Loss Analysis of the Software-based Packet Capturing

Tamás Skopkó
University of Debrecen, Hungary, Debrecen, Hungary, skopkot@unideb.hu

***Abstract*— Gigabit per second and higher bandwidths imply greater challenge to perform lossless packet capturing on generic PC architectures. This is because of software based capture solutions, which did not improve as fast as network bandwidth and they still heavily rely on the OS's packet processing mechanism. There are hardware and operating system factors that primarily affect capture performance. This paper summarizes these parameters and shows how to predict packet loss ratio during the capture process.**

Index Terms — Linux, Software-based packet capturing, libpcap, Wireshark, Communication system traffic.

I. INTRODUCTION

PACKET capturing is an essential part of network traffic monitoring. For network engineers there are several monitoring solutions that allow traffic sampling or capturing. Most advanced tools are capable of passive monitoring that does not affect the monitored traffic itself [1]. Using these hardware-accelerated monitoring devices the traffic of large capacity network backbones can be monitored on-the-fly. These tools feature hardware timestamping of packets and lossless capturing at wire-speed and they are quite expensive. The most professional tools can perform distributed traffic monitoring at multiple points of the network infrastructure. For accurate timestamping, they have to be able to synchronize their internal clocks. Software developers, application programmers and researchers are often interested only in the traffic of a specific host or application and don't need most of the extra functionalities mentioned above. As an alternative, software-based packet capturing is available on most OS's and most architectures.

Timestamps can be generated at well-defined points within the kernel during the data processing. On Linux systems with *MMAP* support, timestamping is performed at enqueuing to or dequeuing from the input packet queue [2]. On other systems (e.g., Windows) timestamps are generated at later processing phases that affect its accuracy. This is not necessarily bad, it depends on what the developer or software engineer wants to know about the packet's arrival time. If she is interested in the time moment when the packet reaches an upper protocol layer (like IP or above), the timestamp generated during software-based packet capturing delivers closer arrival times to the appropriate layer.

In cross-layer protocol analysis, packet flows through the endpoint's network layers are as interesting as their way through the network infrastructure. From the point of view of an application developer or protocol researcher, timestamps generated in the kernel seem to be more adequate.

Libpcap-based packet analyzer applications (e.g., *Wireshark* [3]) are very common, since they are available on a wide range of systems and most of them are free and open source. Using these tools, packets can be saved for later analysis into PCAP or PCAPng file. These formats are flexible, since they are supported by most of the traffic analysis softwares.

II. PROBLEM DEFINITION

Software-based capture solutions did not improve as fast as network bandwidth and computing hardware. However 1 ns timestamp resolution can be reached on a lot of systems, packet processing and capturing overhead is still significant. We cannot therefore expect linear scalability by improving only a specific system parameter (i.e., CPU frequency).

Dumpcap itself uses relatively small system resources, however it is executed on a general purpose system that shares its resources between the running processes. Accordingly, it is not trivial to separate packet processing and capturing tasks from any other system and user processes, even on systems with multiple processors. Unfortunately, most of the device drivers in the Linux kernel are not able to share the processing task of a single NIC queue between two or more CPU cores [4]. Since the trend is to add more cores to a CPU rather than raising its frequency, serialized packet processing is a serious limit for the scalability of software-based packet capturing.

Although general client applications don't invoke intense network traffic, server applications handling large number of users and data flows, can produce excessive network traffic even on a single interface. Moreover, flow identification is generally done only during offline analysis of the saved traffic, thus filtering traffic to specific flows cannot be done during the capturing process.

The goal of this paper is to present a method to determine and improve the packet capturing performance

of a generic Linux system. It seems to be trivial that we hardly can expect lossless performance on fully saturated 1 Gbps+ links with generic hardware layouts, but if we can consider and adjust the most important system parameters, we may capture all the packets of our interest.

A measurement system has been set up and was driven to the extremes by targeted network traffic to imitate aggregated network connections.

III. The Linux Packet Processing

A. Data flow of software-based capturing

Figure 1 shows the schematic of data flow during packet capturing. Frames are received from the physical layer by the NIC driver. It pushes them towards to the kernel. The drivers typically operate in two different modes. In interrupt mode the kernel is interrupted after receiving a certain number of packets. In polling mode the kernel queries the NIC driver about the received frames. These two modes can be combined to better adopt the traffic intensity and optimize system load and this technique is named NAPI on Linux [5][6]. The importance of these modes is dual. Firstly, since software timestamps are generated at the enqueueing process, they reflect that time instead of the moment of arrival at the physical layer. Moreover, polling causes bursty packet forwarding towards the kernel and thus there is a need for appropriate sized buffers to handle them.

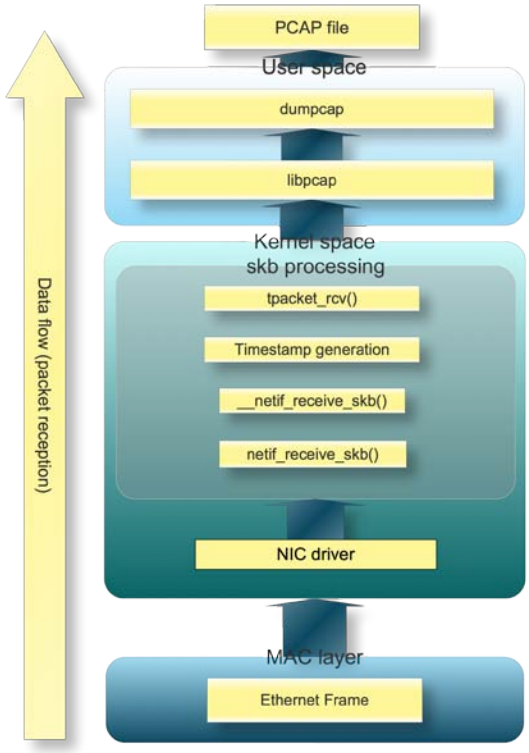

Figure 1. Data flow from physical layer to the libpcap-based capture application

After enqueueing, packets are available for user applications. Dequeueing is the process when they receive their own packets from the kernel. And also, after the capture application can access the packets. Using libpcap in MMAP-supported Linux systems this invokes filling up packet metadata called *TPACKET_V2* and copying data from kernel to user space [6].

B. Packet buffers

The packets pass through a number of buffers (queues) while they get saved. The very first buffer is the memory on the NIC itself. Its size is relatively small and cannot be altered. The first in-RAM queue is the RX ring buffer in the NIC driver. If the driver supports it, its size can be changed using the *ethtool* application. In most of the drivers one NIC queue can be assigned to one CPU [6].

In case of non-NAPI drivers, the next queue is the *backlog* in the network stack. This is a per-CPU buffer storing the packets before they get processed by the kernel. Its size can be specified by the kernel parameter *net.netdev_max_backlog*. NAPI-enabled drivers work different, they are not sensitive to the backlog buffer setting but the *net.core.netdev_budget* that controls the number of packets being transferred during one polling cycle.

TABLE I.
PACKET PROCESSING-RELATED KERNEL PARAMETERR

Parameter and purpose	Default value
net.netdev_max_backlog: Size of the pre-processing queue of the kernel	1000
net.core.netdev_budget: Maximum number of packets taken from an interface during one polling cycle	300
net.core.wmem_max: Maximum size of the general packet transmission buffer size (all packet types).	131071
net.core.rmem_max: Maximum size of the general packet reception buffer size (all packet types).	131071
net.core.wmem_default: Default size of the general packet transmission buffer size (all packet types).	126976
net.core.rmem_default: Default size of the general packet reception buffer size (all packet types).	126976
net.core.optmem_max: Maximum ancillary buffer size allowed per socket	20480

Further buffers may be present on the processing path, depending on the upper layer protocols. If any of the buffers becomes full, packets will be lost. Table I shows the most important kernel parameters associated with kernel level packet buffers. Their default values on the measurement system in this paper are also specified (see Section IV).

The applications open sockets for their network connections. Each connection has its own buffers, for both sending and receiving. Their size can be controlled at OS

level by the parameters *net.core.rmem_default* and *net.core.wmem_default* and can be limited by the *net.core.rmem_max* and *net.core.wmem_max*.

Packets can contain ancillary data related to protocols (e.g., timestamps) [4]. The OS controls an additional buffer for handling this information, which can be limited via the *net.core.optmem_max* parameter.

IV. The Measurement

In measurement Session 1, packet processing and capturing performance of the measurement PC was investigated. In Sessions 2 to 4, improvements targeting the performance are presented and evaluated. Though the synthetic traffic generated during the sessions does not necessarily show a real life scenario, it makes the measurements repeatable and also makes possible to reach the performance boundaries faster.

A. Environment description

For the measurements, a simple network configuration with two direct attached endpoints has been set up. One of the endpoints was a generic PC, the other one was a custom FPGA-based packet generator device. The generator hardware ran a dedicated firmware for packet transmission: it listens for an UDP packet with the command describing the size and amount of Ethernet frames to be sent, as well as the size of inter-frame gap (IFG) between them. Despite the existence of a large number of network performance testing software, I decided to use a dedicated generator device, since its primary purpose is to produce packet flows with a precise timing in hardware [7]. It guarantees that no packet will be lost at the generator side and they will be sent with uniform interval.

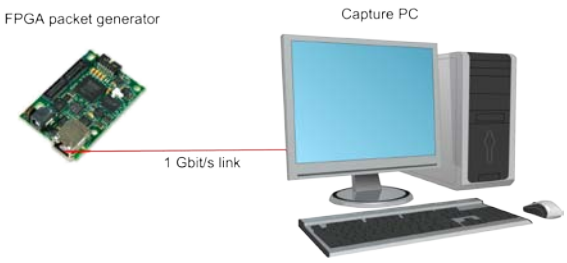

Figure 3. The measurement setup

The PC is based on an Intel Core-i7 870 CPU with 4x1 GB 1333 MHz DDR3 memory on the motherboard. It also featured an Intel 82571EB PCI Express Gigabit Ethernet NIC card. As operating system, Linux kernel version 2.6.39.2 was used, the NIC card was driven by the *e1000e* module. The driver operates in NAPI mode and module parameters were left at their default values [8]. However this NIC has multiple hardware RX/TX queues, the driver allows only one queue per interface. Driver design change would be needed to make use of multiple input queues in the kernel [9].

TSC clocksource was selected for lowest overhead timestamping [10]. Traffic was captured using the *dumpcap* application of the *Wireshark* 1.6.2 package. CPU load was monitored with the *mpstat* tool. The network connection was a direct link through 2 meters of an AMP Cat6A patch cable.

B. Session 1

Similarly to every following session, this one is made up from series of measurements. In every step, capturing performance of the traffic of a specific packet size and IFG was monitored. Packet sizes ranged from 72 to 1200 bytes, incremented by 8. IFG was fixed at 12 bytes. The packet size contains the frame headers without the preamble and checksum fields. Each measurement was run for at least 5 seconds to surely fill the buffers. All of the kernel parameters related to packet buffers were kept at their default values.

In lossless term, capture performance showed poor results, almost regardless of packet size. Kernel packet loss (continuous red line) shows the ratio of lost packets against generated packets, dropped by the kernel since those packets could not be processed in time. Dumpcap packet loss (continuous blue line) represents the ratio of dropped packets against kernel-processed packets, after they were processed and enqueued by the kernel but they were not processed in time by the capture application itself.

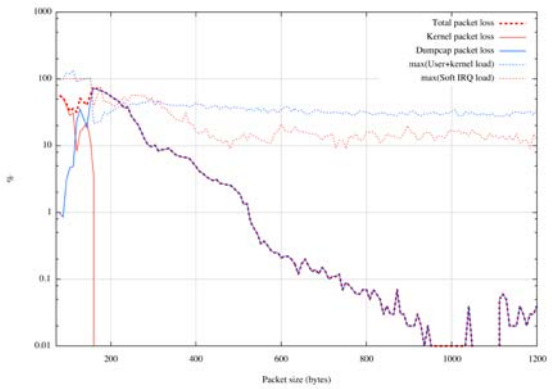

Figure 2. Results of measurement Session 1 - Packet loss ratio and system load

At small packet sizes (below 168 bytes) heavy system load was detected on the cores doing packet processing due to the complex capturing tasks. Since these processes are not parallelized in the current software environment (see Section II), only two of the 8 available cores participated the job. We can see that at the smallest packet sizes, dumpcap lost very few packets. Because most of the packets were dropped before enqueueing, dumpcap had to process relatively small number of packets, so it was able to do its task. The larger packet sizes the fewer kernel packet drops are occurred.

Of course, when capturing larger packets, fewer packets were lost. However, my expectation for such a powerful

system was lossless packet capture at least for larger packet sizes.

I summarized user and kernel space load because of dumpcap involves user as well as kernel space loads (two separate processes while packet data is copied from kernel to user space). And this is also the reason why this sum is sometimes higher than 100 %.

C. Session 2

In this session, I tried to ease dumpcap's tasks. During traffic monitoring and cross-layer protocol analysis, whole packet data is slightly needed. We can truncate them to a specific size and still keep important header and protocol data. This step reduces the amount of memory copies and makes processing faster. Dumpcap has an option called *snaplength* to do truncation. I repeated the measurements for three different snaplength sizes: 256, 128 and 96 bytes. Greater snaplength sizes may hold application protocol headers, but 96 bytes are still enough to keep at least transport layer information. A developer or researcher should consider the depth of inspection.

If compared to the original measurement (constant red line), the smaller snaplength the fewer lost packets by dumpcap. Of course, for smallest packet sizes this option does not really help since there is not much data to be cut down (Fig. 4).

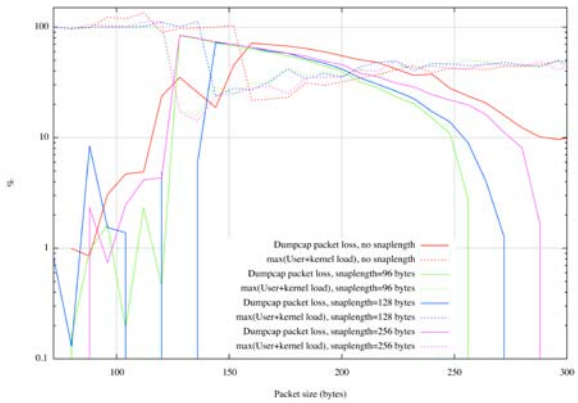

Figure 5. Results of measurement Session 2 - Snaplength option efficiency

We should note that summarized user and kernel CPU load caused by the application is not significantly reduced. This means that high CPU load is not the only cause for packet losses. Small buffers may also raise difficulties.

D. Session 3

Since realtime monitoring of the buffer state is not easy to implement, I had to estimate the values of the scale parameters to extend kernel buffer sizes by. Default buffer sizes may deliver good performance in a typical desktop environment with low intensity connections.

During packet capturing, additional higher level buffers are utilized by the capture application in promiscuous interface mode. For traffic intensity in the measurements, buffers in the kernel had to be extended 64-fold to significantly improve the performance at kernel level: a more intense traffic (consisting of smaller packets, with continuous green line) could be processed with less loss (Fig 5). However, total packet loss (dashed green line) was still high.

TABLE II.
MODIFIED KERNEL BUFFER PARAMETERS FOR EXTENDED BUFFERS

Parameter	Modified value
net.core.netdev_budget	19200
net.core.wmem_max	8388544
net.core.rmem_max	8388544
net.core.wmem_default	8388608
net.core.rmem_default	8388608
net.core.optmem_max	1310720

In the second half of this measurement session, the application buffer has also been extended. Dumpcap's internal buffer is called capture buffer. It is primarily intended to compensate the latency of I/O subsystem during writing the trace data to disk. Since I needed statistical information instead of the trace file itself, I used memory disk during the capturing according to the technique in [11]. This fact makes storage performance a less relevant question in this paper. Table II shows the modified values of the corresponding kernel parameters.

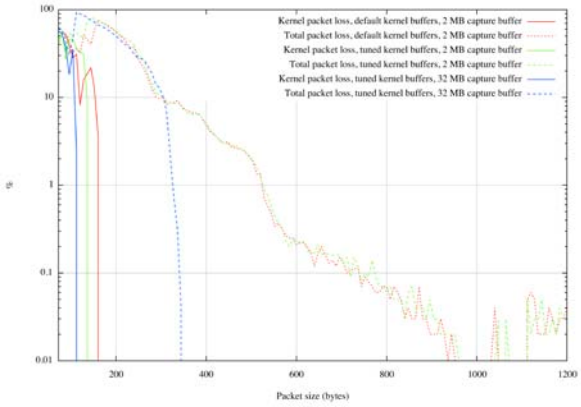

Figure 4. Results of measurement Session 3 - The effect of extended kernel and capture buffers

Since the parameter net.netdev_max_backlog applies only to non-NAPI drivers, in this measurement it was left at the default value. Instead, net.core.netdev_budget has been increased to reduce the number of interrupts and allow more packets to be transferred from the device driver to the kernel at one polling cycle.

Default capture buffer size of 2 MB was extended to 32 MB. This caused unexpected reduction of kernel-related small-sized packet loss (continuous blue line). This step made a serious improvement, since total packet loss became more predictable and the traffic of 352-byte packets at wire speed could be all captured (dashed blue line).

The improvement during kernel buffer adjustments gives to make a conclusion that default kernel buffer parameters could not compensate bursty processing of small sized packets within the kernel.

The result of this session showed again that the more small sized packets enqueued the more dropped by dumpcap.

E. Session 4

In the final measurement session, snaplength option was combined with buffer adjustments. Figure 6 represents the worst versus the best scenario: the original setup with default buffer sizes and without using snaplength option is compared to the measurements with tuned buffers at kernel and application levels and also

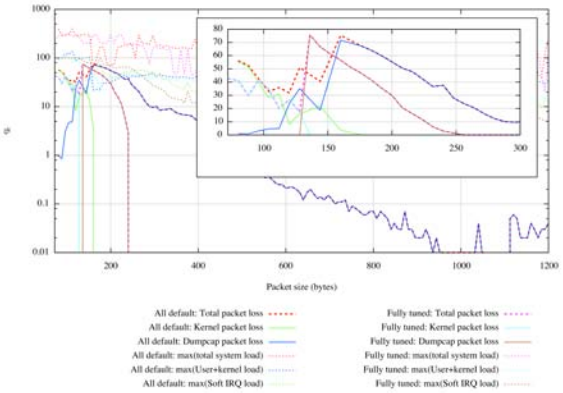

Figure 6. Results of measurement Session 4 - The combined effect of snaplength option and extended buffers

using 96-byte snaplength in dumpcap.

The improvement is the most significant compared to any previous sessions. Varying packet loss ratio at all packet sizes got stabilized and capturing performance got more reliable even at small packet sizes of 200 bytes. After the optimizations, this system could capture traffic of packets greater than or equal to 256 bytes without loss. Although packet loss at small packet sizes could not be reduced greatly, mean packet loss ratio became almost linear.

V. CONCLUSION

During the measurement series, a method for determining the capture performance of a generic Linux PC has been introduced. Reasons of packet loss have been revealed and factors affecting packet processing and capturing performance have been summarized. The measurements showed that default buffer settings are not enough for 1 Gbps+ software-based packet capturing at wire speed. Performance bottleneck caused by the lack of parallel packet processing could be successfully compensated by reducing the amount of memory copies. After optimizing the buffers and enabling snaplength functionality, capture performance was greatly improved and became more predictable.

Future design of NIC drivers should utilize the power of multiple CPU cores. However, multi queuing improves packet processing and capturing performance only if the monitored traffic consists of relatively large number of flows.

ACKNOWLEDGMENT

The work was supported by the TÁMOP 4.2.2.C-11/1/KONV-2012-0001 project. The project was implemented through the New Széchenyi Plan, co-financed by the European Social Fund.

REFERENCES

[1] Endace DAG board. [Online]. Available: http://www.endace.com

[2] P. Orosz and T. Skopko, “Timestamp-resolution problem of traffic capturing on high speed networks,” January 28-30, 2010, ICAI international conference, Eger, Hungary

[3] Wireshark Network Protocol Analyser. [Online]. Available: http://www.wireshark.org

[4] The Linux Kernel. [Online]. Available: http://www.kernel.org

[5] P. Orosz, T. Skopko, and J. Imrek, “Performance Evaluation of the Nanosecond Resolution Timestamping Feature of the Enhanced Libpcap,” 6th International Conference on Systems and Networks Communications, ICSNC 2011, October 23-28, 2011, Barcelona, Spain, ISBN 978-1-61208-166-3, Proceeding p. 220-225.

[6] C. Benvenuti, "Understanding Linux Network Internals", ISBN 0-596-00255-6, 2005

[7] A. Shriram, M. Murray, Y. Hyun, N. Brownlee, A. Broido, M. Fomenkov, and k. claffy, “Comparison of Public End-to-End Bandwidth Estimation Tools on High-Speed Links", in Passive and Active Network Measurement Workshop (PAM), Boston, MA, Mar 2005, vol. 3431, pp. 306--320, PAM 2005.

[8] Intel 1 GbE NIC driver manuals. [Online]. Available: http://www.intel.com/

[9] Yi, P.P. Waskiewicz Jr.: “Enabling Linux Network Support of Hardware Multiqueue Devices”, 2007 Linux Symposium Vol. Two

[10] P. Orosz and T. Skopko, “Performance Evaluation of a High Precision Software-based Timestamping Solution for Network Monitoring,” the International Journal on Advances in Software, ISSN 1942-2628, 2011 Vol 4. No. 1 & 2 p. 181-188.

[11] Y. Klonatos, M. Marazakis, and A. Bilas “A Scaling Analysis of Linux I/O Performance”, Poster at ACM EuroSys Conference, 2011.