Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Gerry Zaugg

A Lightweight Packet Capturer for High-Speed Links

Semester Thesis SA-2004.08
November 2003 to March 2004

Supervisors: Parminder Chhabra, Thomas Dübendorfer, Arno Wagner
Professor: Bernhard Plattner

Abstract

The Internet grows rapidly in both complexity and speed. With the advent of Gigabit Ethernet, traffic characterization becomes a challenging task as current capture tools (*tcpdump*, *ethereal*) are inefficient when working on high-speed interfaces.

In order to better understand traffic pattern and consistently identify malicious code, there is an urgent need for fast packet capturing techniques.

This Semester thesis designs, implements and evaluates a lightweight packet capturer that works on commodity hardware and is able to keep up with Gigabit links.

Contents

1 Introduction **4**
1.1 Overview 4
1.2 Task 4
1.3 Related Work 5

2 Design **6**
2.1 Tcpdump 6
2.1.1 Architecture 6
2.1.2 Problems 6
2.1.3 Performance 6
2.2 Improvements 7
2.3 Turbo Tcpdump 7
2.3.1 Architecture 7
2.3.2 Performance 7
2.4 MAGNeT 7
2.4.1 Architecture 7
2.5 Lightweight Packet Capturer (Lwpcap) 8
2.5.1 Modifications 8
2.5.2 Hooks 8

3 Implementation **12**
3.1 General Overview 12
3.2 Kernel Module 12
3.2.1 Initialization 12
3.2.2 At Runtime 13
3.2.3 Termination 14
3.3 User Space 14
3.3.1 Initialization 14
3.3.2 At Runtime 14
3.3.3 Termination 15

4 Performance **16**
4.1 Experimental Environment 16
4.2 Network Throughput 16
4.3 Results 17
4.3.1 Core Capacity 17
4.3.2 Counting 18
4.3.3 Writing to disk 19

5 Summary and Outlook **22**
5.1 Summary 22
5.2 Outlook 22
5.3 Acknowledgment 23

List of Figures

2.1 Lwpcap Architecture 9
2.2 Packet Reception Path and Lwpcap Hooks 9

3.1 Kernel At Runtime 13
3.2 User Application At Runtime 15

4.1 Experimental Environment 16

List of Tables

2.1 Comparison of the *Lwpcap* Alternatives 11

4.1 Packet Size and Throughput 17
4.2 Counting packets in the NIC driver 17
4.3 Counting packets under kernel 2.4.24 18
4.4 Counting packets under kernel 2.6.0 18
4.5 Writing packets under kernel 2.4.24 19
4.6 Writing packets under kernel 2.6.0 20

Chapter 1

Introduction

1.1 Overview

The world has changed. Country's frontiers smear and disappear, gates are broken down, the gap between rich and poor widens, religious conflicts are rising. The Internet is an allegory for these changes: It knows no frontiers, no obstacles, no limits but for all that it divides between propertied and unpropertied groups. Only a privileged class has access to the new media and as the technical progress proceeds, the poor countries will miss the train to wealth completely.
Many representatives of the new economy think that the Internet has the power to close or at least to reduce the gap between rich and poor. In the best of all possible worlds, this perhaps would be true but life makes me know that this won't happen. Or as the holy bible says: "Those who have, more shall be given, but from those who have not, even that which they have shall be taken away."
Globalization escalates, the world becomes smaller but it doesn't become fairer and for sure, no balance will take place. But the so-called third world won't give up, won't meet their faith. They will fight.
As the world faces increasing intercultural tensions, the Internet faces new threats of so far unknown dimensions: Viruses and worms spread rapidly in the Internet and can infect hundred thousand hosts in a matter of hours. The compromised machines are often used to attack specific target networks. These Distributed Denial of Service (DDoS) attacks have the potential to paralyze backbones and therefore to seriously damage Internet infrastructure.
The DDoSVax project at ETH is intended to detect and analyze DDos attacks. The researchers work out monitoring mechanisms, possible prevention strategies, fast countermeasures due to improved detection schemes and the ability to more concise statements about the origins and targets of an attack.
One of the key issues herein is the identification of traffic patterns to characterize and classify malicious code. As network speeds have been increasing at an incredible rate, doubling every 3-12 months [16], we deal today with Gigabit Ethernet and thus, interception, analysis and storage of data streams becomes a true challenge.
Well tried capture tools like *tcpdump* [2] and *ethereal* [3] (which both are built on top of *libpcap* [1]) that held up remarkably well for many years are showing their age. Today, network speed outstrips the ability of these programs. They are incapable to monitor traffic at Gigabit links and end up dropping thousands of packets. New approaches and strategies have to be developed and implemented to solve the problem.

1.2 Task

The goal of this project is to design and develop a program to capture the full header (and as much payload as possible) of every arriving packet on a Gigabit interface (i.e. considering IPv4 packets: Ethernet header (14 bytes), IP header (20 bytes) and TCP header (20 bytes) result in a minimal capture length of 54 bytes).
The collected data is archived for further analysis. That is, the data is written to disk, sent to another host, copied into memory-mapped files, parsed and evaluated in memory or whatever

the user judges as being useful.
The program is for passive measurements only and it should not add significant overhead to CPU utilization when the packet capture is turned on.

1.3 Related Work

The Monitor for Application-Generated Network Traffic (*MAGNeT*) [6] toolkit serves as a model for our design. *MAGNeT* was developed for different purposes, but the architecture can easily be applied to our problem. Basically, the model consists of a kernel module and a user-space application. The module grabs the packets and copies them into a circular buffer in kernel memory. The buffer is exported to user space via a special device. The user program accesses the same buffer and copies the slots into files.
The packets have to be caught immediately after their arrival at the Network Interface Card (NIC) to avoid traveling through the entire network stack. There are different hooks in the protocol stack code to redirect the packets:

- Luca Deri [9] describes a radical approach to fetch the packets already in the NIC driver. Whenever a packet is received on the NIC, the driver directly copies it into the circular buffer instead of passing it to the network stack.
- Kossak and Lifeline [11] explain how to register a user defined packet handler in the protocol handler list. The handler contains a pointer to a function that will be called when a packet of the specified type is received. This function copies the packets into the kernel buffer.
- Bioforge [10] sheds light on the functionality of the Netfilter framework and shows how to associate a function with a particular Netfilter hook. When a packet traverses the hook, the registered function is called and again the packet is put into the buffer.

My solution heavily relies on the *MAGNeT* architecture, on Luca Deri's implementation of his approach and on the three hooks described above. One of them will be installed to capture the packets and to write them into the circular buffer. A user application polls the buffer, reads out the data and writes it into a file.
The journey of a packet through the Linux 2.4 network stack is described in-depth in [12, 13, 14, 15]. These papers are invaluable when getting in touch with the protocol stack.

Chapter 2

Design

In this chapter, we inspect different models to approach the problem and point out their advantages and weaknesses. We shed light on the key issues and discuss important design decisions.

2.1 Tcpdump

2.1.1 Architecture

Tcpdump is a free and easy-to-use monitoring tool. It is designed as a user application relying upon functionality contained in the lower-level *libpcap* library. The core of *tcpdump* is a call to the *libpcap* routine `pcap_loop()` which performs a `recvfrom()` system call for each arriving packet to copy it from kernel to user space.
Libpcap exploits a special type of socket protocol family: `PF_PACKET`. This type differs from other domains in allowing an application to send and receive packets dealing directly with the network card driver, thus avoiding the usual protocol stack handling (e.g. IP/TCP or IP/UDP processing). That is, any packet sent through the socket will be directly passed to the Ethernet interface, and any packet received through the interface will be directly passed to the application. This protocol family avoids time consuming flowing through the network stack.
The `PF_PACKET` receive function first tries to restore the link layer header, checks whether a filter has been attached to the receiving socket and applies the filter if present. If the packet is not discarded, it is cloned and a receive function on the appropriate socket is called. There are no unnecessary sanity checks, no kernel data structures, no queues.

2.1.2 Problems

The `PF_PACKET` socket is fast but the problems start when the packets have to be passed from kernel to user space. *Libpcap* must ask the operating system to perform the required packet copy in the network stack. This operation always involves a receive system call (`recvfrom()`) which performs a context switch to kernel mode and copies memory from kernel to user level. This call-and-copy is repeated for each packet. At high network speeds, the context switching and copying overhead becomes a significant burden. Furthermore, every packet is written to disk separately. This stresses the writing head as it jumps all around the disk to write the data.

2.1.3 Performance

It is a well known fact [6, 7] that *libpcap*, *tcpdump* and *ethereal* are inefficient at reliably grabbing packets on Gigabit links. Tests [7] have shown that *tcpdump* can record traffic at speeds not greater than 250 Mbit/sec. My own measurements confirm these results: *tcpdump* loses around 10% of all the packets monitoring a 250 Mb/sec data stream transmitting 100 bytes packets and capturing 64 bytes per packet. The trend intensifies by increasing the packet throughput.

2.2 Improvements

The PF_PACKET socket grabs the packets early in the network stack, but perhaps an even earlier capture is possible.
Do not waste time for meaningless checks and eliminate unnecessary steps.
It would be clever to aggregate multiple packets and write them in one operation instead of copying and writing each packet separately.
Based on my observation and understanding of performance bottlenecks, the following modifications will be useful:

1. Fetch the packets as early as possible in the protocol stack
2. Reduce the number of context switches
3. Copy multiple packets to user space simultaneously
4. Accumulate several packets before writing them to disk

It is useful to separate kernel and user space activities so that the user program doesn't have to interact with the kernel (and vice versa) using system calls. However, we still have to define a synchronization between user and kernel processes as they have to hand over the packets.

2.3 Turbo Tcpdump

2.3.1 Architecture

Phil Wood [5] addressed some of the problems listed above and developed an improved *libpcap* version. He made use of the shared memory ring buffer implemented in the Linux kernel by Alexey Kuznetsov as a patch to the 2.2.x and a kernel configuration option in the 2.4.x kernel. Up to 32768 frames can be allocated in the kernel and mapped to user memory. The handling function of the PF_PACKET socket writes the packets into the ring buffer. *Libpcap* accesses the same circular buffer. Instead of invoking a system call, *libpcap* simply checks whether there is a new frame available. As long as this condition is fulfilled, it writes the packets to disk. There are no context switches and multiple packets are copied per check. But again, every packet is written to disk separately.

2.3.2 Performance

Turbo tcpdump consists of the improved *libpcap* library and the original *tcpdump* package. This combination outperforms pure *tcpdump* by far by only losing 0.2% (200,000 out of 100,000,000) of all packets on a 250 Mb/sec link sending 100 byte packets and capturing 64 bytes per packet. This result is impressive in comparaison to the poor performance of *tcpdump*. However, we aim to further improve on this by making the capture mechanism more reliable.

2.4 MAGNeT

2.4.1 Architecture

MAGNeT was developed to capture network traffic as it progresses through a running protocol stack. It differs from existing tools in that it monitors traffic not only as it enters and leaves the network but also at the application level and throughout the entire protocol stack. Feng et al. [8] suggest that application traffic experiences significant modulation by the protocol stack before it is placed on the network. In order to determine this modulation, *MAGNeT* was designed.
The *MAGNeT* approach consists of both Linux kernel modifications and user application programs. There are several hooks in the protocol stack to allow *MAGNeT* to record events. Each time a network-stack event occurs in the kernel, the relevant data is copied to a circular buffer in kernel space. This kernel buffer is exported to user-space applications via Linux kernel/user

shared memory. That is, a device file serves as an user-level handle to the kernel's shared-memory region. Opening this file causes Linux to create a mapping between the kernel memory region and the user-address space. The user application accesses the circular buffer and maintains a pointer to its current slot in the buffer. When the status field at this slot becomes non-zero, the application reads the data but instead of writing the slot directly to disk, it uses the memory-mapping I/O features in the Linux kernel. Once an empty file exists, the application maps this file into its memory space and then saves data to disk by performing a memory copy between the kernel/user shared memory and the memory region mapped to the file. The status field is set to zero to signal the kernel that the slot is once again available. It then advances its pointer to the next slot in the circular buffer.
The kernel also maintains a pointer to its current slot and whenever an event occurs, it checks the value of the status field before writing to a slot. A non-zero indicates that the slot has not yet been copied to user space and that the kernel buffer is full. In this case, the kernel code increments a count of the number of instrumentation records that could not be saved due to buffer overrun. Otherwise, the kernel code writes a new instrumentation record and advances its pointer to the next slot.

2.5 Lightweight Packet Capturer (Lwpcap)

2.5.1 Modifications

The *MAGNeT* architecture solves all the problems specified in 2.2: (1) Hooks in the protocol stack code are defined to grab the packets early in time, (2) context switches are reduced, (3) packets are copied in bulk as they are available in the circular buffer and (4) the data is not written to disk but memory copied.
By introducing the circular buffer, kernel and user space are decoupled and synchronization is maintained via the pointers to the current kernel/user slots (consumer-producer principle).
We adopt this architecture to our problem.

- As we are not interested in modulation of a packet within the protocol stack but simply want to feed the packets to user level, we drop the hooks defined by *MAGNeT* and specify our own control points at the beginning of the protocol stack (Table 2.1).
- We don't need one but many memory-mapped files as we want to capture much more data than can be saved in a single file in memory. These files are arranged in a ring. The user application memory copies the packets into the files, while a special swapper thread unmaps written files and maps empty files into memory space. There are always two files in memory: the currently in use and the next to be written. As soon as the user program reaches the end of a file, it invokes the swapper thread and switches to the next file. Meanwhile, the swapper locks a mutex, unmaps the written file and maps the next empty file into memory space. If the process requests the next file, it tries to lock the mutex. Whenever the swapper did not yet map the requested file, the process has to wait until the swapper completes its job and unlocks the mutex.

  The file size has to be chosen with caution so that the process doesn't have to wait for the thread. Larger files mean more time for the thread to map/unmap but also more data to be mapped/unmapped. Smaller files mean less time for mapping/unmapping but less data to map/unmap. Once all files have been written into, the round-robin scheme implies that the first file is overwritten.

Figure 2.1 illustrates the architecture of the lightweight packet capturer. The kernel module (Lwpcap) is associated with one of three hooks and writes the packets into the kernel buffer. In the meantime, a user program (User Application) reads out the available slots and copies them to the memory-mapped files.
Figure 2.2 shows the journey of a packet through the modified protocol stack.

2.5.2 Hooks

To understand the location of the different hooks in the protocol stack, we delve into the network stack. We consider kernel 2.4.24 as all development was done in this version but the notes are

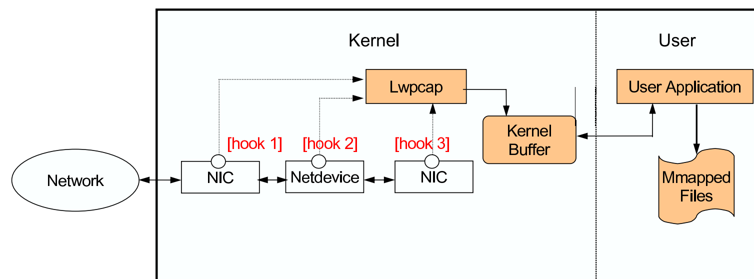

Figure 2.1: Lwpcap Architecture

also true for kernel 2.6.x with minor modifications.

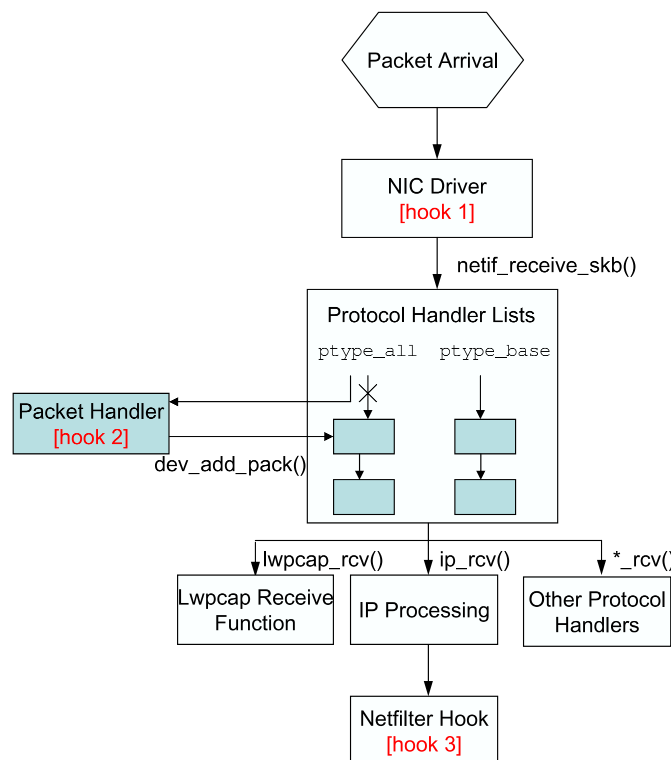

Figure 2.2: Packet Reception Path and Lwpcap Hooks

NIC driver hook

The New Linux API (NAPI) was introduced in release 2.6 (back-propagated to 2.4.20) and provides new techniques to improve network performance. NAPI processes packet events in what is known as device polling method. When a packet arrives at a network interface card, it issues an interrupt request. The operating system serves the request. The NIC driver disables the interrupt and schedules a poll event. That is, all arriving packets are directly written into the card buffer without generating an interrupt request any more. The kernel then polls the device, fetches the collected packets via DMA and copies them into allocated socket buffer structures (skb). As soon as all packets are read out of the network card buffer, the interrupt is re-enabled and the next packet will again generate an interrupt request. This strategy has the advantage avoiding cannibalization of the CPU by interrupt handling activities in case of high traffic rates. Potential problems are card buffer overruns due to low polling frequencies.
The capturing PC we use for testing has two Broadcom 5407 Gigabit Ethernet adapters supporting the Tigon3 chip set. The corresponding Tigon3 driver implements polling since kernel

2.4.20 and therefore, we benefit from NAPI mechanisms.
After fetching the packet from the card buffer into the skb, the socket buffer is passed to the protocol independent device support routine (`net/core/dev.c:netif_receive_skb()` in NAPI), the single gathering point for all the packets collected by the different network card drivers.
Luca Deri's [9] approach intercepts the packets before this generic network reception handler is called. Instead of putting the packet to the network stack, it is passed to a function that directly copies the packet into the kernel buffer. Once the function call returns, the packet is discarded.
Since the packets are intercepted in the NIC driver whereas the other hooks grab them later in the protocol stack, we suggest a high capturing efficiency for the NIC driver hook. However, there are disadvantages in this approach: The NIC driver has to be modified and thus, the solution is not portable. Due to its programming complexity, bugs are an inherent problem.

Protocol handler hook

If the packet is not intercepted by the modified NIC driver hook, it is passed to the network reception handler (`net/core/dev.c:netif_receive_skb()`). This routine runs through two lists of packet handlers, calling the relevant processing functions. That is, there are two lists (`ptype_all` and `ptype_base`) that contain protocol handlers for generic packets (`ptype_all`) and specific packet types (`ptype_base`), respectively. Protocol handlers are registered by invoking the `net/core/dev.c:dev_add_pack()` function which adds a packet type structure to one of the lists (due to the specified protocol type: `ETH_P_ALL` for `ptype_all` and `ETH_P_*` for `ptype_base`). The packet type structure declares the protocol type to handle, the interface to listen at and a pointer to a function which will be called whenever a packet of the specified type arrives at the specified interface.
We will add our packet handler to the generic list (`ptype_all`, protocol type `ETH_P_ALL`) to get hold of all types of packets. When a packet traverses the two lists, our packet handling function (`lwpcap_rcv()`) is called and immediately writes the packet into a slot of the kernel buffer. As we are at the beginning of the first list to be visited, we get the packet before every other handler.
Of course, this hook doesn't grab the packets as early as the NIC driver hook does, but anyway, the performance should be good since the hook is placed right after `net/core/dev.c:netif_receive_skb()` is executed. In contrast to the NIC driver hook, this approach is highly portable and easy to program which makes it a valuable alternative.

Netfilter hook

If an IP packet enters the network stack, the IP packet handler and the corresponding function (`net/ipv4/ip_input.c:ip_rcv()`) in the `ptype_base` list is called. In this routine, the packet passes the Netfilter prerouting control point. Netfilter is a subsystem in the kernel and provides a generic and abstract interface to the standard routing code. This is currently used for packet filtering, mangling, network address translation (NAT) and connection tracking. There are five hooks defined within the protocol stack but we only deal with the first that is called after a packet arrives at the NIC (`NF_IP_PRE_ROUTING` hook in `net/ipv4/ip_input.c:ip_rcv()`, after sanity checks, before routing decision) . Netfilter allows the registration of a hook function on a particular Netfilter hook. If a packet is fed to a Netfilter hook, all the functions defined on this hook are executed. We register a hook function in the prerouting hook. Whenever a packet passes the control point, our function is called and writes the packet into a kernel buffer.
The Netfilter hook is similar to the protocol handler hook: They both register functions at particular places within the network stack, they are similar in performance, they are both portable and easy to program. The Netfilter hook has the additional advantage of being called after some sanity checks are done. Unfortunately, only IP packets (more specific: only IPv4 packets) are captured since the hook is called in the IPv4 packet protocol handler. Furthermore, this approach doesn't work in promiscuous mode as the packets not intended for this host are dropped at the beginning of `net/ipv4/ip_input.c:ip_rcv()`.
Table 2.1 presents different factors affecting the quality of a solution and the rating for the different hooks.

Hook	Performance	Portability	Integrity checks	Programming complexity	Completeness of data flow
NIC	++	-	-	-	+
Protocol handler	+	+	-	+	+
Netfilter	+	+	+	+	-

Table 2.1: Comparison of the *Lwpcap* Alternatives

All in all, the NIC driver hook is the most effective access point but it's not portable and difficult to program. The protocol handler hook and the Netfilter hook are portable, easy to program but perform worse than the NIC driver hook. The Netfilter hook only processes IPv4 packets and fails in promiscuous mode (i.e. it doesn't get the entire data stream).
Comparing and balancing all the advantages and disadvantages, the protocol handler hook is the most reasonable approach because it is fast, elegant, easy to program and guarantees complete data flow delivery.

Chapter 3

Implementation

After the generic description of the architecture, this chapter provides an overview of the implementation. We have a closer look at the program flow and inspect how the different parts interact.

3.1 General Overview

The implementation consists of a Loadable Kernel Module (LKM) and a user-space application (UA). The LKM installs a hook in the protocol stack code intercepting the packets as they pass the control point. The packets are copied into a circular ring buffer in kernel memory. The buffer is exported to user space via shared memory. The user program fetches the available buffer slots and writes them to memory-mapped files.

3.2 Kernel Module

3.2.1 Initialization

A misc device is registered (`drivers/char/misc.c:misc_register()`) serving as a control device for the UA and providing a communication channel to the LKM. Several useful file operations are defined on the misc device:

- `mmap`: Called by user application to map the kernel buffer into user memory
- `ioctl`: Exchange control information (statistics, commands)
- `poll`: Poll the device when waiting for the next slot in the circular buffer to become available

The ring buffer is allocated according to the parameters defined at start time (i.e. the user is able to specify the length of a slot, the number of slots, the capture size and the listening interface). The pages are reserved to not being swapped out.
The buffer size is limited to at most 128KB due to a kernel memory allocation limit (`kmalloc()` is not able to allocate more than 128KB at once). Of course, it would be possible to bypass this limitation by allocating multiple pieces of memory. We set this trick aside to keep the implementation simple. Whenever a user specified buffer exceeds the 128KB limit, the number of slots is reduced to a number that respects the limitation.
The circular buffer consists of two parts:

1. A global information field containing statistical information (number of detected/written/dropped packets), details of the kernel buffer (number of slots, length of a slot), pointers to the current slots of the kernel module and the user application.
2. The slots wherein the full header of each packet and a part of the payload will be saved (number of slots (default: 512) and size of slots (default: 128 bytes) are specified in the information field). The slots contain a status field indicating whether the slot is available or not.

The elected hook is installed or registered.

- For the NIC driver hook, nothing has to be done since the modifications are coded into the Tigon3 driver.
- The protocol handler hook installs a packet handler (`packet_type` struct) in the `ptype_all` list. The `packet_type` structure defines the protocol type (`ETH_P_ALL`) to be handled, the interface to be monitored and the packet handling function to be invoked whenever a packet traverses the network stack. `net/core/dev.c:dev_add_pack()` adds the packet handler to the `ptype_all` list.
- The Netfilter hook binds a function to the Netfilter prerouting control point. The `nf_hooks_ops` structure determines the hook the function is bound to (`NF_IP_PRE_ROUTING`), the priority of the function (`NF_IP_PRI_FIRST`) and the handling function. The interface cannot be specified. That is, the function gets all packets regardless on which interfaces they arrive. The same holds for the NIC driver hook. A test in the handling function eliminates packets of unmonitored interfaces. `net/core/netfilter.c:nf_register_hook()` installs the structure. As already described, only IPv4 packets are grabbed and all packets that are not intended for the local host are discarded.

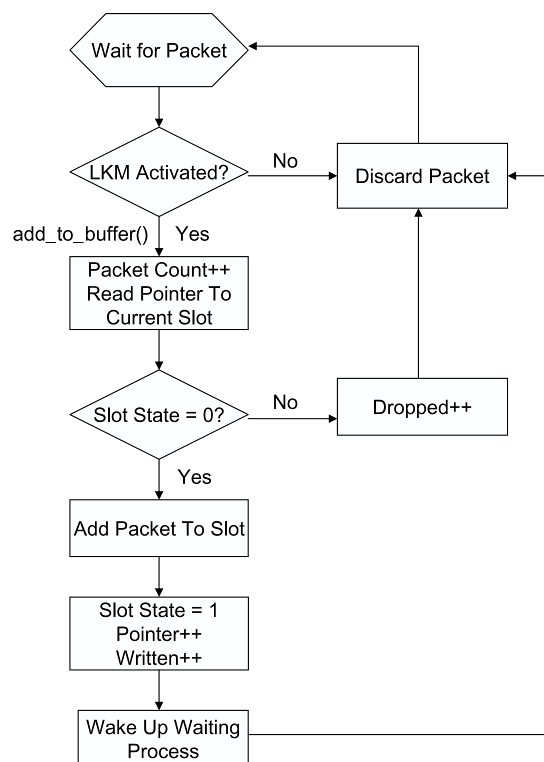

Figure 3.1: Kernel At Runtime

Figure 3.1 illustrates the kernel operations at runtime.

3.2.2 At Runtime

The LKM waits for packets. If the module is not activated, the received packets are dropped immediately. Once the user application activates the module by sending a signal (`LWPCAP_ACTIVATE`) over the communication channel (using the `ioctl` system call), the handling function calls `add_to_buffer()` for each captured packet. After the function returns, the packet is discarded. `add_to_buffer()` increments the packet count in the information field of the kernel buffer, reads the pointer to the current kernel slot, calculates the offset into the

kernel buffer and assigns the memory address to a slot structure. The status of the slot is evaluated. If it is zero (empty), the slot is free and the data is written to the slot. The status is set to one (available), the pointer is advanced to the next slot, the written count is incremented and a potentially waiting process is awakened (`wake_up_interruptible()`). That is, if the user application wants to read a slot but the status indicates empty (no data available), it polls the device and runs into a `poll_wait()`, waiting for a signal sent to a wait queue variable. When the LKM has written the slot, it sends this signal and therefore awakens the UA.
If the slot is full (i.e. the user process didn't manage to fetch the data in time) the packet is dropped and the drop count in the information field is incremented.

3.2.3 Termination

The misc device is unregistered (`drivers/char/misc.c:misc_deregister()`), the functions are unregistered (`net/core/dev.c:dev_remove_pack()` for the protocol handler hook and `net/core/netfilter.c:nf_unregister_hook()` for the Netfilter hook), the kernel buffer is unreserved and released.

3.3 User Space

3.3.1 Initialization

The user-space program first writes the empty files that gather the data. They are mapped into user memory by a swapper thread as capturing proceeds. Different tests have shown that a file size of 200 MB is a good choice. If they are too small the thread is not able to map and unmap the files fast enough. On the other hand, if they are too large, we run into bad performance due to memory management problems.
The misc device is opened to map the kernel buffer, to poll the device and to open a communication channel. Using this channel, external programs are able to activate (start capturing), deactivate (stop capturing) the kernel module, get the packet statistics (packets seen, written, dropped) and adapt the capturing length.
The swapper thread that unmaps the written and maps the empty files is created.
The `LWPCAP_STATISTICS ioctl` is utilized to get a copy of the statistics structure which provides the size of the kernel buffer.
The circular kernel buffer is mapped into memory by invoking the `mmap` file operation on the misc device.
The first two files are mapped into memory and the signal handlers (`SIGINT, SIGTERM`) are installed.
The user application activates the LKM (`LWPCAP_ACTIVATE ioctl`) and jumps into the main loop (`read_kernel_data()`).
Figure 3.2 shows the user application operations at runtime.

3.3.2 At Runtime

In the `read_kernel_data()` routine, the user program continuously checks its pointer to the current slot. If the slot is available, the application memory copies the slot into a file, advances the slot pointer and change the slot status to zero (free). When there is not enough space to write the data, the program signals the swapper thread via a condition variable to unmap the written file and to map a new one. It then switches to the other file in memory that the swapper mapped after the last signaling. If the swapper thread didn't manage to map the new file into memory yet, the program has to wait until the swapper completes the mapping. This mechanism is saved by `phread_mutex_t` structures. The swapper locks a mutex while mapping and unmapping files. The user program performs the file switch as soon as the mutex is not locked by the swapper any longer thus maintaining mutual exclusion.
If the slot is not available, the program polls the misc device and waits (`poll_wait()`) until the LKM awakens the application (`wake_up_interruptible()`) after writing the corresponding slot.

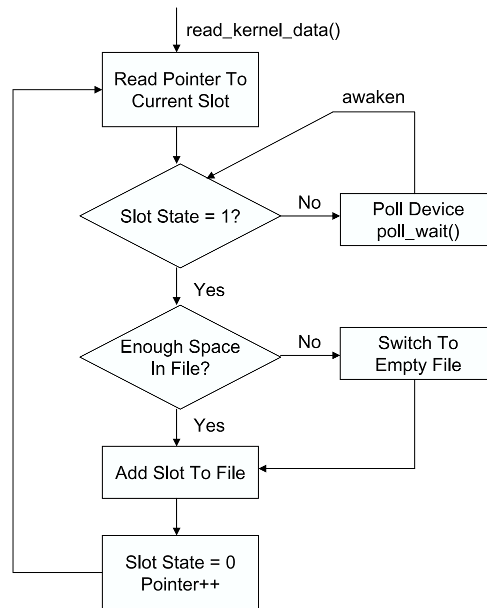

Figure 3.2: User Application At Runtime

3.3.3 Termination

When the user aborts the capture by pressing `Ctrl-c`, the user application sends the `LWPCAP_DEACTIVATE` command to the LKM via `ioctl` and asks for the statistics (`LWPCAP_STATISTICS`). The program prints a report, unmaps the kernel buffer, closes the files in memory, terminates the swapper thread and exits.

Chapter 4

Performance

In this section we perform various tests and discuss the results. We compare *tcpdump* and *turbo-tcpdump* to the three *lwpcap* alternatives and try to understand how particular design decisions influence the outcome.

4.1 Experimental Environment

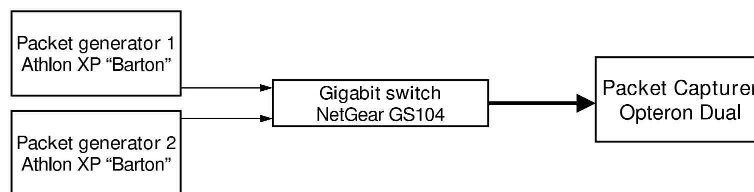

Figure 4.1: Experimental Environment

The test environment we use (Figure 4.1) consists of two identical packet generating PCs (Athlon XP “Barton”, 2.8 GHz, 2 GB RAM, Gigabit Ethernet adapter) connected to a capturing PC (AMD Dual Opteron, 3.4 GHz, 4 GB RAM, PCI-X, Broadcom 5704 Gigabit adapter) over a Gigabit switch (NetGear 4-port Switch GS104, Frame Forward Rate: 1,488,000 frames/sec). Only commodity hardware is used. The packet generators run under Linux kernel 2.4.24 whereas the capturing PC is tested under both 2.4.24 and 2.6.0. Unused services and extraneous traffic are disabled.

4.2 Network Throughput

To generate the traffic, we use *tcpreplay* [4]. This is a tool to replay saved tcpdump files. Other tools like *netperf* or *iperf* were tested, but we prefer *tcpreplay* as it provides high throughput, comprehensive statistical information (number of packets sent, elapsed time, data throughput, packet throughput) and additional features. That is, the *tcpreplay* data stream can be bound to a specific network card and therefore allows sending packets from one interface to the other on the same machine unlike many other tools where the kernel demystifies the trick and uses the loopback interface instead.
Table 4.1 shows that the data throughput decreases dramatically when decreasing the packet size. Small packets lead to high packet throughput but the corresponding data throughput is low because there are too few packets sent to compensate the lack of data introduced by the small packet size (i.e. 1500/64 = 23 times more 64 bytes packets have to be sent to get the same data throughput as the 1500 bytes stream provides).
Capture tools perform worse with higher packet throughput, as they have to deal with more packets and a massive amount of data has to be saved (i.e. they always store the same amount

Packet size	64 bytes	256 bytes	1500 bytes
Packets sent	52,639,180 pkts	25,442,196 pkts	10,262,260
Data throughput	117.84 Mb/sec	339.63 Mb/sec	766.54 Mb/sec
Packet throughput	241,325 pkts/sec	146,315 pkts/sec	66,982 pkts/sec
Elapsed Time	218.13 sec	146.32 sec	153.29 sec

Table 4.1: Packet Size and Throughput

of data per packet, but there are significantly more packets to handle). As a consequence, more data has to be stored.

4.3 Results

The tests evaluate the performance of *tcpdump*, *turbo-tcpdump* and the three *lwpcap* variants: (1) NIC driver hook, (2) Protocol handler hook, (3) Netfilter hook.
Each test configuration runs under Linux kernel 2.4.24 and 2.6.0 as well as packets of size 64 bytes, 256 bytes and 1500 bytes are transmitted. 64 bytes packets are used to get a high packet throughput and thus to stress the applications, 1500 bytes (MTU limit) are sent to get a high data throughput and 256 bytes because the average packet size in the Internet is around 200 bytes [17] and so 256 bytes measurements simulate "real-life" situations. The number of packets sent is also constant for every packet size throughout the tests. The massive differences in the number of packets sent have the pragmatic reason that a measurement takes longer for 1500 byte packets than for 64 byte packets when transmitting the same number of packets. Hence, we send less packets of large size to keep the time for a test more or less constant regardless of packet size.

4.3.1 Core Capacity

Test

To determine whether packets are dropped on the way from the packet generators to the NIC driver of the capturing PC, we simply count the packets arriving at the capturing NIC and discard them immediately. This proceeding yields no packet handling overhead.
The purpose of this test is to check whether the entire data stream arrives at the Gigabit NIC of the capturing PC. It would be possible that packets are dropped at the packet generator's NIC, in the Gigabit switch or at the capturing PC's NIC before we count them in the NIC driver.

		64 bytes		256 bytes		1500 bytes
Kernel 2.4.24	sent:[a]	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:[b]	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:[c]	0	lost:	0	lost:	0
Kernel 2.6.0	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0

Table 4.2: Counting packets in the NIC driver

[a]Number of packets sent by packet generators
[b]Number of packets counted in NIC driver of capturing PC
[c]sent - counted

Interpretation

Table 4.3.1 points out that there is no packet loss before the packets enter the network stack. The NAPI mechanisms work properly and card buffer shortage is not a problem yet.

4.3.2 Counting

Test

Tcpdump, *turbo-tcpdump* and the *lwpcap* alternatives come into play. They count and discard the packets but don't save them to disk (in *tcpdump* this means to write the packets to /dev/null). This measurement has the intention to evaluate the burden of the later introduced writing operation. If a program captures the entire traffic stream without writing the data to disk and loses packets when saving them to disk, then writing is a bottleneck.

		64 bytes		256 bytes		1500 bytes
Tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:[a]	52,613,722	counted:	25,442,196	counted:	10,262,260
	lost:	25,458	lost:	0	lost:	0
Turbo-tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0
Lwpcap NIC driver hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0
Lwpcap Protocol handler hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0
Lwpcap Netfilter hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0

Table 4.3: Counting packets under kernel 2.4.24

[a]Number of packets counted by the application

		64 bytes		256 bytes		1500 bytes
Tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	42,425,603	counted:	24,538,219	counted:	10,262,260
	lost:	10,213,577	lost:	903,977	lost:	0
Turbo-tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0
Lwpcap NIC driver hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0
Lwpcap Protocol handler hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0
Lwpcap Netfilter hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	counted:	52,639,180	counted:	25,442,196	counted:	10,262,260
	lost:	0	lost:	0	lost:	0

Table 4.4: Counting packets under kernel 2.6.0

Interpretation

The results are similar for both kernel versions: *Turbo-tcpdump* and the *lwpcap* variants manage to count all the packets whereas *tcpdump* misses many of them. This is not surprising since *tcpdump* is not designed to keep up with high packet rates.

Tcpdump performs poor for small packets and betters itself when capturing large frames. This is what we expect since high packet throughput means more counting overhead and therefore heavy load on the application.
The performance of *tcpdump* dramatically degrades using kernel 2.6.0. This result is surprising as we expect 2.6.0 to provide significant network performance improvements. *Tcpdump* loses almost a million 256 bytes packets, whereas it manages to count them all under 2.4.24. We conclude that the kernel modifications introduced in kernel 2.6 have a bad influence on *tcpdump*'s performance.

4.3.3 Writing to disk

Test

We write the first 64 bytes of every arriving packet to disk.
This is the crucial measurement as it simulates real-life conditions the lightweight packet capturer will face. The collected data has to be archived and not just thrown away. In *lwpcap*, archiving means to save the data into memory-mapped files.

		64 bytes		256 bytes		1500 bytes
Tcpdump	sent:[a]	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:[b]	52,639,180	detected:	25,442,196	detected:	10,262,260
	not detected:[c]	0	not detected:	0	not detected:	0
	written:[d]	48,808,374	written:	24,950,393	written:	10,247,108
	dropped:[e]	3,830,806	dropped:	491,803	dropped:	15,152
	lost:[f]	7.3%	lost:	1.9%	lost:	0.1%
Turbo-tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:	52,639,180	detected:	25,442,196	detected:	10,262,260
	not detected:	0	not detected:	0	not detected:	0
	written:	52,490,294	written:	25,428,202	written:	10,262,260
	dropped:	148,886	dropped:	13,994	dropped:	0
	lost:	0.3%	lost:	0.06%	lost:	0%
Lwpcap NIC driver hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:	52,597,919	detected:	25,421,444	detected:	10,262,260
	not detected:	41,261	not detected:	20,752	not detected:	0
	written:	52,597,919	written:	25,421,444	written:	10,262,260
	dropped:	0	dropped:	0	dropped:	0
	lost:	0.08%	lost:	0.08%	lost:	0%
Lwpcap Protocol handler hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:	52,549,710	detected:	25,415,451	detected:	10,262,105
	not detected:	89,470	not detected:	26,745	not detected:	155
	written:	52,549,593	written:	25,415,451	written:	10,262,105
	dropped:	117	dropped:	0	dropped:	0
	lost:	0.2%	lost:	0.1%	lost:	0.002%
Lwpcap Netfilter hook	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:	52,510,909	detected:	25,391,485	detected:	10,258,219
	not detected:	128,271	not detected:	50,711	not detected:	4041
	written:	52,508,594	written:	25,391,047	written:	10,258,219
	dropped:	2315	dropped:	438	dropped:	0
	lost:	0.3%	lost:	0.2%	lost:	0.04%

Table 4.5: Writing packets under kernel 2.4.24

[a] Number of packets sent by packet generators
[b] Number of packets detected in the network stack
[c] sent - detected
[d] Number of packets written to disk
[e] detected - written
[f] (not detected + dropped)/sent

		64 bytes		256 bytes		1500 bytes
Tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:	52,639,180	detected:	25,442,196	detected:	10,262,260
	not detected:	0	not detected:	0	not detected:	0
	written:	38,442,025	written:	22,407,716	written:	10,262,260
	dropped:	14,197,155	dropped:	3,034,480	dropped:	0
	lost:	27.0%	lost:	11.9%	lost:	0%
Turbo-tcpdump	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
	detected:	52,639,180	detected:	25,442,196	detected:	10,262,260
	not detected:	0	not detected:	0	not detected:	0
	written:	52,639,180	written:	25,442,196	written:	10,262,260
	dropped:	0	dropped:	0	dropped:	0
	lost:	0%	lost:	0%	lost:	0%
Lwpcap	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
NIC driver hook	detected:	52,639,180	detected:	25,442,196	detected:	10,262,260
	not detected:	0	not detected:	0	not detected:	0
	written:	51,374,736	written:	25,087,979	written:	10,262,260
	dropped:	1,264,444	dropped:	352,217	dropped:	0
	lost:	2.4%	lost:	1.4%	lost:	0%
Lwpcap	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
Protocol handler	detected:	52,639,180	detected:	25,442,196	detected:	10,262,260
hook	not detected:	0	not detected:	0	not detected:	0
	written:	50,809,515	written:	24,411,374	written:	10,165,480
	dropped:	1,829,665	dropped:	1,030,822	dropped:	96,780
	lost:	3.5%	lost:	4.1%	lost:	0.9%
Lwpcap	sent:	52,639,180	sent:	25,442,196	sent:	10,262,260
Netfilter hook	detected:	52,639,180	detected:	25,442,196	detected:	10,262,260
	not detected:	0	not detected:	0	not detected:	0
	written:	51,810,735	written:	25,139,464	written:	10,262,260
	dropped:	828,445	dropped:	302,732	dropped:	0
	lost:	1.6%	lost:	1.2%	lost:	0%

Table 4.6: Writing packets under kernel 2.6.0

Interpretation: Kernel 2.4.24

As can be seen in Table 4.3.3, no application manages to capture all 64 bytes packets.
The larger the packets, the better the performance.
Again, *tcpdump* performs worst and drops millions of packets.
Turbo-tcpdump and *lwpcap* variants perform similar but they lose the packets at different points in the processing chain: *Turbo-tcpdump* (and *tcpdump*) detect all packets in the kernel but drop them before they are written to disk (context switches and/or writing to disk are bottlenecks). The *lwpcap* variants do not drop packets when dealing with the kernel buffer since all packets that are detected and processed in the LKM are also copied to disk (only few drops). This implies that handing over the packets in the kernel buffer works fine. But lots of packets remain undetected. That is, they get lost before they are counted in the hook function. Assuming that all packets arrive at the NIC of the capturing PC, these effects point at problems in the network card buffer or in the memory management when fetching the packets from card buffer into the socket buffer structures. Perhaps the NAPI polling rate is too low and packets are dropped due to card buffer overruns.
Comparing the different *lwpcap* variants, the NIC driver hook performs best (as expected), followed by the Protocol handler hook and the Netfilter hook, but the differences are marginal.

Interpretation: Kernel 2.6.0

Surprising results in Table 4.6: *Turbo-tcpdump* loses no packets in all measurements.
All the other applications perform worse than under 2.4.24. This is a trend already met in the

tcpdump counting measurements. Again, kernel modifications or bugs in the *lwpcap* implementation can add to these results.
As can be seen in Table 4.6, there are no undetected packets in *lwpcap* but a massive number of dropped frames. The picture changes in that *lwpcap* loses the packets at a different stage in packet reception. Under kernel 2.4.24, the packets disappeared before they were counted (no kernel buffer bottleneck) whereas now, all packets are detected in the kernel but they are dropped before copying them into the buffer. That is, the user application doesn't manage to read out the slots fast enough. Because the buffer slots are occupied, the arriving packets are thrown away by the LKM. These results imply that there are synchronization problems between kernel and user programs.
One explanation for the better performance of *turbo-tcpdump* compared to *lwpcap* could be that *turbo-tcpdump* allocates 32768 frames thus having a large buffer space available whereas *lwpcap* is limited to the 128KB circular buffer (resulting in 1000 slots at most). More slots mean more time for the UA to read out the slots before a buffer overrun occurs.
To touch *turbo-tcpdump*'s limits, I ran a 1-billion-packets test using kernel 2.6.0: *Turbo-tcpdump* captured 1,052,394,531 packets (out of 1,052,783,600) transmitting 64 bytes packets and only lost 389,069 packets (0.04%). This is an impressive result but one has to be aware of the fact that in all these measurements, *turbo-tcpdump* only has to save a 120 Mb/sec (15 MB/sec) data stream and therefore the writing head manages to keep pace even when writing each packet separately to disk. This will dramatically change if the stream to be saved is scaled up. Then, the hardware limits are hit and more sophisticated mechanisms (i.e. memory-mapped files) have to be introduced to face the problems.

Chapter 5

Summary and Outlook

5.1 Summary

This Semester thesis provides the design and implementation of a lightweight packet capturer that keeps pace with Gigabit links.
The packets are intercepted on their way through the protocol stack in one of three hooks. In (1) the network card driver, in (2) the `ptype_all` protocol handler list or in (3) the prerouting Netfilter hook. The captured packets are put into a kernel buffer from where a user-space application memory copies them into files.
This architecture eliminates excessive context switching from kernel to user mode and single packet copying and disk writing. Therefore, it disburdens the hardware and improves the performance significantly.
The concluding tests show that the *lwpcap* variants beat *tcpdump* by far. Whereas *tcpdump* loses 7.3% of the packets in worst case (using kernel 2.4.24), *lwpcap* only drops 0.2%. These results prove the assumption that the chosen architecture is appropriate and effective.
The measurements also show that the performance of all applications degrade using kernel 2.6.0 except for *turbo-tcpdump* which does not lose a single packet in these tests.
While working on this thesis, I realized that there are many ways to approach a problem. Most of them end up in a dead end and only a few yield reasonable results. I took the bad paths many times and this experience made me act with caution and I started thinking hard before making a decision in one or the other direction. And so I fumbled in the darkness for the light switch hoping not to leave the right track. I'm now aware of the fact that systematic proceeding goes along with success and provides guidance through the dense fog.
It was my ambition to fully understand the nature of the problem and to find adequate answers to the questions I encountered. Success wasn't always on my side but after all, I found a solution for a difficult problem and I learned a lot about computer science engineering and the Linux kernel.
This is more than one could expect.

5.2 Outlook

The work for the lightweight packet capturer is not finished yet.
It will be the task of my follower to refine the prototype implementation of this Semester thesis and perhaps to embed it into a working framework.
There is a number of unsolved tasks:

- Increase flexibility and usability
- Implementation of a user-friendly interface
- Dynamically adapt various parameters (size of kernel buffer and capturing length) at runtime.
- Improve the mechanisms used to save the data from kernel buffer to disk (e.g. kernel threads to perform all steps)

- Proper integration of IPv6
- Capturing on several interfaces in parallel
- More diversity in processing collected data
- Filtering mechanisms

Much work has still to be done, challenging problems have to be solved.
Time will tell whether I took the right path.

5.3 Acknowledgment

I'd like to thank Parminder Chhabra, Thomas Dübendorfer and Arno Wagner for their support, humor, critical comments and their help that lead me out of the darkness into the light a thousand times throughout this work.
I know it wasn't it wasn't always easy to work with me. Their calmness and gentleness is amazing.
My thanks go to them.

Bibliography

[1] Lawrence Berkeley National Labs.
Libpcap
http://www.tcpdump.org

[2] The Tcpdump Group.
Tcpdump
http://www.tcpdump.org

[3] The Ethereal Group.
Ethereal
http://www.ethereal.com

[4] Aaron Turner, Matt Undy, Matt Bing.
Tcpreplay
http://tcpreplay.sourceforge.net/

[5] Phil Wood.
Turbo tcpdump
http://public.lanl.gov/cpw/

[6] Jeffrey R. Hay, Wu-chun Feng, Mark K. Gardner. (2001).
Capturing Network Traffic with a MAGNeT.
Proceedings of the 5th Annual Linux Showcase and Conference (ALS'01), Oakland, California, November 2001
http://public.lanl.gov/radiant/pubs/magnet/als01.ps

[7] Eric Weigle, Wu-chun Feng. (2002).
TICKETing High-Speed Traffic with Commodity Hardware and Software.
Passive & Active Measurement Workshop (PAM2002), Fort Collins, Colorado, March 2002
http://public.lanl.gov/radiant/pubs/ticket/PAM-2002-TICKET.ps

[8] W. Feng and P. Tinnakornsrisuphap. (2000).
The Failure of TCP in High-Performance Computational Grids.
Proceedings of the 2000 ACM/IEEE conference on Supercomputing, November 2000
http://www.sc2000.org/techpapr/papers/pap.pap174.pdf

[9] Luca Deri. (2003).
Improving Passive Packet Capture: Beyond Device Polling.
http://luca.ntop.org/Ring.pdf

[10] Bioforge. (2003).
Hacking the Linux Kernel Network Stack.
Phrack Magazine, Vol 12, Issue 61, File 13.
http://www.phrack.org/phrack/61/p61-0x0d_Hacking_the_Linux_Kernel_Network_Stack.txt

[11] Kossak, Lifeline (1999).
Building Into The Linux Network Stack.
Phrack Magazine, Vol 9, Issue 55, File 12.
http://www.phrack.org/phrack/55/P55-12

[12] Harald Welte. (2000).
The Journey of a Packet Through the Linux 2.4 Network Stack.
http://www.gnumonks.org/ftp/pub/doc/packet-journey-2.4.html

[13] Gianluca Insolvibile. (2001).
Kernel Korner: The Linux Socket Filter: Sniffing Bytes over the Network.
Linux Journal, Issue 86.
http://www.linuxjournal.com/article.php?sid=4659

[14] Gianluca Insolvibile. (2002).
Kernel Korner: Inside the Linux Packet Filter.
Linux Journal, Issue 94.
http://www.linuxjournal.com/article.php?sid=4852

[15] Gianluca Insolvibile. (2002).
Kernel Korner: Inside the Linux Packet Filter, Part II.
Linux Journal, Issue 95.
http://www.linuxjournal.com/article.php?sid=5617

[16] K. Coffman and A. M. Odlyzko. (2001).
Internet Growth: Is there a "Moore's Law" for data traffic?.
http://www.dtc.umn.edu/ odlyzko/doc/internet.moore.pdf

[17] Lukas Karrer. (2000).
Characterization of Internet Traffic - Interesting Facts and Figures.
http://www.tik.ee.ethz.ch/ huang/teach/ summer00/student/trend-sum.pdf