Improving Passive Packet Capture: Beyond Device Polling

Luca Deri
NETikos S.p.A.
Via Matteucci 34/b
56124 Pisa, Italy
Email: luca.deri@netikos.com

Abstract

Passive packet capture is necessary for many activities including network debugging and monitoring. With the advent of fast gigabit networks, packet capture is becoming a problem even on PCs due to the poor performance of popular OSs. The introduction of device polling has improved the capture process quite a bit but not really solved the problem.

This paper proposes a new approach to passive packet capture that combined with device polling further improves it and allows, on fast machines, packets to be captured at (almost) wire speed.

Keywords

Passive packet capture, device polling, Linux.

1.Introduction

Many network monitoring tools are based on passive packet capture. The principle is the following: the tool passively captures packets flowing on the network and analyzes them in order to compute traffic statistics and reports including network protocols being used, communication problems, network security and bandwidth usage. Many network tools that need to perform packet capture [tcpdump] [ethereal] [snort] are based on a popular programming library called *libpcap* [libpcap] that provides a high level interface to packet capture. The main library features are:

- Ability to capture from various network media such as ethernet, serial lines, virtual interfaces.
- Same programming interface on every platform.
- Advanced packet filtering capabilities based on BPF (Berkeley Packet Filtering), implemented into the OS kernel for better performance.

Depending on the operating system, libpcap implements a virtual device from which captured packets are read from user-space applications. Despite different platforms provide the very same API, the libpcap performance changes significantly according to the platform being used. On low traffic conditions there is no big difference among the various platforms, whereas at high speeds the situation changes significantly. The following table shows the outcome of some tests performed using a traffic generator [tcpreplay] on a fast host (Dual 1.8 GHz Athlon, 3Com 3c59x ethernet card) that sends packets to a mid-range PC (VIA C3 533 MHz, Intel 100Mbit ethernet card) connected over a 100 Mbit Ethernet switch (Cisco Catalyst 3548 XL) that is used to count the real number of packets sent/received by the hosts. The traffic generator reproduces at full speed (~80K pkt/sec) some traffic that has been captured previously, whereas the capture application is a simple application named *pcount* based on libpcap that counts and

discards, with no further analysis, the captured packets.

Traffic Capture Application	Linux 2.4.x	FreeBSD 4.8	Windows 2K
Standard Libpcap	0.2 %	34 %	68 %
Mmap Libpcap	1 %		
Kernel module	4 %		

Table 1. – Percentage of captured packets (generated by tcpreplay)

The experience learnt from this experiment is:

- At 100 Mbit the simplest packet capture application is not able to capture everything (i.e. there is packet loss).
- During the above experiment, the host used for capturing packets was not responsive when the sender injected packets on the network.
- Out of the box, Windows and *winpcap* [winpcap], the port of libpcap to Win32, perform much better than other popular Unix-like OS.
- Linux, a very popular OS used for running network appliances, performs very poorly with respect to other OSs used in the same test.
- *Libpcap-mmap* [libpcap-mmap], a special version of libpcap that exploits the mmap() system call for passing packets to user space, does not improve the performance significantly.
- Implementing a Linux kernel module based on *netfilter* [netfilter] improves the performance significantly, but still under Linux most of the packets are lost. This means that Linux spends most of its time moving packets from the network card to the kernel and very little from kernel to user-space.

An explanation for the poor performance figures is something called *interrupt livelock* [Mogul]. Device drivers instrument network cards to generate an interrupt whenever the card needs attention (e.g. for informing the operating system that there is an incoming packet to handle). In case of high traffic rate, the operating system spends most of its time handling interrupts leaving little time for other tasks. A solution to this problem is something called *device polling* [Rizzo]. Polling is a technique for handling devices, including network cards, that works as follows:

- When a network device receives a packet it generates an interrupt to request kernel attention.
- The operating system serves the interrupt as follows:
    - It masks future interrupts generated by the card (i.e. the card cannot interrupt the kernel).
    - It schedules a task for periodically polling the device to service its needs.
    - As soon as the driver has served the card, the card interrupts are enabled.

Although this technique may seem not to be effective, in practice it gives the operating system control over devices and it prevents devices from taking over control over the kernel in case of high traffic rate. FreeBSD implements device polling since version 4.5 whereas Linux has introduced it with NAPI (New API) part of kernel 2.6. Note that the network device driver must be polling-aware in order to exploit device polling. The

author has repeated the previous traffic experiment on Linux and FreeBSD (Windows does not seem to support device polling nor to provide facilities for enabling it) using device polling with the following outcome.

Traffic Capture Application	Linux 2.6 with NAPI	FreeBSD 4.8 with Polling
Standard Libpcap	5.6 %	99.9 %
Mmap Libpcap	Not Working	
Kernel module	99.5 %	

Table 2. – Percentage of captured packets (generated by tcpreplay) using kernel polling

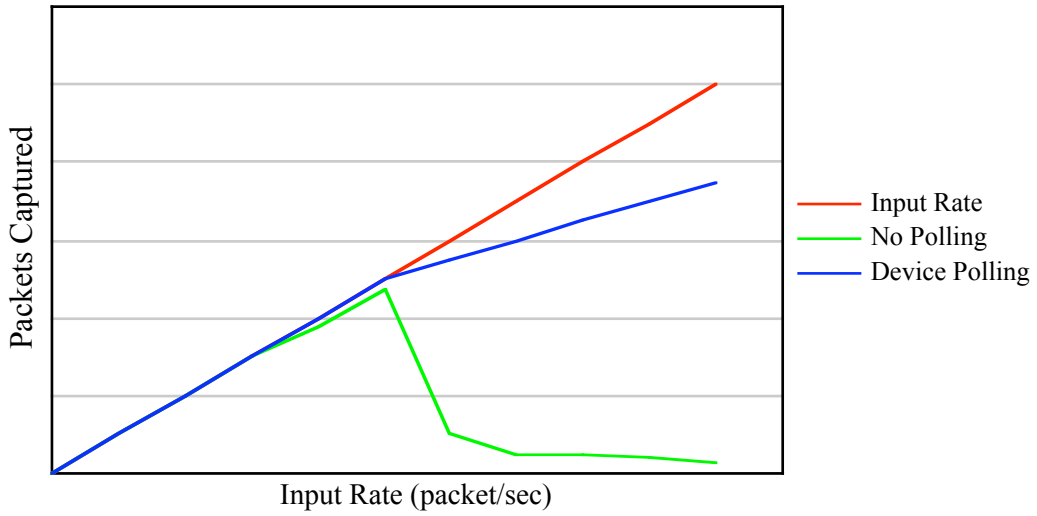

Figure 1. – Packet Capture Performance: Polling vs. non- polling

The experience learnt from this experiment is:

- Device polling is a very good solution to both improves packet capture performance and system responsiveness under high traffic load.
- Even with kernel polling, FreeBSD performs significantly better than Linux running the same userspace libpcap-based application.
- The Linux kernel module is almost as fast as the userspace FreeBSD application.

The following chapter explains why device polling is just the starting point and not the ultimate solution.

2. Beyond Device Polling

As shown in the previous chapter, the very same simple libpcap-based application performs significantly different on Linux and FreeBSD. Using on the same experiment setup a DoS (Denial of Service) application named *stream.c*[1] as traffic generator, the gap

[1] stream.c can be downloaded from http://www.securiteam.com/unixfocus/5YP0I000DG.html.

between the two operating systems is even larger.

Traffic Capture Application	Linux 2.6 With NAPI	FreeBSD 4.8 With Polling
Standard Libpcap	0.8 %	74.7 %
Mmap Libpcap	Not Working	
Kernel module	9.7 %	

Table 3. – Percentage of captured packets (generated by stream.c) using kernel polling

After running all the experiments, the author realized that:

- Linux needs a new speed bump for efficiently capturing packets at high network speeds.
- FreeBSD performs much better than Linux, but still at 100 Mbit it looses more than 25% of the incoming packets.
- Kernel device polling is not sufficient for capturing (almost) all packets.

Doing some further measurements, it seems that most of the time is spent moving the packet from the adapter to the user-space through the kernel data structures and queues. The mmap-version of libpcap reduced the time spent moving the packet from the kernel to user-space but has not improved at all the trip of the packet from the adapter to the kernel. Therefore the author has designed a new packet capture model based on the following assumptions and requirements:

- Given that network adapters are rather cheap hence that is not too costly to allocate a network adapter only for passive packet capture, the goal is to maximize packet capture performance and not reduce the overall costs.
- As the adapter does not have to send packets, the time spent checking whether there's something to transmit can be avoided simply disabling data transmission.
- Device polling proved to be very effective, hence it should be exploited to improve the overall performance.
- For performance reasons, it is necessary to avoid passing incoming packets to the kernel that will pass then to userspace. Instead a straight path from the adapter to the user space needs to be identified in order to avoid the kernel overhead.

The idea behind this work is the following:

- Create a circular buffer directly into the adapter driver and not into the kernel as libpcap-mmap does.
- The buffer is allocated when the adapter driver is initialized and deallocated then the adapter is deactivated.
- Whenever a packet is received from the adapter (usually via DMA, direct memory access), the driver copies it into the circular buffer instead of passing it to the kernel (on Linux this is implemented by the netif_rx). If the buffer is full the packet is discarded.
- The circular buffer is exported to userspace applications via a special file that supports the mmap system call.

- Userspace application that want to access the buffer, open the file, then call mmap on it in order to obtain a pointer to the circular buffer.
- The kernel copies packets into the ring and moves the write pointer forward. Userspace applications do the same with the read pointer.
- New incoming packets overwrite packets that have been read by userspace applications. Memory is not allocated/deallocated by packets read/written to the buffer but just overwritten.
- The buffer length and bucket size is fully configurable by the user when the driver is initialized.

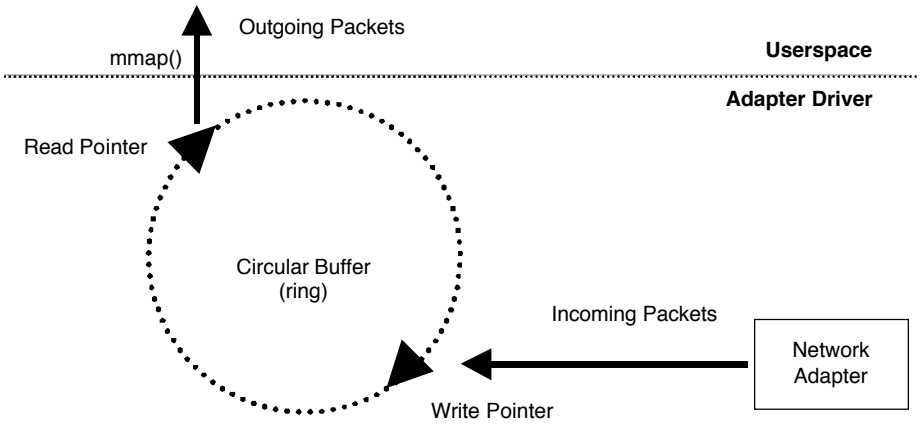

Figure 2. Adapter Driver Architecture

The advantage of a circular buffer located into the driver is many, including:

- Packets are not queued into kernel network data structures.
- The mmap primitive allows userspace applications to access the circular buffer with no overhead due to system calls such as in the case of socket calls.
- Even with kernel that does not support device polling, under strong traffic conditions the system is usable. This is because the time necessary to handle the interrupt is very little compared to normal packet handling.
- Implementing packet sampling is very simple and effective as sampled packets do not need to be passed to upper layers then discarded as it happens with conventional libpcap-based applications.

The main difference with respect to normal packet capture is that applications that rely on libpcap need to be recompiled against a modified (ring/mmap-aware) version of the library as incoming packets are stored into the buffer and no longer in the kernel data structures. For this reason the author has implemented a subset of the libpcap on top of the circular buffer in order to read packets from the buffer and not from the real network adapter.

In order to evaluate the performance of the circular buffer, the author has implemented the architecture on Linux 2.6 modifying the Intel Gbit ethernet driver[2] that uses NAPI.

[2] The original Intel code is available at http://sourceforge.net/projects/e1000/.

The modified driver has been tested in the same environment used previously in order to evaluate the performance advantage with respect to the original driver. The following table shows the test outcome.

Traffic Capture Application	Linux 2.6 with NAPI	Linux 2.6 with NAPI and Circular Buffer	FreeBSD 4.8 with Polling
Standard Libpcap	5.6 %		99.9 %
Mmap Libpcap	Not Working		
Kernel module	99.5 %		
Ring-based Libpcap		> 99.9 %	

Table 4. – Percentage of captured packets (generated by tcpreplay) using kernel polling

Basically the new driver has captured everything with the exception a few packets that are probably lost at the beginning of the capture process because if the test is repeated several times with a various number of packets the number of lost packets is more or less constant. At 100 Mbit, using this driver, there is no longer a performance gap between FreeBSD and Linux as it used to be, and basically the PC can now capture all the traffic. An interesting test outcome is that a ring-based application in the userspace is (slightly) faster that the same application coded into the kernel.

A userspace application that has to access a mapped memory buffer can do it in two ways: with or without ring polling.

Packet Handling with Polling

```
fd = fopen("/proc/net/ring", "rw");
..
ringBufferPtr = mmap(NULL, ringSize,
                PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, 0);

slotId = &ringBufferPtr->slotId;

while(TRUE) {
  /* Loop until a packet arrives */
  if(ringBufferPtr->slot[slotId].isFull) {
    readPacket(ringBufferPtr->slot [slotId]);
    ringBufferPtr->slot [slotId].isFull = 0;
    slotId = (slotId + 1) % ringSize;
  }
}
```

Packet Handling without Polling

```
fd = fopen("/proc/net/ring", "rw");
..
ringBufferPtr = mmap(NULL, ringSize,
                PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, 0);

slotId = &ringBufferPtr->slotId;

while(TRUE) {
  if(ringBufferPtr->slot [slotId].isFull) {
    readPacket(ringBufferPtr->slot [slotId]);
    ringBufferPtr->slot [slotId].isFull = 0;
    slotId = (slotId + 1) % ringSize;
  } else {
    /* Sleep when nothing happens */
    pfd.fd = fd;
    pfd.events = POLLIN|POLLERR;
    pfd.revents = 0;
    poll(&pfd, 1, -1);
  }
}
```

Table 5. – Packet Retrieval in userspace Applications: poll() vs. polling

From the performed tests, it seems that the use of polling at user-space is not really effective in terms of performance gain. After repeating the same test several times, the conclusion is that the use of user-space polling does not seem improve the overall performance, or at best that the performance gain is not easy to measure. For this reason, all the ring-buffer tests have been performed using the poll() system call that has the advantage of keeping the system load low with respect to user-space polling that exhausts all the available CPU cycles.

In order to better evaluate the performance of the circular buffer, a new test setup on a Gbit link has been setup. The traffic collector is the same PC used in the previous tests equipped with an Intel Gbit ethernet card, whereas the traffic sender is a Sun Fire 480U running several instances of the DoS application previously used. The Sun sent 4 million packets at more that 160.000 pkt/sec (~480Mbit) towards the collector. The following table shows the test outcome.

Traffic Capture Application	Linux 2.6 with NAPI and Circular Buffer	FreeBSD 4.8 with Polling
Standard Libpcap		3.7 %
Ring-based Libpcap	< 0.01 %	

Table 6. – Percentage of packet (generated by stream.c) loss using kernel polling

This test outcome shows that:

- The previous assumption on the ring buffer was correct: the Linux circular buffer implementation looses a little number of packets per run, mostly at the beginning of the capture process. FreeBSD, while still performing very well, is loosing many more packets than Linux on the same test experiment.
- At Gbit speeds, Linux with the circular buffer captures almost everything with very little packet loss on a low-end collector PC.
- The implemented solution significantly both reduced the trip of the packet from the network card to the traffic capture application, and decreased the load on the kernel.
- The combination of Gbit card/driver is more efficient than the 100 Mbit card/driver. This is also because modern cards implement interrupt mitigation: the driver can instrument the card to send one interrupt each n packets, hence reducing the driver work.

3. Future Work

Currently, the circular buffer has been implemented on Linux. However from performance tests (e.g. see table 3) it is obvious that a vanilla FreeBSD system is much more efficient than a vanilla Linux system when used for packet capture. The author

plans to port his work to FreeBSD to see whether the performance improvement measured on Linux can be replicated under such operating system.

Other work items include:

- The evaluation of this work at full Gbit speeds and above, to understand how the circular buffer performance changes with higher network speeds.
- The positioning of this work with respect to commercial cards such as the DAG card [dag].
- Evaluate what is the performance improvement moving the BPF filter code to the driver so that packet filtering is performed directly into the driver.

4. Final Remarks

This paper has described the design and implementation of a new approach to packet capture whose goal is to improve the performance of the capture process at Gbit speeds. The validation process has demonstrated that:

- A circular buffer in the network driver has significantly improved the overall capture process at both 100 Mbit and Gbit.
- Userspace applications based on the circular buffer are faster than an equivalent kernel module not using the ring code.
- Packet loss is very minimal if any even using PCs with limited CPU power.

5. Availability

The circular buffer code is distributed under the GPL2 license and can be downloaded free of charge from the ntop home page (http://www.ntop.org/) and other mirrors on the Internet.

6. Acknowledgment

The author would like to thank Charles Spurgeon <c.spurgeon@mail.utexas.edu> for his challenging requirements, suggestions and support throughout this research work.

7. References

[dag] *The DAG Project*, Univ. of Waikato, http://dag.cs.waikato.ac.nz/.

[ethereal] G. Combs, *Ethereal*, http://www.ethereal.com/.

[libpcap] Lawrence Berkeley National Labs, *libpcap*, Network Research Group, http://www.tcpdump.org/

[mogul] J. Mogul and K. Ramakrisnan, *Eliminating Receive Livelock in an Interrupt-Drivern Kernel*, Proc. Of 1996 Usenix Technical Conference, 1996.

[netfilter] H. Welte, *The Netfilter Framework in Linux 2.4*, Proceedings of Linux Kongress 2000.

[pcap-mmap] P. Wood, *libpcap-mmap*, Los Alamos National Labs, http://public.lanl.gov/cpw/.

[rizzo] L. Rizzo, *Device Polling Support for FreeBSD*, http://info.iet.unipi.it/~luigi/polling/, BSDConEurope Conference, 2001

http://info.iet.unipi.it/~luigi/polling/, BSDConEurope Conference, 2001.

[salim] J. Salim and others, *Beyond Softnet*, http://www.cyberus.ca/~hadi/usenix-paper.tgz, Proceedings of 2001 Usenix Annual Technical Conference, Boston, 2001.

[snort] M. Roesch*, Snort – Lightweight Intrusion Detection for Networks*, Proceedings of Usenix Lisa '99 Conference, http://www.snort.org/.

[tcpreplay] A. Tuner and M. Bing, *tcpreplay*, http://tcpreplay.sourceforge.net/.

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

[winpcap] F. Risso and L. Degioanni, *An Architecture for High Performance Network Analysis*, Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001), July 2001.