design with a complete set of experiments validating the different design choices as well as the overall performance of the framework. The performances of the framework have first been analyzed considering the case in which only the identification of the application protocol is required. After that, we studied the case in which, after that the application protocol has been identified, the extraction and processing of the data contained inside the application payload is required, by using the demo application described in section 3.6.

Unit tests have been developed together with the framework in order to check the correctness of some features like: parsing of network and transport headers, IP defragmentation, TCP stream reassembly and protocols inspectors. These tests work by reading network packets from some *pcap* [87] files provided with the framework and by comparing the obtained results with the expected ones.

All the performance related tests, in order to be reproducible, have been executed over traffic stored in *pcap* files. Moreover, in order to be able to test the framework under the maximum load conditions possible, we read the file containing the traffic directly from the local machine memory instead of reading them from the network. In this way we are able to avoid possible bottlenecks due to the specific packet reading technologies and we can isolate our results from external factors that are not dependent from our work. Furthermore, we would like to avoid to shift this problem to the one caused by the bottleneck due to the limited bandwidth of the I/O transfers from the disk. For this reason, when reading the file, we first load the entire *pcap* in main memory and then we start analyzing it.

It's important to point out that this doesn't affect the validity of our results and that, modelling the capture of the packets in this way, we are testing our framework in the worst conditions possible. In a real situation indeed the latency to read a packet will be, in general, higher than the one that we have in our case where the packet is already present in memory and we simply need to dereference a pointer. Accordingly, the framework will have an input bandwidth lower than the one used to execute these tests that, consequently, can be considered as an analysis of the worst case.

For the different tests we used the following datasets:

CAIDA This dataset [88] contains traffic captured from a monitor located in a data center in Chicago, and connected to a 10GigE backbone link of a Tier1 ISP between Chicago and Seattle. The dataset has been kindly provided by CAIDA association [89] and contains about 24 millions of IPv4 packets distributed in 1428689 IPv4 flows[1]. The payload of the packets is not present for privacy reasons. However, this can be used to compare the proposed hash functions, as in this case we need to have data only up to the transport protocol header.

Sigcomm This dataset [90] contains a collection of the wireless IP traffic captured over the entire period of a three day conference. It contains about 16 millions of IPv4 packets distributed in 298449 IPv4 flows. Also in this case the payload has been cut off from the packets. Consequently, also this dataset will be used only to compare the different hash functions we proposed.

Synthetic This dataset contains 1428043 synthetic IPv4 packets. These packets are distributed over 13314 HTTP flows, each of which correspond to a transfer of a large file between two hosts.

Darpa This dataset [91] has been collected by the Cyber Systems and Technology Group of MIT Lincoln Laboratory and was commonly used to evaluate intrusion detection systems. It contains 1308081 IPv4 packets distributed in 38985 IPv4 flows.

Local This dataset contains traffic captured from a laptop connected to an home network. It contains 524761 IPv4 packets distributed in 17939 IPv4 flows and 34635 IPv6 packets distributed in 1670 IPv6 flows.

All the tests have been executed over a NUMA machine composed by two Intel Xeon E5-2650 @ 2.00GHz nodes with 8 hyperthreaded cores on each of them. Each NUMA node has 16GB of main memory, a shared 20MB L3 cache and private 256KB L2 and 32KB L1 caches for each core.

[1]The dataset contains a low number of IPv6 flows so the analysis is concentrated on IPv4 flows only.

In all the described experiments, when error bars are present, they represent the standard deviation from the mean. Error bars will not be shown for the results characterized by a negligible standard deviation.

4.1 Hash functions analysis

We now compare the hash functions we implemented to access the hash table containing the data about the flows. This comparison have been done in order to verify if more complicated hash functions, which in general are characterized by an higher latency, are also characterized by a better distribution uniformity with respect to simpler but faster hash functions.

First of all, we compare the distributions of the functions using the metric described in [92]. This metric evaluates the uniformity of an hash function using the following formula:

$$\frac{\sum_{j=0}^{m-1} \frac{b_j(b_j+1)}{2}}{\frac{n}{2m}(n+2m-1)}$$

where b_j is the number of items in $j-th$ slot, m is the number of slots, and n is the total number of items. The more this ratio is close to one, more the function is close to the uniform hash function.

In figure 4.1 we present the results obtained for the functions provided by the framework, varying the load factor of the hash table. For this comparison we used the **CAIDA** dataset. As we can see, all the four hash functions have a distribution which is very close to the uniform one.

Furthermore, in figure 4.2, we illustrate the comparison of the execution times of the parts of the framework directly influenced by the computation of the hash functions. As we can see, as a consequence of the good uniformity of all the proposed functions, the average time spent in accessing the collision lists is the almost the same in all the four cases.

On the other hand, the time spent in computing the function changes and, as we will see later, also if it may seem negligible with respect to the table access time, it has a non negligible effect on the bandwidth of the framework.

Similar results have been obtained also using the other datasets. However, due to space constraints, we will not show them here but we will directly show their effect on the total bandwidth of the framework (expressed in millions of packets per second) in figure 4.3.

As expected, the difference in the time spent to compute the hash function is reflected over the total bandwidth of the framework.

Moreover, from figure 4.3 we can analyze the behaviour of the framework under different kinds of traffic. As we can see, the dataset which exhibit

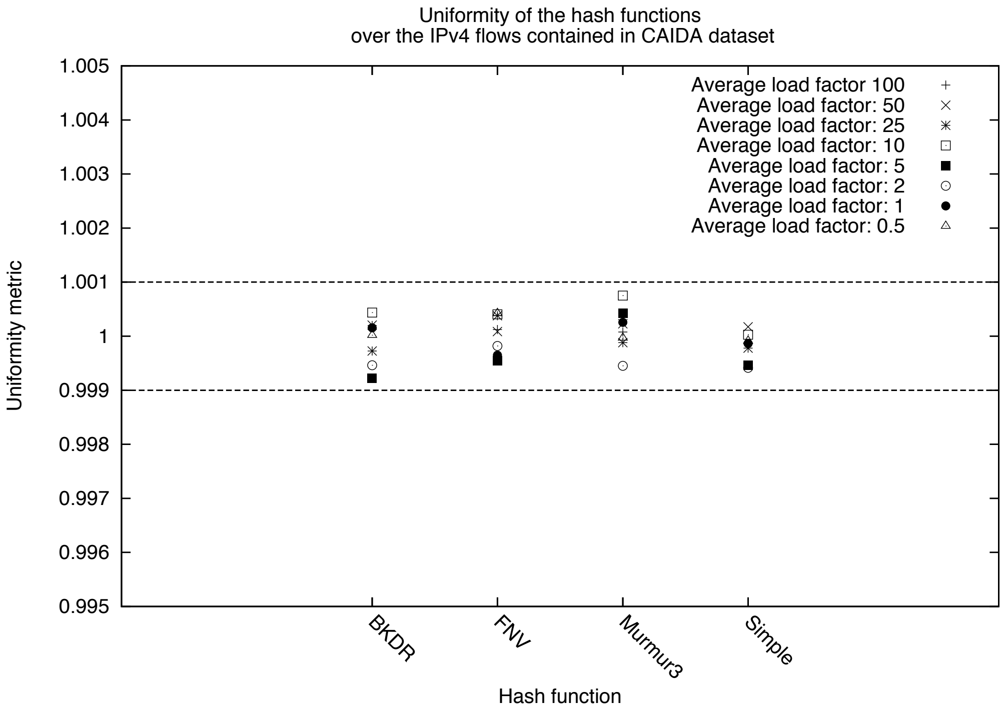


Figure 4.1: Hash functions uniformity over the flows contained in **CAIDA** dataset

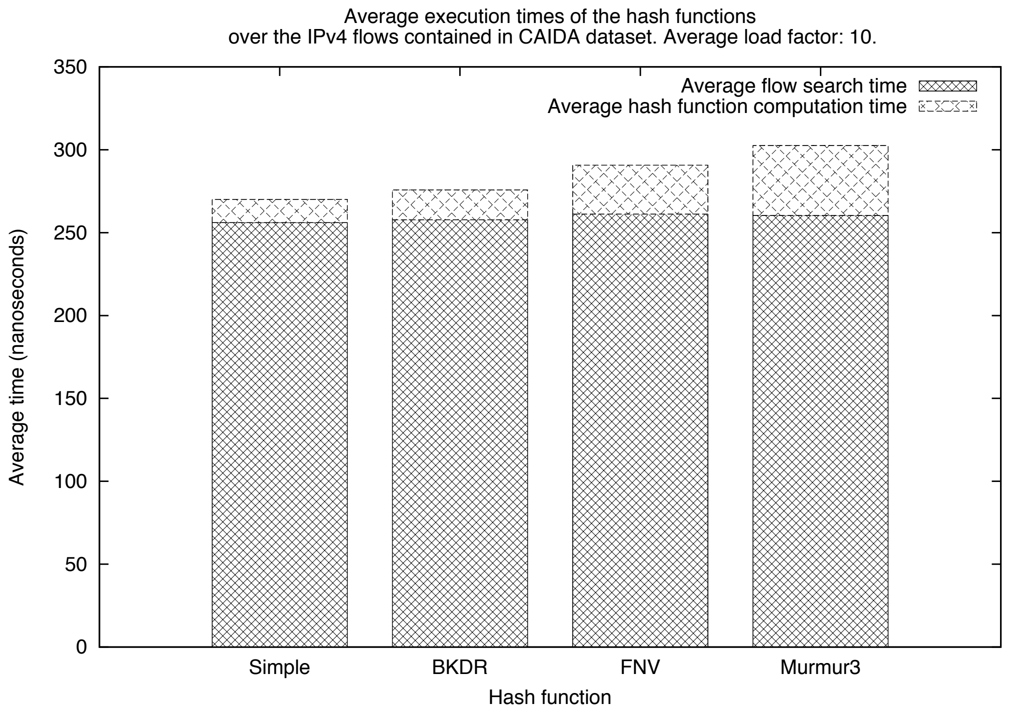


Figure 4.2: Hash functions execution times over the flows contained in **CAIDA** dataset

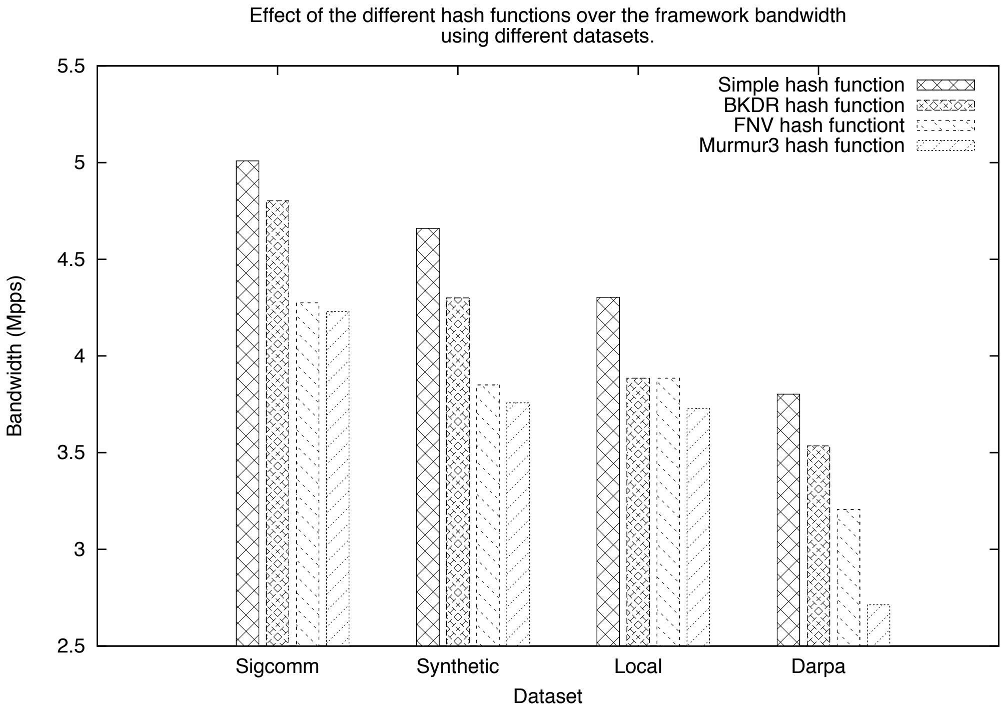


Figure 4.3: Comparison of the impact of the different hash functions over the framework bandwidth (in millions of packets per second)

the highest bandwidth is the **Sigcomm** dataset, due to the absence of the packet payload. Consequently, the protocol inspectors will never be called for this dataset, while we still execute the operations relative to network and transport header parsing, flow data retrieval and IP and TCP normalization.

Concerning the **Synthetic** dataset, it is characterized by flows with a long duration and thus, as we have described in Chapter 2, we will execute the inspectors only on the first packets of each flow. Then, once it has been identified, the inspectors will no more be called for the flow.

Eventually, **Local** and **Darpa** datasets require more computational effort to be processed, since they are characterized by shorter flows and thus the inspectors will be called more frequently.

In this section we have seen that the simplest and fastest hash function among the ones proposed present a distribution uniformity comparable to that of the other functions. Moreover, this has been verified both by analyzing in isolation the parts of the framework which are directly influenced by the hash function and by considering the impact on the overall bandwidth.

4.2 Analysis of Move To Front technique

As we described in section 3.3, the collision lists are kept sorted from the most recent to the least recent flow. This should have a twofold advantage:

- From one side it allows, in principle, to keep the most frequently updated flows in the first positions of the collision list, thus reducing the latency required to find a flow in the hash table.
- On the other hand, when scanning the table to find expired flows, we can usually avoid to scan each collision list entirely. Indeed, since each list is sorted, we can start scanning the list from the end up to the point in which we find a flow which is not expired.

Accordingly, we would like to evaluate if this technique actually increases the overall bandwidth of the framework with respect to the case in which it is not used. In order to assess the advantage of this technique, we made our test over the **Sigcomm** dataset varying the load factor of the hash table. Indeed, since the latency required to find a flow increases when the average length of the collision lists increase, the advantage should be more noticeable incrementing the load factor.

In figure 4.4 we can see that using MTF we achieve significantly better results. Moreover, as expected, this improvement increases while increasing the average load factor of the table.

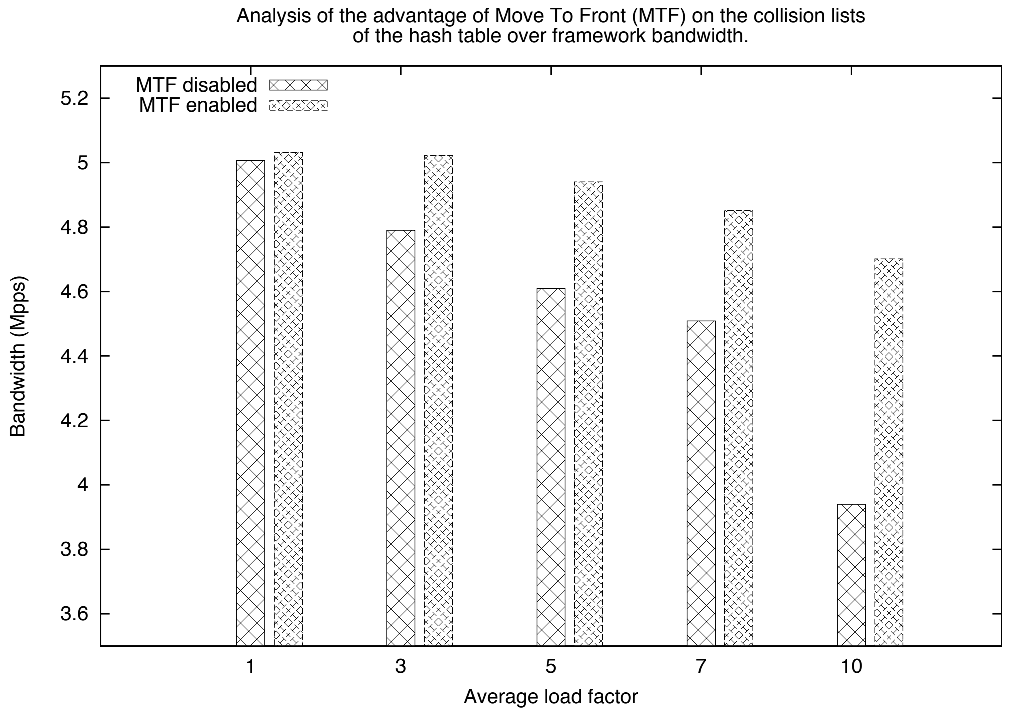


Figure 4.4: Analysis, using Sigcomm dataset, of the impact of MTF strategy over the global bandwidth of the framework varying the average load factor of the table

In this section we have thus validated the implementation choice we made in section 3.3 by analyzing its impact on the overall bandwidth of the framework.

4.3 Comparison of L3 farm scheduling strategies

As we described in section 2.2, when the emitter is parallelized and replaced by the L3 farm, we should consider the possibility that it could reorder the received packets. Therefore, also if they arrive in the correct order to the framework, they could be received out of order by the L7 worker and, consequently, we could incur in the latency caused by TCP normalization also when it is not really needed.

To avoid this problem, we proposed a scheduling strategy which preserves the order of the packets. For this reason in figure 4.5 we compare the two

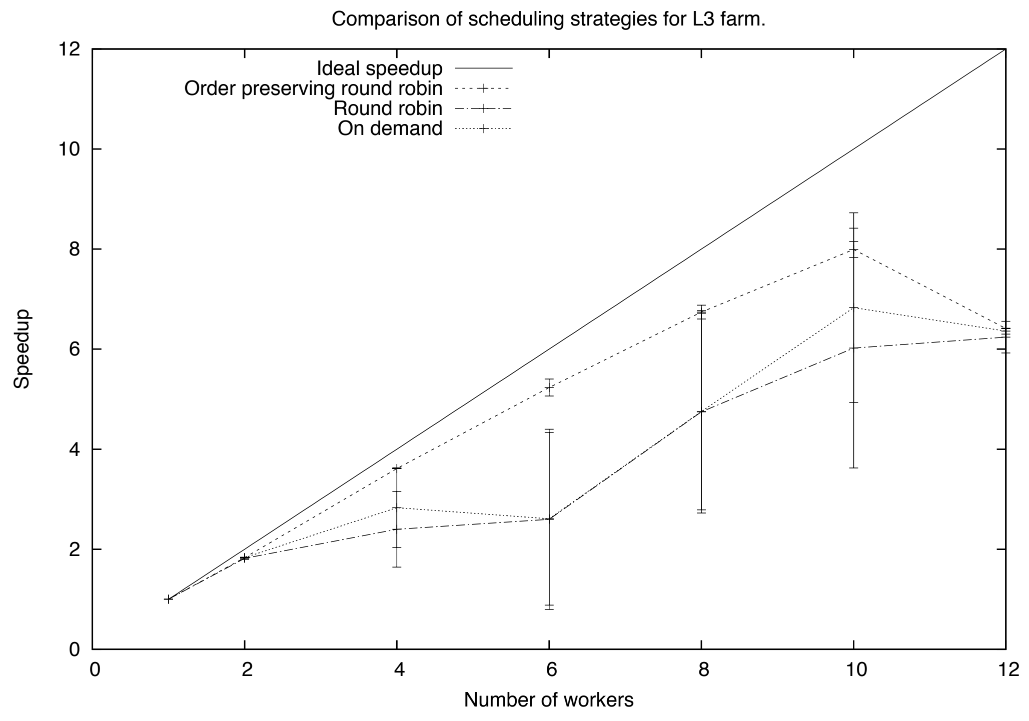


Figure 4.5: Comparison of the impact of the different scheduling strategies for the L3 farm

non ordering strategies (round robin and on demand), with respect to the ordering scheduling (ordered round robin).

As we can see, the ordering scheduling strategy provides better results with respect to the other two solutions. Moreover, when a strategy which doesn't preserve the order is used, the results are characterized by a much higher standard deviation. This happens because in the different runs the packets could be scheduled in different way and thus incurring in a different cost due to the TCP normalization. Furthermore, increasing the parallelism degree (and thus increasing also the number of workers activated for the L3 farm) we increase also the possible ways in which the packets can be scheduled and, as we can see from the figure, this is reflected on the standard deviation.

For these reasons, when TCP normalization is activated, an ordering strategy should be used to schedule the tasks in L3 farm. Indeed in this section we have shown that, when scheduling strategies which don't preserve the order of the packets are used, we could have a worse speedup due to an increase in the cost of TCP reordering. Consequently, the framework will have a lower bandwidth with respect to the one we have when using an order preserving scheduling strategy.

4.4 Speedup

We will now analyze the speedup achieved thanks to the parallelization of the framework. In the analysis, we will separate the case in which we perform only the protocol identification, from the case in which Network Intelligence (NI) capabilities are required and thus all the packets will be processed also after that the protocol of the corresponding flow has been identified.

4.4.1 Protocol identification

First of all, we will analyze the speedup of the framework when only the protocol identification is performed.

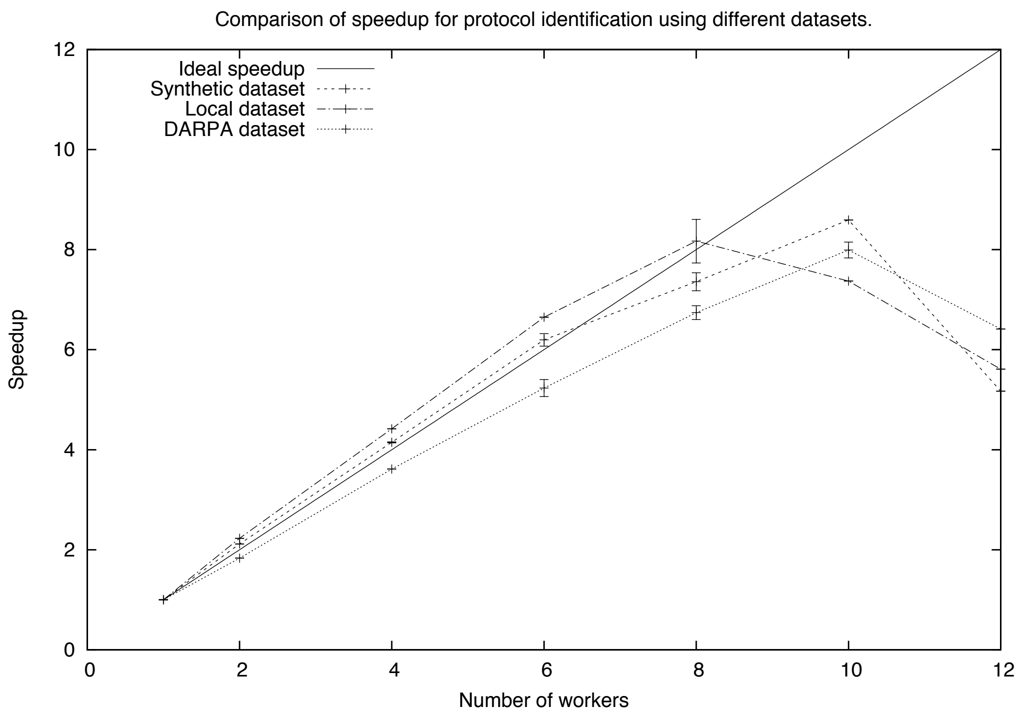


Figure 4.6: Comparison of the speedup for protocol identification using different datasets

Usually, the protocol of an application flow will be identified by inspecting only the first packets carried by it. For example, considering an HTTP flow, the framework will usually identify the protocol by inspecting only the first packets of the flow (which carry the HTTP header). For this reason, for all the remaining packets of the flow, the cost of the inspection will not be payed and the framework will incur only in the latency due to the access to the hash table (which is always needed to check if the flow have been already identified). This is the reason why, in average, the processing of a packet is a very fine grained operation. For example, on the architecture used to run our tests, in many cases it has a latency of $O(100)$ ns. Consequently, as we can see from figure 4.6, there are cases in which we are not able to achieve the ideal speedup.

On the other side, there are cases in which the framework achieves a speedup greater than the ideal one. As we described in section 1.2, this is due to a better temporal locality exploitation. For example, let's consider the hash table that is partitioned among the workers of the L7 farm. In this case, when more than one node is used, each node will access only to a smaller part of the table which, hopefully, will fit in the lower levels of the memory hierarchy.

In this section we have shown that, despite the very low latency of the protocol identification process, the framework is still able to achieve a good speedup. However, due to the low latency of the protocol identification process, we reach a point where the emitters become bottlenecks and thus the framework reaches the saturation. Anyway, as we will see in section 4.4.2, when NI capabilities are required, the latency of the packet processing increases, allowing thus to achieve better results.

4.4.2 Processing of extracted data

We now analyze how the speedup is affected when packet processing capabilities are required, trying to find a lower bound to the processing grain which allows the framework to achieve a good speedup. In figure 4.7 we compare the speedup of the framework varying the latency of the processing function which will be executed for each identified packet.

As we can see, also for very fine grained processing functions, the framework is characterized by a good speedup. Moreover it's worth noting that, as we will see in section 4.4.3, "real" processing functions are characterized by latencies which may also be two order of magnitude greater than the ones we used to make this analysis. Consequently, as we will see, in real monitoring environments the framework is able to reach a speedup very close to the ideal one.

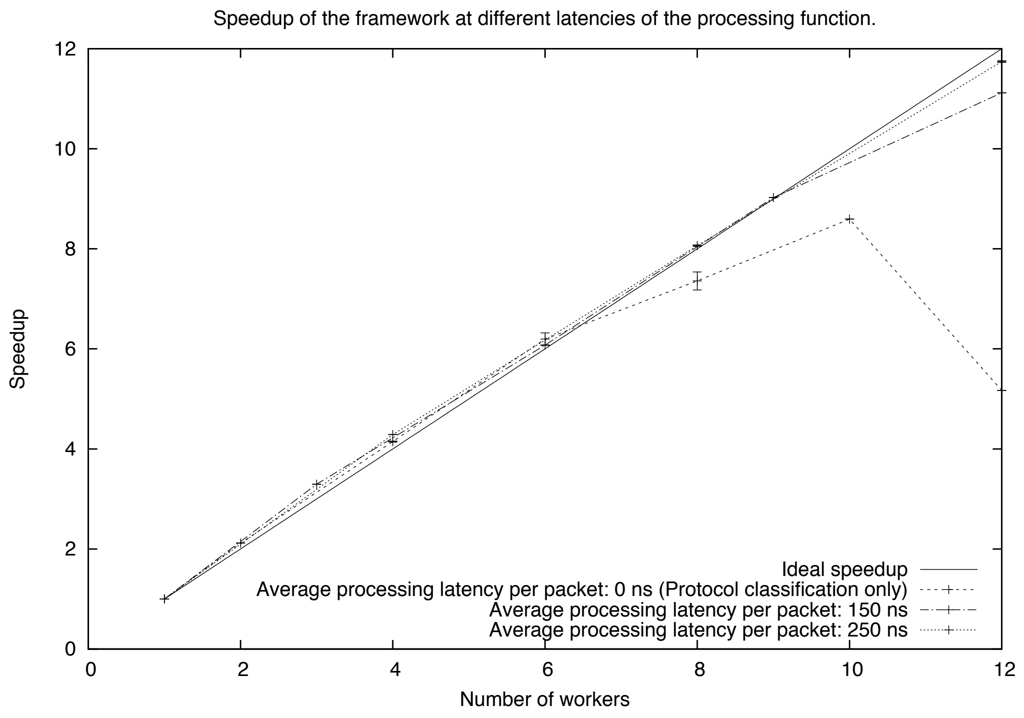


Figure 4.7: Comparison of the speedup when packet processing capabilities are required

4.4.3 Application speedup

Eventually, we analyze the capabilities of the framework under a real monitoring environment as the one described in section 3.6. We briefly recall that the application we built over the developed framework allows to search patterns inside the HTTP packets traveling over the network by using the Aho/Corasick algorithm. Our experiments shown that using a database of 1781 patterns with an average length of 274 characters each, in average, this operation has a latency of $O(10)$ microseconds per packet.

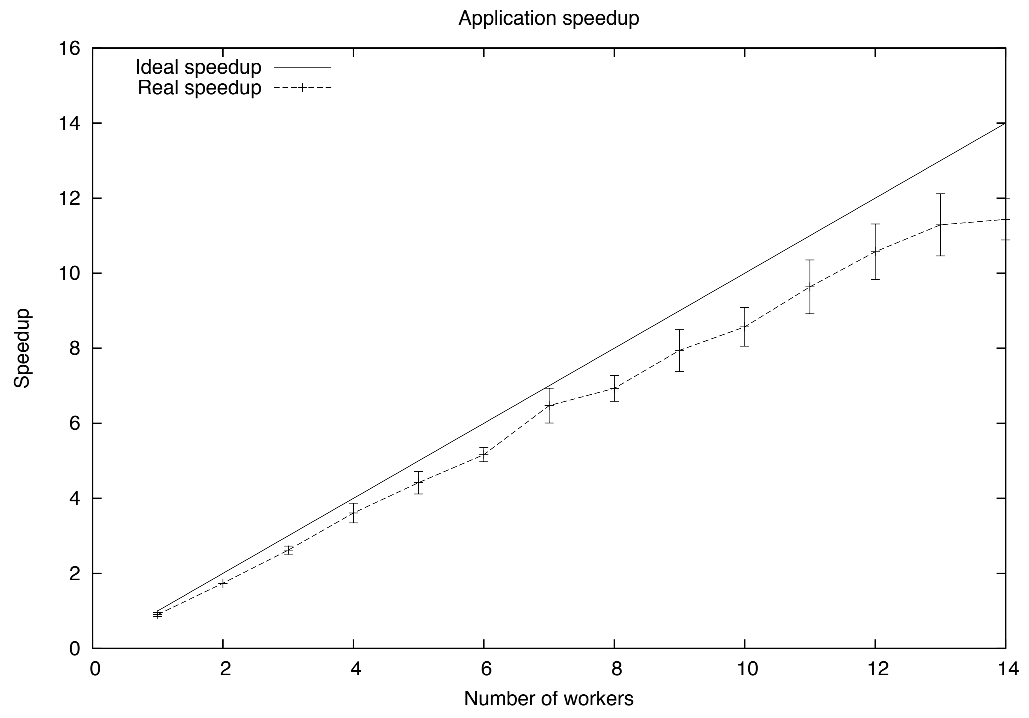


Figure 4.8: Comparison of the speedup of the HTTP payload pattern matching application

As we can see from figure 4.8, the application exhibits a good speedup up to 14 workers.

4.5 Assessment

In this section, we will evaluate the results we obtained using our framework, comparing them with the ones obtained with other existing software and hardware tools.

4.5.1 Comparison with other sequential software solutions

With respect to common and well known existing tools, as shown in Table 4.1, instead of focusing on the number of supported protocols we decided to characterize this work by providing a support for current multicore hardware and by providing to the application programmer the possibility to specify the callback to be used to process specific data and metadata carried by the protocol.

	Our framework	OpenDPI/ nDPI	libprotoident	l7filter
IPv4 and IPv6 normalization (section 3.2.1)	Yes	No	Yes	Yes
“Stateful” mode (section 3.3)	Yes	No	Yes	Yes
TCP normalization (section 3.4.1)	Yes	No	Yes	Yes
Callbacks (section 3.5)	Yes	No	No	No
Multiprocessor support (Chapter 2)	Yes	No	No	No
Supported protocols	10	117/141	250	112

Table 4.1: Comparison of the features of our framework with respect to well known DPI libraries

Having illustrated the main contributions of our framework with respect to popular software tools, we now compare them from the point of view of the achieved performances.

Concerning nDPI, we made the comparison by using both nDPI and our framework over the same machine. Moreover, in order to have a fair comparison, we disabled from our framework the features that are not supported by nDPI and we disabled from nDPI the protocols inspectors not implemented in this work. Furthermore, since nDPI doesn't have its own way to store the flow data, we slightly modified the hash table used by our framework in order to store the data required by nDPI.

Under these conditions, we tested the two framework over different datasets, comparing them over both synthetic and real traffic. Moreover analyzing the code of nDPI we saw that, also if the protocol of the flow has been already identified, it still try to identify the other packets belonging to the same flow. Accordingly, we compared our results with two different versions of nDPI, the native one and the one in which we call the library only if the flow has not been yet identified, obtaining the results shown in figure 4.9

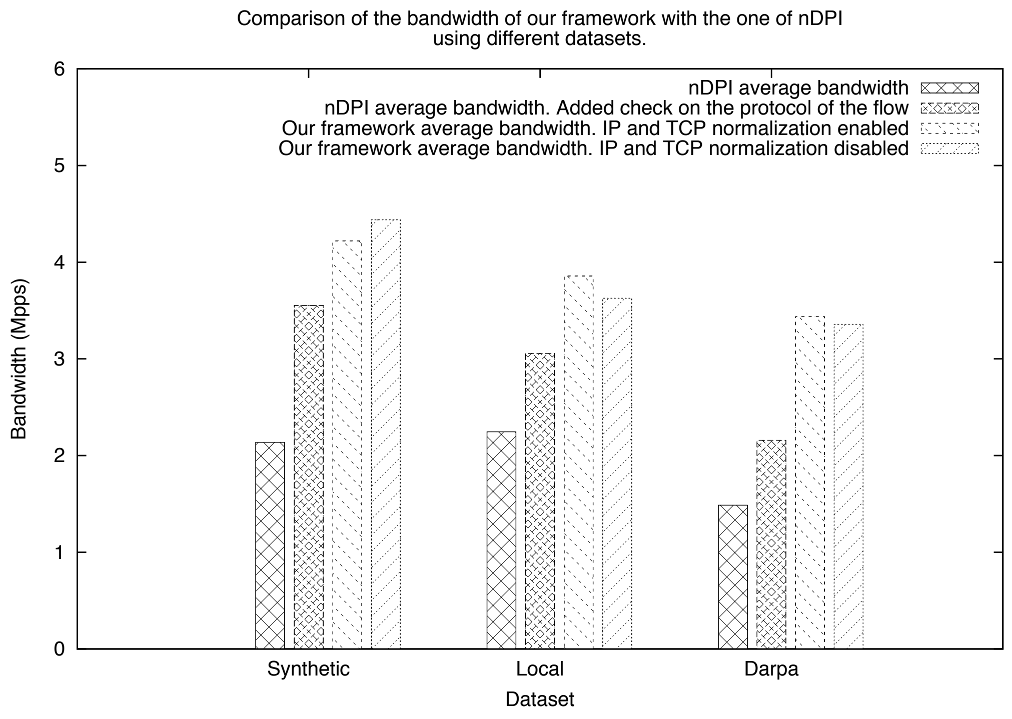


Figure 4.9: Comparison of the bandwidth (in millions of packets per second) of our framework with respect to nDPI over different datasets

As we can see, our framework exhibits always better results, also when IP and TCP normalization is enabled. Moreover it's worth noting that there are cases in which, enabling the TCP normalization, we get an higher bandwidth with respect to the cases in which it is disabled. This derives from the fact that in the former case, when out of order packets are received, we delay the analysis of the flow up to the point in which the stream is in the right order. Consequently, we execute the inspectors a lower number of times and over a longer segment and, since we usually doesn't need to scan the entire segment but only the first part, the cost due to the normalization is amortized and we globally have an higher bandwidth. However, this effect is not present for the **Synthetic** dataset since it contains already ordered data.

Furthermore in this comparison we should take into account that, although we disabled not required protocols, since nDPI has been designed to support more application protocols, its internal data structures (e.g. that used to store the flows information) are usually bigger than the ones we use in our framework. Accordingly, we need to consider that this could be one of the reasons why nDPI exhibits lower performance, since it gains less advantages from spatial locality.

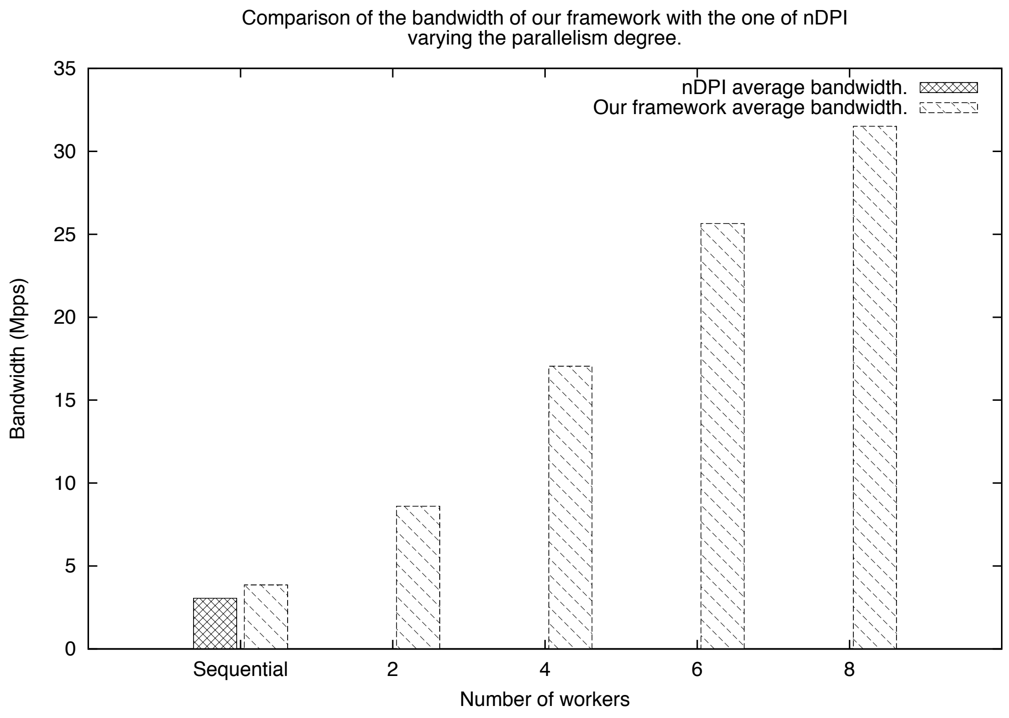


Figure 4.10: Comparison of the bandwidth (in millions of packets per second) of nDPI with respect to our framework varying the number of workers of the farm

Moreover nDPI provides only the possibility to process the packets sequentially and, if multicore support is required, it should be implemented from scratch. Accordingly, in figure 4.10 we compare the bandwidth manageable by nDPI with that of our framework, varying the number of workers of the farm.

Concerning the other tools, according to the authors of [22], l7filter and libprotoident achieve a bandwidth respectively of 2.421 Gbps and 7.752 Gbps. Under presumably similar hardware and using different kinds of real traffic dataset, as shown in Table 4.2 our framework has demonstrated to be able to reach from 5.89 Gbps up to 17.35 Gbps.

However, in this comparison we must consider that these tools support much more protocols with respect to the ones we implemented in our prototype.

Dataset	Bandwidth (Gbps)
Darpa	5.89
Local	17.35

Table 4.2: Bandwidth achieved by our framework using different real traffic datasets

Anyway, the qualitative results should still be valid and our approach, coherently with the considerations done in [22], from the point of view of the achieved bandwidth, should be located between libprotoident and l7filter. Indeed, since Libprotoident is a Lightweight Packet Inspection (LPI) approach and looks only at the first four bytes of the payload, it will probably have an higher bandwidth. However, from a functional point of view, analyzing only the first four bytes could be a limitation since they will not be able to sub classify the protocols or, in general, to extract the data carried by the application.

Concerning l7filter, it exhibits a significant lower bandwidth, mainly due to the inefficiency of regular expression matching with respect to fixed string matching as the one performed in the other cases.

Moreover the main advantage of our approach is that, as shown in section 4.4.1, differently from these tools we are able to exploit the underlying multicore architecture and thus to achieve an higher bandwidth.

Comparison of the speedup with existing software solutions To evaluate the validity of a *structured parallel programming* approach, we will compare the speedup achieved by our framework with that achieved by other DPI tools present in literature. However, since the source code of the other tools is not available, we make our consideration over the results found in the respective works. In Table 4.3 we show the best results obtained for each work in terms of speedup.

Tool	Number of threads	Speedup
[24]	16	10
[25]	14	6.25
[26]	5	1.75
[27]	3	1.98

Table 4.3: Speedup achieved by tools which provide multicore support

As we can see, the other existing tools suffer from limited speedup, also for relatively low parallelism degrees. On the other hand our framework, as shown in section 4.4.1, is able to reach, in the best case a speedup of 8.17 using 8 farm workers when only the protocol identification is required. Moreover for more complex tasks, as the one described in section 3.6, we are able to reach a speedup of 12 with 14 farm workers.

Furthermore, as also stated from their developers, these problems are often related to design choices. Consequently, also if the results presented in the other works are collected using different machines from the one we used, we can still reasonably suppose that from a qualitative point of view, the results would not change significantly.

In this section, we have shown that we are able to often manage higher bandwidths with respect to the other existing DPI tools and, at the same time, to provide better results in term of speedup thanks to an accurate design which exploits the concepts of *structured parallel programming* theory. Although in some cases it is difficult to make a fair performance comparison that takes all the algorithmic, hardware and used datasets details into account, we clearly shown the efficiency of our approach with respect to those used in other works.

4.5.2 Comparison with hardware solutions

Many hardware solutions are proposed in literature. In most cases they perform pattern matching on the packets both by using Field Programmable Gate Arrays (FPGAs) [17, 37, 38] or by using Content Addressable Memories (CAMs) [39, 18, 19, 40].

Pattern matching can aim both to search for application signatures or to search generic patterns inside the payload, similarly to what the application we described in section 3.6 does.

Some of these solutions reach, in the best case, a bandwidth of 20Gbps [40]. Anyway, we need to consider that this is the rate in isolation of the pattern matching process and thus doesn't account the cost of all the processing related to the parsing of the network and transport headers, hash table lookup and TCP normalization. However, these are all tasks that should be considered and implemented in a real environment, since their absence could lead to the impossibility to find these patterns.

Considering only the protocol classification, we are able to reach the same rate of the CAM solution [40] (20Gbps) over the **Darpa** dataset by using only four workers. Similar results have also been obtained using the other datasets. However, we have to consider that the algorithms used in the two cases are different. Our solution indeed has a better knowledge of the protocols and of the structure of the packet and, consequently, each inspector usually needs only few byte before deciding if the protocol matches or not. Conversely, the

CAM solution searches for a signature inside all the packet payload and thus it usually scans it entirely before moving to the next signature.

For these reasons, a more significant comparison can be done by evaluate the two solutions over a pattern matching application as the one we described in section 3.6 and which have implemented using the framework developed in this thesis. In both cases, the results are taken by considering the matching over a database of 117 patterns with an average length of 67 characters each.

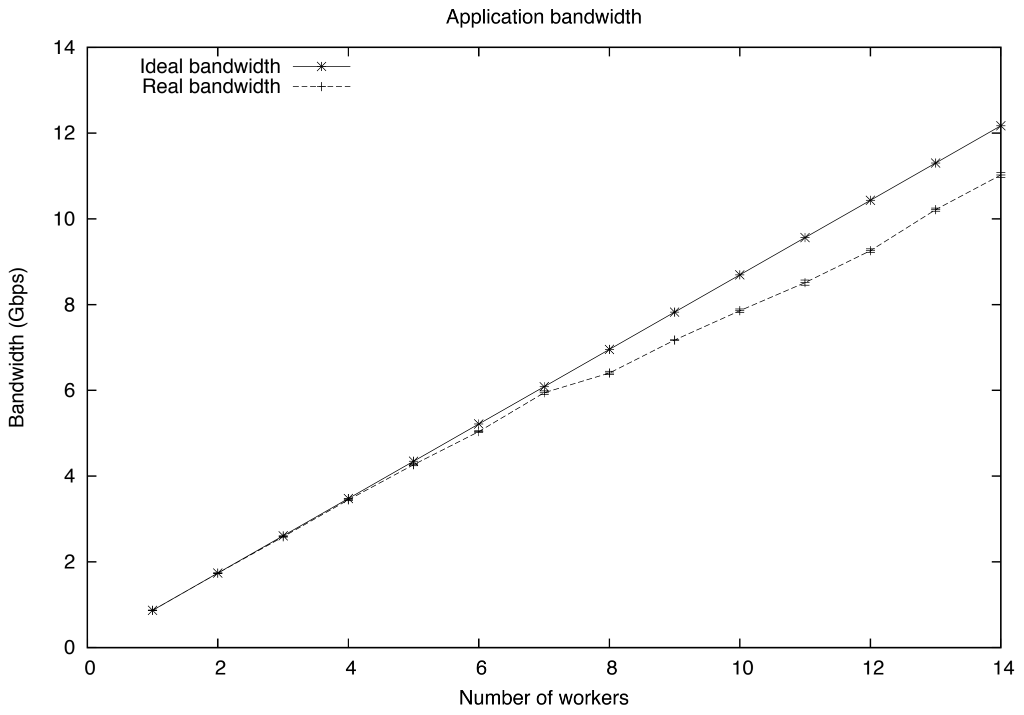


Figure 4.11: Bandwidth of the HTTP payload pattern matching application

Under these conditions, as we can see from figure 4.11, we are able to reach a rate of 11Gbps by using 14 workers. This is a positive result since we are able to achieve comparable rates to those achieved by dedicated hardware solutions and, at the same time, to provide the flexibility and ease of use which characterize software solutions with respect to the ones implemented through dedicated hardware. Moreover, as we said before, we should consider that our results also account the additional overhead due to all the other parts related to the packet processing and not only to pattern matching. From the point of view of the effectiveness, this allows our work to identify the patterns independently from the fact that the data are reordered or segmented while, using only a CAM or FPGA solution, this would not be possible since it doesn't have any knowledge about the structure of the packets and simply search "blindly" in the traffic which travels over the network. Furthermore our solution allows not only to develop pattern matching applications but also any other type of traffic management applications which require to access and process specific parts of the packet payload.

Lastly it's important to notice that, in principle, nothing prevents our framework to take advantage of this kind of solutions for some specific parts of the protocol identification process by offloading them to FPGA or hardware coprocessors. As an example, considering the demo application described in section 3.6, the application programmer could decide, if needed, to offload the pattern search inside the HTTP payload to an hardware coprocessors and thus to take advantage from dedicated hardware performances.

Chapter 5

Conclusions

The original goal of this thesis was to explore the possibility to apply *structured parallel programming* theory to support the implementation of a DPI framework capable to manage current network rates by exploiting commodity multicore architectures. To achieve our target, we implemented these concepts using FastFlow, a parallel programming library which provides very low latency communication mechanisms to support high bandwidth streaming applications.

Our work allows to identify the application protocol carried by the packet by inspecting its contents up to the application layer and by checking the bytes situated in some specific locations or, in more complex cases, by implementing a full protocol parser. Since such kind of accurate identification requires to maintain some state about each application flow, we designed and implemented efficient data structures to store such data that could be easily partitioned among different threads when multicore support is required. Promising results have been obtained for both the sequential and the parallel version of the framework.

Concerning the sequential framework, we validated the approach by comparing the results we obtained with those achieved by well known DPI libraries: we are able to achieve comparable, and also better results in term of number of packets processed per second.

To support an high degree of flexibility so that the framework can be used for different monitoring applications, we allow the application programmer to specify which part of the protocol metadata or of the data carried by the protocol are actually needed and how to process them. To correctly perform this kind of tasks, it may often be required that the data are processed in the same way they are sent by the application. Consequently, we implemented IP and TCP normalization to rearrange and manage fragmented or out of order data. Since this is a risky task which could be affected by security threats, this phase required careful design and implementation choices. We validated the flexibility and efficiency of our framework by using it to implement an application which searches some specified patterns inside all the HTTP traffic travelling over the network, performing thus a task similar to that performed by common Intrusion Detection/Prevention Systems (IDS/IPS).

In order to reach the target of this thesis and to efficiently exploit the underlying multicore architecture, the framework has been designed as a farm and, to avoid any kind of unnecessary synchronization mechanisms, the flow data have been partitioned among its workers. Since in some cases the emitter may be a bottleneck, we provided the possibility to parallelize it by means of another farm, obtaining thus a solution composed by two pipelined farms. To avoid the reordering of the packets, we proposed and validated an order preserving strategy for the L3 farm, allowing to prevent unnecessary costs due to TCP reordering and leading thus to better results.

We then validated our approach by showing that we are able to achieve a good speedup also when only protocol identification is required. However, since under certain traffic conditions it could be a very low latency task, we reach the saturation around 8-10 workers.

Moreover we studied the capabilities of our approach in the cases in which, after the protocol has been identified, further payload processing is required. Consequently, we searched the lower bound to the latency of the processing function which allows the framework to achieve a good speedup, founding that it can be achieved also by using functions characterized by a latency two or three order of magnitude lower than that required by real processing applications. Our results have been further validated by comparing the pattern matching application we developed using our framework, with a similar one implemented over dedicated hardware. The comparison showed that we are able to reach around 11 Gbps versus the 20 Gbps reached by the hardware solution. However, the dedicated hardware solution only offers pattern matching capabilities without any knowledge about the structure of the packets, thus providing a lower accuracy to the one we provided implementing also flow management and TCP normalization. Furthermore, our solution is characterized by the flexibility and ease of use which represent one of the biggest advantages of software solutions with respect to dedicated hardware ones. Anyway, nothing prevents our framework to use these hardware solutions to perform some specific tasks.

Our work could still be improved by addressing some problems such as:

Flow unbalancing Even if the hash functions we proposed present a good distribution of the flow key, the flows themselves could still be unbalanced. Accordingly, also if the number of flows managed by the different workers is equally distributed, we could still have an unbalance from the point of view of the managed packets. To solve this problem, the hash table we proposed in our implementation could be modified to dynamically resize is partitions accordingly to the current load of each worker of the farm.

Resizing of hash table Despite the hash functions we proposed are characterized by a good uniformity of their distributions, we could still have cases in which one collision list is particularly unbalanced with respect to the others. Accordingly, we could need some mechanism to dynamically resize the hash table and to redistribute the flows contained in it.

Adaptivity As we described in section 1.2.3 it is very difficult to foresee the average values of the interarrival time or the latencies of the stages of the processing, since they could be influenced by many different factors. Accordingly, when an high efficiency is required and under-provisioning and over-provisioning need to be avoided, our implementation could be modified in such a way that it can dynamically increase or decrease the number of activated threads and, possibly, also to change the parallelism form. This is one of the reasons why, instead of providing one distinct hash table for each worker of the farm, we decided to use a single hash table partitioned among them by means of bounds.

Support for more protocols The provided protocol inspectors may be improved and extended with data and metadata extraction capabilities. In this work, we provided the possibility to extract and process data and metadata only for HTTP protocol. However, the framework has been designed and implemented to support it possibly for all the implemented protocols. Furthermore, inspectors for other protocols could be added, either by using payload inspection techniques or by using statistical approaches.

These problems have not been addressed in this work due to time and size constraints of the thesis. The starting goal of the thesis has been fully achieved, proving the validity of a *structured parallel programming* approach to solve this kind of applications using commodity hardware.

Bibliography

[1] FastFlow - Website. `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`, 2012. [Online; accessed 23-December-2012].

[2] Inc. eSoft. White Paper - Modern Network Security: The Migration to Deep Packet Inspection. `www.esoft.com/content/pdf/dpi-migration-whitepaper.pdf`, 2012. [Online; accessed 23-December-2012].

[3] Elfiq App Optimizer. `http://www.elfiq.com/appoptimizer`. [Online; accessed 14-January-2013].

[4] Meraki - Application QoS. `http://www.meraki.com/technologies/application-qos`. [Online; accessed 14-January-2013].

[5] Clifton Phua. Protecting organisations from personal data breaches. *Computer Fraud and Security*, 2009(1):13 – 18, 2009.

[6] Martin H. Bosworth. ChoicePoint Settles Data Breach Lawsuit. `http://www.consumeraffairs.com/choicepoint`. [Online; accessed 14-January-2013].

[7] US Federal Communications Commission. Children's Internet Protection Act. `http://www.fcc.gov/guides/childrens-internet-protection-act`, 2012. [Online; accessed 23-December-2012].

[8] Research note - Facebook: Measuring the cost to business of social notworking. `http://nucleusresearch.com/research/notes-and-reports/facebook-measuring-the-cost-to-business-of-social-notworking/`, 2009. [Online; accessed 23-December-2012].

[9] Cisco IOS Netflow. `http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html`, 2012. [Online; accessed 23-December-2012].

[10] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy*. Society Press, 2002.

[11] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Automated traffic classification and application identification using machine learning. In *Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary*. IEEE Computer Society, 2005.

[12] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. *SIGCOMM Computer Communication Review*, 37(1):5–16, January 2007.

[13] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. *SIGCOMM Computer Communication Review*, 41(4), August 2010.

[14] Qosmos DeepFlow Probes - Ready-to-Use Network Intelligence. `http://www.qosmos.com/new-qosmos-deepflow-probes-deliver-the-power-of-network-intelligence-in-ready-to-use-formats/`. [Online; accessed 14-January-2013].

[15] G. Ziemba, D. Reed, and P. Traina. RFC 1858 - Security Considerations for IP Fragment Filtering. `http://tools.ietf.org/rfc/rfc1858.txt`, 2012. [Online; accessed 23-December-2012].

[16] Sarang Dharmapurikar and Vern Paxson. Robust TCP stream reassembly in the presence of adversaries. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. USENIX Association, 2005.

[17] Monther Aldwairi, Thomas Conte, and Paul Franzon. Configurable string matching hardware for speeding up intrusion detection. *SIGARCH Computer Architecture News*, 33(1):99–107, March 2005.

[18] Yaron Weinsberg, Shimrit Tzur-david, and Danny Dolev. High performance string matching algorithm for a network intrusion prevention system (nips). In *High Performance Switching and Routing*, 2006.

[19] Sherif Yusuf and Wayne Luk. Bitwise optimised CAM for Network Intrusion Detection Systems. In *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications*. IEEE, 2005.

[20] OpenDPI - Website. `http://www.opendpi.org/`, 2012. [Online; accessed 23-December-2012].

[21] l7filter - Website. `http://l7-filter.clearfoundation.com/`, 2012. [Online; accessed 23-December-2012].

[22] Shane Alcock and Richard Nelson. Libprotoident: traffic classification using Lightweight Packet Inspection. `www.wand.net.nz/~salcock/lpi/lpi.pdf`. [Online; accessed 12-February-2013].

[23] nDPI - Website. `http://www.ntop.org/products/ndpi/`, 2012. [Online; accessed 23-December-2012].

[24] Yunchun Li and Xinxin Qiao. A parallel packet processing method on multi-core systems. In *Proceedings of the 2011 10th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*. IEEE Computer Society, 2011.

[25] Terry Nelms and Mustaque Ahamad. Packet scheduling for deep packet inspection on multi-core architectures. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2010.

[26] Tan Li, Fan Yang, Jie Yang, Yinan Dou, and Huanhao Zou. Research of dpi optimization on multi-core platform. In *Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on*, 2010.

[27] Junchang Wang, Haipeng Cheng, Bei Hua, and Xinan Tang. Practice of parallelizing network applications on multi-core architectures. In *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009.

[28] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, January 2013.

[29] Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Quantifying the accuracy of the ground truth associated with Internet traffic traces. *Computer Networks*, 55(5):1158–1167, April 2011.

[30] Andrew W. Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. In *PAM*, 2005.

[31] Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. ACAS: automated construction of application signatures. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*. ACM, 2005.

[32] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5th edition, 2009.

[33] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*. USENIX Association, 2001.

[34] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, 1998. [Online; accessed 14-January-2013].

[35] Gianni Antichi, Domenico Ficara, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. Counting bloom filters for pattern matching and anti-evasion at the wire speed. *IEEE Network*, 23(1):30–35, January 2009.

[36] George Varghese, J. Andrew Fingerhut, and Flavio Bonomi. Detecting evasion attacks at high speeds without reassembly. *SIGCOMM Computer Communication Review*, 36(4):327–338, August 2006.

[37] Gerald Tripp. A finite-state-machine based string matching system for intrusion detection on high-speed networks. In *EICAR 2005 Conference Proceedings*, 2005.

[38] Tran Ngoc Thinh, Tran Trung Hieu, Van Quoc Dung, and S. Kittitornkun. A FPGA-based deep packet inspection engine for Network Intrusion Detection System. In *9th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2012.

[39] Fang Yu, Randy H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols*. IEEE Computer Society, 2004.

[40] Mansoor Alicherry, M. Muthuprasanna, and Vijay Kumar. High speed pattern matching for network IDS/IPS. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*. IEEE Computer Society, 2006.

[41] Snort - Intrusion Detection and Prevention system. `http://www.snort.org/`, 2012. [Online; accessed 23-December-2012].

[42] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*. USENIX Association, 1999.

[43] Chaofan Shen and Leijun Huang. On detection accuracy of L7-filter and OpenDPI. *Third International Conference on Networking and Distributed Computing*, pages 119–123, 2012.

[44] WAND - Network Research Group. The case against L7 Filter. `http://www.wand.net.nz/content/case-against-l7-filter`, December 2012. [Online; accessed 14-January-2013].

[45] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[46] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

[47] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. P3L: A structured high level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[48] Marco Aldinucci and Marco Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, October 2007.

[49] M. Vanneschi. *Course Notes of High Performance Systems and Enabling Platforms - Master Program in Computer Science and Networking*. 2011.

[50] The Muenster Skeleton Library Muesli - A Comprehensive Overview. `http://www.ercis.org/de/node/230`, 2009. [Online; accessed 13-February-2013].

[51] Philipp Ciechanowicz and Herbert Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*. IEEE Computer Society, 2010.

[52] OpenMP - Website. `http://openmp.org/wp/`. [Online; accessed 14-January-2013].

[53] Message Passing Interface (MPI) - Website. `http://www.mcs.anl.gov/research/projects/mpi/`. [Online; accessed 14-January-2013].

[54] S. Ernsting and H. Kuchen. Data parallel skeletons for GPU clusters and multi-GPU systems. In *Proceedings of ParCo 2011*. IOS Press, 2012.

[55] Marco Aldinucci, Massimo Torquati, and Massimiliano Meneghin. Fastflow: Efficient parallel streaming applications on multi-core. *Computing Research Repository*, abs/0909.1187, 2009.

[56] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient smith-waterman on multi-core with fastflow. In Marco Danelutto, Tom Gross, and Julien Bourgeois, editors, *Proceedings of International Euromicro PDP 2010: Parallel Distributed and network-based Processing*, Pisa, Italy, February 2010. IEEE.

[57] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st international conference on Scalable information systems*. ACM, 2006.

[58] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Domain-specific optimization strategy for skeleton programs. In *Proceedings of Euro-Par 2007*. Springer, 2007.

[59] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. *SIGPLAN Notices*, 44(1):177–185, January 2009.

[60] Johan Enmyren and Christoph W. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM, 2010.

[61] CUDA - Website. `http://www.nvidia.com/object/cuda_home_new.html`. [Online; accessed 14-January-2013].

[62] OpenCL - Website. `http://www.khronos.org/opencl/`. [Online; accessed 14-January-2013].

[63] M. Danelutto, L. Deri, and D. De Sensi. Network monitoring on multicores with algorithmic skeletons. In *Proceedings of ParCo 2011*. IOS Press, 2012.

[64] Massimo Torquati. Single-Producer/Single-Consumer queues on shared cache multi-core systems. *Computing Research Repository*, abs/1012.1824, 2010.

[65] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proceedings of 18th International Euro-Par 2012 Parallel Processing*. Springer, 2012.

[66] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[67] John Giacomoni. Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2008.

[68] HPC Advisory Council. Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing. `http://www.mellanox.com/blog/2009/10/interconnect-analysis-infiniband-and-10gige-in-high-performance/`, 2009. [Online; accessed 12-February-2013].

[69] Mellanox Technologies. InfiniBand clustering. Delivering better price/performance than Ethernet. `http://www.mellanox.com/pdf/whitepapers/IB_vs_Ethernet_Clustering_WP_100.pdf`. [Online; accessed 12-February-2013].

[70] Infiniband trade association Website. `http://www.infinibandta.org/`. [Online; accessed 2-January-2013].

[71] Myricom Website. `http://www.myricom.com/`. [Online; accessed 2-January-2013].

[72] Libzero. http://www.ntop.org/products/pf_ring/libzero-for-dna/. [Online; accessed 2-January-2013].

[73] PFRing with Direct NIC Access. http://www.ntop.org/products/pf_ring/dna/. [Online; accessed 2-January-2013].

[74] Augonnet Cédric. Interval-based registration cache for zero-copy protocols. http://cedric-augonnet.com/wp-content/uploads/2012/02/Rapport-StageM1.pdf, 2007. [Online; accessed 12-February-2013].

[75] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*. IEEE Computer Society, 1998.

[76] Myrinet. Myri10GE performances. http://www.myricom.com/scs/performance/Myri10GE/. [Online; accessed 2-January-2013].

[77] Performances of PFRing with Direct NIC Access. http://www.ntop.org/pf_ring/pf_ring-dna-rfc-2544-benchmark/. [Online; accessed 2-January-2013].

[78] W. Simpson. RFC 1853 - IP in IP Tunneling. http://tools.ietf.org/html/rfc1853, 2012. [Online; accessed 23-December-2012].

[79] Information Sciences Institute University of Southern California. RFC 791 - Internet Protocol Specification. http://tools.ietf.org/rfc/rfc791.txt, 2012. [Online; accessed 23-December-2012].

[80] Fowler/Noll/Vo hash function. http://www.isthe.com/chongo/tech/comp/fnv/. [Online; accessed 2-January-2013].

[81] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[82] Murmur3 hash function. http://code.google.com/p/smhasher/. [Online; accessed 2-January-2013].

[83] Joyent Inc. HTTP Parser. https://github.com/joyent/http-parser, 2012. [Online; accessed 11-January-2013].

[84] NLnet Labs - Signature based antivirus. http://www.nlnetlabs.nl/downloads/antivirus/, 2012. [Online; accessed 14-January-2013].

[85] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[86] D. Moore, C. Shannon, and k. claffy. Characteristics of fragmented traffic on Internet links. In *Internet Measurement Workshop (IMW)*, 2001.

[87] Libpcap - Website. `http://www.tcpdump.org/pcap.htm`, 2012. [Online; accessed 14-January-2013].

[88] CAIDA Association. The CAIDA UCSD Anonymized Internet Traces 2011 - 19/05/2011. `http://www.caida.org/data/passive/passive_2011_dataset.xml`. [Online; accessed 23-December-2012].

[89] CAIDA - Website. `http://www.caida.org/home/`, 2012. [Online; accessed 23-December-2012].

[90] Anand Balachandran, Geoffrey M. Voelker, Paramvir Bahl, and P. Venkat Rangan. CRAWDAD trace ucsd/sigcomm2001/tcpdump/08292005 (v. 2002-04-23). `http://crawdad.cs.dartmouth.edu/ucsd/sigcomm2001/tcpdump/08292005`, April 2002. [Online; accessed 14-January-2013].

[91] Massachusetts Institute of Technology: Lincoln Laboratory. DARPA intrusion detection evaluation data set. `www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data/1999/training/week1/index.html`. [Online; accessed 14-January-2013].

[92] Alfred Aho. *Compilers, principles, techniques, and tools.* Addison-Wesley Pub. Co, Reading, Mass, 1986.