On the feasibility of 40 Gbps network data capture and retention with general purpose hardware

Guillermo Julián-Moreno
NAUDIT HPCN S.L.
Madrid, Spain
guillermo.julian@naudit.es

Rafael Leira
Univ. Autónoma de Madrid (UAM)
Madrid, Spain
rafael.leira@uam.es

Jorge E. López de Vergara
UAM & NAUDIT HPCN S.L.
Madrid, Spain
jorge.lopez_vergara@uam.es

Francisco J. Gómez-Arribas
UAM & NAUDIT HPCN S.L.
Madrid, Spain
francisco.gomez@uam.es

Iván González
UAM & NAUDIT HPCN S.L.
Madrid, Spain
ivan.gonzalez@uam.es

ABSTRACT

New Ethernet standards, such as 40 GbE or 100 GbE, are already being deployed commercially along with their corresponding Network Interface Cards (NICs) for the servers. However, network measurement solutions are lagging behind: while there are several tools available for monitoring 10 or 20 Gbps networks, higher speeds pose a harder challenge that requires more new ideas, different from those applied previously, and so there are less applications available. In this paper, we show a system capable of capturing, timestamping and storing 40 Gbps network traffic using a tailored network driver together with Non-Volatile Memory express (NVMe) technology and the Storage Performance Development Kit (SPDK) framework. Also, we expose core ideas that can be extended for the capture at higher rates: a multicore architecture capable of synchronization with minimal overhead that reduces disordering of the received frames, methods to filter the traffic discarding unwanted frames without being computationally expensive, and the use of an intermediate buffer that allows simultaneous access from several applications to the same data and efficient disk writes. Finally, we show a testbed for a reliable benchmarking of our solution using custom DPDK traffic generators and replayers, which have been made freely available for the network measurement community.

CCS CONCEPTS

•**Networks** → **Network monitoring;** *Network measurement;*
•**Computer systems organization** → *Multicore architectures;*

KEYWORDS

Network monitoring, traffic storage, multicore architecture, packet storage, off-the-shelf systems, DPDK, SPDK, NVMe

SAC'18, Pau, France

DOI: 10.1145/3167132.3167238

ACM Reference format:
Guillermo Julián-Moreno, Rafael Leira, Jorge E. López de Vergara, Francisco J. Gómez-Arribas, and Iván González. 2018. On the feasibility of 40 Gbps network data capture and retention with general purpose hardware. In *Proceedings of ACM SAC Conference, Pau, France, April 9-13, 2018 (SAC'18),* 9 pages.
DOI: 10.1145/3167132.3167238

1 INTRODUCTION

The advantage in passive network monitoring is its capability to access the packets that are being transmitted between hosts for inspection either online or at a later time. Online capture of network traffic may be used to detect attacks, anomalies or a drop in the quality of service.

However, some network measurement algorithms cannot be executed online because of time restrictions, data requirements or human input. Detailed analysis of networks such as the one done in [11] or forensic analysis [4] using either specialized tools or general-purpose ones such as Wireshark require more time and the *a posteriori* knowledge of which traffic intervals to study, and in most cases they are not fast enough to run online or require human input and control. In these situations, timestamping of the incoming frames is required to be able to establish causality relationships between frames and also to extract measurement statistics of the network.

Therefore, any monitoring application must be capable of both feeding online applications with packets on the fly, and also storing them in a manner that is easily accessible for analysts. Additionally, it should capture the highest number of network possible frames with accurate timestamping as possible to minimize information loss.

The capture process can be done using specialized hardware, but a "software capture process" using general purpose hardware is usually the preferred approach due to the lower costs and increased flexibility. There are several software solutions that allow the capture and storage of traffic for 10 Gbps networks but not for higher speeds, such as the mentioned 40 Gbps or above. One of the reasons is the steep increase in complexity of the data capture at these higher data rates. For example, while capturing at 10 Gbps can be possible using just one CPU core, at 40 Gbps the hardware requirements are too high to achieve such data rate. As an illustration, in the worst case, two consecutive packets of

minimum size (64 Bytes including CRC) would arrive with a gap of just a few 16.8 ns between them. If we compare this to a successful access to the L3 cache in an Intel Skylake CPU at 3.40 GHz, it would represent more than the 75 % of the time between those two packets [8]. These facts emphasize the need to make use of parallelism and multiple cores.

A straightforward parallelism using, for instance, the NIC queues for workload distribution across cores (Receive Side Scaling or RSS technology), could be an option to solve this problem. However, as we will discuss along this article, it is not the best approach in certain situations and a more sophisticated approach is needed to cover all these cases.

Throughout this paper, we present a system able to capture, timestamp, and store network traffic at 40 Gbps, working on Intel 40 GbE network interface cards and using SPDK (Storage Performance Development Kit) to store network traffic directly to NVMe disk arrays with an architecture that goes beyond the usual practices on capture drivers at lower speeds. Additionally, we include some techniques to reduce the traffic supported by applications allowing capture even in systems without high-speed storage systems. The application of these results have other multiple gains: less TAPs or SPAN ports are needed, monitoring cabling is reduced, and the overall hardware complexity of the network measurement system is also reduced, cutting down the total monitoring costs, both in terms of hardware and operation. Furthermore, the presented architecture could be expanded and scaled to capture at higher rates in the future.

The rest of this paper is organized at follows. The next section examines the state of the art in capture and storage engines, followed up by our proposed design and our implementation in section 3. Later, in section 4 we explain our main use cases and which test methodology has been applied to check and measure our system. Then, the results are shown in section 5. Finally, in section 6 we discuss the obtained results and present the final conclusions.

2 RELATED WORK

The literature is extensive for traffic capture and storage in 10 GbE networks, but not so much in 40 GbE or higher networks. Moreover, those works cover only either reception of network traffic at 40 Gbps, or storage of general data at high speeds. In our study, we have not found an integrated system capable of both receiving the traffic and storing it directly to disk for later analysis.

2.1 High speed network capture

Most high-performance network capture systems developed in the last years are focused on 10 GbE networks, as reviewed on [16]. Their main ideas, such as bypassing the Linux network stack, batch processing, optimization for Non-Uniform Memory Access (NUMA) architectures and memory mapping, are still useful but not sufficient for capture in 40 GbE networks.

In the literature, the few solutions that are able to capture traffic at that speed do not propose new architectures and instead extend already developed systems. One approach is the use of several 10 GbE NICs [21, 22]: it allows linear scaling with appropriate NUMA allocation policies [19], but it needs more hardware (NICs and an appropriate splitter for the incoming traffic) and the resource usage is not particularly efficient as it cannot scale up or down easily.

The second approach is using 40 GbE NICs with multiple RSS queues in the NIC to distribute the workload between different cores. The main representatives here would be the Data Plane Development Kit (DPDK[1]), PF_RING, netmap and PFQ. We have not found a benchmark comparing those at 40 Gbps rates, only standalone performance figures for DPDK, netmap and PFQ, in both cases without timestamping of the frames nor storing the capture frames. DPDK achieves 40 Gbps forwarding frames of size 128 bytes or greater [7] (a scenario less demanding than traffic capture) using 4 cores, and PFQ achieves capture (without storage nor timestamping) at 40 Gbps for frames of size 256 bytes or greater using 5 RSS queues [2]. Regarding netmap, we did not find performance assessments in the literature, although its web page[2] claims that it can reach 30 Mpps (equivalent to line rate at 145 bytes per frame at 40 Gbps) in unspecified conditions.

FlowScope [10] is another solution that was published after our work was finished. It uses MoonGen and DPDK as a base for building a 100 Gbps aggregated traffic capture system with several NICs and RSS queues. The authors claim that their system is capable of capturing to memory at line rate for frames of size 128 bytes or higher. However, they do not evaluate in deep the performance when writing to disk that traffic data. Moreover, its implementation can only save small peaks, since its storage media is not fast and large enough and can only store packets at 16 Gbps.

There are, however, two main downsides when using RSS. First, the packets are assigned to each queue based on a hash function which usually takes into account the IP addresses and ports of each packet. For normal operations, this is a very advantageous approach as it tends to place packets pertaining to the same flow in the same queue. But it also requires tweaking to avoid a non-uniform distribution among cores that leaves relatively unoccupied while overloading others. There are algorithms that ensure uniform distribution of flows among RSS queues [23] but these do not account for scenarios that may be present when monitoring an entire network with few dominant flows in terms of bandwidth. In this scenario, where having all the packets of a flow distributed to the same core is not essential, replacing RSS distribution by a simpler algorithm might be beneficial.

The second issue is the suitability of this approach when the purpose of the capture is a later analysis of the traffic. By using multiple queues, packets and their timestamps can be unordered on reception, something that can severely affect analysis applications. Also, the zero-copy schemes they use may not be especially efficient when the focus is the storage of the traffic, as we will discuss later.

Overall, to the best of our knowledge, there is not any software-based system for capturing, timestamping and storing traffic at 40 Gbps using 40 GbE NICs. Moreover, there is room to explore an architecture that, while using parallelism, does not use RSS queues and achieves high performance storage and precise timestamping with minimal disorder.

[1] http://www.dpdk.org
[2] http://info.iet.unipi.it/~luigi/netmap/

2.2 High speed storage

The two most prominent storage solutions in this field are incidentally named the same: NVMeDirect. The first one [20] use smart network cards capable of moving the data directly to and from a NVMe drive, without any CPU intervention other than the initial configuration. They claim to reach, using several drives and network cards, aggregated data rates above 100 Gbps. However, this project is oriented to data access and sharing one NVMe drive with a set of computers in a cluster. It has not been designed nor used to store packets from a network interface.

The second project [12] is very similar to the Storage Performance Development Kit (SPDK)[3] project used in this work. Unlike the other project discussed above, the key idea here is to exploit the performance of NVMe drives bypassing the operating system with a user-level driver, following the idea used in DPDK with network cards. The advantages of a user-level driver are clear: easier debugging, failure protection (memory protection), and less context switches. During our development, we have decided to use SPDK instead of NVMeDirect. Despite that [12] shows a slightly better performance than SPDK, NVMeDirect is currently on an early stage and partially unmaintained, whereas SPDK currently has a very active development.

2.3 High speed traffic generation

Apart from designing and developing the capture system, we also need to test it and measure its performance in several scenarios. However, it is difficult to have a high-speed real network with enough traffic to perform this type of testing. These networks are owned by ISPs, telecommunication operators and large enterprise networks, and user data is usually protected by existing regulations. Therefore, the only possibility is to use traffic generators and players.

One of the most used high-speed traffic generator is the DPDK-based PKTGEN-DPDK[4]. Nevertheless, due to its generality, this tool cannot perform at the maximum theoretical link speed: PKTGEN-DPDK can only send small PCAP files, which makes it difficult to saturate a 40 Gbps link. A solution to send larger PCAP files is DPDK-REPLAY[5]. However, this tool is currently deprecated (it uses DPDK-1.8.0, and the current version is 17.8), and it is only optimized for 10 Gbps traffic. In our tests, it only achieved around 5 Gbps when using CAIDA traces [3].

[3] http://www.spdk.io/
[4] https://github.com/pktgen/Pktgen-DPDK
[5] https://github.com/marty90/DPDK-Replay

A more specific tool could leverage all the available system resources and perform better. In section 4 we will show our own solution to achieve fast traffic generation, with two different tools.

3 PROPOSED DESIGN AND IMPLEMENTATION

Throughout this section we will explain the proposed one-copy design to achieve 40 Gbps capture, first explaining the architecture of the base HPCAP driver and why is it still useful for our case, and then describing the architecture in each stage of the capture.

3.1 Previous architecture

The proposed work builds on a previous capture driver for 10 Gbps [14], which provides the base for high-performance capture and storage. It consists of a modified version of the original drivers for the NIC, of which the most important part is the replacement of the original reception logic with our custom one.

One of the differentiating factors of this solution with respect to other such as DPDK or the Linux packet `mmap` interface [1] is the one-copy design. Usually, capture systems are zero-copy: the NIC copies the received frames via Direct Memory Access (DMA) and then the client application reads from them. However, if the main focus is storage, this approach is less efficient. When writing to disk, it is desirable to copy all the frames in a contiguous manner and write them in blocks to maximize performance. But it is impossible to know the packet sizes beforehand so the card will not be able to copy all the packets in a contiguous memory region.

Therefore, it is more efficient in this case to use the one-copy approach: the driver copies the frames contiguously from the NIC ring to an intermediate buffer that client applications can read (client reading mechanisms will be further discussed in section 3.4).

For better management of the stored traffic, our system writes timestamped files of 2 GB, allowing the analyst a quick location of the desired segment of traffic, and also facilitating easy rotation of old captures. To enforce write operations of constant size (thus improving performance) and avoid splitting packets between files, a final padding packet is inserted during the capture when there is not enough space in the 2 GB file to write the next incoming frame.

The architecture for multiple readers is also maintained. With it, one interface accepts several applications reading the same traffic, supporting scenarios such as having one application analyzing live traffic and another separate one copying it to permanent storage [17]. This approach allows the storage applications to operate in blocks without reading every packet to determine the size of each file, therefore improving efficiency.

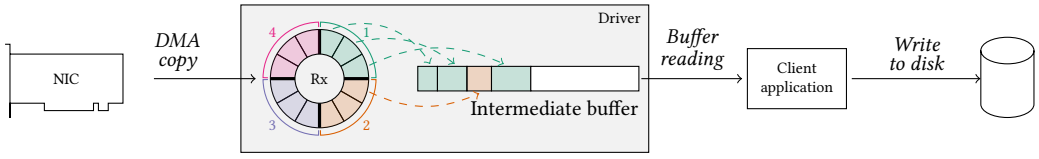

Figure 1: A schematic of the architecture of our driver: The NIC copies the new frames to the RX ring via DMA. Each thread will copy the frames from its assigned sector to an intermediate buffer. Client applications can map that buffer to their memory space to read and store the frames.

3.2 Reading from NIC

The usual approach that NICs use in reception is to write the packets in the RX ring in host memory using DMA. As discussed previously in section 2.1, the main approach for parallelism in traffic reception is the use of multiple RSS queues, so that the CPU can read by using multiple cores. But there is another possibility: if we disable RSS queues and use just one reception ring we remove any potential problem with uneven distribution of the traffic between queues.

A first approach to read from this ring with multiple threads could be first-come-first-served (FCFS), where each thread reads the first available descriptor. However, this would cause two problems: continuous synchronization between threads that would decrease performance, and disorder in the copied packets. The alternative implemented for this driver is the division of the ring in several sectors, assigning one to each thread as in fig. 1. Thus, no synchronization mechanism is needed as every thread knows exactly which descriptors can and cannot be read and the process become cache-friendly with a predictable latency and throughput. Furthermore, the only disorder of the frames can happen on the boundary of each sector, when two threads are copying their corresponding frames at the same time without respecting their relative order.

Synchronization is only needed for one action: the ordered return of ownership to the NIC of the read descriptors as required by its specification, so it can fill them again with new packets. This ownership change is preferred to memory allocations/deallocations of descriptors in order to improve performance. As the network card requires the descriptors to be returned in order, our system ensures by means of an atomic variable that each thread can only release those in its assigned sector after all the descriptors in the previous sectors have been returned.

For testing purposes, we also implemented the option to use RSS queues and assign each consumer thread to a different ring. In this case, the implementation is straightforward as there is no need for synchronization between threads at this stage, and in our tests the performance was similar if the incoming traffic was such that the workload was evenly distributed. However, this is not the preferred approach due to, as discussed previously, uneven distribution and increased disorder of frames and timestamps.

3.3 Write to intermediate buffer

A critical step in the reception of traffic is to write the intermediate buffer from which the clients will be able to read, either in a packet-by-packet fashion or writing in blocks to disk. All the reception threads will write to this buffer, so an efficient synchronization method is required to enable high-performance packet capture.

The abstract model for this buffer is a multiple reader - multiple writer queue. However, several aspects of the driver design allow the implementation of shortcuts and optimizations that improve performance. The basic synchronization tool is an atomic variable that points to the offset in the buffer where the next packet should be written. This variable is incremented by each thread before writing the packet, thus making the allocation immediately effective for all threads. As the size of the intermediate buffer is controlled by the driver, there is no need to add further synchronization mechanisms to ensure that the offset stays inside of the available memory range: the driver can allocate, using Linux hugepages, a buffer with a size that is a divisor of 2^{32} bytes. Thus, a 32-bit variable for the offset can be used by performing modulo operations, and it will overflow appropriately without extra controls that would imply additional synchronization and poorer performance.

The same idea allows the control of the offset inside the file for padding control. As explained in section 3.1, our driver uses padding to ensure that packets are not split between files, allowing direct writes to disk. This is still a desirable feature when the target storage system uses NVMe disks: it divides frames into fixed-size blocks that can be easily indexed. To keep this capability in the driver, the padding algorithm needs to be adapted to the new write scheme. The problem to solve is what to do when a thread allocates space for a packet that would be split into two files. A possible approach would be reverting that allocation and writing only the padding to fill up the file. However, this would require more synchronization mechanisms that would slow down the system.

The better approach is, instead, to write padding at the beginning and at the end of the file, marking appropriately that the allocated space is not a valid frame. The decision of which thread should write the padding is delicate: a thread could have allocated insufficient space to write the padding header at the beginning of the file, or another one could have allocated too much previously leaving no space for the padding footer at the end of the file. We do not describe the full algorithm here due to space reasons, but the main idea is to compute remaining and used spaces in the buffer and have a thread write the padding as soon as it is sure there is no space for packets or padding to be written by other threads.

3.4 Client reading

The next step in a usual multiple reader queue would be the synchronization of the readers and the writer threads. The architecture of our driver allows a simple mechanism to meet this goal.

To avoid overwriting, the driver has a global offset inside the buffer marking the last byte that was read by all listeners. This allows the calculation of the available space s in the buffer and in turn makes a simple mechanism to avoid overwriting: each thread writes at most $\lfloor s/n \rfloor$ bytes in a batch, where n is the number of threads. This avoids unnecessary readings of atomic variables each time a packet is received (although reading atomic variables is fast, it can affect performance in the demanding scenario of 40 Gbps capture).

The second issue —writer synchronization— is solved by the architecture of the driver and the Linux system call mechanism. Consider the worst possible situation of reading unwritten regions: the reading application would have to perform the system call to the driver through an `ioctl`, a thread should update the offset counter, and as soon as that value is written the system call process should read the offset variable. However, even in that worst case, the kernel still has to finish the system call and return the execution flow to the client application, an operation ranging on the order of tens of microseconds, at least. On the other hand, the receiver thread is running without possibility of being stopped or scheduled out of the processor, and the only task it has to do is to copy the frame to memory. Thus, even in the worst case the time between

the moment the thread increases the counter of written bytes in the buffer and any possible read of the memory region is enough for the thread to actually write those bytes. Therefore, no extra synchronization mechanisms are required.

3.5 Writing to disk

Each client, as explained in the previous sections, has an offset pointing to the first byte it can read and the number of bytes available to read, and will notify periodically to the driver how many bytes have already been read. There is complete freedom on how that data is read and processed by the client: in this paper we only describe the two approaches used in the client applications performing the tests.

The first approach is the one inherited from the 10 Gbps HPCAP driver. This application copies the data from the intermediate buffer directly to files in the disk in blocks of 4MB. This approach is effective as long as the filesystem is capable of writing at capture speed. For 10 Gbps, this was accomplished with an array of mechanical disks in RAID-0 configuration [15]. Unfortunately, at the time of developing this work, hardware RAID controllers for NVMe disks are not available on the market, and software RAID controllers are not fast enough to capture at 40 Gbps and do not take advantage of the maximum achievable speed of a NVMe drive array. This made us use a second approach to overcome these limitations: using the Intel SPDK framework, which gives efficient read/write access to the disk by using the optimal parameters provided by the manufacturer.

In order to be able to access an NVMe disk efficiently, it is necessary to perform the read and write operations according to the restrictions of the disk. Some of those parameters are the number of blocks to read in each transaction, or the depth and number of queues to use. These are usually provided by the manufacturer and in any case they can be retrieved using SPDK tools. In order to handle stored data, we have developed our own filesystem that adapts to the restrictions and configuration of the NVMe disks and to the requirement of distributing files among disks without using excessive synchronization.

The new `hpcapdd-spdk` tool distributes regions of the intermediate buffer between the different threads, using a size multiple of the optimal write size for the NVMe, so that each thread is responsible of writing to one disk. This solution requires one writing thread for each NVMe disk and it achieves the maximum possible throughput, and at the same time the smallest synchronization cost. The different files are managed by using the sector 0 of each NVMe disk, allowing a small, limited number of files but enough to efficiently use the custom software NVMe-RAID.

3.6 Filtering and range selection

Although the architecture previously described allows the capture at high rate in a scalable fashion, we have nevertheless developed features to allow our software to run not only in high-end hardware but also in more flexible environments.

The filtering system is designed to be simple and fast. It allows the definition of several "strings" of bytes and their position in the frame, with a limit of 128 rules. The driver checks each packet for the presence or absence of those strings in the specified positions and then copies or discards the packet as configured by the user. As an example, this solution allows discarding UDP and ICMP packets or capturing only the packets with source or destination port 80 or 8080.

Another possibility for reducing the disk workload of the capture is to store a range selection of the bytes of the frame. This allows the user selecting which bytes are going to be saved, thus removing information deemed unnecessary. For example, the system could store only the first 200 bytes of each packet, or only the TCP header (in this case, a fixed range of bytes must be provided as the driver does not read nor interpret any header nor field in the frame).

3.7 Timestamping

Implementing a precise timestamping method is a great challenge and in no way trivial. Given that the time between frames is on the order of nanoseconds, it is necessary to use libraries with a response time at most in the order of tens of nanoseconds[6], which is an important overhead in both the reception and sending processes, and of course, it also affects the precision of the resulting timestamp.

Retrieving the current time from the system to timestamp a packet using user-level calls would increase the inaccuracy: performing a system call has a variable delay and oscillations between hundreds or even thousands of nanoseconds per call. This makes user-level timestamping using the system clock a non-adequate method for packet timestamping.

Therefore, we have decided to timestamp all the received frames on arrival in the driver polling threads, retrieving the current time using the `getnstimeofday()` kernel function before the received packet is copied to the intermediate buffer. The other approach (timestamping all frames in hardware) is, unfortunately, not available in all NICs. We have also decided to avoid timestamping only each N packets (batch timestamping) as those approaches have worse results regarding accuracy [18].

4 TESTING METHODOLOGY

The testing of the system consisted of two servers connected by a single QSFP+ DAC cable, where the first computer acted as a traffic generator and the other one as the traffic receiver. For the reception, four different types of tests have been made. The first type is done without any clients reading from the buffer, ensuring that the architecture is capable of pulling all the received frames at line rate. The second type adds a client reading from the buffer and then discarding the frames, which allows the testing of the maximum reception rate without disk storage. The third type of test includes the storage application saving all the traffic to NVMe disks. Finally, the fourth one consists in sending files with previously captured traffic at maximum rate through the network interface and storing them in the disks.

We used two different files: a trace from the CAIDA [3] dataset, with a size of 222 GB; and another one with traffic captured in our university student laboratories, with a size of 4.3 GB. The average frame size in these files was of 787 and 910 bytes respectively. Both are sent in a loop to transmit 1 TB of traffic.

[6] https://github.com/hpcn-uam/hptimelib

Our driver was configured in the reception server with different number of receiving threads, appropriately distributed in the NUMA architecture for optimal performance. For best results, we also configured the driver so that it allocates the memory region for the NIC ring in the same NUMA node where the card is placed, and the intermediate buffer in the other node, where the client applications are executed. We also ensured that the assigned cores were isolated to avoid interferences caused by the Linux scheduler.

4.1 Hardware used

Regarding the hardware, we used the three servers shown in table 1. One server was used for traffic generation and two for reception, of which only *RX Server 2* had an array of 6 NVMe Intel DC P3600 800 GB SSD drives.

Table 1: Specifications of the servers used for testing. HyperThreading was disabled.

	Traffic generator	*RX Server 1*	*RX Server 2*
CPU	Intel Xeon E5-1620 v2	Intel Xeon E5-1620 v2	2 × Intel Xeon E5-2630 v4
Clock	3.70GHz	3.70GHz	2.20 GHz
Cores	4	4	2 x 10
Memory	32 GB	32 GB	2 x 64 GB
NIC	Intel XL710	Intel XL710	Intel XL710
Storage	SATA RAID	SATA RAID	6 x NVMe
Est. cost	7,000€	7,000€	10,000€

To ensure optimal results, we measured the performance of the NVMe array configured as a RAID-0 software unit and also by using SPDK. In both cases we used a chunk size of 128 KB, which is the optimal write size for each drive and produces the best performance in sequential writing. The benchmark results are shown in fig. 2 along with the theoretical maximum performance (the specification [5] shows that each drive should have a maximum rate for sequential writes of 1 GB/s). The software RAID achieved a maximum performance of only 31 Gbps, which represents only a 65 % of the maximum theoretical rate and is not enough to capture at 40 Gbps. The SPDK framework improves considerably the performance, writing at 45 Gbps (93 % of the theoretical maximum disk write speed), which is enough for our purposes.

During the tests, we used the Intel XL710 network adapter. In order to achieve optimal performance, we disabled the following characteristics: checksumming, segmentation and fragmentation offloading, hashing and VLAN filtering. Although these features might be useful for normal usage of the NIC, in our specific capture scenario they are not required (the traffic is going to be stored as-is, without reassembling of flows nor checksums verifications) and they would slow down the capture and provide erroneous time measurement; therefore they are kept disabled.

We also performed tests with a Mellanox MT27500, another 40 GbE NIC. It has the advantage of being able to timestamp frames in hardware, a method more precise than doing it in software. However, we found a significant bottleneck in the card. After integrating the *mlx4_en* driver with our system, we performed the basic test (just reception of the frames without copying them to the intermediate buffer, the less demanding scenario possible) and we only achieved line rate capture without losses for frames with large size (600 bytes), something that did not happen with the Intel XL710 NIC. The capture rate seemed to be limited by the card: the improvement in reception rate was minimal when changing from one to two consumer threads, and inexistent with even more threads. By disabling almost all the features of the card (including those mentioned to for the Intel NIC, and also the RSS distribution system) we achieved line rate capture in the least demanding scenario for frames of size 300 or higher (for reference, in the same scenario with the Intel XL710 we capture all frames of size 100 and higher), but the issue with the non-improvements with scaling persisted. Therefore, we decided to not include the Mellanox results in this study.

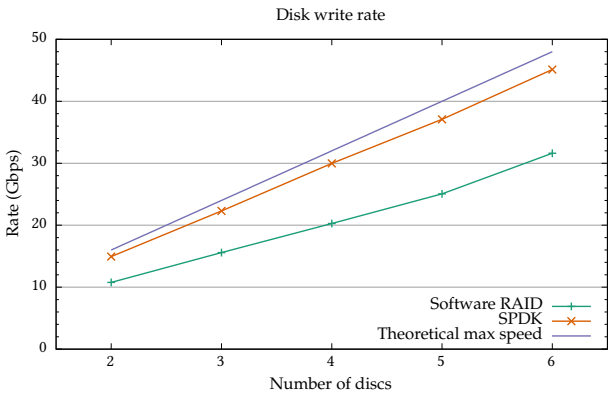

Figure 2: Performance of the NVMe disk array.

4.2 Traffic generation

Two independent tools have been coded to carry out the tests, the first one for the generation of synthetic traffic[7] and the second one to have a fast PCAP replay tool[8]. Both tools have been developed using DPDK, since it supports our 40 Gbps card and is well suited for our purposes of traffic generation.

Our developed tools improve between 10% and 40% the performance of the state-of-the-art tools using small packets, with a single queue. However, that is not enough to saturate a link at 40 Gbps, so we used multiple send queues, with each queue associated to an independent thread and core.

The stress application sends synthetic traffic so its implementation is simple: based on parameters provided by the user (packet length, MAC address, etc.), a single packet is built in memory and later sent using the multiple queues at maximum rate. Our measurements (figure 3 on the facing page) show that the application is capable of generating and sending up to 42 million packets per second in our traffic generator, therefore saturating the 40 Gbps link with packets of length 96 bytes or greater.

Sending a PCAP file at 40 Gbps represents a tougher challenge compared to synthetic traffic since it requires to read the file at the sending speed with almost negligible latency. Our tool maps the target file in memory (using Linux's mmap call), requesting

[7] https://github.com/hpcn-uam/iDPDK-LatencyMetter
[8] https://github.com/hpcn-uam/iDPDK-PcapSender

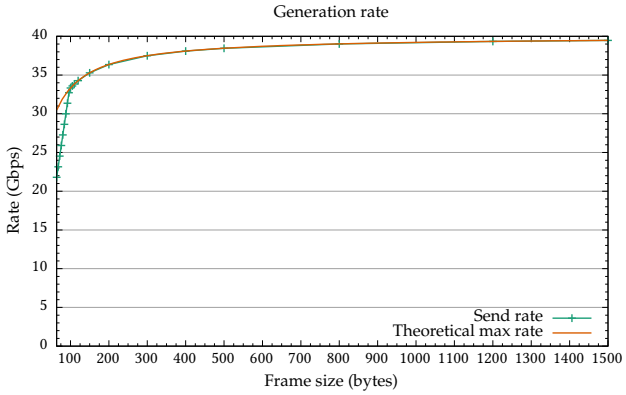

Figure 3: Synthetic traffic rates achieved.

the operating system to preload the entire file if possible (that is, if there is enough memory available). The application reads the looped file infinitely, allocating each packet descriptor and copying its PCAP payload on the fly. This allows sending files as large as the available RAM of the computer, or even larger if the backing storage system is fast enough. In fact, we were able to transmit at 10 Gbps using 10 HDDs in a RAID-0 configuration and a single transmission thread. The high speed of the RAID system may not even be needed if the traces are truncated, as the CAIDA ones are. There, frames truncated at IP level and the TCP/UDP layer is added during runtime. Therefore, to send those traces at 40 Gbps we only need to read the file at a rate of about 2.3 Gbps.

4.3 Timestamping accuracy

In order to measure timestamp accuracy, we placed two Intel XL710 NICs in our *RX Server 2* as using two different servers would lead to errors caused by the drift between the CPU clocks [13]. We used the *LatencyMetter* application to send timestamped traffic to evaluate our system with the same approach used in [18].

Accuracy of the timestamping system is affected by the behavior of the Intel NIC regarding the descriptor updating. As explained in [9], the Intel XL710 NIC works with batches of descriptors for improved performance: it will only post received frames to the RX ring by groups of eight descriptors, and that affects timestamping accuracy as the interarrival times between packets of the same batch are going to be smaller than the real value.

We observed these differences in our tests: we sent timestamped packets of size 64 bytes with our traffic generator at the maximum possible rate and compared the interarrival time between them with the time as measured by our driver. The average error was of 1,738 nanoseconds with a standard deviation of 3,296 ns. However, when we measured the error just in one out of eight packets (that is, only once per batch) the average fell down to 55 ns with a standard deviation of 287 ns, which is similar to timestamp accuracies measured in the literature [18].

Therefore, the effect of batching in the NIC introduces a non-negligible error in the timestamping accuracy. But given that we are not using RSS queues, the card copies the frames in an ordered manner to the RX ring, avoiding unordered timestamps necessary for multimedia protocols that use several concurrent flows.

4.4 Measurements conducted

In the tests, we measured the traffic generation rate as reported by the generation application counters, and the captured bandwidth and loss percentage as measured by our driver. We also monitored specific counters on the card using `ethtool` to measure the NIC performance, specifically `rx_dropped` and `port.rx_dropped`. The first one reveals packets lost because the driver was not fast enough. The second one is also relevant because, according to the Intel XL710 documentation [9], it counts packets lost because "*something is not fast enough in the slot/memory/system*" and in some tests revealed bottlenecks on the card itself for small frames. This is not unexpected, as Intel only claims wire-rate performance for packets of size 128 bytes and higher [6] and the exact same issue has also been observed in other works [10].

5 RESULTS

The results of the different cases discussed in the previous section are presented below. The purpose of the first test (fig. 4) was to ensure that our architecture was capable of extracting the frames from the NIC rings at line rate, without any application reading them. In this case we observed a bottleneck in the hardware: almost all of the losses came from the `port.rx_dropped` counter, representing frames that were dropped not by our driver but by the card not being able to service all the frames. Our system is capable of reading all the packets received by the NIC with just two reading threads, and it can be seen that the system does not present losses for frame sizes greater than 105 bytes.

The second test was performed by having a client process connected to the driver and reading frames from the buffer without storing them, to measure the maximum reading capacity. There, we achieved line-rate capture and timestamping at frame sizes of 300 bytes with 4 receiver threads and just one thread reading from the buffer. (fig. 5)

The third test (fig. 6) consisted on the reception of frames and their storage in a NVMe disk array. This final test was performed on *RX Server 2* as it was the only one with an NVMe drive array. We had to increase the number of receiving threads to counterbalance the lower CPU speed. With this system we were able to store at line rate the incoming traffic for frames of 300 bytes and above.

Additionally, we also tested the performance when using RSS queues to measure the performance effects of our approach. We observed the same performance as long as the traffic was evenly distributed between queues, which shows that our approach has no performance side-effects.

Finally, we performed tests reproducing files containing previously captured traffic, as described in the previous section, sending them in a loop until at least 1 TB was received. As can be seen in table 2, our testing architecture is capable of sending those files at maximum rate and capturing them with only negligible losses in the CAIDA trace and no losses at all in the University trace.

We also tested the filtering features of the driver. Due to the simplicity of these filters, they do not worsen significantly the capture performance. Their final effect on performance depends on how many packets pass the filters and are copied to memory, which is the most expensive task. Similarly, the range selection

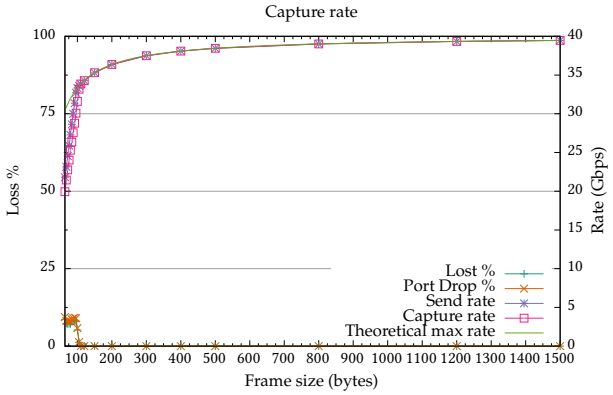

Figure 4: Results of the first test: retrieval of the frames with the NIC.

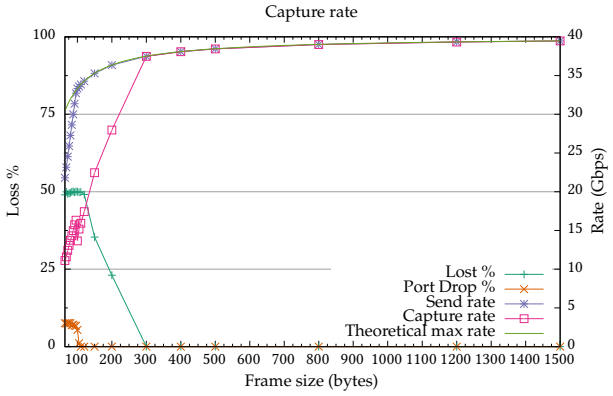

Figure 5: Results of the second test: writing of the frames to a null device.

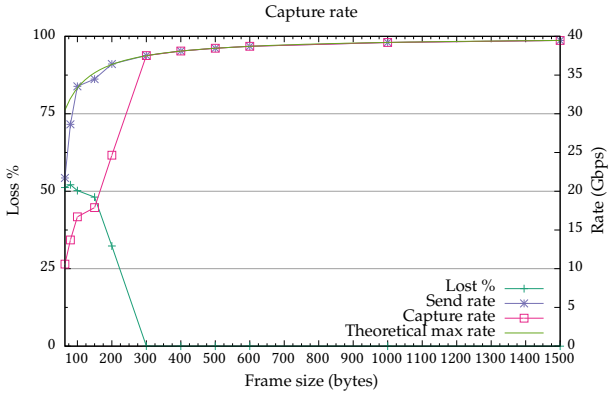

Figure 6: Results of the third test: traffic storage using SPDK.

Table 2: Performance on reception of traffic capture files (test 4).

Name	Size	Avg. frame size	Send rate	Loss %
CAIDA	222 GB	787.91 B	39.78 Gbps	<0.01
University	4.3 GB	910.08 B	39.82 Gbps	0

improves performance depending on the average packet size. We observed maximum performance gain when the average packet size is approximately three or more times the number of bytes being saved. In that scenario, the performance is equivalent to having no client applications.

6 CONCLUSIONS

Throughout this paper we have presented a system capable of capturing, timestamping and storing network traffic at 40 Gbps with negligible losses when testing it with real traffic traces using general purpose hardware, such as an Intel NIC and an array of NVMe drives. The code has been made available[9] under the GPL license.

From the results presented above, we can see that there is a payoff when not using RSS parallelism in the capture and reducing as much as possible the number of synchronization points: in the base case, we have achieved optimal performance (to the point that the bottleneck is in the network card, CPU and hardware system) by using one RSS queue and two threads divided into different memory sectors, which enables straightforward scalability without adding more sincronization points.

Furthermore, our solution using fixed sector assignments behaves better in terms of balancing than if we were using RSS queues: it does not need configuration nor tweaking of parameters to avoid asymmetric workloads and ensures uniform distribution among the threads even when the conditions of the network change over time. It also reduces possible errors in the timestamping as the frames are copied by the card to only one reception ring, so that our driver processes them in the order they arrived.

Although the one-copy mechanism adds some overhead to the capture process, it allows high performance storage of the received traffic. In our tests, with 6 NVMe disks we achieved 40 Gbps rate in sequential writing using a multi-threaded writer with one thread per disk when the frames transmitted are of size 300 bytes and greater. We also tested the validity of this system when using real-world traffic from previously captured network traces. These results can be improved with faster off-the-shelf hardware (the NVMe storage server only had CPUs at 2.20 GHz). Additionally, we have also developed a testbed capable of saturating a 40 GbE link with 96-byte frames of constant size in order to test our system.

However, there is still work to do with this system, such as a detailed comparison of the disorder induced by the use of our approach versus using RSS queues, or testing the scalability of the architecture at higher speeds. Finally, we are planning to use the knowledge acquired with this development and libraries such as SPDK to create traffic generators capable not only of sending synthetic traffic but also reading and transmitting network trace files at 100 Gbps.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under the project TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R), and by the European Commission under the H2020 project METRO-HAUL (Project ID: 761727).

[9] https://github.com/hpcn-uam/hpcap40g

REFERENCES

[1] Ulisses Alonso Camaró and Johann Baudy. 2016. *Linux packet_mmap.* Technical Report. Linux Foundation. https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt

[2] Nicola Bonelli, Stefano Giordano, and Gregorio Procissi. 2016. Network traffic processing with PFQ. *IEEE Journal on Selected Areas in Communications* 34, 6 (2016), 1819–1833. https://doi.org/10.1109/JSAC.2016.2558998

[3] Caida. 2016. The CAIDA UCSD Anonymized Internet Traces 2016 - 130000-131000 Chicago. (2016). http://www.caida.org/data/passive/passive_2016_dataset.xml

[4] Vicka Corey, Charles Peterman, Sybil Shearin, Michael S Greenberg, and James Van Bokkelen. 2002. Network forensics analysis. *IEEE Internet Computing* 6, 6 (2002), 60–66. https://doi.org/10.1109/MIC.2002.1067738

[5] Intel Corporation. 2014. Intel SSD DC P3600 Series (800GB, 2.5in PCIe 3.0, 20nm, MLC). (2014). https://ark.intel.com/products/80999/Intel-SSD-DC-P3600-Series-800GB-2_5in-PCIe-3_0-20nm-MLC

[6] Intel Corporation. 2014. *Product Brief: Intel Ethernet Controller XL710 10/40 GbE.* Technical Report. Intel. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-xl710-brief.pdf

[7] Intel Corporation. 2016. *DPDK Performance Report Release 16.11.* Technical Report. Intel. http://fast.dpdk.org/doc/perf/Intel_DPDK_R16_11_NIC_performance_report.pdf

[8] Intel Corporation. 2016. Intel 64 and IA-32 architectures optimization reference manual. (2016). http://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[9] Intel Corporation. 2016. *Intel Ethernet Controller X710/XL710 Linux Performance Tuning Guide.* Intel. http://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/xl710-x710-performance-tuning-linux-guide.pdf

[10] Paul Emmerich, Maximilian Pudelko, Sebastian Gallenmüller, and Georg Carle. 2017. FlowScope: Efficient Packet Capture and Storage in 100 Gbit/s Networks. In *IFIP Networking 2017.* Stockholm, Sweden. http://dl.ifip.org/db/conf/networking/networking2017/1570334712.pdf

[11] Wolfgang John and Sven Tafvelin. 2007. Analysis of Internet backbone traffic and header anomalies observed. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement.* ACM, 111–116. https://conferences.sigcomm.org/imc/2007/papers/imc91.pdf

[12] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16).* USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[13] M. Lévesque and D. Tipper. 2016. A Survey of Clock Synchronization Over Packet-Switched Networks. *IEEE Communications Surveys & Tutorials* 18, 4 (Fourthquarter 2016), 2926–2947. https://doi.org/10.1109/COMST.2016.2590438

[14] Víctor Moreno. 2015. *Harnessing low-level tuning in modern architectures for high-performance network monitoring in physical and virtual platforms.* Ph.D. Dissertation. Escuela Politécnica Superior UAM. https://repositorio.uam.es/bitstream/handle/10486/674401/moreno_martinez_victor.pdf

[15] V. Moreno, J. Ramos, J. L. García-Dorado, I. González, F.J. Gómez-Arribas, and J. Aracil. 2015. Testing the capacity of off-the-self systems to store 10GbE traffic. *IEEE Communications Magazine* 59, 9 (2015), 118 – 125. https://doi.org/10.1109/MCOM.2015.7263355

[16] Victor Moreno, Javier Ramos, Pedro M. Santiago del Río, José Luis García-Dorado, Francisco J. Gomez-Arribas, and Javier Aracil. 2015. Commodity Packet Capture Engines: Tutorial, Cookbook and Applicability. *IEEE Communications Surveys & Tutorials* 17, 3 (thirdquarter 2015), 1364–1390. https://doi.org/10.1109/COMST.2015.2424887

[17] Víctor Moreno, Pedro M. Santiago del Río, Javier Ramos, David Muelas, José Luis García-Dorado, Francisco J. Gómez-Arribas, and Javier Aracil. 2014. Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems. *International Journal of Network Management* 24, 4 (2014), 221–234. https://doi.org/10.1002/nem.1861

[18] Víctor Moreno, Pedro M. Santiago del Río, Javier Ramos, Jaime J Garnica, and Jose Luis García-Dorado. 2012. Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines. *IEEE Communications Letters* 16, 11 (2012), 1888–1891. https://doi.org/10.1109/LCOMM.2012.092812.121433

[19] MV Vinu Paul, Raktim Bhattacharjee, and R Rajesh. 2014. Traffic capture beyond 10 Gbps: Linear scaling with multiple network interface cards on commodity servers. In *Data Science & Engineering (ICDSE), 2014 International Conference on.* IEEE, 194–199. https://doi.org/10.1109/ICDSE.2014.6974636

[20] QLogic. 2016. NVMe Direct: Next-Generation Offload Technology. White Paper. (Oct. 2016). http://www.qlogic.com/Resources/Documents/WhitePapers/Adapters/WP_NVMeDirect.pdf

[21] Pedro M Santiago del Río, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil. 2012. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 Internet Measurement Conference.* ACM, 65–72. https://doi.org/10.1145/2398776.2398784

[22] Martino Trevisan, Alessandro Finamore, Marco Mellia, Maurizio Munafo, and Dario Rossi. 2017. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. *IEEE Communications Magazine* 55, 3 (2017), 163–169. https://doi.org/10.1109/MCOM.2017.1600756CM

[23] Shinae Woo and KyoungSoo Park. 2012. Scalable TCP session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, South Korea, Tech. Rep. TR-symRSS* (2012). https://an.kaist.ac.kr/~shinae/paper/2012-srss.pdf