Peer-to-Peer Detection

or

Beyond Naive Traffic Classification

Diploma Thesis

François Deppierraz
`francois@ctrlaltdel.ch`

Professor: Hervé Dedieu
Mandator: Jaime Nebrera

Date: December 8, 2006
Status: Final

Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Project Description

Diploma project description

Most current NetFlow probes are usually able to classify traffic using only transport layer informations, more specifically only the TCP/UDP well-know port numbers. With the increasing use of P2P applications, VoIP and other applications using dynamic port numbers, multiples sessions and sometimes encrypted payload, this kind of basic classification is not reliable anymore. Therefore other methods have to be used. For example, pattern matching on the data payload or statistical analysis of the packet flows is necessary.

As the need for higher bandwidth arise, applying pattern matching on data payload becomes quite resource intensive and cannot be handle by a single common CPU anymore. Companies like Sensory Networks, which is a Eneo partner, has developed specific hardware to handle such tasks. An implementation using this hardware in an open source NetFlow probe called pmacct will be done.

If time permits, the statistical approach using only transport-layer informations to classify flows at the collector level could be studied and implemented. The current research in this domain is going quite fast and many papers are available.

Requirements: Linux network, C Programming, NetFlow

Support: Besides IICT teachers, the student will get support from Eneo specialists and partners

Specifications

More reliable IP traffic classification using pattern matching and/or statistical analysis

1. Reading and understanding of the protocols to detect
2. Discussion of the different methodologies available
3. Implementation of one of those methodologies
4. Benchmarks and measures of the chosen method
5. Writing of a research report

Contents

Abstract

Most of the **traffic classification** currently implemented on IP networks is based on the assumption that the layer 4 (TCP or UDP) port numbers present in IP packets headers can define the layer 7 protocol of the communication. Unfortunately this assumption is no longer true because of many protocols using dynamic ports including many Peer-to-Peer protocols, VoIP and so on.

To solve this issue at least two ways exist, (1) trying to apply better algorithms to the transport layer information we already have and (2) analyzing more informations than now, like the whole packets including payload.

This first solution called behavioural analysis is based on ongoing research which tries to model the way a Peer-to-Peer network is working and infers some common characteristics of Peer-to-Peer traffic.

The main focus of this diploma has been on the second solution, called **Deep Packet Inspection**. It works by using protocol signatures, usually in the form of regular expressions, which are applied on the payload of IP packets to determine the protocol used. This feature can be integrated with current network probes.

Because the pattern matching operation on traffic can quickly become very complex and resource intensive, some companies have produced hardware dedicated to it. Most of the time this hardware is based on a FPGA which implements a regular expressions matching engine.

During this diploma, in the office of *Eneo Tecnología* in Seville, a network probe doing traffic classification with hardware accelerated pattern matching has been developed. This development was based on (a) the free software pmacct, a multi protocol network probe developed by Paolo Lucente and (b) the NodalCore C-2000 Serie card as well as the Linux C API from the company *Sensory Networks.*

The result of the development is a working but performance-wise suboptimal implementation which can serve as a proof of concept and has given ideas about future works on the subject.

Acknowledgments

To Lauren for her support and for bearing with my absence during those three months far away.

To my family for their lasting support.

To all the nice guys working at Eneo for their hearty welcome and their help during the project : Angél, Ale, Elio, Jaime, Pablo, José Alberto and Jota.

To Paolo Lucente for his sleep-less nights when replying to my mails and all the work he did on pmacct.

To Hervé Dedieu for his cheerful support.

A mis queridos compis de piso : Antonio, Dawid, Mariana, Marina, Janin and Saidjah. Muchas gracias por compartir su buen humor y la limpieza.

To Saitis Network and especially Nicolas Désir for everything including — more precisely during that project — access to such a wonderful production networking lab.

To Luca Deri for his help and code (nProbe) during the pre-project.

To Matthew Gream of *Sensory Networks* for reminding me not to overflow buffers and for taking the time to write huge interesting mails.

To Grégoire Galland for managing the printing of the report.

To Joao Alves and Léonard Gross for their LaTeX tips and tricks.

To Pascal Gloor for his C programming tips.

Chapter

1

Introduction

The diploma project presented in this document has been done during about three months at the end of 2006 in the office of *Eneo Tecnología*[1], a small network security company located in the beautiful city of Seville in Spain.

Peer-to-peer detection takes place into the framework of general network monitoring and security. The project goal defined by Jaime Nebrera, CEO of Eneo Tecnología, was to design and implement an hardware accelerated network probe based on the free software project **pmacct** (see [Luc]) and using the hardware and development kit provided by the company *Sensory Networks*.

This document is made up of three parts. The first part I on page 3 named **State of the Art** will present some definitions as well as existing tools used in the network monitoring domain in which this project fits. Research is quite active in this domain too, and thus some interesting ongoings researches will be presented.

The second part II on page 30 named **Technologies** will describe in details the different technologies which have been studied and extensively used during this project. Regular expressions as well as hardware acceleration will be presented. The basis of concurrent programming will be explained and development techniques such as debugging and profiling will be presented.

The last part III on page 57 named **Hardware Accelerated Network Probe** will describe the actual development effort which has been done during the implementation of a hardware accelerated network probe doing traffic classification using regular expressions. The implementation has been based on hardware from *Sensory Networks*. The results of benchmarks will also be presented.

[1]`http://www.eneotecnologia.com/`

Part I

State of the Art

Chapter

2

Peer-to-Peer

This chapter will give on overview of the so called *Peer-to-Peer* technologies, their history and some applications using it. We will try to depict the similarities and differences between the *Peer-to-Peer* model and the usual *Client-Server* model.

2.1 Definition

A peer-to-peer (**P2P**) network consists of multiple **nodes** (also called peers) taking part in it and **acting both as clients and servers** at the same time. Whereas P2P looks like new technology, it is basically nothing else than a specific way to use the old client-server model.

Where the client-server model imposes a great differentiation between each node function — the server node provides a service which is accessed by one or many clients — whereas in a P2P network every node will act as both a server and a client in the same time.

2.2 History

The term P2P appeared in the late nineties with the first applications like *Napster* (see section 2.3 on the next page) available to the general public, this triggered a hype on the technology, thus driving the development of new applications.

But the underlying paradigms used by P2P applications have been in use at

Protocol	**Search**	**Data Transfer**	**Authentication**
HTTP	Search engines (**C**)	Single server (**C**)	HTTP Auth (optional)
Napster	One website (**C**)	Other clients (**D**)	One central server (**C**)
Gnutella	From an ultra-peer (**D**)	Other clients (**D**)	None
eDonkey	Multiple servers (one per network) (**C**)	Other clients (**D**)	None
Bittorrent	Search engines (**C**)	Other clients (**D**)	None
Skype	Single server (**C**)	Other clients as relays (**D**)	Single server (**C**)

Table 2.1: P2P protocols characteristics

least since the beginning of Usenet[1] around 1979 under the name of distributed computing.

The rapid development of P2P applications has been driven by the increasing popularity of Internet and the availability of always increasing bandwidths.

2.3 File Sharing Applications

The first applications taking advantage of P2P technologies where mostly file sharing applications using a centralized index of the files and clients available through the network. The following applications focus primarily on giving their users to ability to exchange files between them but without the need the store the file in central location. Instead, the files themselves are directly exchanged between nodes.

The table 2.1 gives some comparisons points between HTTP (see [rfc2616]) and the different P2P protocols described in this section. In P2P protocols usually multiple functions are available such as search, data transfer and sometimes authentication. Each of those functions can work in a centralized or distributed way, the letters **C** and **D** describe that. Only the features widely available in current versions are described, for newer features see section 2.5 on page 7.

[1]A network used the deliver articles in different newsgroups between news servers distributed on the Internet

Napster

Napster has been the first popular file sharing application implementing P2P technologies, though in a very basic way. Each node registered itself on a centralized server and transmitted the list of files he was sharing. Searching through the available files in the network was done on a web interface but the data transfers took place directly between nodes.

Napster's topology was **overly centralized**, this has led to multiple legal problems which are outside the scope of this work. The current trends in file sharing applications is to avoid any centralized element in the network while trying to keep up with the performances usually experienced with centralized networks.

The whole history of Napster is available on [Wik06h].

Gnutella

Gnutella was one the first **fully distributed** P2P protocol made available in the early 2000 by the company *NullSoft* which has just been bought by *AOL*. Soon after release, *AOL* decided to stop the project because they feared legal liability for copyrighted material exchanged on the network. But the software was already in the wild and compatible open-source clients have been developed by reverse-engineering[2] of the protocol.

More information on the protocol is available on [Wik06f] for general information and [gnu] for the protocol development effort.

eDonkey

The *eDonkey* protocol relies on multiple **centralized servers**, one for each community, each connected clients will send a list of files he is sharing. One of the specificity of *eDonkey* was the utilization of file chunks which can be shared even before having the whole file and can be downloaded in parallel from multiple peers.

This protocol is supported in the following well-known clients : *eMule*, *MLDonkey*, *eDonkey2000* and *Morpheus*. More information is available on [Wik06d].

[2]Reverse-Engineering means understanding the protocol used by an application by analyzing it without access to the source code

Bittorrent

The main specialty of the *Bittorrent* protocol is that it supports only file transfer and does not support any file searching capability. The HTTP protocol is commonly used to retrieve a `torrent` file which contains meta data about the file, the address of the tracker[3] and SHA-1 checksums of each file chunk.

One of the main advantage of *Bittorrent* relying on the publication of `torrent` files on websites, email or other means outside of the P2P network itself is that the **authenticity** of the file can be better controlled. This allows companies and open source projects to release their software using *Bittorrent* without caring about fakes files.

More information is available on [Wik06a].

2.4 P2P Telephony

Skype

Skype is a Voice over IP (**VoIP**) application developed by the creators of *Kazza*. It has become widely used mostly because of the following features :

- His aptitude to bypass firewalls and NAT routers
- A great voice quality, achieved by using wideband codecs

But *Skype* is a **proprietary** product using proprietary protocols which brings major drawbacks :

- The security and privacy of the software is unknown and difficult to prove
- No interoperability with standard compliant products (such as SIP or H.323 phones)

Researchers have tried to reverse-engineer the protocol used by *Skype* [BS04] or the application itself [BD06].

Media Streaming

The idea of streaming media such as audio or video over the Internet is not new but the increase in bandwidth available at home has given it a boost during the last years.

[3]The tracker serves as a directory of the nodes currently sharing chunks of the file

Technically speaking the use a *Multicast*[4] infrastructure will fit perfectly the needs of media broadcasting over the Internet but multicast is not, and sadly has not much chance of being, implemented on Internet in it's current form because, besides some technical difficulties which can hopefully be fixed, it can change radically the economical model of content distribution on the Internet.

Economically speaking, if everyone on the Internet would be able to send a multicast stream to the entire world using only his own Internet access at home this would break the business case of many ISP which in addition of providing Internet access to their customer also provide servers hosting for the content providers. This idea is quite interesting in term of potential services provided on the Internet but his realization would require ISP to deploy multicast on their network and open that from the Internet and thus losing hosting customers which is not likely to happen.

The is one of the reason why P2P networks for content streaming have been used as a fall back solution due to the unavailability of multicast.

2.5 Future directions

The implementation of P2P protocols in wide use these days usually require some kind of centralization to handle available informations lookups. Such a centralization can threaten the network in case of legal action directed to the owner of the servers or in case of DoS[5] attacks for example.

Moreover, the protocol as well as data are usually transmitted in clear, expect for *Skype* and thus can be inspected, for example, by the network probe developed in this project or filtered by firewalls. Moreover, privacy issues can arise depending on the data transfered with those means. More and more protocols are beginning to implement data encryption and authentication as well as anonymity using cryptographic methods.

Distributed Network

Most current applications using P2P protocols require some sort of centralized server.

- Each *eDonkey* network require a central server, see the RazorBack2 case [Far06]
- *Bittorrent* files are downloaded and searched on websites, see the "May 2006 police raid" section of [Wik06i]
- *Skype* use a centralized authentication service, see [BS04]

[4]In a multicast network, a host can send packet which will be received by many hosts instead of only one when using unicast

[5]Denial of Service

In a fully distributed network there is no need for a centralized server anymore, all or some of the nodes creating the network will play this role. The drawbacks is that fully distributed P2P network is quite more complicated to build and scale. The *Gnutella* protocol (see section 2.3 on page 5) is an example of such protocol.

Other P2P protocols have been extended to support distributed operations, usually these extensions were developed as a result of some legal action against known servers. One of the example is the *Kademlia* algorithm (see [Wik06g]) which has been used to overcome the central server requirement of *eDonkey* and *Bittorrent*.

Cryptography Usage

One of the current trend in P2P protocols is using cryptography in the protocol for different reasons like avoiding traffic analysis which is used for firewalling or traffic shaping, to increase the privacy of users or even to give users fully anonymity when using the protocol.

The first category of cryptography usage in P2P protocol is simple **obfuscation** of the protocol data. This is mainly done to bypass layer 7 pattern matching detection equipments such as routers or firewall of some ISP which decided to filter or shape P2P traffic. Usually those implementation are quite weak in term of security because their goal is only obfuscation. This type of encryption has been implemented in the *Bittorrent* protocol (see [Wik06b]).

Another type of cryptography usage is done by P2P application such as *Waste* (see [was]) which use **asymmetrical cryptography** to secure all the network links between clients as well as authenticating clients entering the network. This application is mostly designed for small groups of users which need services like instant messaging and file transfer inside the group.

The last category presented is **overlay networks** taking advantage of P2P-like protocols and cryptography to create completely anonymous networks running on top of Internet called **overlay networks**. The goal of those projects is to protect freedom of speech by allowing people to access and publish information without any traceability. Those networks are using multiple layers of encryption and are routing both the requests and responses through multiple hops in the network. Only the last hop is able to decrypt the message and all the intermediaries only know about the node before him and the following.

Multiple projects using that kind of overlay networks exist, widely known examples are *Tor* (see [tor]) and *Freenet* (see [fre]). In the case of *Tor* which allows Internet access from within the overlay network, it is even possible to use usual P2P applications with it. The performances are usually much lower than with a direct Internet access because of the multi-hop routing.

Chapter

3

Network monitoring

This chapter will show different protocols and systems in use today to monitor networks of different sizes, ranging from the one of a small company to an international ISP network spanning multiple continents.

3.1 Introduction

Network monitoring plays an important role in today network management. IP Networks are now being used to access the Web, communicate using emails, transfer scientific data, carry voice calls, video on demand and so on. This convergence on a single network, which is best-effort by design, requires careful monitoring to minimize outages and improve the quality of service experienced by the users.

3.2 SNMP

The *Simple Network Management Protocol* has been standardized by the IETF as a mean of getting status informations about network equipments as well as modifying the configuration of those. Due to many reliability and security issues in the initial design of the protocol, it is mostly used theses days in read-only mode as a mean to query network equipments for informations about their status such as interfaces counters (`ifInOctets` and `ifOutOctects`) or system informations such as uptime (`sysUpTimeInstance`) or Operating System version (`sysDescr`). More information on the protocol itself can be found in [Wik06k] and [rfc1157].

Many available tools gather statistics using SNMP to give network administrators a view on how their network is currently running. The best known tools working this way are Mrtg (see section 6.1 on page 21), Cacti (see section 6.1 on page 22) or proprietary tools such as HP OpenView. Usually informations such as interface counters are polled from a central management station running one of the aforementioned tool each 5 minutes and displays graphs showing the derivate of those values while handling the counter overflows[1].

One of the drawbacks of monitoring a network by relying only on SNMP is the coarse granularity of the gathered informations, all you can get is the total traffic on each interface of your network equipments. There is no way to differentiate between different protocols, different IP addresses and so on.

3.3 NetFlow

NetFlow is a protocol created by *Cisco Systems* which allows network equipments such as routers and switches to export statistics about the traffic flowing through them. Different versions of the protocol exist, but we will focus primarily on NetFlow v5 which is commonly used on network equipment at the moment and IPFIX (see [rfc3917]) which is the IETF standard based on NetFlow v9 (see [rfc3954]) which is used to transport more detailed informations than NetFlow v5.

Flow Definition

The usual definition of a flow is an unidirectional communication made of IP packets sharing the following 5-tuple of attributes :

- Source IP address
- Destination IP address
- IP protocol (TCP, UDP, ICMP, etc.)
- Source port
- Destination port

Usage

NetFlow provides a way to get informations about every flow routed through a network in a simple and efficient manner. Many Internet Service Providers (ISP) use it to gather statistics about the traffic routed through their network, to bill their customers, to detect security attacks and so on.

[1]Those counters are commonly 32 bits long and thus cycles after 2^{32} bits which is 4 GB. Newer network equipments have to use 64 bits counters to avoid cycling twice during the same polling interval.

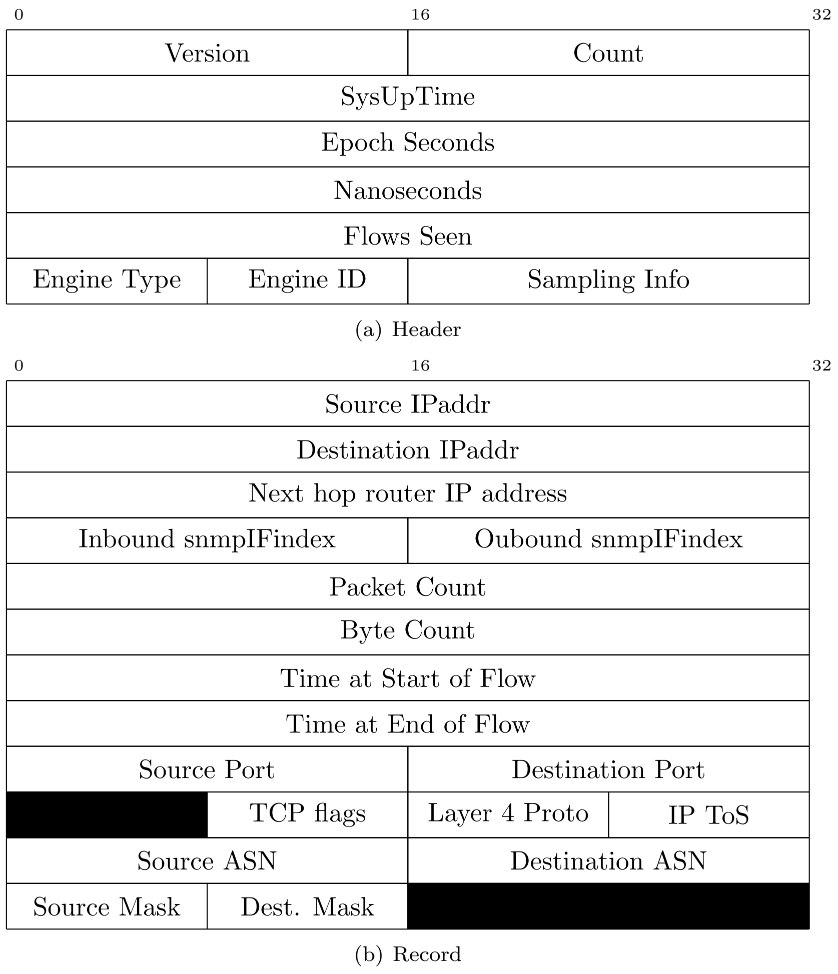


Figure 3.1: NetFlow v5 Packet Format

In comparison to dedicated traffic analyzing equipment such as IDS or specialized packet sniffers, NetFlow has the main advantage of running on the same hardware as the one forming the network without requiring adding dedicated equipment at each interconnection point.

NetFlow v5 packets

NetFlow v5 packets contain a basic set of informations about flows which have been created according to the definition presented before. The NetFlow v5 protocol is not extensible but has the main advantage of being simple and easy to implement.

NetFlow v9 packets

The NetFlow v9 protocol has been designed to be fully extensible, it allows generators and collectors to exchange all the informations they requires by supporting templates. Of course, both the generator and the collector must understand the templates to be useful but if a specific template is unknown, the specific information using this template can be discarded while keeping the rest.

3.4 sFlow

sFlow is a multi-vendor alternative to NetFlow which has the following advantages :

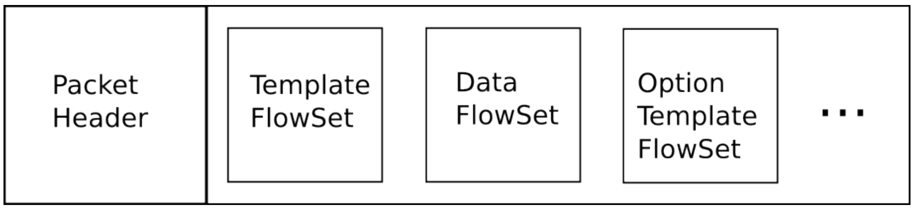
- Supports other protocols than IP such as Ethernet directly, IPX or Appletalk
- More extensive support for including BGP4 informations such as Communities and AS-Path (see [vB02])
- Built-in support of sampling to be able to cope with high-speed networks

We are not going in much details because this protocol won't be used in this project. More informations are available in [sfl].

3.5 Intrusion Detection Systems

Intrusion Detection Systems (IDS) usually uses traffic analysis methods to warn system administrators of security attacks on their systems. Most current IDS are using a set of rules applied to the traffic and are raising alerts in case of match. Those rules are often created based on security advisories of known security holes. One of the most well-known open-source IDS software is Snort (see [sno]).

One the method often used pattern matching on traffic payload is regular expressions matching (see chapter 8 on page 30) or simple strings matching. That is why those system have the same performance problems as traffic classification and can also take advantage of hardware acceleration (see chapter 9 on page 35).



(a) Packet structure

0	16 32
Version	Count
SysUpTime	
UNIX Secs	
Sequence Number	
Source ID	

(b) Header

0	16 32
FlowSet ID = 0	Length
Template ID	Field Count
Field Type 1	Field Length 1
Field Type 2	Field Length 2

(c) Template FlowSet

0	16 32
FlowSet ID = Template ID	Length
Record 1 - Field Value 1	Record 1 - Field Value 2
Record 1 - Field Value 3	...
Record 2 - Field Value 1	Record 2 - Field Value 2
Record 2 - Field Value 3	...
Record 3 - Field Value 1	...
...	Padding

(d) Data FlowSet

0	16 32
FlowSet ID = 1	Length
Template ID	Option Scope Length
Option Length	Scope 1 Field Type
Scope 1 Field Length	...
Scope N Field Length	Option 1 Field Type
Option 1 Field Length	...
Option M Field Length	Padding

(e) Options Template FlowSet

Figure 3.2: NetFlow v9 Packet Format

Chapter

4

Traffic Analysis

4.1 Introduction

Traffic analysis requires access to the traffic flowing through a network, that is why it has usually to been done on network equipment such as routers or switches or even directly on the links using passive probes.

Various traffic analysis methods exist, some give access to the full packet payload or others only give statistical informations about packets. Traffic analysis can be embedded in network equipment (NetFlow for example see 3.3 on page 10) or on dedicated equipments such as packet sniffers, `tcpdump` running a machine connected to the mirroring port of a switch.

4.2 Traffic classification

Well known ports

One of the easiest traffic classification method is transport layer analysis based on port numbers. Originally, in a plain client-server IP architecture, each server uses a specific TCP or UDP port number which has been allocated for the service by IANA[1]. These well known ports are defined in the range 1 to 1023.

Let S_1 be a TCP session between two hosts.

[1]Internet Assigned Numbers Authority

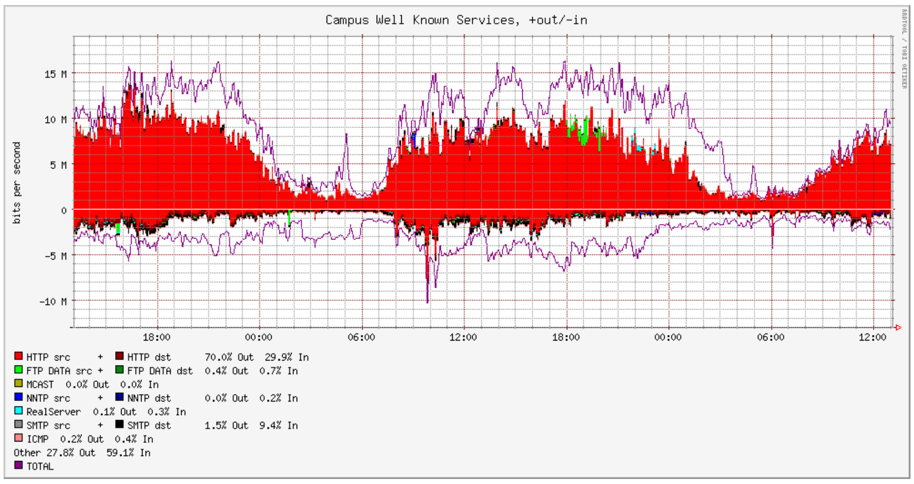


Figure 4.1: Traffic classification using well known ports

$$S_1 = (ip_1, port_1) \Leftrightarrow (ip_2, port_2)$$

The session is defined by two parameters for each host, the IP address ip_x and the port number $port_x$. Now to able to determine the protocol which is running and the function of each host (client or server) it is sufficient to find which of the two port numbers is less than 1024, this will define the server and a simple lookup with this port in the IANA registered port list or `/etc/protocols` under Unix will give us the protocol used.

On figure 4.1 is an example of a well known ports based traffic classification done on a small ISP network[2]. The traffic categorized as **Other** is the one using non-standards ports, in that example it is 27.8% in output and 59.1% in input. This is only example a simple example but shows well why better techniques are required.

Deep Packet Inspection

Deep Packet Inspection (DPI) consists in using all the informations available on packet from layer 2 to layer 7 to do traffic classification. These methods use protocol signature which are usually either fixed strings or regular expressions (see chapter 8 on page 30). Both are matched on the packet payload and can sometimes be associated with other layers specifications such as layer 3 protocol, port numbers or even specific IP addresses (known centralized servers for example).

[2]Saitis Network, AS6893

This method was used during the development part of the project which is described in part III on page 57.

4.3 Packet Capture

This section will give an overview of different packet capture mechanisms available under Linux. The job of a packet capture is to get packet from a network interface, possibly using promiscuous mode[3] and transmit it to an application which will care of the traffic analysis itself.

Libpcap

Libpcap is a widely used packet capture library available under Unix which has been developed for the `tcpdump` packet sniffing utility but which is now used by many other projects.

Under Linux, Libpcap uses the *Linux Packet Filter* (see [Ins01]) feature of the kernel because, when using sockets, only the packet payload is transmitted to user space applications by the kernel which has already handled the layers 3 and 4 protocols decoding. This feature allows applications to receive the whole packet include layer 2 headers.

Libpcap-mmap

A patch is available on [libb] to add support in libpcap for the shared ring buffers available in the Linux kernel (using the kernel option `CONFIG_PACKET_MMAP` since kernel 2.4). It is more efficient because it avoids data copy between user and kernel space, the user space processes can thus directly access the ring buffer using the `mmap` system call.

Streamline

Streamline is a networking subsystem working in the Linux kernel. It works at high-speed because all the filtering and data manipulation is done directly in the kernel instead of being pulled up to user space before manipulation. In practice, that means that a user space application gives a command like `filter tcp traffic to port 80 -> reassemble tcp session -> apply regular expressions -> send the result to user space`. This feature allows applications requiring very specific operations on traffic to perform quite faster.

[3]A special network interface mode which will give access to all packets coming to the interface without, in the case of Ethernet interfaces, filtering on the destination MAC address.

Hardware acceleration is also usable by Streamline which can take advantage of network processor such as Intel IXP1200 (see section 9.1 on page 37). The architecture of Streamline allows applications to create specific code destined to the kernel or the network processor which will apply specific functions to the traffic.

nCap

nCap is a packet capture and transmission system developed by Luca Deri (the author of nProbe too, see chapter 7 on page 24) which is compatible with libpcap and was designed to work with commodity hardware. In capture mode, the idea is to shorten at maximum the path between the network interface and the applications handling the captured traffic. To do that nCap is implemented directly in the Ethernet interface driver[4] and will feed a shared buffer with the monitoring applications, thus bypassing completely the kernel itself.

[4]At the moment only Intel 1Gbps and 10Gbps interfaces are supported

Chapter

5 Behaviour Identification

This chapter will present an overview on current traffic identification methods relying only on **transport layer informations** — which are already available on monitoring systems based on NetFlow v.5 (see section 3.3 on page 10) for example — to do traffic classification. Most methods are using statistical analysis of the flow informations to be able to determine at which class of traffic a flow belongs.

No development has been done on this type of traffic identification during this project but those techniques are nonetheless quite interesting and usually easier and cheaper to deploy because they do not require access to the packet payload and can leverage an existing NetFlow infrastructure.

5.1 Ongoing Research

This section will give an overview of some research papers which have been published on the subject of traffic classification using transport layer informations.

Transport layer identification of P2P traffic

The paper entitled "Transport layer identification of P2P traffic" (see [KBFc04]) by Karagiannis et al. presents a method which focus is P2P detection and is based on two main behaviours :

1. Utilization of both TCP and UDP within a defined time frame between the same pair of source and destination IP addresses
2. Connections to as many different ports than different IP addresses

Some mechanisms based on known port numbers of non-P2P traffic were added to decrease the number of false positives.

BLINC : Multilevel Traffic Classification in the Dark

A second paper of Karagiannis et al. (see [KPF05]) presents a more general traffic identification method which does not specifically focuses on P2P traffic and, this time, without using any a priori information such as well known port numbers.

This classification is based on an approach taking into account three different levels. The **social** level made of the interactions between hosts, the **network** level analyzes the role (provider, consumer of both) of the hosts and the **application** level analyzes the interactions between hosts on specific ports trying to identify applications.

Identifying Known and Unknown P2P Traffic

Another paper, entitled "Identifying Known and Unknown P2P Traffic" (see [CM]) which followed the two previous papers is only using the two following intrinsic characteristics of P2P networks :

1. A large network diameter
2. Many nodes acting both as clients and servers

5.2 Advantages

In situations where techniques based on packet payload analysis are not available due to legal, privacy or technical concerns, behaviour identification can be of great interest.

Data processing can be delayed because the flow informations can easily be stored in databases and the classification algorithm can be run at any time. With payload identification this is usually not possible because there is too much data to store and therefore the payload analysis has to be done in real-time.

Behaviour identification can also work with informations coming from multiple monitoring points on a network whereas payload analysis cannot use such information aggregation.

New P2P protocols can be detected using such a method where payload identification requires prior reverse-engineering of the protocol, what is usually not trivial.

5.3 Drawbacks

Implementing an infrastructure able to handle payload analysis can provide some groundwork for implementing more services taking advantage of the classification such as service dependent traffic shaping, anti-virus or anti-worms firewalling and so on.

Usually behaviour identification results are much less specifics than with payload identification. For example, P2P detection based on behaviour identification will only tell if a specific flow was generated by some P2P applications, in which situations, payload identification can tell the name of the P2P application which generated the flow.

5.4 Future Work

One interesting future project would be to integrate a NetFlow collector and viewer such as NfDump/NfSen (see sections 6.2 on page 22 and 6.2 on page 23) using the algorithms of the research papers presented in section 5.1 on page 18. This could allow network managers to have clearer view of the traffic flowing through their network than with port-based analysis.

Chapter

6

Existing Tools

This chapter will describe a bunch of tools, all available under free software licenses, which are using the different protocols and techniques presented in the chapter 3 on page 9.

6.1 SNMP

Mrtg

The Multi Router Traffic Grapher (MRTG) is a software developed by Tobias Oetiker while working at the ETHZ[1]. It can generate graphics like the one on figure 6.1 from SNMP interface counters.

The project homepage is available on [Oeta].

[1]Swiss Federal Institute of Technology Zürich

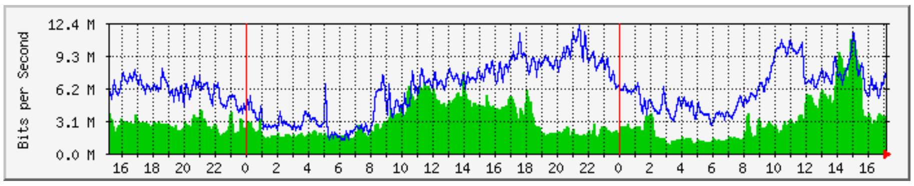
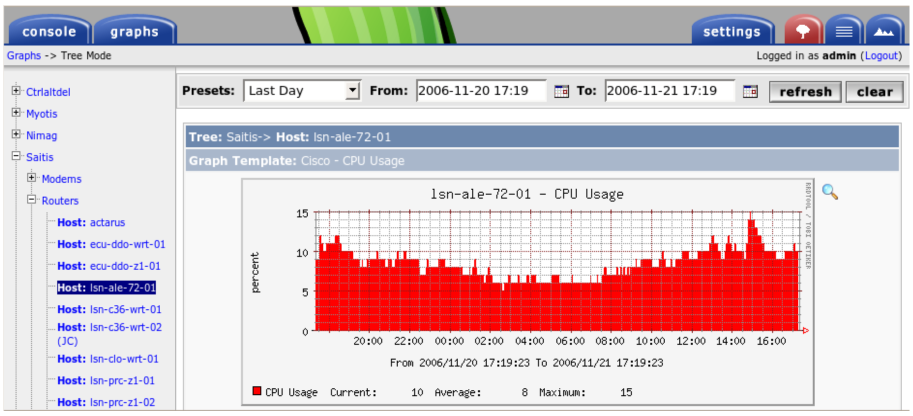


Figure 6.1: Example of MRTG graph generated from data retrieved with SNMP



Presets: Last Day **From:** 2006-11-20 17:19 **To:** 2006-11-21 17:19

Tree: Saitis-> **Host:** lsn-ale-72-01

Graph Template: Cisco - CPU Usage

```
From 2006/11/20 17:19:23 To 2006/11/21 17:19:23

🟥 CPU Usage  Current:     10  Average:      8  Maximum:     15
```

Figure 6.2: Screenshot of Cacti web interface

Cacti

The Cacti project is network graphing solution composed of a poller which will record data fetched with SNMP in RRD files (see section 6.3 on the following page) and a web interface allowing his configuration and the visualization of the generated graphs.

The figure 6.2 shows an example of the web interface. The project homepage is available on [cac].

6.2 NetFlow

nProbe

nProbe is a network probe using libpcap and exporting flows informations using the NetFlow protocol. This software will be dissected in great details in chapter 7 on page 24.

NfDump

NfDump is a NetFlow collector developed by Peter Haag who is working in the security team of Switch[2]. It was designed to receive NetFlow packets sent by network equipments, store them in optimized binary files and give access to this information in an efficient manner.

[2]The Swiss Education & Research Network

The key features of NfDump are :

- Supports all current version of the NetFlow protocol : v5, v7 and v9
- Supports the application of `tcpdump`-like filters on stored flows
- Can generate "Top N" statistics for IP addresses, ports, AS numbers and router interfaces

The project homepage is available on [Haaa].

NfSen

NfSen is a web frontend for NfDump also developed by Peter Haag. It allows easy access to the data stored by NfDump, network managers can use it to have a global view of their network and can create profiles when dealing with a specific problems (e.g. one computer has been cracked, what was his traffic before the incident ?).

The followings are the key features of this software :

- Graphs generation using the RRD library (see section 6.3)
- Profile support allows to create specific profiles based on a filter which can create graphs. This can even be done using already stored data.
- Supports plugins which need access to the stored NetFlow data

The project homepage is available on [Haab].

6.3 The Round Robin Database

The Round Robin Database (RRD) is a logging and graphing library associated with utilities (RRDTools) developed by Tobias Oetiker. It was developed to generalize the file format used in Mrtg (see section 6.1 on page 21).

The file format used by RRD allows to keep samples in fixed sized database which handles the **time aggregation** of the data itself. For example, you can create a RRD file which contains 228 5-minutes records (one day), 336 30-minutes records (one week) and 365 1-day records. With this file all you need is to update the data each 5 minutes and the weekly and yearly records will be automatically calculated.

This concept is quite interesting for network related data because the precision required for recent events is usually greater than the one of a yearly average and with RRD all this data housekeeping is handled automatically.

The project homepage is available on [Oetb].

Chapter

7

nProbe

This chapter will present *nProbe* v4, an extensible NetFlow probe released under the GPL (see appendix G on page 124) by Luca Deri.

7.1 Features

The version 4 of nProbe which was analyzed possess the following features :

- Flows export using the following protocols : NetFlow v5, NetFlow v9 and IPFIX (see section 3.3 on page 10
- Gigabit speeds support
- Multi-threaded design
- Plugin architecture for classification purposes

7.2 Design

This section describes the design analysis of nProbe which has been done by reading the source code. The goal was to be able to understand correctly the way taken by an incoming packet in the probe and how it was handled by the multiple threads.

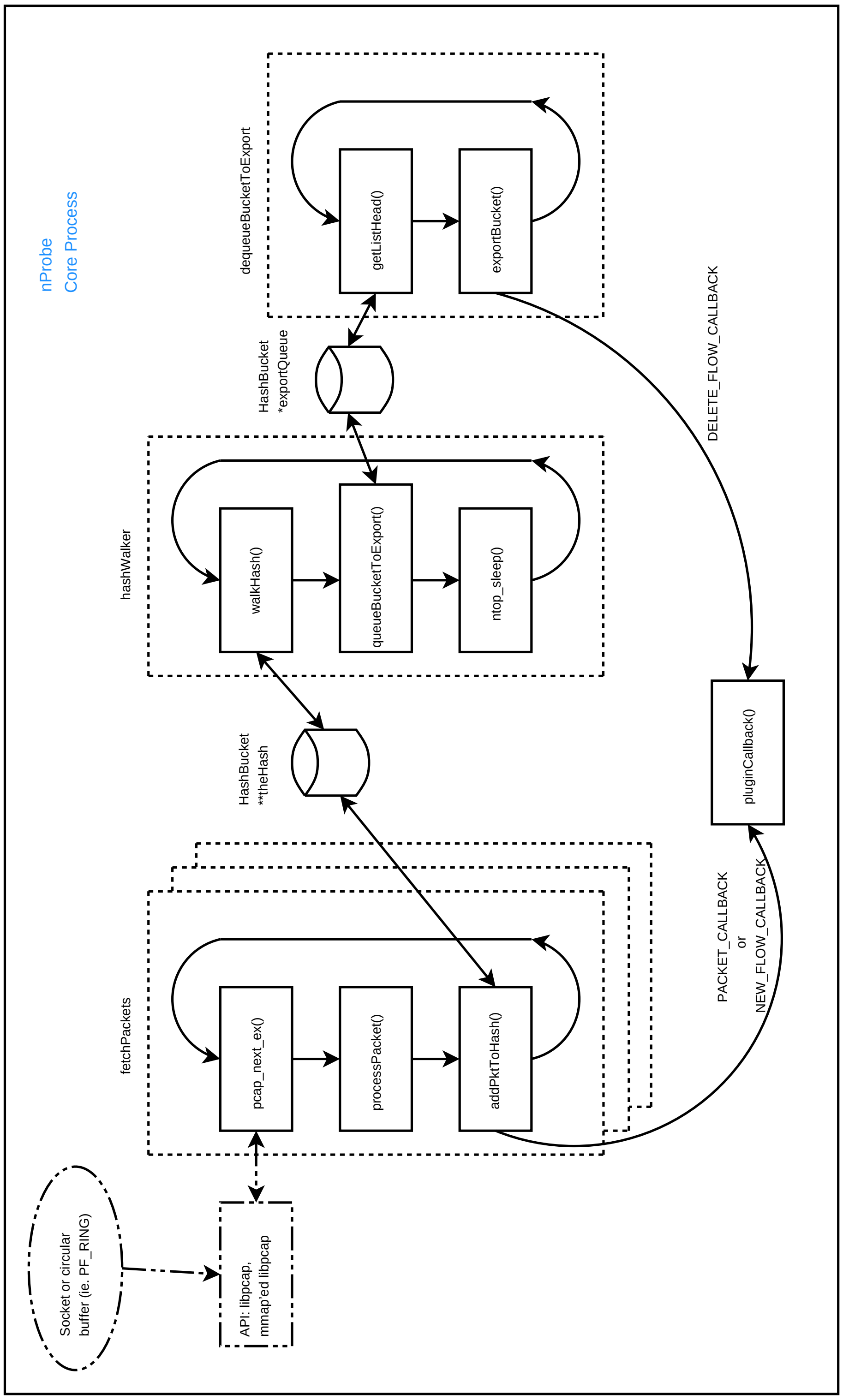


Figure 7.1: nProbe multi-threaded design

Event	Function name
Plugin initialization	`xxxPlugin_init`
New packet received	`xxxPlugin_packet`
A flow has been emitted	`xxxPlugin_delete`
A flow need to be exported	`xxxPlugin_export`
Return if the given NetFlow v.9 template is supported	`xxxPlugin_get_template`
Return the NetFlow v.9 templates supported	`xxxPlugin_conf`

Table 7.1: nProbe plugin API, with `xxx` substitued by the plugin name

The core process of nProbe is using three different types of threads, presented on figure 7.1 on the previous page and named according to their main function. At the moment, only the packet capture thread (`fetchPackets`) can be used as a thread pool (see section 10.9 on page 46) with multiple threads.

- `fetchPackets()`
  This thread group captures network packets using the libpcap function `pcap_next_ex()`, processes them by analyzing headers, buffers packet fragments, etc. and finally adds it to the shared flow hash table `theHash`.
- `hashWalker()`
  This thread parses the `HashBucket` to detect expired flows and adds them to the shared queue `exportQueue`.
- `dequeueBucketToExport()`
  This thread processes the `exportQueue` and converts the flows to NetFlow data.

7.3 Implementation : Layer-7 plugin for nProbe

The implementation of a layer-7 plugin in nProbe project (see [Der]) has been done during the pre-project phase of this work. It is presented here because the layer-7 filter patterns description can also be of interest when using `pmacctd` which supports for those patterns too.

nProbe plugin API

The plugin API in nProbe is described in the file `README.plugins` included in nProbe 4.0 tarball. Basically nProbe has several hooks which calls plugins methods when the events described in table 7.1 occurs.

Layer-7 patterns

The l7-filter project [l7fa] has developed more than 100 patterns for matching standards protocols (like HTTP, SMTP, POP3, etc.), reverse-engineered protocols (P2P protocols for example) and even traffic generated by worms such as *CodeRed* or *Nimda*.

File Format

The file format of L7-filter patterns is fully described in the "L7-filter Pattern Writing HOWTO", see [l7fb]. Only the important informations will be presented here.

An example pattern to match HTTP traffic is showed in figure 7.2 on the following page. Every line beginning with an hash sign is a comment. On our example the name of the pattern is defined on line 19 and the regular expression on line 23.

The first part is metadata about the pattern, on line 1 is the protocol name, on line 2 some tags about the pattern quality (such as undermatch, great, overmatch) and speed (veryfast, notsofast, slow), on line 3 the groups in which the protocol belongs can be listed. And finally line 4 a link to a wiki page about the protocol.

```
# HTTP - HyperText Transfer Protocol - RFC 2616
# Pattern attributes: great notsofast superset
# Protocol groups: document_retrieval ietf_draft_standard
# Wiki: http://protocolinfo.org/wiki/HTTP
#
# Usually runs on port 80
#
# This pattern has been tested and is believed to work well.
#
# this intentionally catches the response from the server rather than
# the request so that other protocols which use http (like kazaa) can be
# caught based on specific http requests regardless of the ordering of
# filters... also matches posts

# Sites that serve really long cookies may break this by pushing the
# server response too far away from the beginning of the connection. To
# fix this, increase the kernel's data buffer length.

http
# Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF (rfc 2616)
# As specified in rfc 2616 a status code is preceeded and followed by a
# space.
http/(0\.9|1\.0|1\.1) [1-5][0-9][0-9] [\x09-\x0d -~]*(connection:|content-type:|content-length:|date:)|post [\x09-\x0d -~]* http/[01]\.[019]
# A slightly faster version that might be good enough:
#http/(0\.9|1\.0|1\.1) [1-5][0-9][0-9]|post [\x09-\x0d -~]* http/[01]\.[019]
# old pattern(s):
#(http[\x09-\x0d -~]*(200 ok|302 |304 )[\x09-\x0d -~]*(connection:|content-type:|content-length:))|^(post [\x09-\x0d -~]* http/)
```

Figure 7.2: Layer-7 filter project pattern file format

Part II

Technologies

Chapter
8

Regular Expressions

8.1 Introduction

A *Regular expression*, also called *pattern*, is a string which describes a set of strings. It does so in a much more compact way than listing all the possibilities thanks to its great expressive power. The mathematic basics of regular expressions have been described by the mathematician Stephen Kleene in the 1950s under the name *regular sets*. Since around 1966, they appeared in the Unix world in quite a few utilities, from text editors to programming languages and finally were in other environments as well. More details are available in [Wik06j] and [Fri06].

In the context of network security and monitoring, regular expressions have found many purposes like traffic analysis (see chapter 4 on page 14) and intrusion detection (see section 3.5 on page 12).

8.2 Syntax

A regular expression is composed of two types of characters, **literals** and **metacharacters**. Literals are matched directly while metacharacters are specials characters which are interpreted in the regular expression. Those metacharacters are what makes regular expressions so powerful[1].

[1]As well as, sometimes, difficult to understand

	Meaning	Character
Line	Start of line	`^`
	End of line	`$`
Class	Character class	`[]`
	Negation	`!`
	Any character	`.`
	Alternation	`|`
Quantifiers	Optional item	`?`
	Repetition (1 to n)	`+`
	Repetition (0 to n)	`*`
	Intervals	`{min, max}`
	Backreferences	\x

Table 8.1: Metacharacters used in regular expressions

8.3 Examples

For the sake of showing the way a regular expression is used to match a known protocol, we will use expressions taken from the l7-filter project. Note that those patterns are using two features specific to the l7-filter project. First, they are case insensitive and secondly, hexadecimal characters can be written as `\x` where `x` is their hexadecimal value.

Matching POP3 traffic

Let's begin with a simple pattern which matches the POP3 protocol used to retrieve mails from a server. The regular expression is the following :

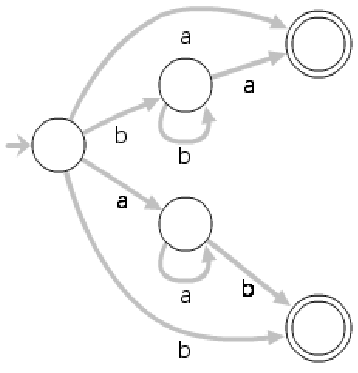
`^(\+ok␣|-err␣)`

It is made of 2 parts separated by the alternation character `|` and will match when any of the parts match at the beginning of a line. In fact this expression is equivalent to matching the following expressions :

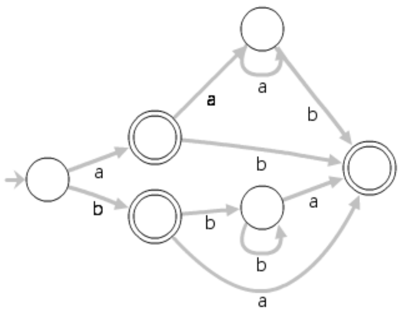
- `^\+ok␣`
  will match the string "`+ok␣`" at the beginning of a line
- `^\-err␣`
  will match the string "`-err␣`" at the beginning of a line

Matching Bittorrent traffic

The following regular expression is used to match Bittorrent traffic (see section 2.3 on page 5).



(a) NFA



(b) DFA

Figure 8.1: Example of regular expression compilation for `a*b|b*a`

```
\x13bittorrent␣protocol|d1:ad2:id20:|\x08'7P\)[RP]|^azver\x01$|
^get␣/scrape?info_hash=
```

This pattern makes use of the alternation character `|` and is thus composed of 5 different parts which each can generate a match of the entire expression :

- `\x13bittorrent␣protocol`
  will match the character whose value equals to 0x13 followed by the string `"bittorrent␣protocol"`
- `d1:ad2:id20:`
  will match the string `"d1:ad2:id20:"`
- `\x08'7P\)[RP]`
  will match the character whose value equals 0x8 followed by the string `"'7P)"` and followed by the character `'R'` **or** the character `'P'`
- `^azver\x01$`
  will match the string `"azver"` followed by the character whose value is 0x1. The whole being alone on a single line
- `"^get␣/scrape?info_hash="`
  will match the string `"get␣/scrape?info_hash="` at the beginning of a line

8.4 Regular Expression Matching

The regular expression compilation is done by converting the regular expression into a nondeterministic finite automaton (**NFA**) or a deterministic finite automaton (**DFA**) depending on the implementation used. The matching is

	Regular expression of length n	
	Processing complexity	Storage cost
NFA	$O(n^2)$	$O(n)$
DFA	$O(1)$	$O(\Sigma^n)$

Table 8.2: Worst case comparison of DFA and NFA for a single regular expression

done using the compiled expression , the input data is fed into the automaton which will generate matches when the input data reaches an accepting state (double circles on the figures). The examples on figures 8.1(a) on the preceding page and 8.1(b) on the previous page should makes that clearer.

Efficiency Considerations

The number of states generated by a regular expression depends on the number and type of metacharacters used, the type of matching automaton used (NFA or DFA) and the total size of the expression.

The table 8.2 taken from [YCD+] shows the worst processing complexity and storage cost depending on the type of automaton used, Σ represents the set of input symbols, which in network monitoring applications is the extended ASCII set which is composed of 256 characters.

The constant $O(1)$ processing time taken by DFA pattern matching engines makes them quite attractive for hardware implementation. This is the type used in NodalCore Pattern Matching Engine (see section 8.5 on the next page). But care has to be taken because the memory available on hardware pattern matching engine is limited and thus the exponentials $O(\Sigma^n)$ storage cost can quickly get us in troubles.

8.5 Implementations

This section presents two implementations of regular expressions matching engines used during this project. The first one, Henry Spencer C implementation, is used in the nProbe layer 7 classification (see section 7.3 on page 26) as well as in pmacct layer 7 classification (see section 13.3 on page 59). The second one is a hardware implementation presents in NodalCore hardware.

Henry Spencer C Implementation

This implementation is based on a NFA and was primarily chosen by the layer7-filter project (see [l7fa]) because it was easier to port in kernel space than the GNU Regular Expression library (see [RSH]). The same implementation has been used in all the applications using patterns coming from the layer7-filter project to avoid having to convert the patterns because of differences between implementations.

NRPL — NodalCore Regular Pattern Language

The implementation of the Pattern Matching Engine (PME) in NodalCore hardware is based on a DFA and is described in [Senb]. This implementation contains a number of important characteristics which needs to be taken into account when writing patterns :

- All patterns have to be grouped in a single one using action tags
- The start of line operator `^` means the beginning of a stream and not a new line, to match a word at the beginning of a line this syntax should be used : `^WORD|\nWORD`
- A new operator `@` is added to the syntax which is similar to the dot operator `.` but has limitations making it more efficient than the dot

Action Tags

Because the NodalCore hardware usually works with a single pattern, it uses action tags to raise the correct events depending on the part of the pattern which matched. These action tags are included in the pattern using the `!` metacharacters following by the event ID.

In the following example the first part of the pattern will match the string `HTTP` and raise event number 1 and the second part will match the string `Bittorrent` and raise event number 2.

```
HTTP!1|Bittorrent!2
```

Patterns Optimization

The optimization of layer7-filter patterns to work efficiently with the NodalCore PME is another project which is done by another student, Elio, for *Eneo Tecnología.*

Chapter

9

Hardware Acceleration

This chapter will explain why hardware acceleration has become necessary with today's network equipments, the way it is usually used, the different technologies available and the some products as well as research projects.

Hardware acceleration has been seen as necessary primarily due to two factors :

1. The increasing bandwidth of network links.
2. The willingness to do much more operations, like packet filtering, QoS[1], Intrusion Detection (see section 3.5 on page 12) on IP packets in addition to basic IP routing.

Doing a naive comparison between the increase in common network interfaces speed versus processors speed we can see that network interfaces speed have been increasing faster than processor speeds. Taking for example which hardware was available in the beginning of the nineties : Intel 486 processors clocked at 25 MHz and Ethernet 10-BaseT at 10 Mbps. When compared to actual standards in 2006 : Intel Pentium 4 at 3 GHz and Ethernet 10Gbe at 10 Gbps. We have a ratio of 120 between processor speeds[2] and of 1000 between network interface speeds.

[1]Quality of Service

[2]Yes, Pentium IV have more instructions than 486, that's why the comparison was called naive.

9.1 Technologies

FPGA

Field Programmable Gate Array (FPGA) is a type of semiconductor device containing logic gates (AND, OR, XOR or NOT) associated with other components such as memory, clock and even standard processors in some models. FPGAs have been invented around 1984 by one of the founder of *Xilinx*, Ross Freeman.

The connexions between logic gates as well as other components can be easily reprogrammed. That is the main difference of FPGA compared to ASIC[3] which, once built, cannot be reprogrammed. FPGA are usually slower than their ASIC counterparts but cheaper for small series.

Programming a FPGA is normally done with a high-level language such as VHDL[4] (see [Wik06n]) or Verilog[5] (see [Wik06m]), which is then compiled to generate a *netlist* describing the interconnection of the components. Using a place-and-route tools (usually proprietary and hardware specific) this netlist can be converted to a binary configuration file which can be uploaded to the FPGA.

According to [Mac06] the higher clock speed of actual FPGA chips is between 200 MHz and 400 MHz which is between ten or fifteen times slower than actual processors speeds. But the main advantage of FPGA is **parallelization**, a single FPGA chip can be configured as **multiple application-specific processors** working in parallel.

More informations about FPGA are available in [Wik06e].

Applications

Network acceleration cards such as COMBO6 from the Liberouter project and the NodalCore C-2000 by *Sensory Networks* both are built around a FPGA.

Another interesting application of FPGAs is a massively parallel cryptography cracking technique using a cluster of FPGAs presented in [HM06] to be able to crack WPA[6] used on wireless access points quite faster than previous software based methods.

[3]Application Specific Integrated Circuit

[4]VHSIC Hardware Description Language, developed by the US Department of Defense inspired by the ADA programming language

[5]A hardware description language inspired by the C programming language syntax

[6]Wi-Fi Protected Access, a wireless encryption standard

Network Processors

Network processors are special purpose processors which are optimized to handle common operations on packets such as routing table lookup and routing, checksumming, payload inspection and so on. The advantage is that multiple network processors can be used in parallel, for example one for each network interface or group of interfaces depending on the power of the network processor. That way, the main processor of the router should mainly be used to configure the network processors and handle high-level management of the router.

There are many different products available under the name *network processor*. Sometimes multiple specific purpose processors are grouped on the same PCI board which can be used with a commodity PC, network processors directly connected with network interfaces are available too. Networks processors can be used to replace ASIC usually used in high-end routers, they can then provide much better extensibility by reprogramming.

One widely known model of network processor is the *Intel IXP1200* which is based around a StrongARM host processor with six independent processors called *microengines* which are optimized for network operation.

9.2 Products

Liberouter

Liberouter (see [liba]) is a project of *Cesnet* (see [ces]), an association which runs the academic network of the universities of the Czech Republic. The goal of the project was to develop a multi-gigabit IPv4 and IPv6 router based a common PC using hardware acceleration. This development lead to the release of the *COMBO6* card based on a *Xilinx* FPGA (see 9.1 on the previous page).

One of the very interesting point of the project is that both the software and the hardware design are released under an open-source license. After the development of the *COMBO6* card, many different research projects have been launched on topics such as high-speed NetFlow probe (see 3.3 on page 10), IDS (see 3.5 on page 12), Packet Capture and so on.

Endace DAG Serie

The company *Endace* (see [end]) is selling hardware acceleration cards dedicated to packet capture and monitoring for multiple interface types ranging from E1[7] to SDH[8], to 10 Gbps Ethernet. All those cards are containing the

[7]European telephony standard interface at 2 Mbps
[8]Synchronous Digital Hierarchy, optical interfaces define by an ITU standard

network interface itself and are working in a fully passive mode.

An interesting point is that the *DAG* serie cards are natively supported by the upstream version of libpcap.

Sensory Networks NodalCore

The hardware acceleration product line of the company *Sensory Networks* consists of the *NodalCore* C-2000 Serie of PCI cards which are dedicated to operations such as pattern matching, data decompression or message digest generation.

Applications like SAA Proxy which are taking advantage of the hardware are also sold be the company as well as a complete C API under Linux in user space or in kernel space (see appendix B on page 100).

Chapter

10

Concurrent Programming

This chapter will try to explain the broad concept of concurrent programming, its advantages as well as the drawbacks brought by that technique. This topic has been quite important during the software development process of this project.

10.1 Introduction

Concurrent programming allows **different tasks** of a single application to run in a concurrent manner. The tasks could be running on the same processor, different processors in the same computer or even multiple computers interconnected by a network. This technique is used quite often these days, like for example, on desktop computers to allow the user to execute multiple applications at the same time, on servers to reply multiple clients at the same time, on embedded systems to handle different types of asynchronous events at the same time.

Because, usually, computers have a number of processors much smaller than the number of applications running at the same time, the Operating System (OS) has to juggle between all the running applications. This is done by running a single process on the processor during a certain time slice, saving the process state, loading the state of another process and running it from the exactly where it was left before. This whole procedure is handled by the OS and the operation of giving the processor to the next process is called a **context switch**.

Depending on the application, multiple concurrent tasks can offer a **performance gain** for the whole application and/or a **simpler design** than when using single task.

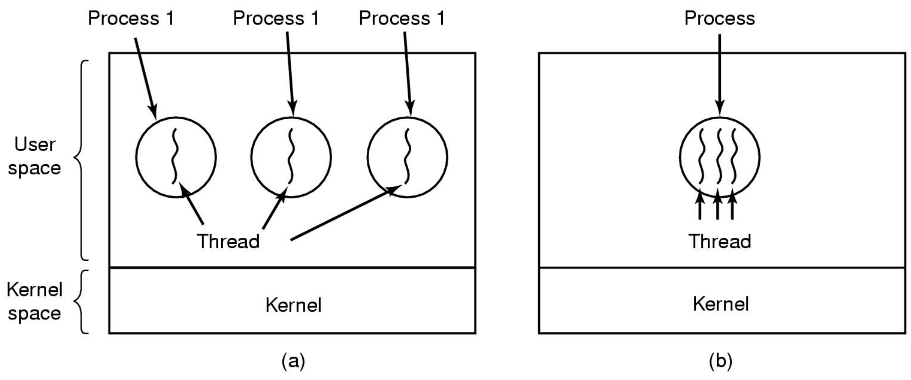


Figure 10.1: (a) Multiple processes each with a single thread (b) Multiple threads in the same process

The performance gain is mostly due to the application executing code while waiting for an I/O operation to finish. Because I/O operations are usually handled by others devices than the CPU, a hard disk or a network interface for example, the CPU can do better than stupidly waiting for the I/O operation to finish.

The design of an application using multiple concurrent tasks can be more logical for applications which are intrinsically composed of multiple operations which are running in parallel. Graphical User Interfaces (GUI) are a good examples because many operations could be running while the interface must respond quickly when the user uses his mouse to click somewhere. Applications which depend on multiple external events, a data acquisition applications for example, could use on task for each type of events it acquires and a task to process the received events. That way we can easily avoid missing events while processing is done.

10.2 Processes versus threads

The basic execution unit in concurrent programming is the **task** but most moderns Operating Systems provides two different types of tasks. The **processes** and the **threads**. By default, each process is running in his own address space (in which the global variables are located) whose access is carefully protected by the OS to avoid different processes interfering with each other. Each process contains, by default, a single execution thread. When using multiple threads in the same process they share the same address space and thus can easily interfere with each other. The figure 10.1 taken from [Tan01] shows (a) multiple processes each with a single thread and (b) one process with multiple threads.

User space and kernel space threads

Multi-threading can be implemented inside the kernel (kernel space), outside (user space) or even sometimes in an hybrid way. The main difference between user and kernel threads appears when calling a **blocking function** (such as `read()` system call), the **user thread** calling this function will **block the whole process** and thus all the other threads too, but the **kernel thread** will be the **only one to block** and the other threads of the process will continue to run. That is why user threads are usually using only non blocking system calls.

10.3 Synchronization primitives

Semaphores

A semaphore is one of the most basic synchronization primitive invented by Edsger Dijkstra, it can be considered as a **counter** associated with a processes **waiting queue**. It supports two operations whose names vary quite a bit in the literature, `V()` and `P()` in the original Dijkstra's theory, sometimes `up()` and `down()` and in the standard C library used under Linux : `sem_post()` and `sem_wait()`.

If the value of the counter is 0, the `sem_wait()` operation will block the calling task by putting it in the waiting queue until the counter value increase. Each task waiting in the queue will usually be freed in a FIFO[1] order.

In practice, semaphore are used in one of the following situations :

- A fixed number n of shared resources can be accessed at the same time, the semaphore will be initialized with n and each task will do a `sem_wait()` before accessing the shared resource and a `sem_post()` after to release the resource.
- A piece of code (called critical region) has to be accessed by only one task at the same time. A semaphore is initialized with 1 (in that special case is called **mutex**), every task entering the critical region has to do a `sem_wait()` and a `sem_post()` when leaving it.

Condition Variables

Condition variables allow tasks to wait for specific condition to become true before continuing to run, another task can then send a signal after having modified the condition and one of the waiting tasks will be run. It is also possible for a signal to be broadcasted to all waiting tasks.

[1]First In, First Out

The use of condition variables is a clean and efficient way to avoid using busy waiting (see section 10.7 on page 44 for a specific condition to become true.

10.4 POSIX Threads

The POSIX Threads library, also called pthread, is the most popular thread library available under Linux and many other Unix systems. It provides an abstraction layer for each different multi-threading environments. Under Linux, the interface of this library is defined in `/usr/include/pthread.h`

It supports the different synchronization methods explained before such as mutexes, semaphores and condition variables. This is the library which has been used during the implementation described in chapter 16 on page 66.

By using kernel threads it can take advantage of multi-processors systems but therefore requires thread-safe but also reentrant code (see section 10.8 on page 45).

Programming with POSIX Threads is explained in great details in [pth] and also in [BNF96].

Linux Kernel Implementation

The implementation of kernel threads in recent versions of the Linux Kernel (2.6 serie) is greatly described in [BC05]. They use the so-called lightweight processes (LWP), which are different processes sharing resources like the address space. The system call `clone()` is used to create them but most of the time only an abstraction library like pthread is using these system call.

10.5 GNU Portable Threads

GNU Portable Threads (or GNU Pth) is a user space thread library with which focuses on being portable across Operating Systems without too much effort. With user space thread libraries like this one, the thread scheduling — which is handled by the library — is cooperative, meaning that thread have to implicitly or explicitly hand out the CPU to the scheduler.

POSIX Threads Emulation

The GNU Pth library supports a POSIX Thread emulation mode which allows programs compiled with the POSIX Thread library to be run using GNU Pth without any recompilation. To do that, a shared library called `libpthread.so` is provided when GNU Pth is compiled with the `--enable-pthread --enable-shared` configure flags. Afterwards the `LD_PRELOAD` environment variable can be set like that to make use of the emulation layer :

```
export LD_PRELOAD=~/tmp/pth-2.0.7/.libs/libpthread.so
```

But don't forget that GNU Pth is library using user space threads while pthread is usually using kernel space threads and that blocking I/O cannot be handled the same way.

10.6 Problems

This section presents two of the most common problems encountered when dealing with concurrent programming, namely race conditions and deadlocks.

Race conditions

To show the problem of race conditions, let's take a simple example like the incrementation of a shared variable which, in C for example, looks like that :

```
value += 1;
```

The problem is that such a simple statement will be translated to three instructions — this depends on the architecture — by the compiler. Now we have two threads running this code at the same time, we can be lucky if the scheduler runs them like on figure[2] 10.2(a) on the next page. But like Murphy's Law[3] says it can too run them like on figure 10.2(b) on the following page resulting in an incorrect result.

To avoid race conditions all shared variables have to be locked with one of the synchronization primitives available (see section 10.3 on page 41).

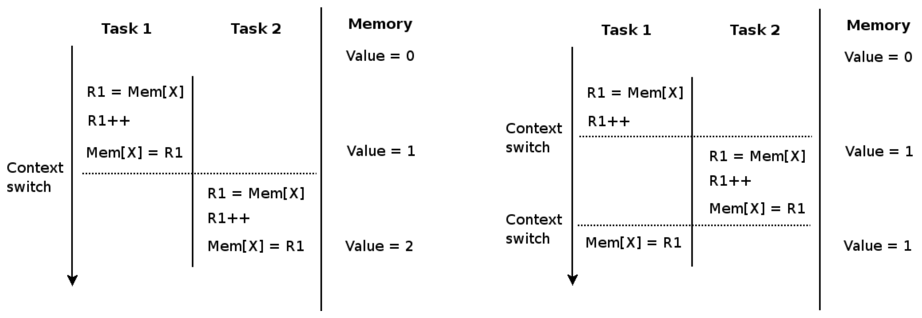
Deadlocks

A deadlock happens when two or more tasks are each waiting for another task to release a resource (unlocking a mutex for example). Nothing can happen because the task are all blocked and the program is frozen.

The following quote taken from [Wik06c] explains the problem in a nice way.

[2] R1 is any register and MEM represents the main memory

[3] If anything can go wrong, it will



(a) Good luck (b) Bad luck

Figure 10.2: Two tasks incrementing the same global variable

> This situation may be likened to two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler, a **deadlock** occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil, before he can give up the ruler. Both requests can't be satisfied, so a deadlock occurs.

To avoid deadlocks, many complex algorithm exists and are listed in [Wik06c]. But for simple cases, they can be avoided by designing the locking mechanism correctly and in coherent manner between different threads.

10.7 Things to avoid

This section will present problems which were encountered during the development of this project as well as the ways to avoid them.

Busy Waiting

When a thread or process has to wait for a external event, which is usually triggered by another thread or process, do not try to use a loop such as the one in the following example instead of using a condition variable or a semaphore.

Let's take the example of two threads, the first one has a specific work to do and when it finishes it will set the global flag `finished` to true. Meanwhile, the second thread has nothing to do and will only wait for the first one to finish. Now, imagine that the second thread use the following `while` loop to wait for the first one.

```
while (!finished);
```

Now, what can happen is that when the scheduler runs the second thread, it will loop in a useless way until the end of his time quantum before the first thread, which has some real work to do, can use the CPU. Then the global work which the program has to do will take longer.

As a side note, trying to use a syscall which gives back the control to the scheduler — such as `sched_yield()` — at each iteration of a waiting loop in a multi-threaded program usually won't schedule another thread of the same program but another process running on the system and thus can penalize the performance of the multi-threaded application.

10.8 Thread-safety and Reentrancy

Definition

A function is

- **Thread-safe** only if
  1. it uses locking for all the external data it uses
- **Reentrant** only if
  1. it does not hold static data between calls
  2. it does not return a pointer to static data
  3. it uses only the data provided by the caller
  4. it does not call non-reentrant functions

More information and examples is available in [aix99].

Usage

Thread-safety is mandatory when a function can be called by multiple threads (but not necessarily run at the same time) because without it the shared data structure will usually become corrupted and the behaviour of the program will usually become unpredictable which is always nice to avoid. Thread-safety applies to **every threads libraries** presented here.

Reentrancy is required when a function can be **run at the same time** or can be preempted during a call, usually on multi-processors systems. That mean that the system will be executing multiple instances of the function at the same moment with different data. Reentrancy applies only to thread libraries which are actually running code really in parallel, on multiple processors. For the different thread libraries presented before, it applies to POSIX Threads because they are using kernel threads but not to GNU Portable Threads because it use user space threads only and cannot take advantage of multi-processors systems.

10.9 Design pattern : Thread Pool

A thread pool is a design pattern often used in concurrent programming because it has the following two advantages :

- The overhead of thread creation and destruction is avoided because the threads are usually created only once at initialization, or when the size of pool is increased
- It allows to keep the number of concurrent threads running under control, thus avoiding taking the whole system down by loosing too much time in context switches instead of doing some real work

The interface of a thread pool can look like that :

```c
/* Allocate the thread pool and create count threads */
thread_pool_t *allocate_thread_pool(int count);

/* Wait for threads to finish, destroy them and free memory */
void desallocate_thread_pool(thread_pool_t *pool);

/* Ask the thread pool to execute function, this function may block if
 * no threads are currently available */
void send_to_pool(thread_pool_t *pool, void *function, void *data);
```

Chapter

11 Debugging

This chapter will present an overview of the tools which were used during this project to help with the debugging task during the software development which took place for this project. All the tools used are coming from free software projects and are running under Linux, the OS on which the whole development took place.

11.1 C Preprocessor and `printf()`

One of the simplest way to debug a program is to add calls to the `printf()` function in order to display informations about the current state of the program. Because it is quite easy to generate quite a bit of output using that techniques, it is usually used in conjunction with the `#if` C preprocessor command which allows those calls to be compiled in the program only when necessary, during the debugging phase for example.

Example

This example shows — in quite naive way — how to watch the effect of a function call on our variable `data`.

```
#include <stdio.h>

void myfunction(int *data)
{
  *data += 1;
}

```

```
int main(int argc, char **argv)
{
  int data = 0;

#if DEBUG
  printf("DEBUG Before: %d\n", data);
#endif

  myfunction(&data);

#if DEBUG
  printf("DEBUG: After: %d\n", data);
#endif

  printf("data = %d\n", data);
}
```

The result of this program will be different depending of the compilation flags given to the compiler.

```
francois@gordo:/tmp$ gcc -o test -DDEBUG test.c
francois@gordo:/tmp$ ./test
DEBUG Before: 0
DEBUG: After: 1
data = 1
francois@gordo:/tmp$ gcc -o test test.c
francois@gordo:/tmp$ ./test
data = 1
francois@gordo:/tmp$
```

Advantages and Drawbacks

This technique has the following advantages :

- No overhead at all when compiled without the `DEBUG` defined.
- No special software required, no special kernel support.

As well as some drawbacks like :

- It is quite easy to generate too much debug information and it is then quite difficult to bring out the relevant informations.
- The debug informations will only contain what was explicitly printed and it is quite frequent to miss the interesting information because it was not logged.

11.2 GNU gdb

The GNU Debugger (gdb) is part of the GNU system and was primarily developed by Richard Stallman. It is the most widely used debugger under Linux as well as many other Unix systems. It allows developers to execute programs in it while tracing his behavior, stopping it at specific points in the program (called breakpoints), examine and modify variable as well as calling functions.

It contains support for POSIX Threads (see 10.4 on page 42) and thus allows to watch the current state and stacktrace of each thread running in the application. The manual of gdb is available online, see [gdb].

Example

```
#include <stdio.h>

int main (int argc, char **argv)
{
  int *ptr = 0;
  *ptr = 1;
}
```

```
francois@gordo:/tmp/debug$ gdb ./test
GNU gdb 6.4.90-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...

(gdb) run
Starting program: /tmp/debug/test

Program received signal SIGSEGV, Segmentation fault.
0x0804833f in main () at test.c:6
6         *ptr = 1;
(gdb) bt
#0  0x0804833f in main () at test.c:6
(gdb) quit
The program is running.  Exit anyway? (y or n) y
francois@gordo:/tmp/debug$
```

11.3 Custom classifier module

A custom `pmacctd` classifier module has been developed during this project in order to catch some difficult to reproduce bugs (Hint: remember what was said in section 10.6 on page 43 ?) which caused memory corruption in the pmacctd core process.

The idea was to generate IP packets with a known payload (bytes values incrementing from 0 to 254), send them from the traffic generator host and use a simple classifier which will trigger a segfault exception — using the `abort()` syscall — when it received a packet which payload has changed.

The advantage of raising a segfault is that it can be catched directly in the debugger (`gdb` for example, see 11.2 on the preceding page) to find which packet made the corruption happen and thus have better chances to spot the place where the corruption happens.

Chapter

12 Profiling

This chapter will present methods and tools which can help during the profiling phase of a software development. Profiling is a way to get **timing informations** about a program to be able to spot inefficiencies and find where the **bottlenecks** are located.

Usually the profiling process has to be the lighter possible to avoid interfere with the usual workflow of the program. Three different methods will be presented each working at a different level.

The first one, OProfile is using a kernel module as well as a users pace daemon that gathers the profiling informations from the kernel module. It is able to profile the whole system including the kernel, the libraries and user space applications.

The second category is **user space debuggers** like *GNU gprof* or *qprof*.

12.1 OProfile

OProfile is a system profiler which uses a **kernel module** making use of performance counters found in common processors to be able to profile the whole system while keeping the impact on performances minimal.

An interesting tool called `opgprof` is available in the OProfile package and is able to generate `gmon.out` files compatibles with GNU gprof but without the need to include profiling instrumentation during the compilation of the program to profile (see section 12.2 on the next page).

The documentation and usage examples are available on [opr].

12.2 GNU gprof

The GNU Profiler (gprof) is a profiler working with special instrumentation compiled into the program. This instrumentation will add some code the each function calls and react to the `ITIMER_PROF` timer signal sent by the kernel, it will then dump useful informations to a gprof specific file called `gmon.out`.

Because gprof requires this instrumentation to be compiled in the program, during the compilation a special flag (`-pg`) has to be given to GCC[1]. Afterwards each time the program is run, it will dump profiling informations to the `gmon.out` file which can be decoded using the `gprof` command.

Multi-threading

By default gprof under Linux is only able to profile the main thread of a multi-threaded program which is usually not quite interesting, especially in our case (see implementation on chapter 16 on page 66).

Fortunately a workaround exists on [Hoc]. The instrumentation compiled into the program dumps informations to the `gmon.out` file when receiving a special timer signal `ITIMER_PROF` which is sent by the kernel. By default, under Linux, the signal received by the parent process of multiple threads are not sent to each threads. This workaround simply use a shared library (loaded using the `LD_PRELOAD` environment variable) which redefines the `pthread_create()` function to add support for threads to receive this timer signal too.

Example

Here a simple slow and useless program called `slow.c`.

```c
#include <stdio.h>

void myfunction(void)
{
  usleep(100);
  return;
}

int main(int argc, char **argv)
{
  int i;
  for (i = 0; i < 10000; i++) myfunction();
  return 0;
}
```

[1]The GNU Compiler Collection

```
francois@gordo:/tmp$ gcc -o slow slow.c
francois@gordo:/tmp$ time ./slow
real    0m40.104s
user    0m0.000s
sys     0m0.000s
francois@gordo:/tmp$ gcc -o slow -pg slow.c
francois@gordo:/tmp$ ./slow
francois@gordo:/tmp$ ls -l gmon.out
-rw-r--r-- 1 francois francois 366 2006-12-04 22:49 gmon.out
francois@gordo:/tmp$ gprof --brief slow
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00    10000     0.00     0.00  myfunction

                        Call graph

granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00   10000/10000       main [8]
[1]      0.0    0.00    0.00   10000         myfunction [1]
-----------------------------------------------
```

12.3 Custom Code

A simple timing module has been developed in C in order to help with the fine-grained profiling of specific parts of the `pmacctd` code. The goal was to be able to easily measure the execution time of any part of the code we were interested in.

Usage

The usage is quite simple, the following steps are required in order to know the time taken by a specific section of a program :

- Define a timer variable using the `struct mytimer` data structure
- Call `start_timer()` at the beginning of the interesting section
- Call `stop_timer()` at the end of the section, this will print the result with an attached message directly to the standard output (`STDERR`).

Integration Example

This module can be integrated the following way in an existing program in order so as to measure the time taken by a specific section of the code.

```
int function(void)
{
  struct mytimer t0; /* Initialize the timer variable */

  start_timer(&t0); /* Start the timer */

  ... /* Code of the interesting section */

  stop_timer(&t0, "name_of_the_interesting_session"); /* Display the result
       */

  return 0;
}
```

Example of Results

All the results printed to the standard output have the same format in order to be easily parsed afterwards to generate statistics. The format is `TIMER:%s:time_spent` where `%s` is a free form string defined when calling `stop_timer()` and `time_spent` is the time in micro-seconds measured between the two function calls.

```
TIMER:function:0x809d1a0:12774
TIMER:function:0x809d280:8
TIMER:send_to_pool::80
TIMER:function:0x809d0c0:12810
TIMER:send_to_pool::3
TIMER:send_to_pool::3
TIMER:function:0x809d520:905
TIMER:send_to_pool::744
TIMER:function:0x809cfe0:13608
TIMER:function:0x809d1a0:818
```

This result can easily by parsed by any scripting language to generate more summarized statistics. This is exactly what has been used to generate the table 17.10 on page 82.

C Implementation

The following C implementation is based on the `gettimeofday()` system call which achieve a resolution of 1 microsecond on our system, the details are available in appendix section D.1 on page 107.

```
#include "sys/time.h"
#include "time.h"
#include "stdio.h"
#include "stdarg.h"

/* Data structure */
struct mytimer {
  struct timeval t0;
          struct timeval t1;
};

/* Prototypes */
void start_timer(struct mytimer *);
void stop_timer(struct mytimer *, const char *, ...);

/* Functions */
void start_timer(struct mytimer *t)
{
  gettimeofday(&t->t0, NULL);
}

void stop_timer(struct mytimer *t, const char *format, ...)
{
  char msg[1024];
   va_list  ap;

  gettimeofday(&t->t1, NULL);
  va_start(ap, format);
  vsnprintf(msg, 1024, format, ap);
  va_end(ap);

   fprintf(stderr, "TIMER:%s:%d\n", msg, (t->t1.tv_sec - t->t0.tv_sec) *
        1000000 + (t->t1.tv_usec - t->t0.tv_usec));
}
```

Part III

Hardware Accelerated Network Probe

Chapter

13

Pmacct

This chapter presents a free software project called **pmacct** which is developed by Paolo Lucente and licensed under the *GNU Public License* (GPL) (see appendix G on page 124).

The name pmacct stands for *Promiscous Mode IP Accounting Package* and has become a set of different tools which can link many different technologies such as packet capture (using libpcap), flows protocols (NetFlow or sFlow), databases (MySQL, PostgreSQL, etc.). It can be used to interface multiple network monitoring protocols in consistent way.

The **pmacct** project is now composed of the following daemons :

`pmacctd`
: A promiscuous mode network probe based on **libpcap**

`nfacctd`
: A NetFlow v5 and v9 collector (see 3.3 on page 10)

`sfacctd`
: A sFlow collector (see 3.4 on page 12)

During this project we were especially interested in `pmacctd` because this is the only daemon which has access to the full packet payload by its use of libpcap.

13.1 Features

To be able to handle IP accounting on high-speed networks, the following features are implemented in the pmacct software suite :

Aggregation

Information can be aggregated as flows (see 3.3 on page 10) but also using specific aggregation methods like subnet aggregation, AS aggregation and so on.

Filtering
Filters can be applied on informations to be able to select only the relevant information. Filtering is configurable for input data — for example using a libpcap filter with `pmacctd` — or independently for each plugin.

Sampling
Sampling allows to analysis only a fraction of the traffic which will be statistically relevant so that we can interpolate later to get an approximation of the full traffic.

13.2 Plugins

Multiple output plugins can be run at the same time in order to export data to different places — for example to a database for billing purposes and to a memory table to use it in a graphing application. The following plugins are available in the pmacct distribution :

`imt`
In Memory Table plugin : Data is stored in main memory and can be accessed using the `pmacct` command-line tool. This plugin can be used with other softwares like Cacti (see 6.1 on page 22) to generate graphs for example.

`mysql`
Export data to a MySQL database which schema is configurable.

`pgsql`
Export data to a PostgreSQL database which schema is configurable.

`print`
The results are printed directly on the standard output and can be parsed by a script for example. This plugin was used during benchmarks by redirecting its output to `/dev/null`.

`sqlite3`
Export data to a Sqlite database which schema is configurable.

`nfprobe`
A plugin which generates NetFlow data to be sent to another NetFlow collector.

`sfprobe`
A plugin which generates sFlow data to be sent to another sFlow collector for example.

13.3 Traffic classification

Since version 0.10.0 `pmacctd` had classification features relying on packets payload using one of the following ways:

Patterns
: Regular expressions patterns fully compatible with those of the l7-filter project (see section 7.3 on page 27).

Shared libraries
: External classification modules which are loaded at runtime using the shared libraries mechanism. That means than external modules can be plugged easily without the need for recompilation.

Shared libraries

Shared libraries offer a way to do stateful analysis of the packets in a flow. At the time of this writing only two classifiers using this technique are distributed, on for *eDonkey* traffic and the other for RTP.

The implementation call a specific function (`u_int32_t classifier(...)`) which must return 1 in case of match and 0 in the other cases. The name of the protocol which has been matched is defined as a fixed value (`char protocol[]`) in the library source code.

It is possible to register new traffic classes using the `pmct_register()` function and then return a number greater than 1 which will mapped to the corresponding class. This method was used in our NodalCore classifier module described in chapter 15 on page 63.

Chapter

14

Sensory Network Hardware

The goal of this chapter is to describe the *NodalCore C-2000 Serie* of hardware acceleration cards for network security applications. This hardware card is already used in some Eneo products such as multi-services network appliances taking advantage of the acceleration the filter HTTP request for known patterns. The picture of the card we used — NodalCore C-2000 Ultra — is available on figure 14.1.

14.1 Features

The following features are provided by the current firmware (called bitstream) but because the product is based on a FPGA (see section 9.1 on page 36)

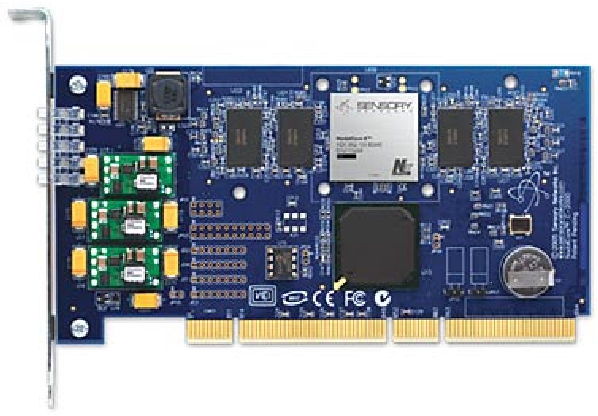


Figure 14.1: Sensory Networks NodalCore C-2000 Card

new features can be added with a firmware upgrade. Each current feature is available as one of those channels :

Pattern Matching Engine (PME)
A pattern engine engine working with regular expressions. This is the channel which was used during this project.

Massive Memory Architecture (MMA)
A string matching engine more efficient than the Pattern Matching Engine but working only with fixed strings.

Content Decoder
This channel principally allows decoding of mails using MIME-compatible formats such as Base64 and Quoted-Printable.

Decompression
This channel allows decompression of file compressed with different standards.

Message Digest
Can generated a message digest from a stream of data. This is can be used in virus detection for example.

14.2 Hardware specifications

This hardware is available as a PCI card which contains the following main components

- Xilinx Virtex-4 XC4VLX60 FPGA
- 144 MB of RAM (1 Gbit)

14.3 Pattern Matching Engine

The Pattern Matching Engine (PME) understands regular expressions in the NRPL format (see 8.5 on page 34). The regular expressions have to be compiled using the `ncore-compiler` tool available in the SDK[1].

14.4 Performances

In this section we will establish a performance baseline of the Sensor Network hardware which will permit to discuss how well our implementation takes advantage of the performances offered by the hardware.

[1]Software Development Kit

The performance of a pattern matching acceleration card like this on is mainly related to the bitrate it can handle. This bitrate depends on many factors such as the size of the data chunks written to the card, the use of the intrinsic parallelism offered by the hardware and the match rate.

14.5 Theoretical performances

The theoretical pattern matching performances of the NodalCore C-Serie depends on the firmware (called bitstream in Sensory's documentation) used. Usually firmwares have names such as Core*X* with the *X* meaning the raw data scanning bitrate in Mbps. For example, the Core1500 firmware which was installed by default should allow the scanning of 1.5 Gbps of data. This performance can also depend of the features used with the cards. In our case we will focus on the Pattern Matching Engine.

The firmware allows multiple streams of data to be handled in parallel, for the NodalCore C-2000 Ultra card we used, 12 independent channels are available.

With the NodalCore C-2000 card with the firmware Core1500 we have a theoretical throughput of 125 Mbps per channel.

Practical considerations

In the section "Optimizing Performances" of [Sena], the following performance limiting factors are described :

- The size of data chunks written to the hardware
- The use of multiple streams at the same time
- The match frequency

Some benchmarks have been designed (see section 17.3 on page 74) to get an idea of the influence of the first two parameters on the throughput of the card. The match frequency has not been benchmarked but according to our discussion with *Sensory Networks* that must not be a problem if the number of matches per packet doesn't excess one per packet which will hopefully be the case with well designed patterns.

Chapter

15

Implementation: Pmacct NodalCore classifier

This chapter will present the implementation of a classifier shared module for `pmacctd` which is using hardware acceleration via the NodalCore API. The goal of this first implementation was to integrate calls to the NodalCore API in the current codebase of pmacct to be able to let the hardware card handle the pattern matching which was precedently done by pmacct using a standard regular expression library, the Henry Spencer's implementation described in section 8.5 on page 34.

15.1 Limitations

It has been decided that this first implementation will not take into account the parallelism offered by the hardware so we are only using a single data stream at the moment. And we are neither taking advantage of doing other operations with the host CPU while to card is working, so the CPU will be idle. Both those restrictions will, for sure, penalize the performance of the system but the main goal is to understand the inner working of the NodalCore API and the potentials implications on a third-party application like `pmacctd`.

15.2 Pmacct classifier extension

The shared library classifier API available in pmacct will be used, thus giving the advantage of avoiding to modify the core of the software. This shared library can even be compiled separately and only require the headers available to pmacct classifiers (`common.h` in the `pmacct-classifiers` archive).

Variables prototypes
`char protocol[]`
`char type[]`
`char version[]`
Functions prototypes
`int init(void **extra)`
`u_int32_t classifier(struct pkt_classifier_data *data, int len, void **context, void **rev_context, void **extra)`

Table 15.1: pmacct classifier extension API

The pmacct classifier API requires the following methods to be defined in each extension :

- `init`
  the initialization method which is called at startup when the extension is loaded
- `classifier`
  the main method called at each classification tentative by the pmacctd core

Moreover, three variables have to be defined in each extension :

- `protocol`
  the name of the protocol matched by the extension. In our case this is a bit useless because we match many different protocols using the same extension. In the implementation it is defined as `ncore` but is not really used by pmacctd.
- `type`
  the type of the extension. At the moment the only type recognized if the `"classifier"` type.
- `version`
  the version number of the module for documentation purposes.

The C prototypes of the two functions and three variables which are required for each extension are given in table 15.1 in addition to the functions exported by the classifier API to be used in extensions. These exported functions are mostly related to protocol names management.

15.3 NodalCore classifier module

The implementation of a NodalCore classifier module has been done during this project using the NodalCore API wrapper described in section B.2 on page 101 which eased the initialization of the pattern matching engine. The goal was to keep this classifier simple and program it in a thread-safe way (see section 10.8 on page 45) because it had to be usable with the multi-threaded `pmacctd` described in chapter 16 on the following page.

15.4 Distribution

All the code presented in this chapter is kept internal to *Eneo Tecnologá* because it uses the NodalCore API which is covered by a NDA.

Chapter

16

Implementation: Pmacct multi-threaded core

During this second implementation, whose design has been greatly discussed with Paolo Lucente, author of pmacct ([Luc]), it has been decided that the easiest way to achieve a well performing integration of the Sensory hardware as a classifier module of `pmacctd` was to implement multi-threading the core process of `pmacctd`.

16.1 Single-threaded Design

The current official version of `pmacctd` core process is single-threaded, as seen on figure 16.1 on the next page, the whole packet processing from the libpcap callback (`pcap_cb()`) to the plugins output (`exec_plugins()`) is done serially and is synchronized by libpcap. Each received packet will trigger the whole processing.

16.2 Multi-threaded Design

Because usually hardware acceleration based on FPGA is highly parallel — in our case the NodalCore hardware card supports 12 pattern matching engines — we needed to handle multiples packets at the same time in the core process which would require a major rework. To avoid that the idea was to encapsulate each packet in a thread during the phase of aggregation into flows and classification.

In order to keep the number of threads constant and avoid a system freeze due

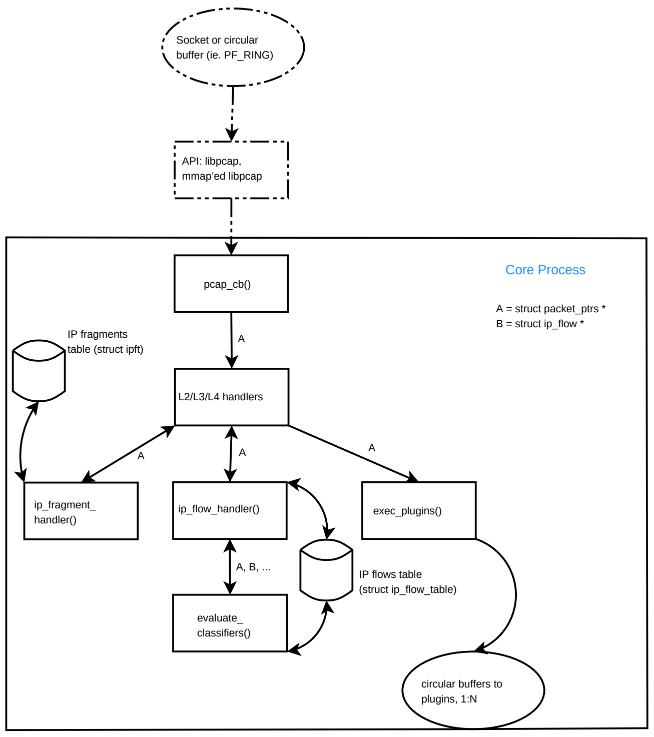


Figure 16.1: Pmacct single-threaded core design schema

to too much threads running, for example, in case of a too high packet rate, a thread pool (see section 10.9 on page 46) has been implemented and used. This is represented on figure 16.2 on the next page by the multiple dashed boxes. The size of the thread pool is defined by the new configuration variable `flow_handling_threads`.

This design requires an extra data copy of the packet and associated informations (the `struct packet_ptrs` data structure) before sending a new packet to one of the thread in the thread pool.

16.3 Shared Data Structures Locking

The following data structures are shared in the thread pool and requires locking :

- `ip_flow_table`
  The IP flows table.
- `channels_list`
  The communication channels with plugins.

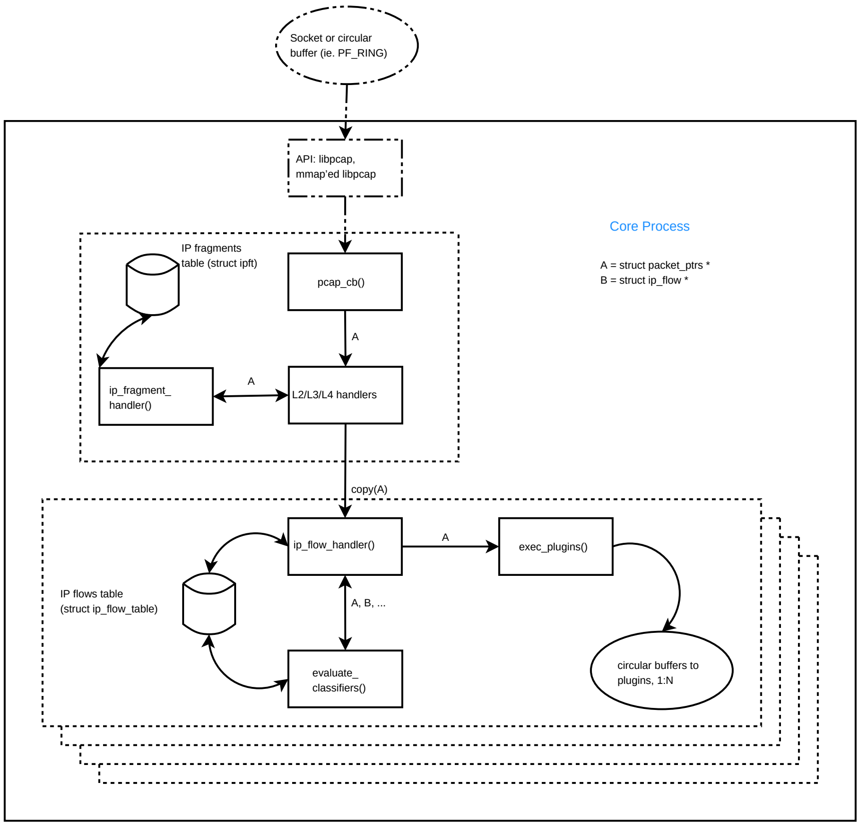


Figure 16.2: Pmacct multi-threaded core design schema

In both cases the locking is done with a single mutex for the whole data structure which means to their access is fully serialized, only a single thread can use them at the same time. This locking policy has been chosen because of the data structure complexity. For example, the IP flows table is composed of a double-linked hashed list and a LRU[1] cache to optimize it when using recent flows which is quite difficult to lock finely.

16.4 Build System

The build system used by the pmacct project is based on the GNU Autoconf and GNU Automake tools which generate almost automatically `configure` scripts and `Makefile` files to compile the software in a portable way across different Unix systems.

The `configure` script accepts flags which are used to enable or disable functionalities at compilation time. It was used to add the `-enable-threads` flag

[1]Last Recently Used

which avoid using the multi-threaded version by default before it was extensively tested and the remaining performances issues fixed (see section 17.6 on page 81).

The usage of this flag will generate `Makefile`s to compile the software using a new compilation flag called `USE_THREADS` which is recognized by the C preprocessor and will compile the suitable code by using `#ifdef` preprocessor commands.

In depth informations about the GNU Autools utilities is available in [GVVT00].

16.5 Future Enhancements

The performance issues described in section 17.6 on page 81 must be resolved, possibly by improving the locking mechanism which might require some modifications of the *IP Flows Table* data structure.

Some improvements can also be mimicked from the nProbe implementation described in section 7.2 on page 24. But some care has to be taken because nProbe uses an aggregation method static, the original flow definition (see section 3.3 on page 10) whereas pmacctd can do custom aggregation.

16.6 Distribution

All the code presented in this chapter has been included in the pmacct project CVS repository under the GPL license (see appendix G on page 124) and will be include in the next official release of pmacct.

Chapter

17 Benchmarks

In this chapter we will have a look at the benchmarks which were done to evaluate the performances of the implementations, described in the chapters 15 on page 63 and 16 on page 66, of an hardware accelerated pattern matching classification working with the `pmacctd` network probe.

The figure 17.1 shows the different elements involved in the software, each element were — when possible — benchmarked separately to get a better understanding of their interactions and try the spot the potential bottlenecks. The following parts have been benchmarked independently :

1. Libpcap
2. NodalCore API
3. Pmacctd single-threaded with and without NodalCore acceleration
4. Pmacctd multi-threaded with and without NodalCore acceleration

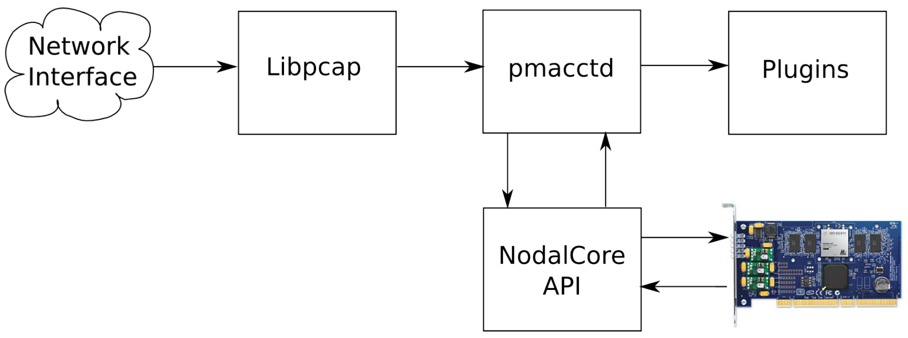


Figure 17.1: The pmacct with NodalCore classifier big picture

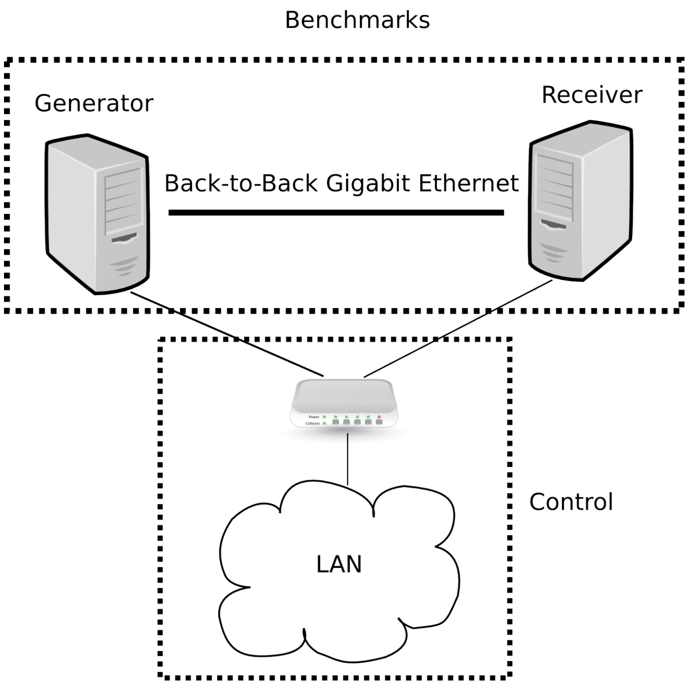


Figure 17.2: Benchmarks testbed schema

17.1 Benchmark Methodology

Testbed Setup

The testbed setup used for benchmarks is described on figure 17.2. It consisted of two computers, whose specifications are in table 17.1 on the next page, the **generator** which generate traffic from previously made real traffic captures replayed with `tcpreplay` and the **receiver** which runs the tested software and discards the traffic afterwards. Both computer are connected to the LAN and remotely accessed using SSH.

A set of Python scripts have been developed to be able to easily reproduce the benchmarks during the code optimization phase. This allows the benchmarks to run automatically during the night for example. These scripts are also using SSH to control the testbed. That way, they could be launched even from outside the office.

Custom Software

Some custom software has been developed for each type of benchmark made. Specific C programs were used for the low-level benchmarks of **Libpcap** and

	Generator	**Receiver**
Model	Nextgate NSAIO46	Dell Poweredge SC1425
CPU	1 Intel P4 3GHz HT	1 Intel Xeon 3GHz HT
RAM	1 GB	1 GB
Harddisk	SCSI	SCSI
Ethernet	1000 Mbps (`sky2`)	1000 Mbps (`e1000`)

Table 17.1: Testbed hardware specifications

the **NodalCore API** whereas more high-level benchmarks like the one of **pmacctd** where remotely controlled by a Python script. All the programs are presented in appendix E on page 109.

Packet generation

The traffic generation has been done using the free software project `tcpreplay`, see [Tur]. The version 3.0.beta7 was used. The maximum throughput of generated traffic achieved using our testbed setup and the `-t` (top-speed) was about 120 kpps and 677 Mbps.

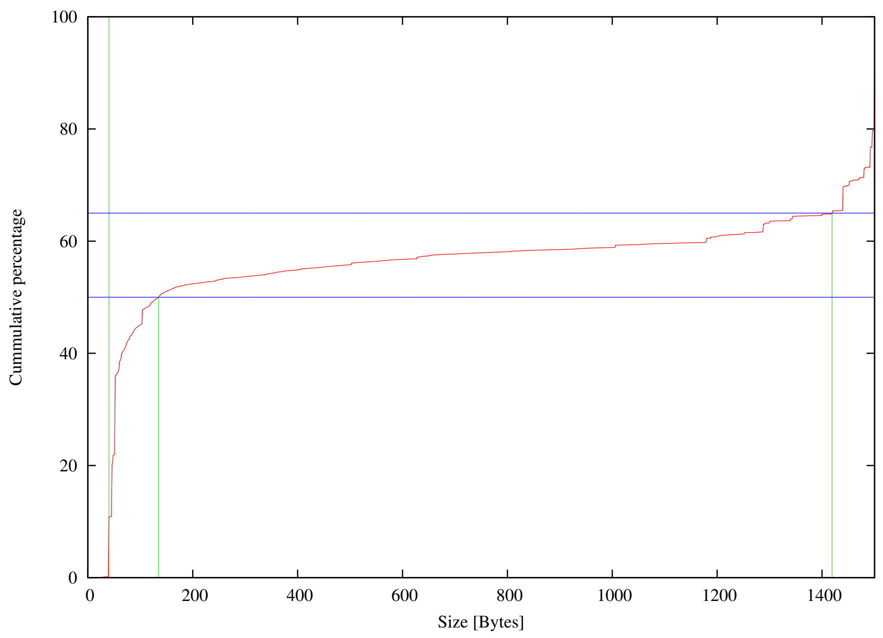
Real Traffic Capture File

In order to get the most realistic results possible from these benchmarks, a pcap capture file has been created using `tcpdump` on the border router of Swiss ISP[1]. The table 17.3(b) on the next page gives the informations about the capture file used during benchmarks. The graph on figure 17.3(a) on the following page shows the cumulative packet size distribution of this capture.

These results are quite similar to those found by *Caida*[2] with captures done at the *NASA Ames Internet Exchange* in 1999-2000 (see [cai]). Except that we do not have a peak in the number of packets around 590 bytes. The following interesting points can be derived from this packet size analysis :

- 50% of the packets are small in the range 40 to 135 bytes;
- 35% of the packets are big in the range 1419 to 1500 bytes;
- The remaining 15% are almost equally distributed between 136 and 1418 bytes.

[1]Saitis Networks, AS6893, `http://www.saitis.net/`
[2]the Cooperative Association for Internet Data Analysis



Duration	5 minutes 19 secondes
Size	655413743 bytes, about 625 MB
Number of packets	947061
Mean Packet Size	676 bytes
Date	Thu Sep 28 11:30:58 CEST 2006

Figure 17.3: (a) Packet Size Distribution (b) Capture File Specifications

Packet Sizes

It is important to define which kind of packet sizes we will be confronted with in order to take this factor into account while doing optimizations. The figure 17.4 on the next page is summary of the packet sizes which can be found on IP network based on Ethernet links without taking into account jumbo frames (see [Dyk99].

Because pattern matching is only done on the data payload of IP packets using protocols UDP or TCP, we will need to handle payload of sizes between 0 and 1472 bytes (maximum payload size using UDP), as according to the graph on figure 17.3(a) the most frequent packet sizes will be in the ranges 40-135 bytes and 1419-1500 bytes, so both extremes of payload sizes will be frequent.

			Payload per packet	
OSI Layer	**Protocol**	**Header**[a]	**Minimum**	**Maximum**
2	Ethernet	14	46	1500
3	IP	20-60[b]	0	1480
4	UDP	8[c]	0	1472
4	TCP	20-X[d]	0	1460

[a]Include trailing checksum in the case of Ethernet
[b]see [rfc791]
[c]see [rfc768]
[d]see [rfc793]

Figure 17.4: Packets Size Limits by Protocol

17.2 Libpcap

The performance of libpcap — the packet capture library used by `pmacctd` to interface with the packet capture mechanism available in the Linux kernel — has been benchmarked using the program described in section E.2 on page 110. The result of the benchmark is that no significant packet loss was experienced for packet rates up to 50'000 pps.

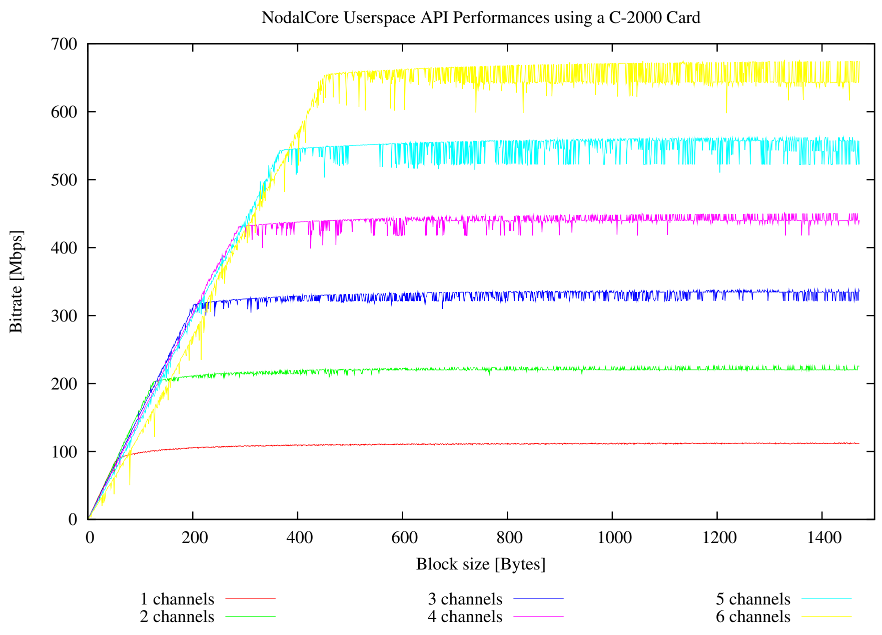
But the packet loss at the libpcap level is highly dependent of the time spent by the callback function before returning because newer packet cannot be processed before that. By the way, it was also a good test to know if the Ethernet settings — like speed and duplex settings for example — were correct.

17.3 NodalCore API and Hardware

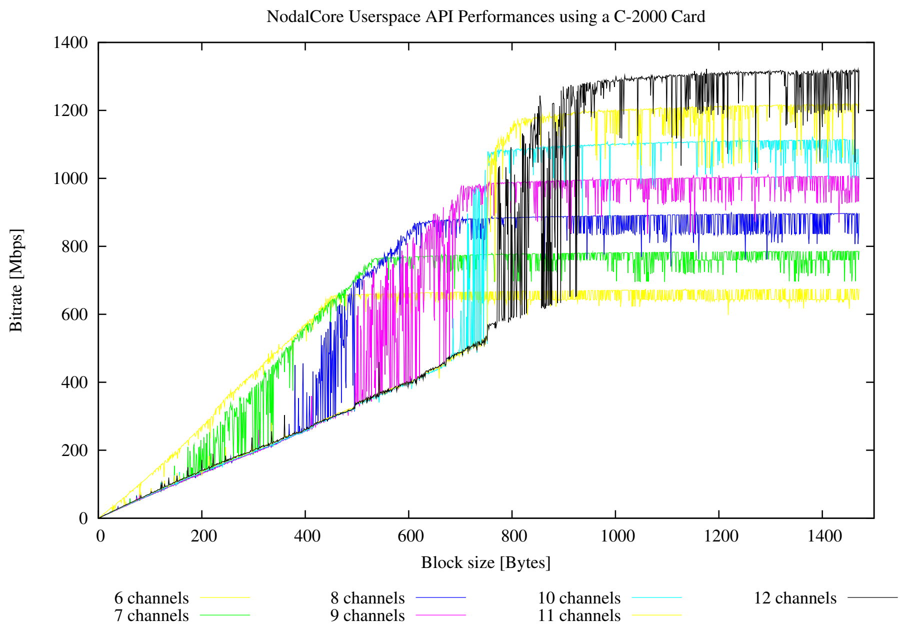
The figure 17.5 on the next page shows the bitrate in function of the block size when using from 1 to 12 streams in parallel on the card. The maximum of 12 channels is due to the card specifications and is certainly defined by the number of gates available on the FPGA divided by the number of gates the pattern matching engine use. The graphs has been done with the benchmarking program described in section E.3 on page 112.

Results

The block size after which throughput becomes stable is about 75-80 bytes per channels. When using between 7 and 12 channels in parallel, our results are quite noisy. The issue has been investigated in great details with *Sensory Networks* but the exact reason was not found at the time of writing.



(a) 1 channels to 6 channels



(b) 6 channels to 12 channels

Figure 17.5: NodalCore C-2000: Throughput versus block size with multiple channels

Linux Realtime Priority

While trying to understand the cause of the noisy graph (see 17.5(b) on the previous page), the use of a Linux kernel scheduling feature has been tried. By calling the `sched_setscheduler()` syscall (on line 11 of the following example), a user space process can ask the kernel to modify the way it will be scheduled. This feature can easily crash the machine if the process use the CPU all the time and has the maximum priority because the scheduler will run only this process and nothing else.

The following code was used to activate this scheduler feature, giving our process the higher priority possible.

```
 1 #include <sched.h>
 2 int set_realtime_priority (void)
 3 {
 4 struct sched_param schp;
 5   /*
 6   * set the process to realtime privs
 7   */
 8   memset(&schp, 0, sizeof(schp));
 9   schp.sched_priority = sched_get_priority_max(SCHED_FIFO);
10
11   if (sched_setscheduler(0, SCHED_FIFO, &schp) != 0) {
12     perror("sched_setscheduler");
13     return -1;
14   }
15 }
```

Unfortunately the use of this feature triggered a bug somewhere in the kernel or in the NodalCore kernel driver and thus didn't gave better results. This bug has been reported to *Sensory Networks*.

```
# dmesg
BUG: soft lockup detected on CPU#0!

Pid: 31090, comm:         ncore-bench1
[...]
```

HyperThreading

HyperThreading is an *Intel* trademark for a technology allowing two threads to be running on the same core at the same time. The idea is that sometimes threads have to wait — when loading a word from memory for example — and this waiting time can be used to run another threads without requiring a real context switch.

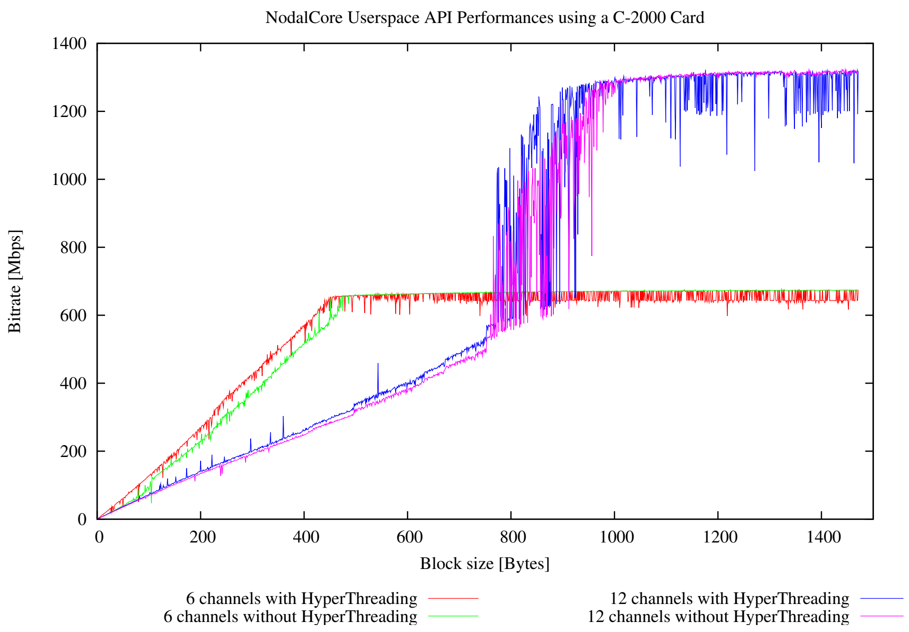


Figure 17.6: NodalCore C-2000: Comparison with and without HyperThreading activated on the host CPU

During the experiments made to find the reason of the throughput instability appearing in our results, the same benchmarks were run after the deactivation of the HyperThreading feature of the host CPU. The results on figure 17.6 shows that the throughput is more stable without HyperThreading but slightly lower. But it doesn't solve the instability for medium sized packets.

Conclusion

We can draw the following conclusions from this analysis of the NodalCore hardware during those benchmarks :

1. The more channels we use the more slowly small packet will be processed, this might be due to the stream switching overhead.
2. Small packets can easily penalize the system. Grouping the I/O operations in a single one with multiple packets can help lowering this overhead but in our case it requires major modifications in the design of `pmacctd`.
3. On our test setup, the HyperThreading feature does not influence much the performances.

17.4 Pmacctd

The official[3] single-threaded `pmacctd` version 0.11.0 has been compared to the multi-threaded version implemented during this project (see chapter 16 on page 66). Different series of tests have been made with or without the NodalCore classifier module described in section 15.3 on page 64.

The following configuration file has been used for all the benchmarks :

```
debug: false
daemonize: false
!
interface: eth1
!
classifiers: /tmp/classifiers
!
pmacctd_flow_buffer_size: 32768000
snaplen: 1500
!
aggregate: src_host,dst_host,class,flows,src_port,dst_port
!
plugins: print
```

The graphs presented in the following sections are using packets per second on the x axis because in a network application this measure is usually more important than the bitrate in term of processing power. But using the mean packet size from figure 17.4 on page 74 which is 676 bytes we can calculate the throughput.

Our graphs are done between 10'000 pps (about 54 Mbps) to 50'000 pps (about 270 Mbps).

Single-threaded versus Multi-threaded : without NodalCore classifier

The following conclusions can be drawn from the graph on figure 17.7 on the following page which compares the performances of both the single-threaded and the multi-threaded version of pmacctd without using any hardware acceleration :

1. The multi-threaded implementation performances are lower than the single-threaded. This is primarily due to the overhead generated by the thread handling (one more memory copy is necessary) as well as locking of the shared data structures.
2. Using only one thread doesn't gives any benefits and only adds unnecessary overhead.

[3]downloaded from the official website, see [Luc]

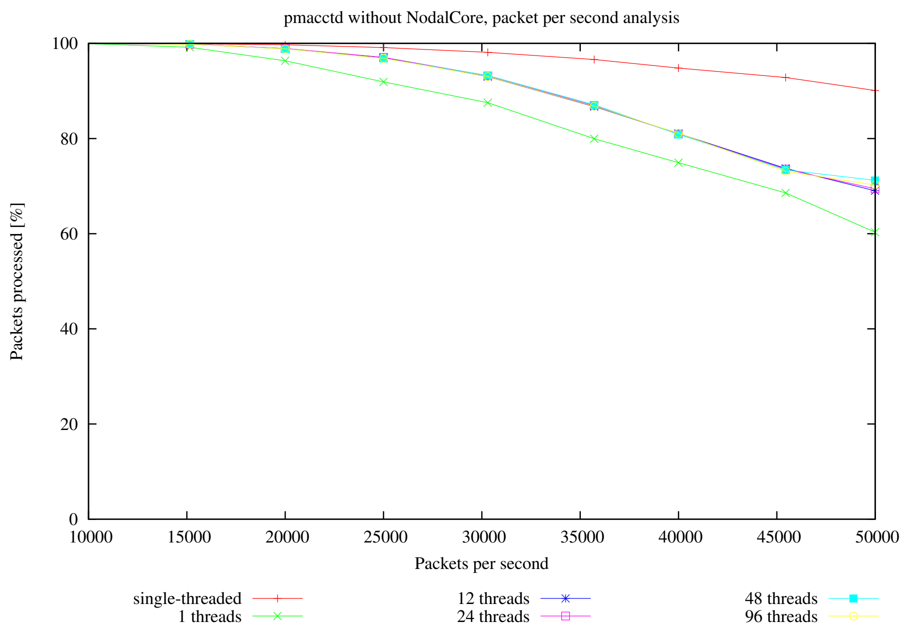


Figure 17.7: `pmacctd` comparison without NodalCore classifier

3. Without using the NodalCore classifier, the number of threads does not impact much the performances of the system.

Single-threaded versus Multi-threaded : with NodalCore classifier

When comparing the single-threaded and multi-threaded version of pmacctd when using hardware acceleration on figure 17.8 on the next page, the following points can be drawn :

1. The single-threaded implementation and the multi-threaded one when running only one thread both are performing pretty bad when using the hardware acceleration, this is due to the processes waiting uselessly while the card is processing the data payload.
2. When using a greater number of threads it performs better but trying to use too many threads can hit some system limits. When using more than 12 threads (the number of independent channels on the card), the waiting threads are acting like temporary buffers.

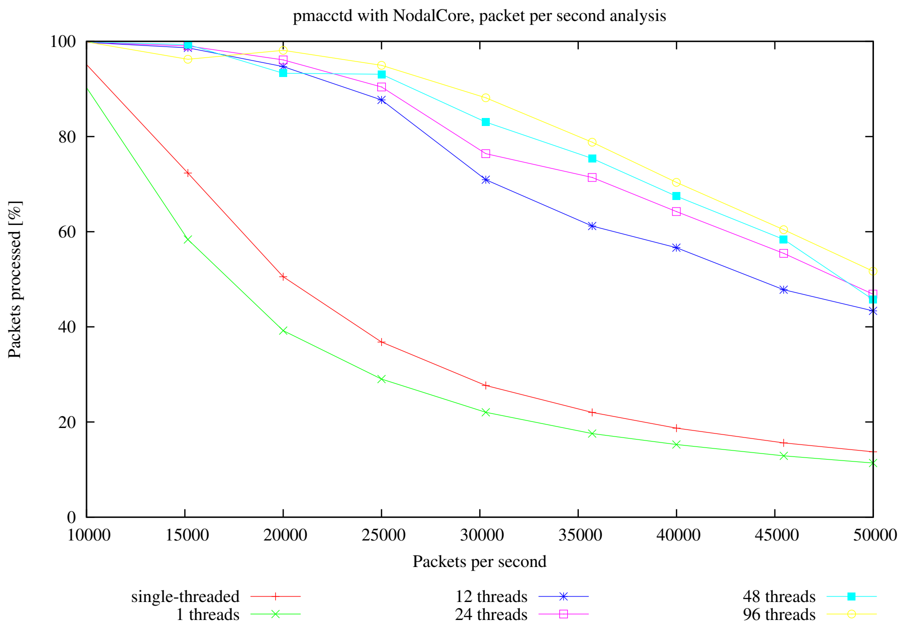


Figure 17.8: `pmacctd` comparison with NodalCore classifier

Pattern Matching Classifier comparison

This time we are comparing the performances of different pattern matching methods in pmacctd.

1. Using NodalCore acceleration with the single-threaded version is totally useless
2. The multi-threaded version using NodalCore acceleration works better but is still slower than software pattern matching. This is certainly because of too much overhead in the multi-threaded version, more on that in the following section.
3. Even though the multi-threaded version using NodalCore acceleration is a bit slower, it keeps some advantage due to the way pattern matching is handled in hardware : its processing time is fixed independently of the complexity and number of regular expressions which is not the case in a software implementation.

17.5 Profiling

The previous results showed that the overhead of the multi-threaded architecture implemented in `pmacctd` was quite big. That is why we had to do some

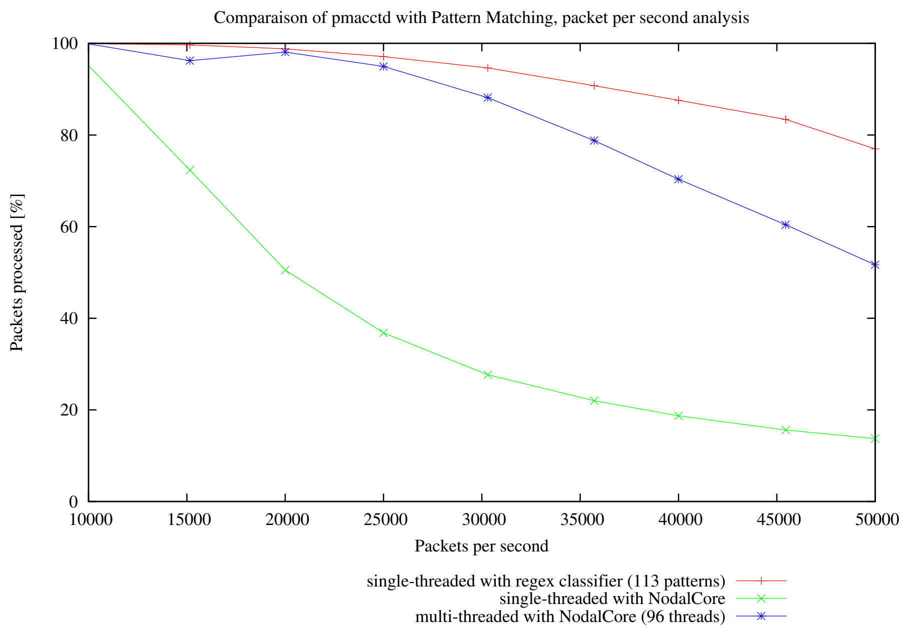


Figure 17.9: `pmacctd` Pattern Matching Classifier comparison

profiling of the code to find the bottlenecks and try to fix them. Different tools have been used during this profiling, most of them described in chapter 12 on page 51.

The table 17.10 on the next page shows the results of a profiling session done using the custom timing implementation described in 12.3 on page 53. These data have been acquired with `pmacctd` running on replayed traffic, like in section 17.4 on page 78 using 48 concurrent threads.

The results from this table shows clearly that the locking of the shared IP flows table is the culprit for the badly performing multi-threaded version.

17.6 Conclusion and Future Work

First of all, all the benchmarks have only been done with packet rates ranging from 10'000 pps to 50'000 ppp which, in our case, is about equivalent to 50 Mbps to 250 Mbps even though the Ethernet interfaces used were able to reach 1 Gbps. This is primarily due to performances limitations of our packet generator.

From the results of libpcap benchmarks (section 17.2 on page 74), we can say that on our test hardware, libpcap itself was not a bottleneck even though it can become one depending of the hardware used as showed by Luca Deri in

		Time				
Section/Function	**Calls**	**Total**	**Min**	**Max**	**Mean)**	**%**
send_to_pool	695823	68.85	0	10008849	98	100
ip_flow_handler	695823	3958.01	7	10048331	5688	100
`ip_flow_table` locking	695823	3789.86	0	10048298	5446	**95.8**
Write to hardware	138336	1.57	4	1857	11	0.03
Match waiting	138336	132.57	32	10008920	958	3.35
Stream index	138336	0.18	0	574	1	0.00
Remaining						0.82

Total in s
Min, **Max** and **Mean** in μs

Figure 17.10: `pmacctd` multi-threaded profiling results

[Der04].

Clearly, there is room for improvement performance-wise in `pmacctd` to be able to handle higher throughput. Here is some interesting topics for future work to optimize the throughput of the system :

- Concurrency improvement of the core process by using a finer-grained locking mechanism for the `ip_flow_table`
- Using an optimized libpcap implementation such as libpcap-mmap which support an in-kernel packet buffer to better handle fluctuation in packet rate (see section 4.3 on page 16) or nCap (see section 4.3 on page 17)
- Some other optimizations in the core process to decrease the time spent for each packet can be possible

Chapter

18

Conclusion

This document should have given a good overview of the different IP classification techniques available these days. It has focused primarily on pattern matching, except for chapter 5 on page 18, and thus working only with protocols which are not fully encrypted. Fortunately, this accounts for most of the protocols actually in wide use and will certainly last a moment because of the significant overhead generated by the use of cryptography.

During this project an implementation of an hardware accelerated pattern matching classifier using NodalCore cards has been made in `pmacctd` which cannot be distributed because the NodalCore API is not available under the GPL. All the multi-threaded development done in the pmacct has been included in the official distribution covered by the GPL. The network probe `pmacctd` supports many features like custom aggregation of packet into flows or multiple output plugins which makes the packet processing more complex. This can be pretty interesting for an integrated appliance which could take advantage of hardware acceleration to free some CPU for other tasks but does not need to handle packet rates too high.

If the goal is developing a high performance NetFlow probe with hardware accelerated pattern matching functionality, the CPU budget per packet is quite small even with today processors (see introduction of chapter 9 on page 35, so it has to be the most simple possible to be efficient. In that case using a less generic single purpose NetFlow probe like nProbe can be more efficient because it has already support for Gigabit speeds and its design is simpler.

During the course of the multiple researches and developments done during this project, a great panel[1] of technologies have been studied and applied. From NetFlow simplicity at first look and future improvements, to regular expressions and their dirty secrets, to high performance FPGA dedicated to

[1]What is proved by the diversified Table of Content of this document

enhance network security or weaken it[2] and we also have been flying through the darkness of concurrent programming and locking issues.

This document should contain enough background informations and bibliographic references to help with the continuation of the project in *Eneo Tecnología*. It has also hopefully opened some new ideas on possible future works. This domain is pretty far-reaching and we have scratched the surface.

[2]Do you remember the example given in section 9.1 on page 36 ?

RFC Bibliography

[rfc1157] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. *Simple Network Management Protocol (SNMP).* RFC 1157 (Historic), May 1990.

[rfc3954] B. Claise. *Cisco Systems NetFlow Services Export Version 9.* RFC 3954 (Informational), October 2004.

[rfc2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[rfc768] J. Postel. *User Datagram Protocol.* RFC 768 (Standard), August 1980.

[rfc791] J. Postel. *Internet Protocol.* RFC 791 (Standard), September 1981. Updated by RFC 1349.

[rfc793] J. Postel. *Transmission Control Protocol.* RFC 793 (Standard), September 1981. Updated by RFC 3168.

[rfc3917] J. Quittek, T. Zseby, B. Claise, and S. Zander. *Requirements for IP Flow Information Export (IPFIX).* RFC 3917 (Informational), October 2004.

Bibliography

Note: For reliability reasons all references to articles available on Wikipedia — a free encyclopedia based on wiki technology which is freely editable — are associated with a web link to the version of the article used during this project and the facts have been verified with other sources.

[aix99] Writing reentrant and thread-safe code [online]. 1999. Available from World Wide Web: `http://www.unet.univie.ac.at/aix/aixprggd/genprogc/writing_reentrant_thread_safe_code.htm`.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* O'Reilly, 2005.

[BD06] Philippe BIONDI and Fabrice DESCLAUX. Silver needle in the skype. 2006.

[BNF96] Dick Buttlar Bradford Nichols and Jacqueline Proulx Farrell. *PThreads Programming — A POSIX Standard for Better Multiprocessing.* O'Reilly, 1996.

[BS04] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. 2004. Available from World Wide Web: `http://www.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-039-04.pdf`.

[cac] Cacti: The complete rrdtool-based graphing solution [online]. Available from World Wide Web: `http://cacti.net/`.

[cai] Packet length distributions [online, cited 24th November 2006]. Available from World Wide Web: `http://www.caida.org/analysis/AIX/plen_hist/`.

[ces] Cesnet [online]. Available from World Wide Web: `http://www.cesnet.cz/`.

[CM] Fivos Constantinou and Panayiotis Mavrommatis. Identifying known and unknown p2p traffic. Available from World Wide Web: `http://theory.lcs.mit.edu/~pmavrom/p2p/`.

[Der] Luca Deri. nprobe — an extensible netflow v5/v9/ipfix gpl probe for ipv4/v6 [online]. Available from World Wide Web: `http://www.ntop.org/nProbe.html`.

[Der04] Luca Deri. Improving passive packet capture: Beyond device polling [online]. 2004. Available from World Wide Web: `http://luca.ntop.org/Ring.pdf`.

[Dyk99] Phil Dykstra. Gigabit ethernet jumbo frames — and why you should care [online]. 1999. Available from World Wide Web: `http://sd.wareonearth.com/~phil/jumbo.html`.

[end] Endace [online]. Available from World Wide Web: `http://www.endace.com/`.

[Far06] Nick Farrell. Razorback2 killed. *The Inquirer*, 2006. Available from World Wide Web: `http://www.theinquirer.net/default.aspx?article=29834`.

[fre] The freenet network project. Available from World Wide Web: `http://freenetproject.org/`.

[Fri06] Jeffrey E. F. Friedl. *Mastering Regular Expressions, 3rd Edition.* O'Reilly, 2006.

[gdb] *GDB User Manual.* The Free Software Fondation. Available from World Wide Web: `http://sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html`.

[gnu] Gnutella protocol development [online]. Available from World Wide Web: `http://www.the-gdf.org/`.

[GVVT00] Tom Tromey Gary V. Vaughan, Ben Elliston and Ian Lance Taylor. *GNU AUTOCONF, AUTOMAKE AND LIBTOOL.* Sams Publishing, 2000. Available from World Wide Web: `http://sources.redhat.com/autobook/`.

[Haaa] Peter Haag. Nfdump [online]. Available from World Wide Web: `http://nfdump.sourceforge.net`.

[Haab] Peter Haag. Nfsen [online]. Available from World Wide Web: `http://nfsen.sourceforge.net`.

[HM06] David Hulton and Dan Moniz. Fast pwning assured, hardware hacks and cracks with fpgas. *Black Hat Briefings 2006*, 2006. Available from World Wide Web: `http://blackhat.com/presentations/bh-usa-06/BH-US-06-Moniz-Hulton.pdf`.

[Hoc] Sam Hocevar. Howto: using gprof with multithreaded applications [online]. Available from World Wide Web: `http://sam.zoy.org/writings/programming/gprof.html`.

[Ins01] Gianluca Insolvibile. The linux socket filter: Sniffing bytes over the network. 2001. Available from World Wide Web: `http://www.linuxjournal.com/article/4659`.

[KBFc04] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and Kc claffy. Transport layer identification of p2p traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, New York, NY, USA, 2004. ACM Press.

[KPF05] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 229–240, New York, NY, USA, 2005. ACM Press.

[l7fa] Application layer packet classifier for linux [online]. Available from World Wide Web: `http://l7-filter.sourceforge.net/`.

[l7fb] L7-filter pattern writing howto. Available from World Wide Web: `http://l7-filter.sourceforge.net/Pattern-HOWTO`.

[liba] Liberouter [online]. Available from World Wide Web: `http://www.liberouter.org/`.

[libb] A libpcap version which supports mmap mode on linux kernels 2.[46].x [online]. Available from World Wide Web: `http://public.lanl.gov/cpw/`.

[Luc] Paolo Lucente. pmacct [online]. Available from World Wide Web: `http://www.pmacct.net/`.

[Mac06] Alistair MacArthur. Fpgas — parallel perfection ?, 2006. Available from World Wide Web: `http://www.celoxica.com/techlib/FPGA.pdf`.

[Oeta] Tobias Oetiker. The multi router traffic grapher [online]. Available from World Wide Web: `http://oss.oetiker.ch/mrtg/`.

[Oetb] Tobias Oetiker. The round robin database [online]. Available from World Wide Web: `http://oss.oetiker.ch/rrdtool`.

[opr] Oprofile [online]. Available from World Wide Web: `http://oprofile.sourceforge.net/news/`.

[pth] Posix threads programming [online]. Available from World Wide Web: `http://www.llnl.gov/computing/tutorials/pthreads/`.

[Rho00] Alex Rhomberg. gettimeofday resolution, 2000. Available from World Wide Web: `http://lists.suse.com/archive/suse-axp/2000-Mar/0115.html`. Post on the *suse-axp* mailing-list.

[RSH] Karl Berry Richard Stallman and Kathryn Hargreaves. Regex - gnu regex library [online]. Available from World Wide Web: `http://directory.fsf.org/regex.html`.

[Sena] *NodalCore C Series Programming Guide*. Retrieved on September 2006 from `https://support.sensorynetworks.com`.

[Senb] *NodalCore Regular Pattern Language*. Retrieved on September 2006 from `https://support.sensorynetworks.com`.

[sfl] sflow forum [online]. Available from World Wide Web: `http://www.sflow.org/`.

[sno] Snort [online]. Available from World Wide Web: `http://www.snort.org/`.

[Tan01] A.S. Tannenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[tor] Tor: anonymity online [online]. Available from World Wide Web: `http://tor.eff.org/`.

[Tur] Aaron Turner. Pcap editing & replay for *nix. Available from World Wide Web: `http://tcpreplay.synfin.net/trac/`.

[vB02] Iljitsch van Beijnum. *Building Reliable Networks with the Border Gateway Protocol*. O'Reilly, 2002.

[was] Waste, anonymous, secure, encrypted sharing. Available from World Wide Web: `http://waste.sourceforge.net/`.

[Wik06a] Wikipedia. Bittorrent — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=BitTorrent&oldid=90534592`. [Online; accessed 28-November-2006].

[Wik06b] Wikipedia. Bittorrent protocol encryption — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=BitTorrent_protocol_encryption&oldid=88715950`. [Online; accessed 22-November-2006].

[Wik06c] Wikipedia. Deadlock — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Deadlock&oldid=91955439`. [Online; accessed 5-December-2006].

[Wik06d] Wikipedia. Edonkey network — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=EDonkey_network&oldid=90438894`. [Online; accessed 3-December-2006].

[Wik06e] Wikipedia. Field-programmable gate array — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=90727367`. [Online; accessed 1-December-2006].

[Wik06f] Wikipedia. Gnutella — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Gnutella&oldid=88934837`. [Online; accessed 22-November-2006].

[Wik06g] Wikipedia. Kademlia — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Kademlia&oldid=89264091`. [Online; accessed 22-November-2006].

[Wik06h] Wikipedia. Napster — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Napster&oldid=91403223`. [Online; accessed 3-December-2006].

[Wik06i] Wikipedia. The pirate bay — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=The_Pirate_Bay&oldid=87039577`. [Online; accessed 22-November-2006].

[Wik06j] Wikipedia. Regular expression — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=88346331`. [Online; accessed 21-November-2006].

[Wik06k] Wikipedia. Simple network management protocol — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Simple_Network_Management_Protocol&oldid=79589022`. [Online; accessed 5-October-2006].

[Wik06l] Wikipedia. Unix time — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Unix_time&oldid=86516247`. [Online; accessed 23-November-2006].

[Wik06m] Wikipedia. Verilog — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Verilog&oldid=89642671`. [Online; accessed 1-December-2006].

[Wik06n] Wikipedia. Vhdl — wikipedia, the free encyclopedia, 2006. Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=VHDL&oldid=89397552`. [Online; accessed 1-December-2006].

[YCD$^+$] F. Yu, Z. Chen, Y. Diao, TV Lakshman, and R.H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection.

Index

List of Figures

List of Tables

Part IV

Appendices

Appendix

A

NodalCore C-2000 Card

A.1 Installation

This chapter describes the installation procedure of a security accelerator card from *Sensory Networks*, the NodalCore C-2000 Ultra. This was done under Gentoo Linux – Eneo's official Linux distribution – even though the drivers were only distributed for RedHat 9 and SUSE Linux Enterprise Server 9. This procedure is inspired by the one documented in the "NodalCore® C-Series Platform Manual".

Two different elements are required to be able to use the card, the kernel drivers and the user space tools. We have used the version 3.3.1.6 of both.

Procedure

The following archives were downloaded from Sensory Networks's support website at `https://secure.sensorynetworks.com/support/`.

```
# ls -l *.tar.gz
-rw-r--r--  1 root root 1160436 Sep 22 11:17 NodalCore
    -3.3.1.6-Redhat9.tar.gz
-rw-r--r--  1 root root 1566222 Sep 22 11:17
    NodalCoreSDK-3.3.1.6-Redhat9.tar.gz
-rw-r--r--  1 root root  119571 Sep 22 11:18 ncore-
    packet-scan-0.2.0.3.tar.gz
```

Kernel Driver

The kernel driver needs to be compiled with the source code of the running kernel.

```
# cd NodalCore-3.3.1.6-Redhat9
# rpm2targz ncore-driver-source-3.3.1.6-Redhat9.noarch.
    rpm
# tar xzf ncore-driver-source-3.3.1.6-Redhat9.noarch.tar
    .gz
# cd usr/src/ncore/
# make KERNEL_SOURCE=/usr/src/linux-2.6.15-gentoo-r1
# make install
```

Once the compiled driver is installed it can be loaded in the kernel and detects if supported cards are installed.

```
# lspci | grep Sensory
01:00.0 Class ffff: Sensory Networks Inc. NodalCore C
    -2000 Content
Classification Accelerator
# modprobe ncore
# dmesg | grep ncore
ncore: ncore0: NodalCore(tm) C-2000.
ncore: 1 NodalCore(tm) C-Series device found.
```

User Space Tools – SDK

```
# cd NodalCoreSDK-3.3.1.6-Redhat9
# rpm2targz ncore-3.3.1.6-Redhat9.i386.rpm
# rpm2targz ncore-devel-3.3.1.6-Redhat9.i386.rpm
# tar xzf ncore-3.3.1.6-Redhat9.i386.tar.gz -C /
# tar xzf ncore-devel-3.3.1.6-Redhat9.i386.tar.gz -C /
```

Because the tools binaries were compiled under RedHat we need to create different symlinks to OpenSSL libraries to get it works under Gentoo.

```
# ln -s /usr/lib/libssl.so.0 /usr/lib/libssl.so.4
# ln -s /usr/lib/libcrypto.so.0 /usr/lib/libcrypto.so.4
```

The required device specials files must be created in `/dev`, one for each card in the machine.

```
# /bin/mknod -m 0666 /dev/ncore0 c 63 0
# /bin/mknod -m 0666 /dev/ncore1 c 63 1
```

A firmware is required for the FPGA on the card, this firmware is called a bitstream and depends on the model and version of the card.

```
# VER=C2000-Ultra.003-pr12-31-3.0.14.0-4
# rpm2targz ncore-bitstreams-$VER.noarch.rpm
# tar xzf ncore-bitstreams-$VER.noarch.tar.gz -C /
```

Loading the driver and the bitstream.

```
# /etc/init.d/ncore start
Starting NodalCore(tm): Loading driver, bitstream (may
    take a few minutes).
# dmesg | grep ncore0
ncore0: Firmware bitstream configuration completed
```

Tests

The tools `ncore-hw-info` and `ncore-diagnostics` are available to check if the card is well configured and working as expected.

```
# ncore-diagnostics
NodalCore C-Series Diagnostic Tool (ncore-diagnostics)
    version 3.3.1.6
Copyright Sensory Networks Inc, 2005. All rights
    reserved.
PCI bus check...                                   [PASSED]
Checking card 0...                                 [PASSED]
Checking driver module present...                  [PASSED]
Module already loaded.
Checking driver version...                         [PASSED]
Bitstream loading check on card 0...               [PASSED]
Bitstream loaded in 0.383 seconds.
C-2000 device information:
Device Name:             C-2000 Ultra
Serial Number:      XXXXXXXXXXXXXXXX
Bitstream Series:                 31
Bitstream Version:            3.0.14
FPGA:                          VLX60
FPGA speed code:                   1
Bank 1 size:                   576Mb
Bank 1 speed code:                 1
Bank 2 size:                   576Mb
Bank 2 speed code:                 1
Card temperature:                 38
FPGA temperature:                 45
RLDRAM memory check on card 0...                   [PASSED]
Pattern matching test on card 0...                 [PASSED]
# ncore-hw-info
```

```
Product Family:                          C-2000
Model Name:                        C-2000 Ultra
Part Number:                        SNI-C2U-00G
Part Revision:                              003
Serial Number:                 XXXXXXXXXXXXXXXX
Bitstream Series:                            31
Bitstream Version:                       3.0.14
Input Channels:                              12
FPGA:                                     VLX60
FPGA speed:                                   1
Pattern Memory:                         1152Mb
Pattern Memory speed:                         1
Bank 1 size:                             576Mb
Bank 1 speed:                                 1
Bank 2 size:                             576Mb
Bank 2 speed:                                 1
```

Software emulation

A software emulation of the NodalCore hardware is available in the SDK, it can be used to simplify development by not requiring the hardware card on the development stations. This feature is fully described in the chapter "Software Pattern Matching" of [Sena].

To use this functionality you have to give a special device number to the `ncInit` function. This special number is given by the `NC_SOFTWARE_DEVICE(num)` macro. The following example opens an emulated NodalCore card.

```c
int main(int argc, char *argv[]) {
  nc_dev_t device;

  ncInit(NC_SOFTWARE_DEVICE(0), &device);
   ...
}
```

A.2 NCore ip_queue benchmark tool

Usage

Ncore-ipq is a users pace benchmark tool using the **ip_queue** facility provided by Netfilter to access packet payload in user space.

Installation

```
# ./configure && make && make install
# modprobe ip_queue
# iptables -A FORWARD -j QUEUE
#
# ./src/ncore-ipq --nc-pattern-db /tmp/toto.db
NodalCore initialised.
```

A.3 NCore Packet Scanner

Usage

ncore-packet-scan is a high performance packet scanning kernel module using the hardware acceleration provided by the NodalCore C-Serie cards.

Installation

```
# tar xzf ncore-packet-scan-0.2.0.3.tar.gz
# cd ncore-packet-scan-0.2.0.3
# NCSRC=/usr/src/ncore/ncore-driver-3.3.1.6/src
# KSRC=/usr/src/linux-2.6.15-gentoo-r1
# ./configure
  --with-ncsource=\$NCSRC
  --with-kernel-source=\$KSRC
[... configuration messages ...]
# make
[compilation messages]
# make install
# depmod -a
# modprobe packet_scan
```

Appendix

B

NodalCore API

This chapter will briefly describe the NodalCore API features which have been used during this project. The whole API is fully explained in [Sena]. In fact, as shown on figure B.1 on the following page, two different NodalCore API are available, one in **User Mode** and the other one in **Kernel Mode**. The only one used during this project was the user mode one because **pmacctd** is a user space application.

B.1 Overview

The NodalCore C-Serie cards (see 14 on page 60) can be used for multiple applications such as pattern matching — the feature used in this project — content decoding, data decompression and data digest generation. To handle those different needs, a **channel** specific to the feature needed has to be created. Once created it can be multiplied in the hardware using **channel pools** to take advantage of the parallelism of the hardware.

Our project is only only the pattern matching feature, we will focus on it. Two kind of pattern matching channels are available : *Pattern Matching Engine* (PME) for regular expressions in the NRPL format (see section 8.5 on page 34) and *Massive Memory Architecture* (MMA) for simple byte-strings. The only pattern matching channel used in this project is the PME because the patterns taken from the *Layer-7 Filter Project* (see section 7.3 on page 27) are only regular expressions.

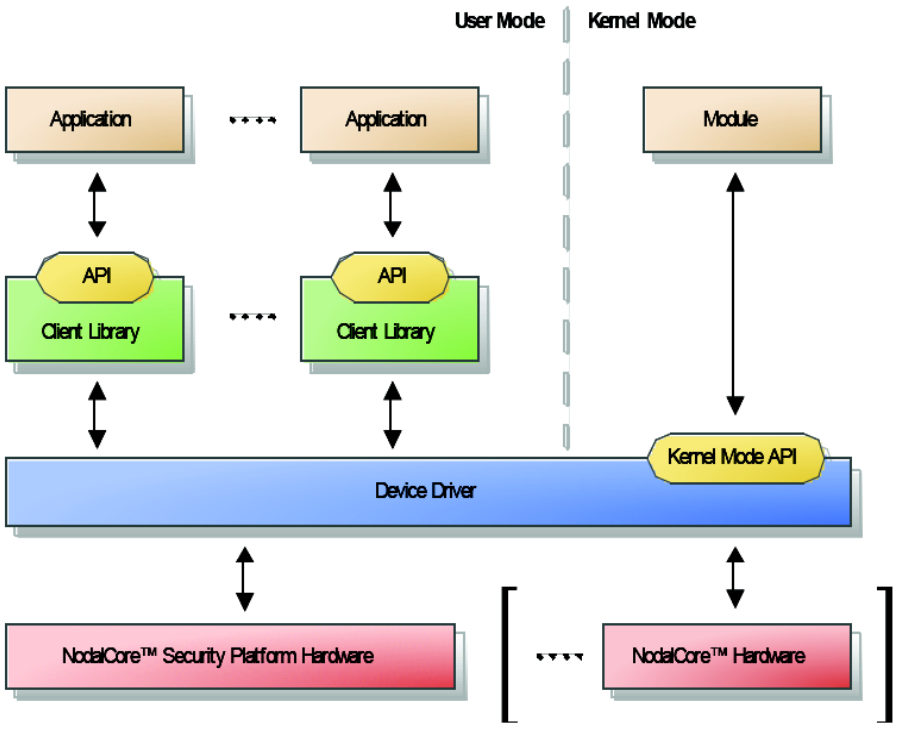


Figure B.1: NodalCore System Architecture, Source : [Sena]

B.2 API Wrapper

A wrapper (made of `nodal_core.c`) was taken from `ncore-ipq` (see section A.2 on page 98) to ease the API usage. The headers (taken from `nodal_core.h` of this wrapper define the following data structures and functions.

Data Structures

Only one data structure is defined in the wrapper, `NcoreContext` stores all informations about the current usage of the hardware card in the application as well as some statistics.

```
struct NcoreContext
{
    nc_dev_t device;

    nc_transform_t pmeTransform;
    nc_transform_t captureTransform;
    nc_channel_pool_t pool;

    nc_active_db_t activeDb;

    nc_stream_t* streams;
    size_t lastUsedStreamIndex;
    size_t streamsLen;

    pthread_t eventProcessor;

    pthread_t* matchers;
    size_t matchersLen;

    u_int64_t bytesProcessed;
    u_int64_t bytesScanned;
    u_int64_t packetsScanned;
};
```

API Initialization

The function `ncoreInit()` takes care of the initialization of the NodalCore API, the loading of the pattern database to the hardware and the launch of the event processor which will call a callback function each time an event is raised by the hardware. The function `ncoreDeinit()` removes the pattern database loaded from the hardware and frees allocated memory.

```
void ncoreInit(int deviceNumber, const char* dbFileName, int
    cacheEvents, struct NcoreContext* data);
void ncoreDeinit(struct NcoreContext* data);
```

Streams Intialisation

The function `ncoreStreamsInit()` takes care of the pattern matching channel pool creation and allocates the number of streams requested which is how data are transmitted into the channels. Cleaning after usage is handled by the function `ncoreStreamsDeinit()` which frees the allocated channel pool and the allocated memory.

```
void ncoreStreamsInit(size_t howMany, struct NcoreContext* data);
void ncoreStreamsDeinit(struct NcoreContext* data);
```

Feeding the Hardware

The main way to send data to the hardware is the `ncoreStreamWrite()` function which writes a data block to the specified stream on the hardware.

Match informations will generate a call to the callback function defined in `nodal_core.c` (`ncoreEventCallback()`) only after `ncoreStreamGetState()` has been called.

The other functions are convenient wrappers to group data write and state request using a single function call as well as requesting a blocking operation. Blocking operations were used in the first part of the `pmacctd` classifier module implementation described in section 15 on page 63).

```c
void ncoreStreamWrite(const unsigned char *data, size_t len, size_t
    streamIndex, struct NcoreContext* context);
void ncoreStreamGetState(size_t streamIndex, struct NcoreContext*
    context);

void ncoreStreamWriteWithState(const unsigned char *data, size_t len,
    size_t streamIndex, struct NcoreContext* context);

void ncoreStreamWriteWithStateBlocking(const unsigned char *data,
    size_t len, size_t streamIndex, struct NcoreContext* context);
void ncoreStreamGetStateBlocking(size_t streamIndex, struct NcoreContext
    * context);
```

Appendix

C Contributions

This appendix lists the different contributions, like bug reports or patches, made to free software projects during this project.

C.1 Libcprops

`libcprops` is a C prototyping library providing data structures like linked-lists, hashtables and vectors as well as a thread pool implementation. This library has been used in the first stages of the development of the multi-threaded version of `pmacctd` (see chapter 16 on page 66). At the end it was replaced by a specifically developed thread pool implementation.

The following bug report has been filed in `libcprops` bugtracker on *Sourceforge.net* including a patch to fix it. The patch has been accepted for next release by the author.

Submission date	2006-10-20
Description	Compilation fix under Debian sarge and MacOS X Tiger.
URL	`http://sourceforge.net/tracker/?func=detail&atid=797946&aid=1581201&group_id=155979`
Status	Accepted for next release

C.2 `libpcap` package in Ubuntu

The `libpcap` package version `0.8_0.9.4-1` available in Ubuntu Dapper Drake has a bug on some Linux systems which render the capture stats wrongs. The number of received packets is counted twice.

Here is an example of the bug while using `tcpdump`, a packet sniffer based on `libpcap`.

```
root@ubuntu:/tmp# tcpdump -n -i eth0 -w test.dump
tcpdump: listening on eth0, link-type EN10MB (Ethernet),
    capture size 96
bytes
45 packets captured
90 packets received by filter
0 packets dropped by kernel
root@ubuntu:/tmp# tcpdump -r test.dump | wc -l
reading from file test.dump, link-type EN10MB (Ethernet)
45
root@ubuntu:/tmp#
```

Submission date	2006-11-21
Description	Packet statistics from libpcap are wrong, number of packets doubled. The bug has already been fixed upstream in libpcap version 0.9.5.
URL	`https://bugz.launchpad.net/distros/ubuntu/+source/libpcap/+bug/72752`
Status	Unconfirmed

Workaround

The problem can be solved by installing the package version 0.9.5-1 coming for Ubuntu Feisty (next official version).

```
root@ubuntu:/usr/src/libpcap/libpcap0.8-0.9.5# apt-get
    install builddep libpcap0.8
[...]
root@ubuntu:/usr/src/libpcap/libpcap0.8-0.9.5# dpkg-
    buildpackage
[...]
dpkg-deb: building package 'libpcap0.8-dev' in '../
    libpcap0.8-dev_0.9.5-1_i386.deb'.
dpkg-deb: building package 'libpcap0.8' in '../libpcap0
    .8_0.9.5-1_i386.deb'.
```

```
 dpkg-genchanges
dpkg-genchanges: including full source code in upload
dpkg-buildpackage: full upload (original source is
    included)
root@ubuntu:/usr/src/libpcap/libpcap0.8-0.9.5# dpkg -i
    ../libpcap0.8-dev_0.9.5-1_i386.deb
root@ubuntu:/usr/src/libpcap/libpcap0.8-0.9.5# dpkg -i
    ../libpcap0.8_0.9.5-1_i386.deb
```

C.3 Pmacct

All the code developed during the implementation of multi-threading in `pmacctd` core process, described in chapter 16 on page 66, has been integrated in the CVS version of pmacct and will be included in the official version 0.11.3 which will be available on the official website of the project (see [Luc]).

Appendix

D

`gettimeofday()` System Call

The `gettimeofday()` system call gives the current system time with a maximum resolution of a microsecond because it returns a structure with two elements, the number of seconds since the Epoch[1] and the associated microseconds.

This system call has been widely used as a way to determine the time difference between two execution points in a program.

D.1 Resolution

The profiling method presented in section 12.3 on page 53 depends on the resolution of the `gettimeofday()` system call under Linux on our test hardware. The following program taken from [Rho00] gives us this value.

The program was run multiple time with different load of the test computer and has always been reporting a resolution of $1\mu s$. So we can conclude that our profiling system has a microsecond resolution.

D.2 Program

This program will repeatedly run `gettimeofday()` during one second and determine how many times it could. The best case is when it can be run 10^6 times in a second giving thus the best resolution possible of $1ns$.

[1]January 1st 1970 at midnight UTC, use as a time base in Unix as well as the Java Programming Language. See [Wik06l]

```
#include <time.h> // for gettimeofday
#include <sys/time.h>
#include <stdio.h>

int main(int argc, char**argv)
{
  struct timeval tv1, tv2;

  gettimeofday(&tv1, NULL);
  do {
    gettimeofday(&tv2, NULL);
  } while (tv1.tv_usec == tv2.tv_usec);

  printf("Difference: %d us\n", tv2.tv_usec - tv1.tv_usec +
         1000000 * (tv2.tv_sec - tv1.tv_sec));

  return 0;
}
```

Appendix

E

Benchmarking Programs

This appendix presents interesting parts of the programs developed during this project which were used to get the benchmarking results described in 17 on page 70. The goal of those explanations is to give a clear overview of how the benchmarks have been carried on because many factors can influence the results.

E.1 Traffic Capture File

The following program `sizedistribution.py` is used to generate the packet size distribution of a given pcap capture file. To do so, it use the `pcapy` libpcap python module. The packet sizes have been analysed between 0 and 1500 bytes (lines 6-7)

```
1  #!/usr/bin/env python
2
3  import sys
4  import pcapy
5
6  MIN = 0
7  MAX = 1500
8
9  """Determine the packet size distribution from a pcap
       capture file"""
10
11 filename = sys.argv[1]
12
13 pcap = pcapy.open_offline(filename)
14
15 # Initialise distribution list
```

```
distribution = []
for i in range(MIN, MAX+1):
  distribution.append(0)

while 1:
  try:
    (header, data) = pcap.next()
  except:
    break
  try:
    distribution[header.getlen()] += 1
  except:
    print header.getlen()

pkts = sum(distribution)

cum = 0
for i in range(MIN, MAX+1):
  size = i - 14
  if size > 0:
    cum += float(distribution[i])/float(pkts)
    print size, "\t", cum
```

E.2 Libpcap

The following program `pcapcount.c` is based on the code from a libpcap tutorial made by Martin Casado. It will run until receiving a `SIGTERM` signal (usually sent by the `kill` command and ctrl-C), and then print the number of packet it has received.

The callback function (defined between lines 27-29) does only one thing : the incrementation of a global variable. That is why almost no packets were lost during our benchmarks (see 17.2 on page 74) because packet loss appears when this callback function returns after a packet has become ready in the BPF[1] kernel buffer used by libpcap.

The configuration of a signal handler (line 49) with the `signal` allows a specific function (`quit` defined on line 31 to be called when the process is `kill`ed or that a CTRL-C is sent by the user.

```
1 /***************************************************************
2 * file :    pcapcount.c
3 * date:     2001-Mar-14 12:14:19 AM
4 * Author: Martin Casado
5 * Last Modified: Mon Nov 20 13:20:43 CET 2006
```

[1]Berkeley Packet Filter, a framework in the kernel able to capture packets on existing network interfaces

```
* Modified by Francois Deppierraz on Tue Nov 14 10:21:15 CET 2006
*
* Description: Display the number of packets captured on SIGTERM
*
*************************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <signal.h>

int count = 0;

/* callback function that is passed to pcap_loop(..) and called each time
 * a packet is recieved */

void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr,const
    u_char* packet) {
    count++;
}

void quit(int signum)
{
  printf("%d\n", count);
  exit(0);
}

int main(int argc,char **argv)
{
    int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    const u_char *packet;
    struct pcap_pkthdr hdr;      /* pcap.h */
    struct ether_header *eptr;   /* net/ethernet.h */
    bpf_u_int32 localnet, netmask;

    /* Init signals */
    signal(SIGTERM, quit);

    /* open device for reading */
    descr = pcap_open_live(argv[1],BUFSIZ,0,-1,errbuf);
    if(descr == NULL)
    { printf("pcap_open_live(): %s\n",errbuf); exit(1); }

    pcap_loop(descr,0,my_callback,NULL);

    fprintf (stdout,"\nDone processing packets... wheew!\n");
    return 0;
}
```

E.3 NodalCore API and Hardware

The following benchmarking program uses the NodalCore API to write 10 MB of dummy data (only 1's) to the NodalCore C-2000 card using different numbers of streams in parallel and different write sizes. The goal is to be able to have real numbers for the performance penalty generated by the use a small sized writes operations described in section 14.5 on page 62.

```
#include <stdio.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#include <stdlib.h>

#include "nodal_core.h"

//#define READ_SIZE 1073741824 /* 1 GB */
//#define READ_SIZE 104857600 /* 100 MB */

#define MAX_STREAMS 12
#define READ_SIZE 10485760 /* 10 MB */
#define MEAN_NUM 1

int verbose = 0;
int state_events = 0;
int writes = 0;

struct NcoreContext ncoreContext;

long double doit(int nc_streams, int block_size)
{
  char buf[block_size];
  unsigned int read_size = 0;
  int idx = 0;
  long double delta;
  struct timeval t0, t1;

  memset(buf, 1, block_size);

  gettimeofday(&t0, NULL);

  if (verbose) printf("Vamos\n");

  while (read_size < READ_SIZE) {
    ncoreStreamWriteWithState(&buf, block_size, idx, &ncoreContext);
    writes++;
    read_size += block_size;

    if (verbose) {
      if ((read_size/block_size) % ((READ_SIZE/block_size)/50) == 0) {
        printf("X");
        fflush(stdout);
      }
    }

    idx = (idx + 1) % nc_streams;
  }

  /* Wait for the last event */
  while (writes > state_events);

  if (verbose) printf("\nTerminado\n");

  gettimeofday(&t1, NULL);
  delta = (t1.tv_sec - t0.tv_sec) * 1000000 + (t1.tv_usec - t0.tv_usec);

  state_events = 0;
  writes = 0;

  return delta;
}

int main(int argc, char **argv)
{
  long double duration, mbps;
  int streams, block_size;
  int i;

  /* Use realtime priority */
  set_realtime_priority();

  /* NodalCore init */
  ncoreInit(0, "/root/toto.db", 0, &ncoreContext);
  ncoreStreamsInit(NC_STREAMS, &ncoreContext);

  /* TESTS */

  // Header
  printf("# Block size");

  for (streams = 1; streams < NC_STREAMS+1; streams++)
    printf("\t%d", streams);
  printf("\n");

  for (block_size = 64; block_size < 2000; block_size += 1) {
    printf("%d", block_size);
      for (streams = 1; streams < MAX_STREAMS+1; streams++) {
      mbps = 0;
      for (i = 0; i < MEAN_NUM; i++) {
        duration = doit(streams, block_size);
        mbps += ((long double) READ_SIZE * 8)/(duration);
      }
      mbps /= MEAN_NUM;
      printf("\t%Lf", mbps);
      fflush(stdout);
    }
    printf("\n");
  }

  /* Cleaning */
  ncoreStreamsDeinit(&ncoreContext);

  exit(0);
}


void handleMatchEvent(int idx, int action)
{
  /* Do nothing */
}

void handleStateEvent(int idx)
{
  state_events++;
}
```

E.4 Pmacctd

Here is a stripped-down version of the `generate-stats.py` which was used to compare the percentage of dropped packets of the following programs in function of the packet rate sent by `tcpreplay`.

- `pmacctd` original version 0.11.0
- `pmacctd` multi-threaded with and without NodalCore acceleration
- Libpcap using the program described in section E.2 on page 110
- The `ncore-ipq` program provided by *Sensory Networks*

```
#!/usr/bin/env python2.4

import paramiko
import threading
import sys
import time
import re
import traceback

NB_LOOPS = 1
NB_PACKETS = 947061 * NB_LOOPS
CAPTURE_FILE = '/var/log/francois/real-traffic-saitis.
    rewrited.dump'

class Host:
  env = {}

  def __init__(self, hostname, username, password):
    self.hostname = hostname
    self.username = username
    self.password = password

  def add_env(self, key, value):
    self.env[key] = value

  def get_env(self):
    res = ''
    for key, value in self.env.items():
      res += '%s=%s ' % (key, value)
    return res

GENERATOR = Host('192.168.100.211', 'root', 'XXX')
RECEIVER  = Host('192.168.100.250', 'root', 'XXX')

def run_command(host, cmd):
  t = paramiko.Transport(host.hostname)
  t.connect(username=host.username, password=host.
      password)
  chan = t.open_session()
  chan.exec_command(host.get_env() + cmd)

  status = chan.recv_exit_status()

  if status != 0 and status != -1:
    print "Error: command '%s' failed with status %d" % (cmd, status)
    error = chan.makefile_stderr('r+').read()
    print error
    sys.exit(1)

  res = chan.makefile('r+').read()
  chan.close()
  t.close()
  return res

class command_runner(threading.Thread):
  def __init__(self, host, cmd):
    self.host = host
    self.cmd = cmd
    threading.Thread.__init__(self)

  def run(self):
    run_command(self.host, self.cmd)

def run_background(host, cmd):
  t = command_runner(host, cmd)
  t.start()

def tcpreplay(pps):
  res = run_command(GENERATOR, '/var/log/francois/tcpreplay -d 1 -i eth1 -p %d -l %d %s 2>&1' % (pps, NB_LOOPS, CAPTURE_FILE))

  try:
    real_pps = re.findall(', ([\d\.]+) pps', res)[0]
    (pkts, bytes, seconds) = re.findall('Actual: (\d+) packets \((\d+) bytes\) sent in ([\d\.]+) seconds', res)[0]
    (bps, mbps, pps) = re.findall('Rated: ([\d\.]+) bps, ([\d\.]+) Mbps/sec, ([\d\.]+) pps', res)[0]

    return (pkts, bytes, seconds, bps, mbps, pps)
  except:
    print "Something failed with tcpreplay:"
    print res
    traceback.print_exc()

def run_pmacctd_mt(pps, threads=12):
  run_command(RECEIVER, 'perl -pi -e "s/^flow_handling_threads: (\d+)$/flow_handling_threads: %d/" /root/pmacct-ncore/pmacct/pmacct-0.11.0-mt/examples/test-live.conf' % threads)

```

```
    run_background(RECEIVER, 'ulimit -c unlimited; /root/pmacct-ncore/pmacct/pmacct-0.11.0-mt/bin/pmacctd-mt -f /root/pmacct-ncore/pmacct/pmacct-0.11.0-mt/examples/test-live.conf > /dev/null')

    time.sleep(20)
    real_pps = tcpreplay(pps)[5]
    time.sleep(5)

    res = run_command(RECEIVER, 'kill -USR1 $(cat /tmp/pmacctd.pid) && sleep 1 && egrep "packets received by filter|dropped by kernel" /var/log/syslog | tail -2')

    received = re.findall('(\d+) packets received by filter', res)[0]
    dropped = re.findall('(\d+) packets dropped by kernel', res)[0]

    run_command(RECEIVER, 'kill $(cat /tmp/pmacctd.pid)')
    run_command(RECEIVER, '; while :; do if [ -f /tmp/pmacctd.pid ]; then echo "Waiting"; sleep 1; else exit 0; fi; done')

    return (real_pps, received, dropped)

def test_pmacctd_mt(threads, packet_rates):
    print "# pps",
    for thread in threads:
        print "\treal pps\t", thread,
    print

    for pps in packet_rates:
        print pps,
        for nb_threads in threads:
            while 1:
                res = run_pmacctd_mt(pps=pps, threads=nb_threads)

                # Test if tcpreplay was fast enough
                precision = float(res[0])/float(pps)
                if precision > 0.97 and precision < 1.03:
                    break;
                else:
                    sys.stderr.write("tcpreplay is lagging or going to fast, %s instead of %s\n" % (res[0], pps))
            print "\t", res[0], "\t", float(res[2])/NB_PACKETS,
            sys.stdout.flush()
    print
    sys.stdout.flush()

try:
  tests = ['pmacctd-mt']

  threads = [12, 24, 48, 96]
  packet_rates = range(10000, 50000+5000, 5000)

  if 'pmacctd-mt' in tests:
    print "# Pmacct multi-threaded without NodalCore"
    run_command(RECEIVER, 'rm -f /tmp/classifiers/ncore.
        so')
    test_pmacctd_mt(threads, packet_rates)
    print
    print

except KeyboardInterrupt:
  print "Killing everything, please wait !"
  run_command(GENERATOR, 'killall tcpreplay || true')
  run_command(RECEIVER, 'kill $(cat /tmp/pmacctd.pid) ||
       true')
  run_command(RECEIVER, '/root/pmacct-ncore/pmacct/
      pmacct-0.11.0-mt/clean-processes.sh || true')
  run_command(RECEIVER, 'killall -9 pcapcount || true')
  sys.exit(1)
except:
  traceback.print_exc()
  pass
```

Benchmark Duration

Here is some informations about the time needed for a benchmark run to be able to evaluate which precision we can get and how many different benchmarks could be done given certain time frame. When working on the same development computers as other persons, it is sometimes necessary to be able to give your coworkers an estimate of the time during which they won't be able to use the computers.

A test run of a single receiver program using one pass of the test traffic and with packet rate between 10'000 and 50'000 by increments of 5'000 takes about **10 minutes** to complete.

So with the following constraints :

- Packet rates are tested between 10'000 pps and 50'000 pps by increment of 5'000 pps

- Results are averaged over 5 runs;
- The following applications have to be benchmarked :
  - **pmacctd** single-threaded with three different configurations (no pattern matching, software pattern matching or NodalCore pattern matching);
  - **pmacctd-mt** multi-threaded with 5 numbers of threads and two different configurations (with or without NodalCore pattern matching);
  - **pcapcount** with a single configuration
  - **nprobe** with a single configuration

We need $5 * (3 + 5 * 2 + 1 + 1) = 75$ runs taking each 10 minutes, this is almost 13 hours of benchmarks.

Appendix

F Measures

This appendix presents the results of benchmarks runs commented in chapter 17 on page 70.

F.1 Pmacctd

The column “PPS” means packet per seconds and is the rate at which packet were generated. After being run the packet generator report the exact rate at which it sent the packet, this is the “Real PPS” column.

All the other values on the following tables are ratio of packet loss which was calculated like that :

$$ratio = 1 - \frac{n_{packet\_received}}{n_{packet\_sent}}$$

Those values have been generated using the program described in chapter E on page 109.

Table F.1: Pmacct multi-threaded without NodalCore

PPS	Real PPS	1 thread	Real PPS	12 threads	Real PPS	24 threads	Real PPS	48 threads	Real PPS	96 threads
10000	9999.77	0.000	9999.65	0.000	9999.82	0.000	9999.73	0.000	9999.70	0.000
15000	15150.57	0.008	15150.75	0.001	15150.86	0.001	15150.70	0.001	15150.90	0.001
20000	19999.27	0.037	19998.98	0.010	19999.11	0.011	19998.11	0.011	19998.70	0.011
25000	24998.99	0.075	24998.44	0.031	24999.06	0.029	24998.93	0.032	24999.05	0.030
30000	30302.12	0.128	30301.18	0.070	30300.71	0.067	30299.96	0.069	30300.98	0.068
35000	35712.82	0.195	35712.39	0.133	35711.63	0.131	35711.97	0.133	35711.71	0.136
40000	39997.66	0.254	39996.93	0.191	39997.00	0.185	39997.77	0.188	39997.86	0.189
45000	45449.62	0.308	45451.88	0.254	45451.10	0.270	45449.69	0.270	45448.50	0.268
50000	49992.71	0.392	49990.36	0.296	49991.41	0.280	49988.95	0.280	49993.44	0.280

Table F.2: Pmacct multi-threaded with NodalCore

PPS	Real PPS	1 thread	Real PPS	12 threads	Real PPS	24 threads	Real PPS	48 threads	Real PPS	96 threads
10000	9999.83	0.061	9999.67	0.000	9999.73	0.000	9999.70	0.000	9999.63	0.000
15000	15150.97	0.326	15150.87	0.069	15151.03	0.003	15150.86	0.002	15150.93	0.001
20000	19998.78	0.500	19998.68	0.022	19999.14	0.017	19999.33	0.055	19998.78	0.011
25000	24998.04	0.618	24999.29	0.113	24998.09	0.045	24998.32	0.037	24998.78	0.030
30000	30300.77	0.667	30299.96	0.178	30301.54	0.096	30302.01	0.085	30301.78	0.075
35000	35712.22	0.679	35713.04	0.182	35710.15	0.232	35711.36	0.214	35711.59	0.144
40000	39997.09	0.754	39996.36	0.309	39996.04	0.296	39997.20	0.344	39996.18	0.199
45000	45449.58	0.786	45449.84	0.384	45450.83	0.301	45448.54	0.281	45450.56	0.408
50000	49991.44	0.806	49992.27	0.459	49995.95	0.362	49995.01	0.408	49993.37	0.331

Table F.3: Pmacct single-threaded without NodalCore

PPS	Real PPS	Packet Loss
10000	9999.65	0.000
15000	15150.90	0.000
20000	19998.78	0.003
25000	24999.12	0.009
30000	30300.90	0.018
35000	35710.53	0.033
40000	39996.48	0.049
45000	45448.42	0.070
50000	49993.28	0.099

Table F.4: Pmacct single-threaded with NodalCore

PPS	Real PPS	Packet Loss
10000	9999.76	0.027
15000	15150.79	0.194
20000	19999.14	0.354
25000	24998.95	0.495
30000	30301.34	0.551
35000	35711.85	0.553
40000	39997.31	0.612
45000	45450.14	0.679
50000	49992.34	0.729

Table F.5: Pmacct with l7-filter with 113 patterns

PPS	Real PPS	Packet Loss
10000	9999.71	0.000
15000	15151.20	0.002
20000	19998.62	0.009
25000	24999.02	0.022
30000	30300.92	0.037
35000	35711.85	0.050
40000	39997.05	0.072
45000	45448.61	0.101
50000	49992.73	0.138

Table F.6: Libpcap

PPS	Real PPS	Packet Loss
10000	9999.72	0.0
15000	15151.12	0.0
20000	19999.01	0.0
25000	24998.33	0.0
30000	30299.94	0.000
35000	35709.92	0.0
40000	39995.84	0.0
45000	45450.04	0.000
50000	49990.23	0.0

Table F.7: nProbe

PPS	Real PPS	Packet Loss
10000	9999.61	0.0
15000	15150.67	0.0
20000	19998.78	0.0
25000	24997.98	0.0
30000	30301.30	0.0
35000	35710.16	0.0
40000	39997.31	0.0
45000	45450.78	0.0
50000	49992.43	0.0

Appendix

G

GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

6. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   d) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   e) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

f) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

7. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

8. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

9. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

10. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free

redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

11. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

12. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

13. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

14. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the

Appendix

H

Diary

This appendix is a diary of the work done during the whole project. The work as well as results done each day are presented in chronological order.

18.9.2006

- Meeting with Jaime -> choose one of the 2 projects
  - Pattern matching in hardware using Sensory Networks cards
  - P2P detection using only transport layer informations

19.9.2006

- Choose the first project to begin with (Hardware accelerated NetFlow probe)
- Read the Sensory Network SDK documentation
  - The C API is available under Linux both in user space and kernel space.
  - Analysis of the ClamAV accelerator which uses another (proprietary) daemon based on SAA

20.9.2006

- Internal Eneo svn project: `http://192.168.100.151/ncoreflow`
- Read the SAA (Security Appliance Architecture) documentation
- POSIX Threads `http://www.llnl.gov/computing/tutorials/pthreads/`

21.9.2006

- Async programming under Linux: `http://pdos.csail.mit.edu/6.824-2001/lecnotes/l3.txt`
- Liberouter traffic scanner: `http://www.liberouter.org/ids.php`
- Read ncore-ipq-0.1.0.0 source code
  ncore-ipq is a small benchmark tool used to test the performances of NodalCore cards used as pattern matching accelerator for a user-space application.
  It uses the ip_queue netfilter API to get packets from the kernel, copy them on a Sensory Network card for pattern matching, get the resulting events and reinject the packet in the kernel independently of the result of the pattern matching.

22.9.2006

- Sensory Networks NodalCore C-2000 Ultra card installation and documentation
- Everything is installed on Angel's PC (192.168.100.60) running under Gentoo
- Got ncore-ipq running and matching a simple pattern
- Found and read ncore-skeleton.c in the SDK

24.9.2006

- Shell and file verbatim environments in LaTeX

25.9.2006

- Pmacct classifier shared modules: not usable because protocol name is a static value for each shared library
- Read pmacct-0.11.0/src/classifier.c
- How to compile multiple patterns per group on the NodalCore card ?
  - Pattern pre-processing:
    - Grouping and actions: (regex1)!1|(regex2)!2|...|(regexi)!i
    - Fixes: []...] -> [\]...] and specials characters -> \x..
- ncore-compiler takes ages !
- Sent an email to Matthew Gream <matthew.gream@sensorynetworks.com>

26.9.2006

- Reply from Matthew -> Regex theory

  - star (*) and plus (+) expand the states space dramatically -> compiler performance goes down
  - Solution: Use the @ operator cf. NodalCore Regular Expressions Manual
  - By the way, it use useless to uses parenthesizes around each regex

- Meeting with Jaime and Pablo
  - Implementation of a prototype using pmacct and the NodalCore API without any parallelism (single thread, blocking matching)
- Beginning of the merge of pmacct and ncore-skeleton

27.9.2006

- Decided to use ncore-ipq code (nodal_core.c) as a basis instead of ncore-skeleton
- Working prototype (proto1) of pmacctd doing pattern matching using NodalCore hardware
- No working prototype (proto2) of a shared library ncore classifier for pmacct
- Installed a test development station using the software emulation feature of the NodalCore SDK on a Debian and got ncore-skeleton to work on it
- Read about FPGA: `http://en.wikipedia.org/wiki/FPGA`
- Jota showed me how to use hping as a traffic generator

28.9.2006

- Finished a working prototype of a shared library classifier for pmacct using NodalCore SDK
- Traffic generators:
  - hping3 using random data
  - netperf
  - tcpreplay using real traffic captured using tcpdump
- Created a 5 minutes traffic dump as a pcap file from Saitis Network's Internet access (626 MB in 319 seconds -> 15.7 Mbps average)

29.9.2006

- Traffic generators:
  - TCPopera paper: `http://www.cs.ucdavis.edu/~wu/ecs236/RAID2005_TCPopera.pdf`
- Testbed installation
  - One Dell SC1425 Server (3 GHz Xeon, 1 GB RAM) running Ubuntu Dapper

  - One ??? (3 GHz Xeon, 1 GB RAM) running Lince
  - Both connected in back-to-back using Gigabit Ethernet cards
- Prototype performances are _very_ poor in comparaison to software pattern matching (at 100 Mbps using real traffic):
  - Using software-only pattern matching: 0.2 % of packet loss
  - Using hardware-accelerated: 42 % of packet loss
  - Likely reasons of theses performances:
    - Blocking IO during the whole treatment by the card (MOST IMPORTANT factor, found after tests)
    - Too many little writes to the cards (one packet at a time)
    - Using only one stream without any parallelism

1.10.2006

- Administrative stuff with Heig-VD

2.10.2006

- Looked at the Ring Buffer implementation in ncore-ipq source code
- How to get the classifiers of pmacct asynchronous ?
  - How the flows are handled in pmacct -> Read ip_flows.c
- Test C program using the ring buffer implementation

3.10.2006

- Project description
- LaTeXformatting
- Read "pmacct: steps forward interface counters", a paper of the author of pmacct Paolo Lucente
  - Random thoughts: A libpcap-like packet capture interface which gives access to the result of the pattern matching done previously by the hardware. This means work in kernel-space.
  - "a content-based Pre-Tagging scheme" in section "VII Future works" looks interesting
- Read INTERNALS from pmacct sources
- Analysis of the classifier architecture
  - evalutate_classifiers is called only in ip_flow.c by find_flow() and create_flow() (and the same for IPv6)
  - Calltrace: pcap_cb() -> pptrs.l3_handler -> ip_handler() -> ip_flow_handler() -> find_flow() -> evalutate_classifiers()

4.10.2006

- Random thoughts: Matthew said that using the @ operator may cause wrong positives which have to be discarded using a software regex matching. But that it is usually statistically small so doesn't impact performances. This is true for IDS but I'm not so sure about general pattern matching which, ideally will return a match for each flows !
- Sent an email to Paolo Lucente <paolo@pmacct.net>
  - How to hook an async classifier in pmacct ?
  - What about content-based pre-tagging ?
- A bit of C pointers refresh :)

5.10.2006

- Discussion with Paolo
  - Don't use threads, because of the added complexity and error-prone handling of shared data structures due to concurrency :)
  - But the NodalCore API is multi-threaded...
  - I will dig a bit into containing the API calls in a separate process
- Streams and channel pools differences ?
  - Multiple streams can be opened on a channel pool and are distributed by the API to availables channels

6.10.2006

- Watched Herbert Bos's presentation at RIPE 53 about "Network Monitoring"
  - Talked about the Lobster project (`http://www.ist-lobster.org/`), "Large scale monitoring of Broadband Internet Infrastructures", an european pilot projet
  - DAG Hardware (`http://www.endace.com/`)
- How to implement classification as another process in pmacct ?
  - Packets buffering needed, probably in the core
- UNIX IPC methods

7.10.2006

- Efficient data structures in the C programming language
  - Arrays, Linked lists, Queues

9.10.2006

- Read pthread Tutorial at `http://www.ecet.vtc.edu/~pchapin/pthreadTutorial.pdf`
- Paolo's proposition of implementing threads in pmacct core
  - Plan A: Encapsulate almost the whole core in multiple threads from pcap_cb() to exec_plugins()
  - Plan B: Encapsulate ip_flow_handler() and evaluate_classifiers() in multiple threads (requires data copies)

10.10.2006

- New mail from Paolo
  - More modular approach using three different thread groups
    - The first with only one thread for packet capture, fragment handling and L[234] handling
    - A group of N threads handling the flow handler and the classifier
    - The last with one thread handling the plugins execution
- Next phase: using another capture library (streamline ?)
- Read Multi threaded Design at `http://www.latenighthacking.com/projects/multithreadedDesign.html`
- Read `http://en.wikipedia.org/` on multithreading, reentrancy, atomicity, etc.
- Test programs using POSIX threads
  - Concurrency bugs -> same player shoot again
  - A simple prototype using three different thread pools synchronized by a condition variable

11.10.2006

- Read about GNU Threads (GNU Pth)
- Discussion with Pablo:
  - Flow handler not in the same threads as hardware pattern matching because packets whose flow is already classified doesn't need to wait for the hardware classification threads to finish
- Continued prototype using three thread pools to get my hands on pthreads programming

16.10.2006

- Installed a subversion server with a repository specific for this project, Paolo Lucente has been given access to it

- Reflexion about a single-threaded concept using buffers
  - Refactoring of pmacctd needed because as for now the classifiers calls are nested in the flow handling code -> Solution quite complicated
- Use of a non-preemptible thread library such as GNU Pth can avoid many issues and fits our problem because we can easily avoid using blocking IO operations
- Tried to debug deadlock issues in t3.c test threaded program

17.10.2006

- Paolo agrees that using a library such as GNU Pth can avoid some issues, what about pthread API emulation in GNU Pth ?
  - The pthread API emulation seems to work fine under Linux but fails to compile under OSX
  - GNU Pth won't take advantage of multiprocessor (SMP) architectures
- Thread architecture
  - Not really the "thread pipeline" design pattern because we don't want to use queues between threads
- Pablo said that someone will begin to work with me on the project next week
- Valgrind usage and test program debug

18.10.2006

- Multi-threaded pool test program working
- Generalization of the code to be able to use it easily with multiple thread pools in other programs
- Automake stuff to compile tpmacctd separately of pmacctd

19.10.2006

- Reported a bug when using ./configure –enable-debug, results from the print plugins are completely wrong -> Paolo
- Learned about GNU Developpement tools such as autoconf, automake, etc at `http://autotoolset.sourceforge.net/tutorial.html`
- Implementation of my thread_pool module in tpmacctd
- Bugs:
  - There's still bugs in the flow handling code because tpmacctd and pmacctd gives a different number of flows for the same pcap file but the byte count is correct
  - Segfault sometimes when using more than 1 thread and miss packets
  - Using a sleep() to wait for the plugins processes looks a bit ugly

20.10.2006

- Had a look at the thread pool implementation of the cprops project at `http://cprops.sourceforge.net/`
- Implemented a test program much like the previous one but using cprops thread pool implementation instead of mine
- In current pmacctd design the packet payload is a buffer in the pcap_cb function stack which is discarded as soon as pcap_cb returns. To avoid that we have to allocate a new buffer which will be freed only after the data is sent to the plugins
- Using a correct data copy function (for pptrs), this doesn't crash anymore :)
- Tried tpmacctd using the previous ncore.so classifier module which has been coded as a prototype before:
  - Doesn't crash but no results: Missing data

21.10.2006

- Read about the communication channels between pmacctd and the plugins to understand why it's missing data
- Fixed it ! In fact the channels_list structure was not protected by a mutex

22.10.2006

- The function cp_pooled_thread_get_id() can be interesting to handle the thread private data space
- Response from Paolo:
  - Using a double linked list to check more rapidly for thread availability
  - sched_yield() is part of librt under Solaris

23.10.2006

- Profiling using gprof, how to use it with multi-threaded applications `http://sam.zoy.org/writings/programming/gprof.html`
- Performances of tpmacctd are equals independently of the number of threads -> There's a bug somewhere
  - Calls to find_flow are serialized by a mutex -> This is not granular enough !
- in ip_flow.c two global variables are used: ip_flow_table and flow_lru_list
  - ip_flow_table is hashtable which size is, by default, 16384000 (16 MB). Each record is 32 bytes long, so we use 512000 buckets. This is too much mutexes !

- Did some research about multi-threading safe hashtables
  - `http://www.cs.rug.nl/~wim/mechver/hashtable/`
  - cprops has a thread-safe hashtable implementation

27.10.2006

- Laptop crash -> mainboard failure, que bueno!
- Thought about the ip_flow_table and flow_lru_list locking in ip_flow.c
- Test scripts to evaluate the performance and reliability of tpmacctd

28.10.2006

- Idea: use a single mutex to serizalize everything except evalutate_classifiers(). Quick and dirty be can be interesting

30.10.2006

- Paolo, discussion about how to handle the locking of the ip_flow_table
  - I choose to use a very basic solution, the one of the 28th of october, to use a single lock to serialized everything except evaluate_classifiers().
- Automake stuff to get tpmacctd compile under older Linux (_XOPEN_SOURCE=600 and _GNU_SOURCE flags)
  - Doesn't work, maybe some files missing in the upstream, asked Paolo
- Some performance tests using real-traffic-saitis.dump with tpmacctd and the simple locking mechanism.
- Comparaison of the print plugin output between pmacctd and tpmacctd
  - Different because tpmacctd flushs flow faster due to memory constraints

31.10.2006

- Fixed the autoconf/automake stuff
- Profiling of tpmacctd
  - Using gprof
    - evaluate_classifiers is 98% slower in tpmacctd, but why ?
      - Can be due to the coarse locking in ip_flow_handler_worker(), but the fact that this waiting time is accounted for evaluate_classifiers is strange.
- Tried to integrated libcprops in the pmacct build system

1.11.2006

- Fixed some of the compilation issues when using libcprops included in pmacct

6.11.2006

- Compilation fixes
- NodalCore cards performances bedtest
  - Made a custom crossover 1000Base-T ethernet cable
  - Used tcpreplay to generate traffic using a pcap dump
- Read about libpcap performances limitations
- Profiling of tpmacctd, time used in evaluate_classifiers
  - Idea: Could be thread waiting for the lock before context switch ?

7.11.2006

- Testbed setup
- Performances benchmarks
  - At 10 kpps (about 50 Mbps), tpmacctd is behaving well at first sight
  - Packets mean size: 625 bytes
- Ncore kernel module error:
  - ncore0: Error: stream write: unable to copy from buffer at b0b7da4e (length 1460) for stream 53
  - Modified the kernel driver to the number of bytes which were not copied:
    - ncore0: Error: stream write: unable to copy from buffer at b0283a4e (length 1460, failed 8) for stream 4
- Why is the init function of ncore.c called multiple times ? In fact the init function is called once but there might be some logging bugs

8.11.2006

- Sent a mail to Matthew asking how to handle the buffer full case
- Configure flag –enable-threads -> quite a bit of debug
- I should try to use pthread_cond_timedwait

9.11.2006

- Used pthread_cond_timedwait to try to debug the deadlock problem
- Other classifiers modules can not be thread-safe -> Use a flag to express thread-safeness and serialize if not set ?

10.11.2006

- Trying to fix the bug
- Ethernet frame sizes: min 64 bytes and max 1518 bytes
- Packet size
  - Ethernet header: 14 bytes
  - Ethernet checksum: 4 bytes
  - IP Header: min 20 bytes (RFC 791)
  - TCP Header: min 20 bytes (RFC 793)
    - TCP Payload: max 1460 bytes
  - UDP Header: 8 bytes (RFC 768)
    - UDP Payload: max 1472 bytes
- NodalCore C-Serie raw performances
  - block size greater than about 1800 bytes has the same throuhput
- Libpcap mmap: `http://public.lanl.gov/cpw/`

11.11.2006

- Tried to strip down libcprops but too much complicated !
- Used my thread pool implementation in pmacct
  - After hours of debugging, it seems that it works
- Mail with Paolo: How to handle the conditional compilation of thread_pool.c ?
  - Quick and dirty: use a ifdef ENABLE_THREADS in thread_pool.c
  - Better: define a variable in configure, like the PLUGINS variables
  - Implemented the quick and dirty for now

13.11.2006

- How to handle the configuration of classes definitions and the mapping with action ids ?
- Benchmarks using a python script
  - Pmacct multi threaded without NodalCore classifier
  - Pmacct multi threaded with NodalCore classifier
  - Pmacct single threaded without NodalCore classifier
  - Pmacct single threaded with NodalCore classifier
- Packet size distribution in the pcap capture
- Linux pktgen: `http://linux-net.osdl.org/index.php/Pktgen`
- Run benchmarks during the night, but it failed in the middle :(

14.11.2006

- Libpcap benchmarks, code inspired from `http://www.cet.nau.edu/~mc8/Socket/Tutorials/`
  - Mostly uninteresting because the libpcap slowdowns comes mostly from what runs in the callback function
- Pmacct benchmarks
  - Multi threaded version is slower without using the NodalCore card but faster when using it
  - Made some automatic data collection to generate graphs

15.11.2006

- Idea: how to debug the NodalCore driver issue which might by caused by some memory corruption
  - Using a generated pcap dump with known payload which will be sent to a dummy classifier which will check if the data is correct
  - It will be easier to generate traffic using libdnet because libpcap data file format is quite complicated !
  - In fact, it's even simpler to use hping3
- 16 bytes memory corruption in packet payload
- Fixed the build system using a new variable in configure.in
- Should use a caplen of 1514 bytes to get full payload

16.11.2006

- Some NodalCore benchmarks
- Bugfixing in pmacctd
- The payload processing in classifier.c need to be disabled, 2 solutions
  - Use a configure flag
  - Remove it and migrate the regex matching in a classifier shared module

17.11.2006

- Performances of the multi-threaded are quite bad...
- Tried some nProbe benchmarks
- Maybe it is better

18.11.2006

- Using sched_setscheduler for the NodalCore Benchmark

19.11.2006

- Tried to run pmacctd-threads using GNU Pth pthread emulation API with the use of a environment variable
  - LD_PRELOAD=/tmp/pth-2.0.7/.libs/libpthread.so
  - Works with pmacctd, but not with NodalCore classifier because the NodalCore API uses blocking syscalls
- Discovered dmalloc library to debug memory problems, `http://dmalloc.com/`

20.11.2006

- Configure flag to use dmalloc
  - Many memory problems...
- Using realtime priority with ncore_bench1 trigger a kernel ncore driver bug: "BUG: soft lockup detected on CPU0!"
- Some schemas using inkscape and benchmarks documentation
- Implemented some timing debug informations in pmacctd

21.11.2006

- Have to read `http://perl.plover.com/Regex/article.html`
- Regex theory
- Performance debugging
  - Using sched_yield is just a quite bad idea ! Because, it cause the entire *process* to go back to the tail of the scheduler running queue
  - Used a condition which is triggered when a worker is freed
    - Quick and dirty implementation, just to see if works better

22.11.2006

- Cleaner reimplementation of the thread pool
- Some calculations
  - 50'000 (200 Mbps) packets per seconds means 20 us per packet
  - 20'000 (100 Mbps) packets per seconds means 50 us per packet
- Found the famous bug (when made the ncStreamWrite failed) in the ncore classifier !
- Writing

23.11.2006

- Profiling of pmacctd
  - ip_flow_table locking is the culprit
  - Require a rework of locking, quite difficult
  - Asked Paolo for advice

24.11.2006

- Writing
- Reworked the statistics generation script
  - Handle exceptions better
  - Redone the whole statistics with an everage of 5 measures each time
  - Expect 13 hours of work, hope that it won't crash !
- New tests with GNU Pth in comparaison with Pthread
  - Using GNU PTh Pthread emulation
    - real 1m35.244s
    - user 1m2.236s
    - sys 0m25.990s
  - Using native Pthread:
    - real 0m37.136s
    - user 0m5.604s
    - sys 0m6.684s

25.11.2006

- Benchmarks results, took 475 minutes (about 8 hours)
- Replaying the capture file multiple may not be a bright idea to avoid noise
  - Let's try to run the full tests 5 times and average everything
    - 98 minutes for each run

26.11.2006

- Report writing
- New complete benchmark run using an average over 5 runs for each results

27.11.2006

- Report writing

- Sent a mail the Matthew asking about the strange NodalCore benchmark results
- How does nProbe handle the packet capture using libpcap
  - For each captured packet, the headers a analyzed and the packet is added to a hash table
  - It can be interesting to get some ideas from nProbe's code

28.11.2006

- Report writing
- Response from Matthew
  - He gets better results, maybe the busy waiting is the culprit
  - Sent their test program which use some optimizations
- Sent the patch the Paolo for inclusion in the project

29.11.2006

- Run Sensory's test program and got similar results -> the culprit is hopefully not my test program
- Paolo will include my patch in release 0.11.3 due on 7th of December
- Writing

30.11.2006

- Rerun NodalCore benchmark without HyperThreading -> Cleaner results but still the same artifacts
- Writing

1.12.2006

- Writing
- Sent the report to Hervé Dedieu for comments
- Sent the report to Matthew for NDA-compliance check

3.12.2006

- Writing

4.12.2006

- Response from Matthew with a few minors changes to avoid problems with the NDA
- Only one public version of the report will be done, but the CD-ROM won't contain any NodalCore code
  - Need to remove NodalCore code and docs
  - Need to remove the NodalCore pmacct classifier
- Sent the report to Elio for comments

5.12.2006

- Writing FINISHED, let's proofread that !
- Cleaning of the files and directory structure to prepare the CD-ROM