POLITECNICO DI TORINO
Repository ISTITUZIONALE

An Architecture for High Performance Network Analysis

Original
An Architecture for High Performance Network Analysis / RISSO F.; DEGIOANNI L.. - (2001), pp. 686-693. ((Intervento presentato al convegno 6th IEEE Symposium on Computers and Communications (ISCC 01).

Availability:
This version is available at: 11583/1417041 since:

Publisher:

Published
DOI:

Terms of use:
openAccess
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

23 January 2022

An Architecture for High Performance Network Analysis

Fulvio Risso and Loris Degioanni
Dipartimento di Automatica e Informatica – Politecnico di Torino
Corso Duca degli Abruzzi, 24 – 10129 Torino, Italy
{risso,degioanni}@polito.it

Abstract — *Most Unix systems provide a set of system calls that allow applications to interact with the network directly. These primitives are useful for example in packet capture applications, which need to grab the data flowing through the network without any further processing from the kernel. WinPcap is a newly proposed architecture that adds these functionalities to Win32 operating systems. WinPcap includes a set of innovative features (such as packet monitoring and packet injection) that are not available in previous systems. This paper presents the details of the architecture and it shows its excellent performance.*

1 Introduction

Applications for network analysis rely on an appropriate set of primitives to capture packets, monitor the network and more. While almost all Unix flavours have kernel modules that support at least packet capture, Windows capabilities are not satisfying. There are some available APIs, each one with his own kernel-module; however they suffer of severe limitations. Netmon API, for instance, is not freely available and its extensibility is very limited; moreover it does not allow sending packets. IP filter driver is available only on Windows 2000 and it does not support other protocols but IP; it allows controlling and dropping packets but it does not allow monitoring and generating them. PCAUSA [11] offers a commercial product that provides an interface for packet capture and includes a BPF-compatible filter. However the user interface is quite low-level and it does not provide abstract functions like filter generation.

The growing diffusion of Windows for tasks that traditionally relied on Unix workstations makes these missing features a non-negligible problem, thus limiting the number and the quality of analysis and security tools on that platform. Our efforts focused on the creation of a powerful and extensible architecture for low-level network analysis on Win32 platform, called WinPcap. This architecture is the first open system for packet capture on Win32 and it fills an important gap between Unix and Windows. Furthermore WinPcap puts performance at the first place, thus it is able to support the most demanding applications.

Packet filtering is made up of a kernel-mode component (to select packets) and a user-mode library (to deliver them to the applications). Last component provides a standard interface for low-level network access and allows programmers to avoid kernel-level programming. WinPcap includes an optimized kernel-mode driver, called Netgroup Packet Filter (NPF), and a set of user-level libraries that are `libpcap`-compatible. `Libpcap` [1] API compatibility was a primary objective in order to create a cross-platform set of functions for packet capture. WinPcap makes the porting of Unix applications to Win32 easier and it enables a large set of programs to be used on Win32 at once, just after a simple recompilation. Moreover, since the importance of traffic monitoring, WinPcap provides highly specific system calls for that.

This paper is structured as follows: Section 2 shows the original BSD packet capturing components; Section 3 presents the general structure and the main components of WinPcap, while Section 4 focuses on the implementation issues; a comparison of the performance between BPF (for FreeBSD) and NPF is given in Section 5. Finally, Section 6 summarizes the results and presents some conclusive remarks.

2 BSD Capturing Components

The system primitives for packet capture and network analysis should be powerful, in order to have excellent performance, abstract, to limit the complexity of the interface between user applications and the kernel, and low-level, in order to get and send data directly to the network interface without any interaction with other software layers.

The BSD-proposed approach for packet capture (Figure 1) can be seen as the sum of three main components. The first block, **Berkeley Packet Filter** (BPF) [2] is the kernel-level component for packet

capture. BPF has been widely recognized having the best implementation compared to other similar components available on Unix and it can be seen as a device driver that interacts with the network interface through the interface's driver.

Its architecture is made up of the following components. The **Network Tap** [3] is a function dedicated to snoop all packets flowing through the network. It is followed by the **Filter**, which analyzes incoming packets in order to detect whether a packet is interesting for the user (the user can set capture filters to receive only a subset of the network traffic). A packet satisfying the filter is copied to the **kernel buffer**, which is subdivided in two small buffers (*store* and *hold*) that are used to keep the captured packets for a while. These blocks of memory (whose size is usually 32Kbytes in recent BSD kernels) are allocated at run time at the beginning of the capture process. The first buffer (*store*) is used to keep the data coming from the network adapter and the second one (*hold*) is used to copy the packets to the user buffer. When the store buffer is full and the hold buffer is empty, BPF swaps them. In this way the user-level application does not interfere with the adapter's device driver because the former clears out the hold buffer while the latter fills in the store one.

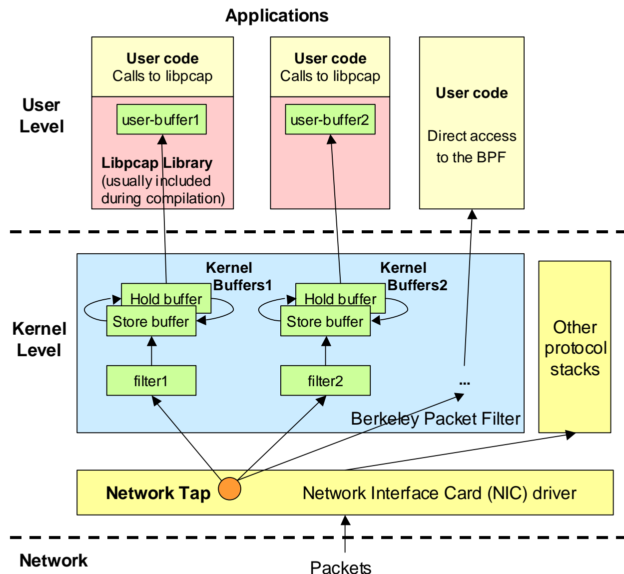

Figure 1. BSD Capturing Components.

`Libpcap` is the main component from the programmer's perspective because it hides the interaction between the application and the OS kernel. `Libpcap` exports a set of functions that can be linked to the user's application and that provide a powerful and abstract interface to the capture process. It includes the filter generation, the management of the **user-level buffer** (that remains hidden to the application), the interaction between user and kernel mode. The User-level Buffer is used to store packets coming from the kernel and, being at user level, it prevents the application from accessing kernel-managed memory. Functions provided by `libpcap` are available only for capturing purposes: this library does not allow sending packets or monitoring the network.

`Libpcap` works in user-space and, being less OS-dependent, it has been successfully ported to several Unix. On the other hand, BPF has been implemented only in a few OS (basically BSD-derived). The lack of a BPF filtering machine in the kernel (for example Solaris[1]) means that all the packets have to be transferred to `libpcap` at user level and that this library has to emulate the functionalities of BPF (filtering and buffering). However native kernel-level packet filtering is far more efficient because the system avoids copying non-interesting packet from the network interface to the user level. Moreover, the kernel-level filtering decreases the number of system calls and the number of context switches between user and kernel level because only useful packets are copied to user-level.

3 Architecture of WinPcap

Since BPF has been proved being a powerful and stable architecture, the basic structure of WinPcap (shown in Figure 2) retains the most important modules shown in Section 2: a filtering machine, two buffers (kernel and user) and a couple of libraries at user level. However, WinPcap has some substantial differences in the structure and in the behavior of the capture stack, and can be seen as the evolution of BPF.

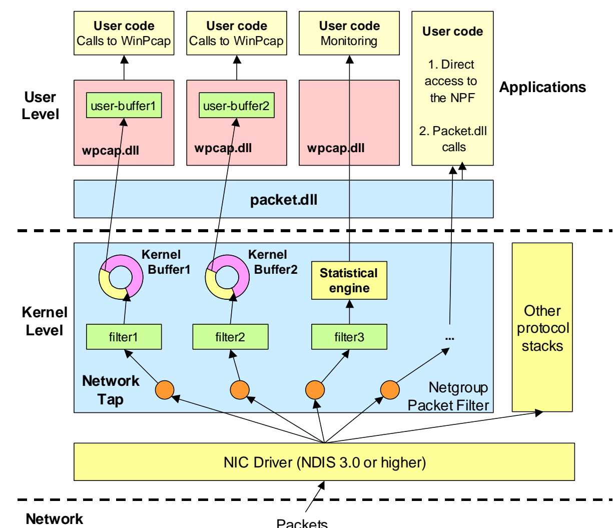

Figure 2. WinPcap and NPF.

[1] Solaris has the *CMU/Stanford Packet Filter* (CSPF) in the kernel; however `libpcap` does not include a CSPF code generator because CSPF does not support advanced features like variable-length headers.

The filtering process is started by a user-level (libpcap-compatible) component that is able to accept a user-defined filter (i.e. "picks up all UDP packets"), compile them into a set of pseudo instruction (i.e. "if the packet is IP and the *protocol type* field is equal to 17, then return true"), send these instructions to the filtering machine, and activate that code. On its hand, the kernel module must be able to execute these instructions; therefore it must have a "BPF virtual machine" that executes the pseudo-code on all the incoming packets. This kernel-side BPF-compatible filtering machine is a key point for obtaining good performance.

An important architectural difference between BPF and NPF is the choice of a circular buffer as kernel buffer. Our implementation of the circular buffer has been optimized to copy blocks of packets at a time. This mechanism is more difficult to manage because data copies no longer have fixed sizes (libpcap uses the same size for kernel and user buffer, 32KB) and because the amount of the bytes copied is updated *while* transferring data from kernel space to user space instead of *after*. Indeed, the copy process (started at user level) frees the portion of the buffer already transferred because the capture portion of the kernel may interrupt it, since it runs at higher priority and it could monopolize all the CPU time. Our implementation of the buffer allows all the memory to be used to store network bursts, while a couple of swapping buffers (such in BPF) allows only half of the memory to be used.

The entire kernel buffer is usually copied by means of a single read(), thus decreasing the number of system calls and therefore the number of context switches between user and kernel mode. Since a context switch requires saving the state of the task (CPU descriptor and task state segment, approximately some hundreds bytes), large transfers decrease the overhead of the process. However a too large user buffer brings no advantages because the context switch overhead becomes negligible while the allocated memory increases linearly.

WinPcap kernel buffer is larger (default 1MB) than BPF one. A small buffer penalizes the capturing process especially in case the application is not able to read as fast as the driver captures for a limited time interval. Such situation is common when data has to be transferred to disk or when the network has bursty traffic. Vice versa, user buffer is smaller and its default value is 256KB. Both kernel and user buffer can be changed at runtime[2].

The size of the user buffer is very important because it determines the *maximum* amount of data that can be copied from kernel space to user space within a single system call. On the other hand, we noticed that also the *minimum* amount of data that can be copied in a single call is extremely important. In presence of a large value for this variable, the kernel waits for the arrival of several packets before copying the data to the user. This guarantees a low number of system calls, i.e. low processor usage, which is a good setting for applications like sniffers. On the other hand, a small value means that the kernel will copy the packets as soon as the application is ready to receive them. This is excellent for real time applications (like, for example, an ARP redirector) that need the better responsiveness from the kernel. The great part of the existing capture drivers has a static behavior that privileges one of the two aspects. On the other hand, NPF is completely configurable and it allows users to choose between either best efficiency or best responsiveness (or any intermediate behavior). WinPcap includes a couple of system calls that can be used to set both the timeout after which a read expires and the minimum amount of data that can be transferred to the application. Packets are copied either when the minimum amount of data is available or the read timeout has expired. By default, the timeout is 1 second and the minimum amount of data is 16K. This capability is often called "*delayed write*"[3].

In general memory copies have to be kept the lowest because of their overhead. WinPcap has the same copy overhead of the original libpcap/BPF and packets are copied two times (from network driver to the kernel, then to user space). Moreover the filtering is performed when the packet is still into the network driver's memory, thus avoiding any copy of non conformant packets in a similar way as the BSD bpf_tap() function does.

3.1 Statistics mode

Packet capturing and network analysis are CPU-intensive tasks because of the high amount of data to process and copy. Our tests will show that the simple operation of capturing small packets on a Fast Ethernet LAN may overwhelm the CPU power of a modern workstation. This prevents from performing any real-time processing on the captured data without losing a large amount of packets. This situation is obviously even worse on faster networks, which require the use of specific hardware to perform the capture.

The most famous approaches to improve speed include improved filtering engines [7] [8] [9] and 1-copy[4] architectures, which avoid copying packets between

[2] pcap_setbuff() allows the user to choose the size of the kernel buffer. BSD allows changing the buffer size by using the sysctl debug.bpf_bufsize shell command. However libpcap overwrites these settings and sets always the kernel buffer to 32Kbyte.

[3] This has not been implemented in Windows 9x and the driver transfers packet from kernel to user space as soon as possible; therefore the CPU load might be higher than in Windows NT/2000.

[4] A packet needs always to be copied from the NIC to the device driver memory. A 0-copy process is obtained when no further copies are performed. A 1-copy process is obtained when the packet is copied one more time before being received by the user-application. Standard tools use a 2-copy process, in which the packet is copied from the device driver memory to the kernel buffer, then to the user one.

kernel space and user space by mapping the kernel buffer in the application's memory [10]. However a 1-copy process decreases the amount of data to be copied but may not be able to decrease the number of system calls between user and kernel mode. If the user reads one packet at a time, the number of system calls (i.e. the number of context switches) is still high and it could vanish the advantages of a shared buffer.

A new idea can be applied keeping in mind that monitoring does not require the packet to be transferred to the application. WinPcap moves monitoring capabilities inside the kernel, thus avoiding any data transfer (and processing) at user level. WinPcap implements a kernel-level programmable statistical module by using the filtering machine available in the NPF, which becomes also a powerful classification engine rather than a simple packet filter. Applications can instruct this module to monitor an arbitrary aspect of the network activity (for example network load, amount of traffic between two hosts, number of web requests per second, etc.), and to receive results from the kernel at predefined intervals.

Statistic mode avoids packet copies and it implements a 0-copy mechanism (statistic is performed when the packet is still in the NIC driver's memory, then the packet is discarded). Moreover, the number of context switches is kept the lowest because results are returned to user-level by means of a single system call. No buffers (kernel and user) are required, therefore they are not allocated when monitoring is launched. As a result, statistics mode is an extremely efficient method to monitor the network and it is able to work without problems on fast networks with the well-known `libpcap` syntax.

WinPcap offers to the programmer a new set of system calls and high-level functions that can be used to launch a monitoring process, making this capability easy to use for a programmer that already knows the `libpcap` API.

3.2 Packets injection

Both BPF and NPF provide writing capabilities that allow the user sending raw packets to the network. However `libpcap` does not use these calls, thus BPF is never used for this purpose; Unix applications often use raw sockets for that. On the other side, Win32 provides raw sockets only in Windows 2000 and they are quite limited; therefore WinPcap is the first library that provides a standard and consistent set of functions for packet injection for all the Win32 flavors. Besides, NPF adds some new functions that allow sending a packet several times using a single context switch between user and kernel mode. Data is copied to the kernel and then sent to the network by means of a single NDIS call.

While WinPcap provides a new set of functions to exploit these features, it does not provide a powerful abstraction for creating packets that need to be generated by hand or by means of other existing tools. However, users can use the Windows version of the well-known Libnet Packet Assembly Library [4], which adds a layer for packet construction and injection on top of WinPcap.

4 Implementation issues

4.1 WinPcap modules

WinPcap is made up of three modules, one at kernel level and two at user level; userland modules come under the form of Dynamic Link Libraries (DLLs).

First module is the kernel part (NPF) (a VXD file in Windows 95/98/ME and a SYS file in NT/2000) that. filters the packets, delivers them untouched to user level and includes some OS-specific code (timestamp management) as well.

Second module, *packet.dll*, is created to provide a common interface to the packet driver among the Win32 platforms. In fact, each Windows version offers different interfaces between kernel modules and user-level applications: *packet.dll* deals with these differences, offering a system-independent API. Programs based on *packet.dll* are able to capture packets on every Win32 platform without being recompiled. *Packet.dll* includes several additional functionalities. It performs a set of low-level operations like obtaining the adapters' names[5] or the dynamic loading of the driver, and it makes available some system-specific information like the netmask of the machine and some hardware counters (the number of collisions on Ethernet, etc.). Both this DLL and the NPF are OS-dependent and change between Windows 95/98 and NT/2000 because of the different OS architectures.

Third module, *WPcap.dll*, is not OS-dependent and it contains some other high-level functions such as filter generation and user-level buffering, plus advanced features such as statistics and packet injection. Therefore programmers can have access to two types of API: a set of raw functions, contained in *packet.dll*, which are directly mapped to kernel-level calls, and a set of higher level functions that are provided by *WPcap.dll* and that are more user-friendly and more powerful. The latter DLL will call the former automatically; a single "high-level" call may be translated in several NPF system calls. Programmers will normally use *WPcap.dll*; direct access to *packet.dll* is required only in limited cases.

4.2 WinPcap and Windows networking

Win32 networking architecture is based on NDIS[6] [5] (Network Driver Interface Specification) standard, the lowest level networking portion of the Windows kernels. NDIS is a specification for building network interface (NIC) drivers and protocol drivers and to handle the

[5] Windows does not have the widely used `ifconfig` Unix utility; this functionality is provided by *packet.dll*.

[6] WinPcap requires the version 3.0 of NDIS at least.

interaction among them. It provides a set of primitives that hide the underlying technology from the upper layers so that a single instance of a protocol stack should be able to exploit different network technologies.

Intuitively, the core of the capture process must run at kernel level and it must be able to access the packets before the protocol stack processes them. BSD executes the capture system before any protocol: BPF is invoked directly by the network card's driver and it requires NIC device drivers to be compliant to something that could be called "BPF driver specification". In other words it requires device drivers to have an explicit call to the BPF tap function, which controls all the packets received (and sent) by the network interface and makes a copy of the ones that satisfy the filter. Such a solution was clearly impossible when creating the WinPcap capture component, because Windows has nothing similar to the BPF driver specification and it does not allow modifying the OS and the NIC drivers in order to add that feature. Therefore, WinPcap locates the network tap as a protocol driver[7] on top of the NDIS structure. Support for different media types is not automatic because NDIS does not isolate completely the underlying layers from the NPF; therefore it has to be built in such a way to support explicitly different media.

The location of the network tap influences the behavior of the capture driver. For example, Point-to-Point Protocol (PPP) uses some auxiliary protocols (LCP, Link Control Protocol, and NCP, Network Control Protocol) in order to configure and maintain the link up. These packets do not reach the NDIS upper layer because they are generated by the underlying software levels, therefore the network tap cannot capture them. The particular extension of the Linux kernel for packet capture, Packet Socket, suffers from the same problem because PPP packets are handled inside the PPP device and not handed to the main networking code. Vice versa, BPF tap is able to capture everything that is being transmitted on the wire because the call to the tap is drowned in the `write()` and `read()` functions of the NIC driver.

The interaction with NDIS is quite complex and this makes the NPF more complicated than the original BPF: BPF interacts with the system through a single callback function; on the other hand, NPF is a part of the protocol stack and it interacts with the OS like any other network protocol. However, NPF is able to obtain excellent results and the tap is even faster than the BPF one. Like BPF, also NPF applies the filter when the packet is still into the NIC driver memory. Another optimization is the synchronous behavior of the NPF. Asynchronous calls are not supported, therefore user-level accesses are always blocking. It follows that NPF does not require a queue in which user-level request are buffered, thus making the driver faster. Moreover a carefully engineered NIC driver can help performance because only a portion of the packet can be transferred from NIC memory to the NDIS driver; a full copy can be avoided if no protocol drivers require the packet.

NPF demonstrates that a "high level" and "clean" capture driver, built coherently with the rest of the networking code, can be as fast as an ad-hoc implementation. Moreover, it proves that it is possible to add an efficient capture facility to an OS without modifying the structure and the interfaces of the kernel, which is essential in commercial systems.

4.3 User-level libraries and porting issues

Porting `libpcap` to Win32 has been relatively easy because of the compatibility between the interfaces exported by BPF and NPF. Basically we had to create a new `pcap-win32.c` file[8] with Windows-dependent code and to write some headers (like those present in the BSD kernel tree in the `net` and `netinet` folders) and system calls (`getnetent()`, `getnetbyname()`, etc.), not available through the Windows Sockets API. This porting resides mostly in the *WPcap.dll* module, which uses the *packet.dll* file instead of accessing directly the NPF. In this way, capture applications that use WinPcap (like WinDump) are independent from the Windows version on which they are running. Since WinPcap is not only a `libpcap` porting, it adds some Win32 specific functions in a separate folder. This guarantees to be able to integrate next releases of `libpcap` as well.

WinPcap has some differences from `libpcap` because of the OS. For example, Win32 applications cannot use the `select()` function on a NPF device in order to know if there are packets that needs to be read, because this function does not work on a normal file in Windows. WinPcap implements a new event, shared between kernel and user mode, that provides a result equivalent to the `select()` one.

5 Performance

This Section aims at giving some indications about the performance of the WinPcap architecture on different OSes. Results obtained by WinPcap on Windows 98 and Windows 2000 are compared to `libpcap/BPF` on FreeBSD 4.1.

The testbed (shown in Figure 3) involves two PCs

[7] NDIS allows the creation of another type of network driver, called intermediate driver, which is located between NIC and protocol drivers. However the implementation of the NPF as an intermediate driver does not bring any advantage unless in some limited cases. What it is certain, instead, is that intermediate drivers are not supported by earlier version of NDIS specification (3.0/3.1) and that their architecture is more complex than protocol drivers.

[8] `Libpcap` keeps major differences among various operating systems in a special file called `pcap-XXX.c`, where XXX is the name of the operating system.

directly connected by means of a Fast Ethernet link. This assures the isolation of the testbed from external sources, therefore allowing more accurate tests. A Windows 2000 workstation generates traffic using a custom tool based on WinPcap, guaranteeing high data rates. Packet sizes have been selected in such way to generate the maximum amount of packets per second, which is usually the worst operating situation for a network analyzer. Average value has been calculated by repeating the tests several times in order to get accurate results.

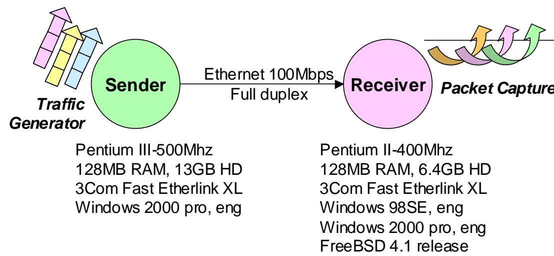

Figure 3. The testbed used in the tests.

OSs under test are installed in different disk partitions on the same workstation to avoid differences due to the hardware. Traffic is sent to a non-existent host in order not to have any interaction between the workstations. The second PC sets the interface in promiscuous mode and captures the traffic using different receiving tools according to the test objectives. Depending on the test, packets received are either dropped by filter, transferred to the application or saved to disk.

The CPU load has been measured by using the `top` program in FreeBSD, the task manager in Windows 2000 and `cpumeter` [6] in Windows 98. The first two tools are included in the operating system, while the third one is available on the Internet.

All the software under test was in the latest available release[9]. Kernel buffer was the default one (1Mbytes) in WinPcap, while `libpcap` was modified to use two buffers of 512Kbytes (instead of the standard 32KB ones) in order to exploit the same amount of kernel memory.

Even if these tests tend to isolate the impact of each subsystem (filtering, copying overhead), results are not able to show exactly the performance of the single component. This is due to the impossibility to isolate each component from interacting each one with the others and with the OS.

5.1 Sending process

First test aims at evaluating the performance of the sending process. This has been done only on Windows 2000 because Windows 95/98 code has not been optimized for that. Figure 4 shows that maximum number of packets per second has been reached when packet size is 88 bytes. This is rather surprising because we should expect the maximum number of packets per second when packet size is minimized (64 bytes on Ethernet). However this does not depend on the NPF so we did not make any further investigation. CPU load never reaches 100% (the sender machine is still ready to accept other commands) and this confirms that the NPF is not the bottleneck. Ethernet is loaded at almost full speed starting from packet size of 400 bytes.

However our findings confirm that the sending process is largely NIC-dependent. A different network card (with a different NIC driver), tested in the same conditions, was able to guarantee maximum 30K packets per seconds.

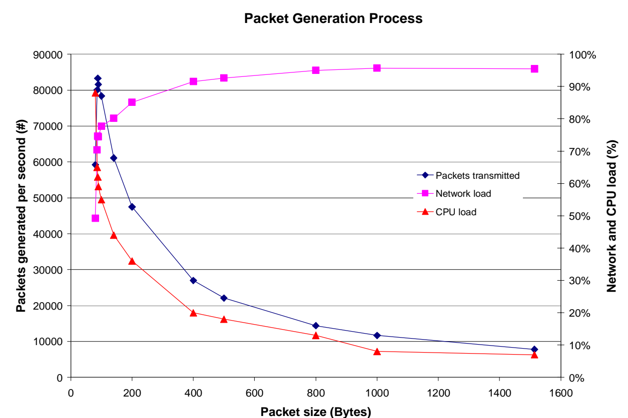

Figure 4. Maximum number of packets per second and corresponding network load.

5.2 Network tap and filtering process

Second test aims at the evaluation of the network tap and the filtering process. Packets are received by the network tap and checked by the filter. Since no packets satisfy the filter, they are discarded without being copied to the application. The filtering code is executed for each packet; therefore CPU usage shows the impact of the filtering process on the system under test. This test has been repeated in two cases: a first filter involved 3 BPF pseudo instructions, while a second one was more complex and involved 13 pseudo instructions[10] before failing.

Results (Figure 5) show that Windows flavors have similar behavior and almost all the packets are received and examined by the filter. Windows 98 consumes more CPU than Windows 2000, even if the load is still at acceptable levels. FreeBSD is by far the worst platform:

---

[9] WinPcap and WinDump (whose porting was based on `tcpdump` version 3.5.2 and `libpcap` version 0.5.2) version 2.1; `tcpdump` version 3.5.2, `libpcap` version 0.5.2; BPF was the one implemented in FreeBSD 4.1-RELEASE.

[10] Short filter was "`ip6`" and expensive filter was "`ether src 2:2:2:2:2:2 and ether dst 1:1:1:1:1:1 and ip and udp`". Packets were created to make the filter failing at the last pseudo-instruction.

the amount of packets captured is about one half of the number of packets sent and even the CPU load has the highest value.

Notice that, since BPF and NPF are quite similar at this level, part of the performance gap is probably due to the OS: in particular, Windows (and above all Windows 2000) seems to be faster in handling hardware interrupts and in all the operations made by NIC driver and NDIS code before calling the NPF tap.

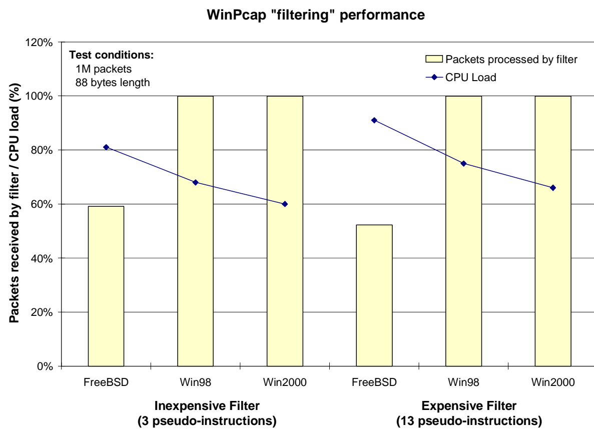

Figure 5. Network tap and filtering process performance.

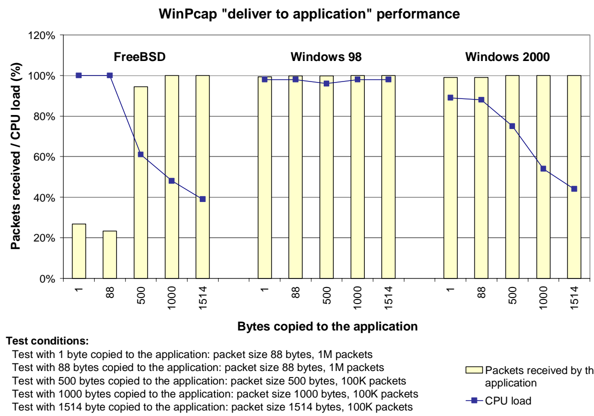

Figure 6. Delivering to the application performance.

5.3 Deliver-to-application performance

A third test involved the creation of a void application that receives packets from WinPcap and discards them without any further processing. This test aims at the evaluation of the entire WinPcap architecture, including the copy process from the interface driver to the kernel buffer and then to the user one. There were no filters in these tests. Results (Figure 6) show that WinPcap is able to deliver almost all the packets received by the network tap to the application and no packets are discarded (dropped) inside the kernel. Vice versa, FreeBSD is not able to bring all the packets to the application, especially in case of the highest values of packets per second. Most of the packets are lost without even reaching the filter; moreover some 20% of the received packets are then dropped by the BPF because the buffer is full.

As expected, CPU load decreases accordingly to the increase of the packet size in both FreeBSD and Windows 2000. Windows 98 has a different behavior because it does not have the *delayed-write* capability. This prevents the kernel from being able to wait for a minimum amount of data and copy a large block of data to user space within a single system call.

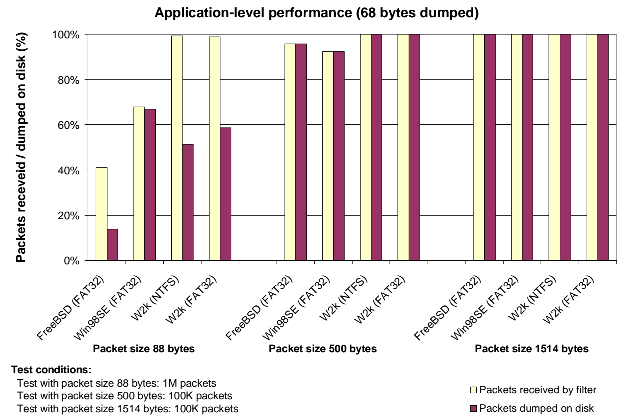

Figure 7. Performance of the complete capturing architecture, dumping 68 bytes of each packet to file.

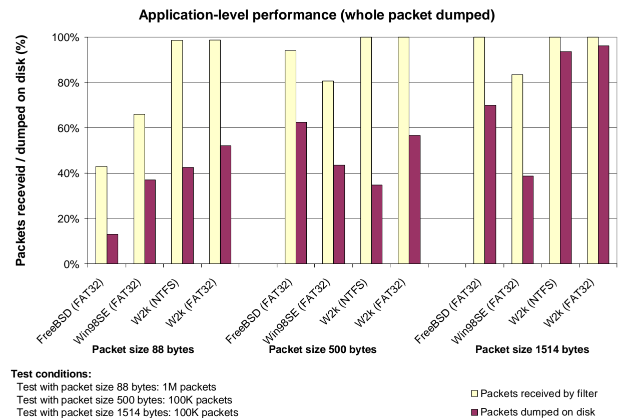

Figure 8. Performance of the complete capturing architecture, dumping the whole packet to file.

5.4 Application performance

Fourth test involves the use of a simple application that uses WinPcap to dump packets to file, which is a typical operation performed by a sniffer. Figure 7 shows the results when, for each packet, a "snapshot" of 68 bytes is saved to file. All the systems suffer noticeable losses when the network is overloaded by a large number of packets per second: a certain amount of packets is lost for the lack of CPU time (a new packet arrives when the tap was processing a previous one), while others are dropped

because the kernel buffer has no more space to store them. Figure 8 shows that results suffer of a non-negligible worsening when the whole packet is dumped to file.

5.5 Monitoring

An ad-hoc program has been used to test the monitoring capabilities of WinPcap. Our tests confirm that CPU load is considerably low and that the results match exactly the ones already shown in Figure 5. Figure 6 shows that the overall cost of a void application that captures at user level is far higher than kernel-mode monitoring. The addiction of a further cost due to the monitoring code will give to the user-level application even worse results. In addiction, user-level monitoring requires a non-negligible amount of memory.

6 Discussion

Tests confirm the excellent performance of WinPcap. The packet generation process, even in presence of a strange behavior (maximum number of packets per second is reached with 88-bytes packets) is highly optimized and it is quite easy to reach the maximum load allowed by a Fast Ethernet LAN. The capturing process also has an excellent implementation and it outperforms the original BPF / `libpcap` implementations. Tests with Windows 2000 show usually better results than the Windows 98 ones because of the larger number of optimizations present in that driver (for example the *delayed write* capability).

While the third test is the one that gives an indication of the overall performance of the WinPcap library, the fourth test is the most interesting from the end-user point of view. This test confirms also that other part of the OS (the most noticeable is the file system managements) may have an importance that is far larger than the packet capture components. In fact, Windows 2000 is able to receive almost all the packets on the network even if a non-negligible part of them are discarded because the impossibility to save them on disk. From this perspective, FreeBSD (that saves data on the same FAT32 partition of Windows 98 and 2000) shows excellent results compared to the ones obtained in the previous tests. The comparison among Windows 2000 saving on an NTFS or FAT32 partition shows that the file system technology itself is of primary importance for the overall performance of the capture process.

Notice that WinPcap has been used with the standard kernel buffer (1MB); in presence of heavy traffic the size of this buffer can be increased by the application through a simple function, improving noticeably the overall performance of the system. Vice versa, `libpcap` does not offer a method to set the kernel buffer and we had to modify it “by hand” in order to set it properly.

These tests show the excellent implementation of the NPF as well as the validity of architectural choices, like the circular kernel buffer instead of the original buffering architecture, the *delayed write* implementation and the *update-space-during-copy* in the kernel buffer. Among the supported platforms, Windows 2000 is the best one for high performance network analyzers. FreeBSD performance are rather surprising: we repeated all the tests with standard (32Kbytes) buffers but we did not get the differences that we expected; a large size for the kernel buffer does not seem to be able to influence substantially the performance of the capture process.

WinPcap has been proved being an excellent choice for the several applications that are based on high performance packet filtering on Win32 platforms.

Acknowledgements

We wish to thank all the people that, in several ways, collaborated with this project. First of all, Piero Viano, Giampiero Alberelli, Fabio Magliano, Antonio Lantieri and Giovanni Meo, who started the implementation as a course work. Thanks to all the Internet people (especially the tcpdump.org and Ethereal mailing lists) who sent us suggestions, bug fixing and source code that helped us to release a better tool. Finally, we would like to thank Microsoft Research that partially supported our work.

Bibliography

[1] V. Jacobson, C. Leres and S. McCanne, libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release June 1994. Available now at http://www.tcpdump.org/

[2] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proceedings of the 1993 Winter USENIX Technical Conference (San Diego, CA, Jan. 1993), USENIX.

[3] Gary R. Wright, W. Richard Stevens, TCP-IP illustrated Volume 2. Addison-Wesley professional computing series.

[4] Mike D. Schiffman, Libnet Packet Assembly System. Available at http://www.packetfactory.net/Projects/Libnet/.

[5] Microsoft Corporation, 3Com Corporation, NDIS, Network Driver Interface Specification, May 1988.

[6] Ricardo Thompson (icardoth@interserver.com.ar), Cpumeter, available on the Internet at http://www.winsite.com/info/pc/win95/sysutil/cpumet12.zip, 1997.

[7] M. Yuhara, B. Bershad, C. Maeda, J.E.B. Moss, Efficient packet demultiplexing for multiple endpoints and large messages. In Proceedings of the 1994 Winter USENIX Technical Conference, pages 153-165, San Francisco, CA, January 1994.

[8] Dawson R. Engler, and M. Frans Kaashoek, DPF: fast, flexible packet demultiplexing, in Proceedings of ACM SIGCOMM '96.

[9] A. Begel, S. McCanne, S.L.Graham, BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture, Proceedings of ACM SIGCOMM '99, pages 123-134, September 1999.

[10] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall (Network Flight Recorder, Inc.), Implementing a Generalized Tool for Network Monitoring, LISA 97, San Diego, CA, October 26-31, 1997.

[11] PCAUSA, Rawether, available at http://www.rawether.net.