Differentiating packet capture mechanisms in Proceedings of IIT.SRC 2019

Tomáš BELLUŠ*

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
xbellust@fiit.stuba.sk

Abstract. This research paper, as a part of Bachelor's thesis, deals with analyzing various network traffic capture tools or libraries for Linux-based systems and choosing the most efficient one for real time network traffic. The main goal is to differentiate between native libraries (e.g. Libpcap), low-level protocol families (e.g. PF_PACKET/AF_PACKET, PF_RING) and other capture mechanisms with possible zero-copy/one-copy features, because efficiently keeping track of network traffic indications, is crucial for security and administrative reasons.

1 Introduction

The key difference in efficient network capture is

- the count of redundant copies of received packet data
- packet post-processing bypass
- buffers shared between kernel and user space.

Distinguishing the most efficient capture mechanism consisted of testing `the elapsed time` of capturing a static count of packets at consistent packet rate and testing for any drop rate of captured packets within a predefined time at various packet rates (`packet capture efficiency`). Testing network topology (see Figure 1) consists of two equal machines with following specifications.

- Operating system: `Ubuntu server 18.04.2 LTS`
- Ethernet Controller: `Intel(R) 82579LM Gigabit Network Connection`
- Process: `Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz, boost 3.70GHz, 4 cores`
- Memory: `8 GB DDR3, 1333 MHz`

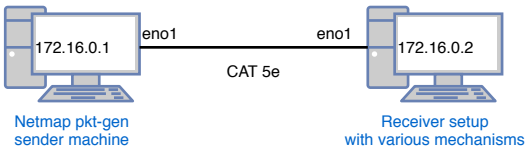

Figure 1. Testing topology

2 Packet capture

The most critical part is real time data capture in network traffic, meaning capturing network packets using a framework or capture library. It is a dependent process of hardware components like Ethernet Controller (NIC), processor performance and its properties (e.g. number of processor cores). Packet capture takes place in the NIC, but packet processing takes place in higher system layers, like the user space for Libpcap library. More efficient mechanisms share data from memory, which falls under the jurisdiction of lower system layer - the kernel space. Though, why is it more effective and efficient to access data in lower system layers? Kernel space is the operation system's (OS) core and an interface to hardware for the OS. More precisely, it is the access to shared kernel space buffers, which altogether bypasses the additional copying and processing present with basic frameworks not utilizing efficient features.

Received data on NIC is stored in NIC buffers. NIC registers keep track of whether the buffers are full and ready for reading or sending. This indication is handled by interrupts from the NIC to kernel. Kernel copies the buffer content to kernel space buffers (called *m_buffs* or *sk_buff*[1]). These buffers are not

---

* Bachelor study programme in field: Computer Engineering
Supervisor: Ing. Dušan Bernát, PhD., Institute of Computer Engineering and Applied Informatics, Faculty of Informatics and Information Technologies STU in Bratislava

[1] These kernel buffers are expensive to create and vary in size. They are complex and include large amount of metadata. [4]

accessible by user processes, but can be copied to the user space or the process accessible memory. This is a typical scenario

- two copies
- interrupts between kernel and user process to access received data on the NIC.

Possible improvements is to use, as mentioned previously, buffers shared between the kernel and the user space, which eliminates one or more copies and interrupts from kernel to user space, leaving only one copy between NIC buffers and shared buffers. Additional improvement is to have a NIC supporting multiple buffers, which splits the load and each buffer is handled by different core simultaneously [3].

A problem arises with the ability to capture all traffic on wire with no increasing delay resulting in packet loss due to lack of buffer space. Solution may be a zero-copy mechanism which utilizes a NIC dependent direct NIC access (DNA) feature or a one-copy mechanism, which utilizes the shared buffers.

3 Mechanisms

Tested mechanisms in scope of this research paper are

- *PF_PACKET*[2] protocol family
- standard Linux library *Libpcap*[3]
- open source framework *Netmap*[4]
- *PF_RING*[5] protocol family by *ntop*

3.1 PF_PACKET

A protocol family or file descriptor (FD) introduced to utilize the shared buffer is bypassing the Linux network stack and the packet socket FD receives full Ethernet frames. This packet socket is utilized by Libpcap since v0.6 release [5].

3.2 Libpcap

Standard Linux packet capture library utilized by Wireshark and Tcpdump. On datalink layer it captures packets with PF_PACKET and in its user space library it breaks down the packet to structures for further processing by the user process.

3.3 Netmap

Netmap targets the processing bottleneck in copying data from NIC to shared buffers. Netmap has its own *pkt_buffs* replacing standard *sk_buffs*, which are preallocated in comparison with *sk_buffs*' constant reallocations [3]. It includes *netmap* kernel module utilizing modified NIC driver kernel module.

3.4 PF_RING

On the socket layer it replaces PF_PACKET. It utilizes ring buffers in shared memory and modified NIC driver kernel modules and the *pf_ring* kernel module. Through Linux New API (NAPI)[6] the kernel module copies polled packets from the NIC to shared ring buffers accessible by user application [2] Enhanced packet capturing is dependent of Intel NICs, but it works for other NICs too.

4 Performance testing

Setting up the testing environment consisted of two bootable Ubuntu server USB sticks with persistent storage. Both with installed Netmap framework with Netmap kernel module utilizing Netmap's patched NIC kernel module matching the original (see Listing 1). Machine generating packets (sender) was setup for running the Netmap's *pkt-gen*. Machine capturing packets (receiver) had, in addition to the Netmap framework, a compiled Netmap-based Libpcap and additionally had configured PF_RING, its modified Libpcap and installed corresponding library *libpfring*.

Listing 1. Netmap setup list of commands for NIC kernel module e1000e

```
sudo apt install linux-source
git clone https://github.com/
    luigirizzo/netmap dev/netmap
cd dev/netmap/LINUX
./configure --no-ext-drivers --
    kernel-sources=/usr/src/linux-
    source-4.15.0 --drivers=e1000e
make && sudo make install
sudo rmmod e1000e
sudo insmod ./netmap.ko
sudo insmod ./e1000e/e1000e.ko
```

[2] http://man7.org/linux/man-pages/man7/packet.7.html

[3] http://www.tcpdump.org/manpages/pcap.3pcap.html

[4] http://info.iet.unipi.it/~luigi/netmap/ or https://github.com/luigirizzo/netmap

[5] https://www.ntop.org/products/packet-capture/pf_ring/ or https://github.com/ntop/PF_RING

[6] NAPI is a performance increase for packet capturing at higher packet rates, by enabling packet polling, therefore decreasing number of interrupts which otherwise would overwhelm the CPU [1]

Using *ethtool*, the network device features are disabled (-K option) for Netmap to correctly interact with its ring buffers. Secondly, the pause frames (option -A) are also disabled, which negotiate the allowed packet rx/tx frequency and therefore limits the receiving of transmitting process. (see Listing 2)

Listing 2. Machine network setup to maximize capturing for both machines

```
sudo ethtool -K eno1 tx off rx off
    gso off tso off gro off
sudo ethtool -A eno1 autoneg off tx
     off rx off
```

After setting up the sender, it could generate up to 1.389 million packets per second (Mpps) with exact options shown in Listing 3.

Listing 3. pkt-gen command executed on sender machine

```
sudo pkt-gen -i eno1 -c 4 -f tx -l
    60 -w 3 -d 172.16.0.2 [-R <rate
    >pps]
```

Table 1. pkt-gen options

option	description
-i	interface used
-c	number of CPUs
-f	function
-l	packet size
-w	wait for link time
-d	destination IP address
-R	packet rate

The receiver was setup with PF_RING as listed in Listing 4 and for both Netmap and PF_RING with compiled modified Libpcaps in their respected directories.

Listing 4. PF_RING setup list of commands for NIC kernel module e1000e

```
git clone https://github.com/ntop/
    PF\_RING dev/pfring
cd dev/pfring/kernel
make && sudo make install
sudo insmod ./pf_ring.ko
cd ../drivers/intel && make
cd e1000e/e1000e-*-zc/src
sudo ./load_driver.sh
# set up libpfring and libpcap
cd dev/pfring/userland/lib
./configure && make && sudo make
    install
cd ../libpcap && ./configure &&
    make
sudo make install
```

4.1 Packet capture efficiency

Testing the packet capture efficiency comprised of sender executing the pkt-gen command (see Listing 3) with

- rate of 1000 pps (see Table 2)
- and maximum rate of 1.389 Mpps (see Table3).

The receiver was capturing packets for 120 seconds and returning basic statistics of captured and dropped packets. For every capture mechanism the process was tested 5 times.

Table 2. Packet capture efficiency median results for 1000 pps

mechanism	packets captured	packets dropped
Libpcap	119963	0
PF_PACKET	120640	0
Netmap	119971	0
PF_RING	119964	0

Table 3. Packet capture efficiency median results for 1.389 Mpps

mechanism	packets captured	packets dropped
Libpcap	86880719	1003
PF_PACKET	117737210	9879085
Netmap	92077024	1143
PF_RING	21476253	0

Dropped packets is read from the mechanism's statistics output (e.g. *getsockopt* or *pcap_stats*), although considering that total packets sent was

$$1.389\ Mpps * 120\ s = 166.68\ Mpkts \quad (1)$$

4.2 Elapsed time of captured packets

Testing the elapsed time (measured in nanoseconds) comprised of sender executing the pkt-gen command (see Listing 3) with no rate provided (resulting rate 1.389 Mpps) and on the receiver side changing the number of packets to be captured. The results are shown in Table 4.

Table 4. Elapsed time of captured packets with median results

mechanism	1K pkts [ms]	100K pkts [ms]	10M pkts [s]
Libpcap	2.292	140.531	13.847
PF_PACKET	5.060	505.313	50.768
Netmap	2.112	129.603	12.965
PF_RING	6.814	388.881	38.673

5 Conclusion

Referring to Packet capture efficiency (see Section 4.1) for the rather small packet rate all mechanisms captured 100% of generated packets. For the maximum packet rate the most packets were captured by PF_PACKET, which are raw Ethernet frames and would still require post processing. On the other hand Netmap-based Libpcap captured the most packets from the Libpcap-based mechanisms.

Although PF_PACKET captured to most packets in the previous test, it was not the fastest mechanism. Netmap managed to capture any number of packets tested in the shortest period of time (see Table 4). PF_PACKET could have been the fastest, but since measuring elapsed time was implemented in python, it might have influenced its performance.

Altogether, Netmap-based Libpcap has shown improvement to its native counterpart in both tests. PF_RING was surprisingly inefficient, which could have been due to improper opening of the network device in Libpcap. Worth mentioning are the native APIs of both Netmap and PF_RING, which are much more efficient and worth further testing.

Acknowledgement: This work was supported by the Ministry of Education, Science, Research and Sport of the Slovak Republic within the Research and Development Operational Programme for the project "University Science Park of STU Bratislava", ITMS 26240220084, co-funded by the European Regional Development Fund. The work was also partially supported by the Slovak Research and Development Agency (APVV-15-0789).

References

[1] corber: NAPI. https://lwn.net/Articles/30107/, 2003, [21.11.2018].

[2] ntop: Vanilla PF RING. https://www.ntop.org/products/packet-capture/pf_ring/, [20.11.2018].

[3] Rizzo, L.: netmap: a novel framework for fast packet I/O. Technical report, Universita di Pisa, https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf [17.11.2018].

[4] Rizzo, L.: netmap: A Novel Framework for Fast Packet I/O. https://www.youtube.com/watch?v=la5kzNhqhGs, 2012, USENIX conference presentation [24.11.2018].

[5] tcpdump group, T.: Summary for 0.6 release. https://github.com/the-tcpdump-group/libpcap/blob/libpcap-0.6.1/CHANGES, [19.11.2018].