Comparing PFQ: A High-Speed Packet IO Framework

Dominik Schöffmann
Betreuer: Sebastian Gallenmüller
Seminar Innovative Internet Technologies and Mobile Communications (IITM) WS2015/16
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: schoeffm@in.tum.de

ABSTRACT

This paper discusses the PFQ framework built for high-speed data transfers on an x86 platform. In order to facilitate the understanding of such a framework the low level mechanics of the Linux kernel are discussed. This includes historic approaches and the state-of-the-art situation. PFQs internal workings are discussed including its functional engine which is unique to this framework. The main concepts of similar frameworks are explained and compared to PFQ. Conducted measurements of PFQs performance are reviewed and compared to other measurements done by the frameworks creator.

Keywords

PFQ Linux network Internet framework

1. INTRODUCTION

The Internet is transferring more and more data by the minute. Traditionally middle boxes such as routers and firewalls were built using specialized hardware in order to speed up the processing and hence be able to handle the growing load.

A more recent approach is to use commodity hardware such as Intel x86 platforms. However normal Operating Systems (OS) kernels are not able to provide the speed needed to fully exploit the hardware provided network link speed.

Furthermore modern networking hardware is able to take away load from the CPU, for example by computing the ethernet CRC32 checksum in hardware. Another improvement is the support of multiple packet queues which allow better utilization of multi-core processors.

In order to actually use all these new possibilities, high performance frameworks need to be developed, tested and used.

The purpose of this paper is to test a framework called PFQ and compare it against other frameworks. In Section 2 the low-level mechanics of the existing network stack of the Linux kernel in which PFQ partly lives are explained. The framework itself is discussed in Section 3. Comparing PFQ to other frameworks is done in Section 4. Section 5 presents the conducted measurements and evaluates the result with respect to previous performance tests.

2. THE LINUX KERNEL

The PFQ framework mostly works inside of the Linux kernel. Also the packet retrieval from the network interface is handled in the standard Linux way. This section covers the communication between the Linux kernel and the Network Interface Controller (NIC), by explaining how this was achieved in the past and today.

2.1 Softnet

In the early days of the Linux kernel the Internet did not consist of networks capable of transferring as much data in a short period of time as it does nowadays. As an addition, the hardware design was much simpler. Most notably there were not as much multi-core processors, as today. As a result multithreading the network stack inside of the Linux kernel was no priority, even thread safety was not given until it was introduced in Linux 2.0 [1]. Thread safety was provided by using a mutex which only allowed one thread to operate inside of the network context.

All this changed with the Linux kernel version 2.3.43 in which multi-core support was added allowing multiple processors to concurrently work on network traffic as it came in from a NIC. The patchset which contained these changes was called “Softnet” [1].

One problem which led to excessive packet loss in high traffic situations however remained. At some point the hardware has to inform the OS, that there are one or more packets to be handled. Meanwhile the network card stores the packets in a DMA memory ring. Up to this point a hardware interrupt was used to signal this event for every incoming packet. This behavior is suitable when packets only arrive every once in a while, but needless and even harmful if lots of packets get received. As soon as a hardware interrupt is caught by the processor it needs to be handled immediately, delaying the work which is currently being done. As observed by Salim et al. [1] this could lead to an unequal usage of computing resources between the kernel and the user space. Therefore packets may have been queued, but never actually been processed by a process interested in these packets in the first place.

2.2 NAPI

As seen in chapter 2.1 issuing an interrupt for every single packet which arrives at the network card does not yield a very good performance for high traffic environments. The exact opposite of this approach would be to use no interrupts at all, but to periodically poll the network card for packets. Obviously the second way will oftentimes add unnecessary latency to the further usage of the transmitted data [1].

The "New API" (NAPI) as presented by Salim et al. [1] provides a hybrid of these two worlds. When the network card is initialized, it is configured to emit an interrupt as soon as a packet arrives. Once this event occurs the interrupt for incoming packets gets disabled and the NIC is inserted into a queue of NICs having unprocessed packets. At some point in time the OS decides to handle the queued interfaces and thus the waiting packets. After all the packets from this interface are retrieved, the interrupt is re-enabled. Packets get dropped if the OS is not capable to schedule the processing of the DMA ring while it has still space left. As soon as it runs out of space no further packets can be written into the RAM, these newly arriving packets are therefore lost.

This behavior mimics the two extremes outlined before in extremely high or low traffic situations. If only a few packets arrive with a long enough time distance an interrupt is send for each packet. When a lot of packets are arriving in a rapid succession the system basically reverts to polling. Thus a good middle ground was found between these two mechanisms.

Up to now the Linux kernel uses the NAPI.

3. INNER WORKINGS OF PFQ

After understanding how the Linux kernel works for retrieving packets from network interfaces, the next upper layer of the used software stack is the PFQ framework itself.

3.1 General Structure

PFQ works inside of the NAPI context making use of the standard way to retrieve packets from network cards. When receiving a packet on the link, PFQ performs multiple steps in order to make this packet accessible to a user space program.

Figure 1 provides an illustration of the content which will be discussed in the following paragraphs.

First there is the so called *packet fetcher* [2] which operates on single packets. The one thing the *packet fetcher* does is saving a pointer to the packet inside of the *batching queue* [2]. Internally the packets are still represented by a *sk_buff* struct as used by the kernel. The whole point of the *packet fetcher* is to speed up the processing afterwards in the next stages by only working on batches of packets instead of processing every single packet on its own.

The *batching queue* is the input to the *packet steering block* [2]. Inside of the *packet steering block* it is decided to which socket the packet is forwarded. Alternatively a packet can be fed back into the Linux kernel or discarded completely. During this process the *functional engine* (discussed in section 3.2) is active. As an output location of the *packet steering block* the *socket queue* is used.

The *socket queue* is the interface between the kernel world and the user space world [2]. It is realized as a wait-free double buffer. While one buffer is being processed by a user space thread, the other buffer is used for storing new packets which are currently coming into the system. This process is facilitated by mapping the buffers into the appropriate memory spaces.

The *user space sockets* are the only instances which only operate in the user space and not inside of the kernel [2]. These sockets provide a way for threads to retrieve packets.

In order to speed up the low-level operations an additional component called an "aware driver"[2] can be used. This kind of driver provides only small code changes to the original vanilla driver. When using such a special driver the Operating System kernel does no longer receive network input. All the incoming packets are processed by PFQ. [2]

At this point it should be noted, that the normal Linux kernel would need to do a lot more preprocessing then PFQ does. For example PFQ does not implement IP reassembling or checks if a TCP segment actually belongs to a currently open connection. It does only provide a thin interface between the networking hardware and the user space processing.

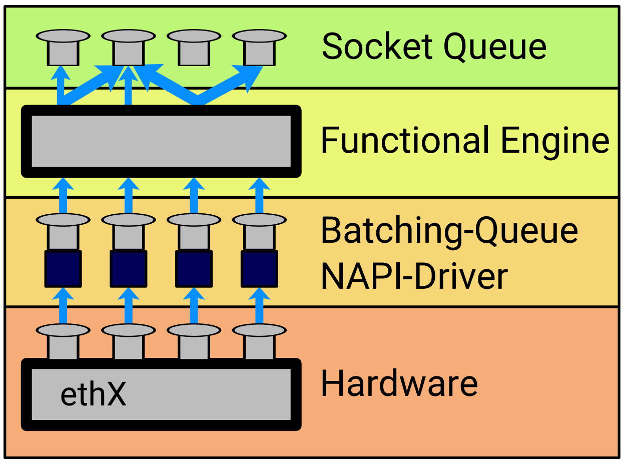

Figure 1: Inner Structure of PFQ, Image was redrawn and adapted, original by Bonelli et al. [2]

3.2 Functional Engine

The functional engine is a feature implemented within the *packet steering block*. This engine determines what to do with a packet. Possible options are: dropping, feeding it back into the kernel, sending it to a group of sockets, all sockets, one randomly chosen socket or forwarding it to a specific socket (a group with only one member socket). If a group is chosen as the destination, it may be decided if the whole group receives the packet, or only one random member, which thus results in a load-balancing situation. The used language is called "PFQ-Lang" and was first introduced by Bonelli et al. [3].

Using a functional approach inside of the kernel provides the capability to run checks against the program in order to verify properties like guaranteed termination (no loops) or that all types correct [3]. This is useful in order to avoid crashes inside of the kernel.

Inside of the program multiple processing steps can be performed, which can result in a conclusion in which direction the packet should be destined. If multiple different decisions are made during this process the last decision is used. One special case is a drop, if a packet is set to be dropped, later steps cannot overwrite this with another action. These steps are also called a “processing pipeline” [3] inside of the functional engine.

A decision on how to handle a specific packet is made by properties of the packet. These properties include the protocols used in the layers 3 and 4. Furthermore source and destination addresses and ports can be investigated. More detailed information about the packet, for example if it is fragmented, can also be used.

Having a filtering mechanism like this, which does not crash and operates inside the kernel does promise a comparably good speed. Therefore middlebox software seems like a sensible application for this type of feature.

4. COMPARISON WITH OTHER FRAMEWORKS

Comparing the basic mechanisms of PFQ to those of other frameworks is important in order to evaluate the performance and fitness for some purpose.

4.1 Netmap

Similar to PFQ Netmap also operates inside the Operating Systems kernel, although most of Netmap is based in the user space. Contrary to PFQ running Netmap means, that the network interface on which Netmap is used is no longer usable for the normal kernel. This limitation is only enforced if an application uses the Netmap framework, otherwise the interface behaves normal. Another difference is, that PFQ can work with vanilla drivers whereas Netmap requires patched drivers, which can be derived from the original Linux drivers [4]. Netmap basically works by letting the NIC write its incoming frames to the user space process memory [2].

4.2 DPDK

DPDK also exclusively uses the network interface which is switched into this exclusive mode whenever a special kernel module is loaded. Said kernel module is named “UIO” and serves as the network cards driver. The only responsibility the kernel module has, is to map the cards memory into the memory space of the process which wants to use DPDK. Obviously this process runs in the user space which means, that the framework parts inside of the kernel space are less then the ones PFQ runs inside of the kernel. This framework does not only provide a fast way to send and receive packets, but a complete framework to realize Data Plane Devices like routers or switches. One example feature which is important to such devices is an efficient implementation of longest-prefix-matching. [4]

4.3 PF_RING ZC

One feature which PFQ adapted from PF_RING ZC is the usage of aware drivers. PF_RING ZC actually was the first framework to propose such an approach [2]. Similarly to PFQ PF_RING ZC also uses shared memory rings for the kernel and the user space. The difference however is, that PFQ uses two such buffers which get swapped, whereas PF_RING ZC only has one buffer [2]. Measurements conducted by Bonelli et al. [2] showed, that PF_RING ZC does not perform as well as PFQ in regards to scaling up to multiple threads and hardware queues.

5. MEASUREMENTS

Measurements of the performance of PFQ were carried out. These include the two basic functions of such a framework, namely sending and receiving packets.

5.1 Setup

The measurements were conducted on two servers connected by a 10 Gbit/s ethernet link. As CPUs the first server used an Intel Xeon E3-1230, the other server an Intel Xeon E3-1230 V2. The used Network Interface Controllers (NICs) were an Intel 82599ES and an Intel 82599EB respectively. No further supporting kernel modules or patches to the network driver were used. The systems were running the Linux kernel version 3.19.

PFQ was compiled from source using the Glasgow Haskell Compiler (GHC) version 7.8.4. The version of PFQ itself was 5.0.4.

5.2 Traffic Generation

Using PFQ traffic was generated on the server with the Intel Xeon E3-1230 processor and the Intel 82599ES NIC. During the experiment the number of threads used for packet generation was incremented from 1 to 4 in steps of 1. The packet size was fixed to 60 Bytes. In another run the size of the packets was set to 128 Bytes, in order to measure if PFQ also performs well in situations with more realistic packet sizes. Using the second setup only 1 and 2 threads were tested. During this test hugepages were not mounted.

The test was performed with the bundled test tool “pfq-gen” using the command line `pfq-gen -l 60 -R -t 0.5.ethtest0 -k 1,...`

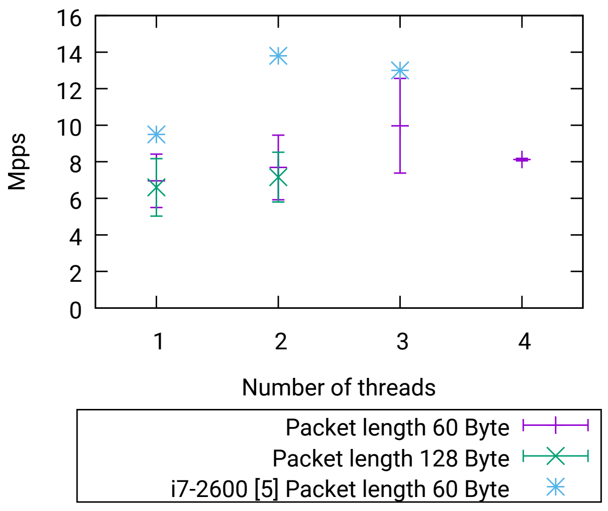

Figure 2: Packet generation

As can be seen in Figure 2, using more than one core does help to generate packets at a faster rate. The framework does also scale with rising packet sizes, since the sending rate did only decrease inside of the error margin. During the experiment it was also observed, that the sending rate dropped below the average at multiple occasions, leading to a quite high standard derivation which is also shown in Figure 2. Notably the average sending rate peaked with three cores and descended with the usage of four cores.

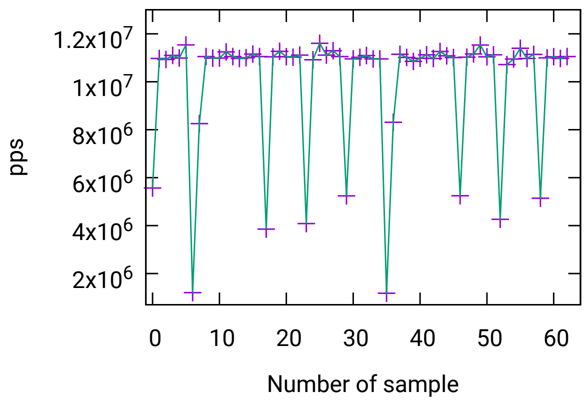

Figure 3: Packet generation on 3 cores

Figure 3 shows the performance of PFQ in packets-per-second for every taken sample of the measurement with 3 cores. It can be observed, that most samples are on one level above 10 Mpps and only a few downwards directed peaks are featured in the graph. These peaks occur relatively regular.

5.3 Traffic Capturing

Capturing traffic was performed on the Intel Xeon E3-1230 V2 processor and the Intel 82599EB Network adapter. The traffic was generated by PFQ running on 3 cores and building packets with a size of 60 Bytes. During the measurements the amount of used hardware queues was raised from 1 to 4. Each queue was bound to one processor core, which results in a multithreading situation inside of the kernel. In contrast to the traffic generation measurement hugepages were mounted.

Analogous to the traffic generation the used tool also was inside of the PFQ test suite. It is called "pfq-counter" and the used command was `pfq-counters -c 64 -t 0.5.ethtest0`. Before issuing the command the PFQ kernel module was reloaded with the appropriate queue number.

Figure 4 illustrates that the packet capturing capabilities of PFQ rise in a linear fashion with the number of queues used. As in the traffic generation test a significant derivation in the measured data was found.

5.4 Comparison with other measurements

Similar benchmarks were performed by Bonelli et al. and the results published in the projects wiki page [5]. Differences between the tests were the kind of processor, and software optimizations. The measurements in this paper were conducted with vanilla drivers, whereas Bonelli et al. [5] used

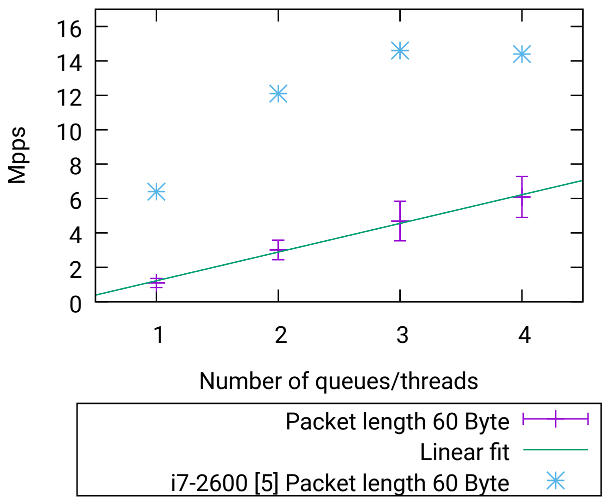

Figure 4: Packet capture

a driver which was optimized by the *pfq-omatic* tool. Furthermore Bonelli et al. loaded a special kernel module which provided support for Direct Cache Access. The used traffic generation and capturing tool were the same and even shared the same options. However the PFQ kernel module was loaded with different options. The measurements conducted by Bonelli et al. used the option "xmit_batch_len=128"[5], whereas the measurements done in this paper used the default value of one. Another big difference were the clock speeds of the used processors, which were lower in the here presented results. Using hugepages might as well have an impact on the performance (it is not stated if Bonelli et al. used hugepages, although this is very probable).

It can be observed, that the measurements conducted in this paper do not yield such a high performance as the previous measurements made by Bonelli et al. [5].

6. CONCLUSION

In this paper historical approaches to handling packets at a low-level were discussed. These included thread safeness and interrupts. Modern operating systems migrated away from issuing one interrupt per packet to issuing one interrupt per batch of packets.

Furthermore an overview of the building blocks of the PFQ high performance I/O framework was given. These included getting the packets from the low-level kernel space, steering them according to a functional engine and enqueuing them to be accessible by user space applications. The internals and capabilities of said functional engine were discussed.

Other frameworks promising similar functionality were examined for similarities and differences. Discussed were Netmap, DPDK and PF_Ring ZC. All provided the same basic functions like sending and receiving packets, but the methods achieving this do differ from one to another. All use some kind of kernel module, although the ratio of work done in the kernel space and the user space vary heavily.

Lastly measurements were taken to determine how many packets PFQ can send and receive on x86 commodity hardware. Inside of these measurements the number of used cores was variated. Within these measurements it was observed, that there were severe drops of the transmission rate. Another important note is, that when capturing traffic there is a linear correlation between the received packets and the number of hardware queues which PFQ was able to use.

7. REFERENCES

[1] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[2] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. On multi–gigabit packet capturing with multi–core commodity hardware. In *PAM'12 Proceedings of the 13th international conference on Passive and Active Measurement*, pages 64–73. Springer, 2012.

[3] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni. A purely functional approach to packet processing. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 219–230, New York, NY, USA, 2014. ACM.

[4] F. W. D. R. Sebastian Gallenmüller, Paul Emmerich and G. Carle. Comparison of frameworks for high-performance packet io. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2015.

[5] Nicola Bonelli. PFQ Benchmarks https://github.com/pfq/PFQ/wiki/Intel-IXGBE-10-20G (last retrieved 8.12.2015)