Traffic Modelling Challenges on 10 Gbit/s Links using Netmap Platform

Hasan Redžović, *Member, IEEE,* Zoran Čiča, and Aleksandra Smiljanić, *Member, IEEE*

Abstract **— Network devices are using a growing number of 10 Gbit/s links. Such throughput upgrade imposes new challenges on creating high-speed packet generators that can provide real network conditions. Traffic generators implemented as software solutions have challenges in achieving a required performance, as they are commonly using a kernel network stack for packet I/O. We describe different techniques for implementing the Poisson and *on-off* traffic model in a software-based traffic generator that is using netmap API as fast packet I/O.**

Keywords **— Traffic Generators, netmap API, IP traffic models.**

I. INTRODUCTION

IP networks face growing performance demands on a year-by-year basis caused by expansion of Machine-to-Machine (M2M) and Internet of Things (IoT) traffic, increase in the number of new users, video streaming and video surveillance services. These demands require development of new high-performance network devices (e.g. switches and routers) with a satisfactory level of reliability. An appropriate testing environment can provide precise feedback of current device capabilities and flaws, which helps developers to implement faster design modifications. Traffic generators create different types of traffic towards network devices and also receive and analyze processed traffic from network devices. In other words, the main goal of traffic generators is to create data flows that resemble real IP traffic and put a network device under a workload that is expected in a real network.

Traffic generators can be implemented as a hardware or software solution. Hardware-based traffic generators can provide a better performance compared to software-based traffic generators. However, hardware-based traffic generators lack flexibility, which can be very important for versatile network device development. Software-based traffic generators are usually relying on Kernel Network Stack (KNS) of operating systems as packet Input/Output (I/O). KNS has a legacy code, which inefficiently executes system-call functions and unnecessary packet copying. As a result, KNS is not well optimized for new network cards with 10 Gbit/s ports. The KNS is the main performance bottleneck of software-based traffic generators, which limits their usability while testing high bandwidth devices and networks. In order to extend the applicability of software-based traffic generators, the problem with performance bottleneck should be solved by switching from KNS to alternative platforms for fast packet I/O, such as netmap [1]. Netmap, in particular, is bypassing KNS and it is using packet batching, static memory allocation, memory sharing and simplify packet representation.

These optimization techniques allow netmap to provide significantly higher packet I/O throughputs compared to the KNS. The netmap platform programing code includes several basic example-applications with a purpose of showing practical usage of netmap API. One of these basic example-applications is traffic generator *pktgen*, which has a very limited set of functionalities - one threaded process that can either receive or send a packet through a single interface. In this paper, we describe the implementation of traffic generator (*nm_tx*) which is using netmap API to achieve a maximal packet throughput on multiple 10 Gbit/s links. Traffic generator *nm_tx* is using the Poisson model and *on-off* model for emulation of real network conditions. In addition, we analyze different techniques for the Poisson model implementation, which needs to comply with specific restrictions of netmap API. This paper is an extended and revised version of the paper presented at Telfor conference [7]. This extended version of the paper describes additional development of *nm_tx*, which includes implementation of *on-off* model.

This paper is organized as follows: In Section 2, we describe netmap architecture and optimization methods. In Section 3, we describe challenges when implementing *on-off* and Poisson models within netmap platform. Section 4 describes different algorithms for generating random numbers that follow the Poisson model. Section 5 analyses the performance of these algorithms. In Section 6, we present implementation of *on-off* and Poisson traffic models within *nm_tx*. Section 7 verifies these implementations. Section 8 concludes this paper.

Paper received May 12, 2019; revised October 7, 2019; accepted October 16, 2019. Date of publication December 30, 2019. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Grozdan Petrović.

This paper is revised and expanded version of the paper presented at the 26th Telecommunications Forum TELFOR 2018 [7].

This work was supported by the Serbian Ministry of Science and Education (project TR-32022), and companies Telekom Srbija, and Informatika.

Hasan Redžović is with the Innovation Center of School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: hasan.redzovic@ic.etf.rs).

Zoran Čiča is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: cicasyl@etf.bg.ac.rs).

Aleksandra Smiljanić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: aleksandra@etf.rs).

II. NETMAP PACKET I/O FRAMEWORK

Netmap is a high-performance packet I/O framework, which uses different optimization methods in order to eliminate various packet-processing overheads and achieve a higher packet throughput compared to the KNS. Netmap achieves a significant performance improvement by applying the following optimization methods:

(*i*) *Simple packet representation* - KNS stores packets in complex data structures, which contain redundant legacy meta-data. In addition, KNS executes per-packet memory allocation/deallocation and multiple packet copies, which are time-consuming operations. Netmap implements a simple packet structure that is composed of few flags and a pointer to the memory location where packets are stored. Within netmap framework, memory is preallocated so that time-consuming per-packet memory allocation and deallocations are eliminated. This optimization method does not limit the implementation of Poisson and *on-off* traffic models.

(*ii*) *Shared memory region* - directly connects applications in user space with ports of Network Interface Cards (NICs). Shared memory region is allocated in a non-pageable (non-swappable) part of system memory and it allows zero-copy packet processing. Packets can be received, processed and sent without being copied. This optimization method does not limit implementation of Poisson and on-off traffic models within netmap framework either.

(*iii*) *Packet batching* - In shared memory region, netmap crates replicas of NIC circular buffers (NIC rings). These replicas are named netmap rings and they point to the same memory locations (packet buffers) as NIC rings. Packet buffers are used for storing packets that are received from network (*Rx*) or that must be sent to the network (*Tx*). As the NIC ring, and its corresponding netmap ring, point to the same set of packet buffers, the applications use netmap ring to access one subset of these packet buffers, while the NICs use the NIC ring to access the rest of the packet buffers. This access control method is imposed with *head*/*tail* slots. The *head*/*tail* mark beginning/end of available slots, either for NICs or applications. The workflow of netmap-based application is as follows. First, all available slots are processed in the netmap ring; then, application triggers ring synchronization for exchange subsets of available slots between NIC and netmap ring. System functions execute ring synchronization process, which is a time-consuming operation. However, this slow process of ring synchronization is performed on the relatively large number of slots. Processing the large number of packets, named batches, per one system function call reduces the overhead. In contrast, KNS copies and executes system functions calls per packet, which is substantially slower.

Unlike in previous two optimization methods, packet batching can significantly influence the implementation of Poisson and *on-off* traffic models within a netmap framework. Increasing the number of packets processed per one system function call (one synchronization) improves the performance, and, at the same time, lowers the precision of traffic emulation.

The role of traffic models is to determine the number of packets that must be generated in each time interval. When a traffic model generates a number that is smaller than desirable for a packet batching optimization method, a traffic generator might struggle to achieve appropriate packet rates.

Taking into consideration the limitation of packet batching optimization method, we have created a traffic generator based on netmap platform (*nm_tx*) that can process a maximal throughput of 14.88 Mpps (Million packets per second) on a 10 Gbit/s port using one CPU core. An equivalent traffic generator with KNS can process only 2 Mpps.

III. TRAFFIC MODELING CHALLENGES

Emulation of IP traffic by any traffic generator requires specification of the number of packets generated in each subsequent time interval. These numbers should be generated according to a specified statistical model in order to emulate realistic network conditions. During the development of *nm_tx*, we have focused on implementing two traffic models: *on-off* and the Poisson model.

The *on-off* model is composed of two states: active and idle. Utilizing *on-off* model, *nm_tx* can either generate and send packets (active state) or sleep (idle state) [2]. The *on-off* model is based on the probability to stay in active or idle state for n iterations. This probability is defined by the following equation:

$$P_r\{X = n\} = \nu(1-\nu)^{n-1}, \qquad n \geq 1 \tag{1}$$

In (1), parameter ν is defined as the probability of transition from one state to another. In *on-off* model, p represents the probability of transitioning from active to idle state and q represents the reverse probability of transitioning from idle to active state. As a result, parameters p and q define the frequency and duration of packet generation. Based on the values of parameters p and q, the frequency of switching between states can be high, and may affect the *nm_tx* performance. For example, parameters p and q could force *nm_tx* to alternate very fast between idle and active state. This fast alternation between states limits the number of packets that *nm_tx* can generate and send in active state. If active state period is brief, and only few packets are generated; then, in order for *nm_tx* to send these packets, packet synchronization must be initialized. Executing packet synchronization for a small number of packets limits the effectiveness of netmap packet batching optimization method.

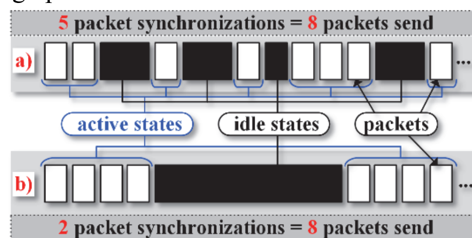

Fig. 1. Packet batching efficiency in the case of fast switching between active and idle states.

Fig. 1 demonstrates this problem by depicting two cases. In case a), switching between states is high, and for sending

eight packets, *nm_tx* executed five synchronizations. In case b), switching between states is slower compared to case a) and for sending the same number of packets, *nm_tx* executes only two packet synchronizations. In order for packet batching to be effective, *nm_tx* should send at least 512 (preferably 1024) packets per packet synchronization. The simplicity of (1) allows relatively easy implementation of *on-off* model from a programing point of view. The challenge comes with finding a solution for packet batching optimization method.

Two methods were combined to make efficient implementation of *on-off* model within *nm_tx*. First, avoidance of values for parameters p and q that cause very fast switching between active and idle states. Second, instead of generating one packet per iteration, *nm_tx* generates a group of packets (e.g. 64 packets) per iteration. This will guarantee that packet batching optimization will be at an acceptable level even if *nm_tx* stays in active state only for one iteration.

The Poisson model is more complex compared to *on-off* model, and it requires an extensive analysis in order to find the most efficient implementation. The probability of k events per unit interval with the Poisson distribution equals (2), where λ is the average number of events per interval and k takes values 0, 1, 2…

$$P(k) = e^{-\lambda} \frac{\lambda^k}{k!} \quad (2)$$

Traffic emulation based on netmap API requires a high value of λ (preferably $\lambda \geq 1024$) which will allow the utilization of packet batching optimization. For example, if λ is smaller than 128, then, the average number of packets for each time interval will be also smaller than 128. As with the case of *on-off* model, this will limit *nm_tx* in generating a sufficient number of packets per system function call, and, consequently, lower the packet throughput of emulated IP traffic. Hence, *nm_tx* must use an algorithm that can generate Poisson random numbers for $\lambda \geq 1024$. In addition, execution time of the algorithm must be sufficiently short, so that it does not affect the traffic generator performance. In the following section, we analyze different algorithms for generating numbers that follow the Poisson distribution.

IV. Analysis of Different Algorithms for Generating the Poisson Distribution

The analysis of the algorithms for generating Poisson random numbers is focused on two parameters: (*i*) execution time; (*ii*) ability to generate random numbers for a larger value of λ – some algorithms calculate $e^{-\lambda}$, which can cause variable overflow. For example, in programing language C, variable type *long double* will overflow (minimal value is 5×10^{-324}) for $\lambda > 700$.

The first algorithm, created by Donald Knuth [3] is shown as a pseudo code in Fig. 2. Knuth's algorithm is very simple to implement and can be efficient for small values of λ. The idea of Knuth's algorithm comes from the fact that the waiting time between two events occurring in thr Poisson process is an exponentially distributed random variable [4], [5]. Thus, Knuth's algorithm counts arrivals in an interval by simulating the times between arrivals and adding them up until the time sum spills over the interval. Knuth's algorithm is simple and easy to implement. It is commonly used despite its poor performance for large arguments. The problem with Knuth's algorithm is that the expected run time of the loop is proportional to λ.

```
set L to e^-λ
set k to 0
set p to 1
set rnd to 0
do
    k++
    set rnd to uniform number [0, 1]
    set p to p × rnd
while p > L
output k - 1
```

Fig. 2. Pseudocode of Donald Knuth algorithm for generating Poisson random numbers.

Besides the poor performance for larger λ, Knuth's algorithm also overflows variable L in Fig. 2 if $\lambda > 700$. However, the problem with variable overflow can be avoided with the following transformations:

$$x_1 \cdot x_2 \cdots x_{i-1} \cdot x_i \cdot x_{i+1} \cdots x_n \leq e^{-\lambda}, \quad 0 < x_i < 1 \quad (3)$$

$$e^{\lambda} (x_1 \cdot x_2 \cdots x_{i-1} \cdot x_i \cdot x_{i+1} \cdots x_n) \leq 1 \quad (4)$$

$$\left(e^{step}\right)^k \cdot e^{\lambda - k \cdot step} (x_1 \cdot x_2 \cdots x_{i-1} \cdot x_i \cdot x_{i+1} \cdots x_n) \leq 1, \quad k = \lambda / step \quad (5)$$

Based on (5), a modified version of Knuth's algorithm (named Junhao's algorithm) was designed, and it is capable of generating Poisson random numbers for any value λ. Junhao's algorithm, shown in Fig. 3, calculates the left side of (5) by multiplying values in steps until it reaches e^{λ}. Incrementally multiplying the value to the left side of (5) guarantees that variables in Fig. 3 will not overflow.

```
set step to 500
set L to λ
set k to 0
set p to 1
set rnd to 0
do
    k++;
    set rnd to uniform number [0, 1]
    set p to p × rnd;
    if p < e and L > 0
        if L > step
            set p to p × e^step
            set L to L – step
        else
            set p to p × e^L
            set L to -1
while p > L
output k - 1
```

Fig. 3. Junhao's algorithm pseudo code. Junhao's algorithm generates random number for any value of λ.

The next algorithm that can generate Poisson random number for any value λ is named Inverse algorithm (Fig. 4). Inverse algorithm only generated one uniform random number in interval [0, 1]. Afterwards, Inverse algorithm sums values based on (2) until the sum overflows the generated uniform number. Inverse algorithm is simple and easy to implement, but suffers from long execution time, similarly to Knuth's and Junhao's algorithms.

The algorithm designed by A. C. Atkinson [6] is based on the normal approximation of the Poisson distribution.

We named this algorithm Lopt and it is shown as a pseudo code in Fig. 5. The error of Lopt algorithm is smaller as the value of λ rises, and, it is recommended for $\lambda > 30$. Unlike previous algorithms, Lopt algorithm is designed to optimize execution time. Still, Lopt has overflow issues with the calculation of *log*(*n!*) which is marked with a red color in Fig. 5. The *log*(*n!*) can be transformed as in (6), and, then it is possible to be calculated without overflow.

$$\log(n!) = \log(1) + \log(2) + \log(3) + ... + \log(n) \quad (6)$$

```
set p to e^-λ
set k to 0
set s to p
set rnd to uniform number [0, 1]

while rnd > s
   k++
   set p to p × λ / k
   set s to s + p

output k - 1
```

Fig. 4. Pseudo code of Inverse algorithm.

```
set n, v, y, x, lhs, rhs to 0
set c to 0.767 – 3.36/ λ
set beta to π/ sqrt(3× λ)
set alpha to beta × λ
set k to log(c) - λ - log(beta)
while 1
   set u to uniform number [0, 1]
   set x to (alpha - log((1.0 - u)/u))/beta
   set n to round(x)
   if n < 0
      continue
   set v to uniform number [0, 1]
   set y to alpha - beta × x
   set lhs to y + log(v/pow(1.0 + e^y, 2))
   set rhs to k + n × log(λ) - log(n!)
   if lhs ≤ rhs
      break
output n
```

Fig. 5. Pseudo code of Lopt algorithm.

However, calculation based on (6) is very slow and downgrades Lopt algorithm execution time. A better solution is to represent factorial as a gamma function ($\Gamma(n+1) = n!$). Then, applying asymptotic approximation of *logΓ(x)* leads to (7) that can provide fast calculation of *log(n!)*.

$$\log\Gamma(x) \approx (x - \frac{1}{2})\log(x) - x + \frac{1}{2}\log(2\pi) + \frac{1}{12x} \quad (7)$$

The error of asymptotic approximation decreases with larger values of arguments. Thus, the optimal solution is to precalculate *log(n!)* for $n \leq 256$, and to use (7) for $n > 256$.

Finally, GSL – GNU Scientific Library for C and C++ programing languages has functions for generating Poisson random numbers. Fig. 6 shows usage of this library in C. Function *gsl_ran_poisson* is using a number of different methods for efficient generation of Poisson random values. The downside of the GSL libraries is that they are not in the standard distribution of the Linux operating system; so, they must be installed in order to provide traffic generators.

```
gsl_rng_env_setup();
const gsl_rng_type *T = gsl_rng_default;
gsl_rng *r = gsl_rng_alloc(T);
return (int)gsl_ran_poisson(r, lambda);
gsl_rng_free (r);
```

Fig. 6. Generating Poisson random numbers with GSL numerical library.

V. Performance Analysis of Poisson Distribution Algorithms

The algorithms were implemented in C programing language. The execution time of algorithms was analyzed by measuring difference in timestamps before and after algorithm call. The execution time was measured in nanoseconds. During testing, each algorithm is called million times to generate a random discrete value and all tests were repeated for values of λ: 20, 50, 100 and 5000. Execution time distribution for $\lambda = 20$ is shown in Fig. 7. Fig. 7. shows that Inverse algorithm has the best execution time, even better than Knuth's algorithm. Inverse algorithm generates a uniform random number only once per call which is more efficient than other algorithms that generate a uniform random number in each loop iteration.

Overall, as expected, measurement results in Fig. 7 show that simpler algorithms have lower execution times, if λ has small values ($\lambda \leq 20$). Fig. 8 and Fig. 9 show measurement results for $\lambda = 50$ and $\lambda = 100$, respectively. As λ rises, Inverse, Knuth's and Junhao's algorithms have longer execution times then Lopt and GSL based algorithm. Based on results in Fig. 8 and Fig. 9, Lopt and GSL algorithm should be used for $\lambda > 100$. Fig. 10 shows execution time distribution for Lopt and GSL algorithms for different λ values (800, 1500 and 5000). As λ continues to rise, Lopt execution time stays fixed, while GSL gradually rises. For $\lambda > 800$, Lopt algorithm has the lowest execution time.

VI. Implementation of the Poisson and *ON-OFF* Model within *NM_TX*

As mentioned in Section 3, traffic emulation on netmap API requires larger values of λ (usually $\lambda \geq 1024$) in order to achieve a high packet throughput. Based on analysis presented in Section 5, *nm_tx* was developed with Lopt algorithm as this algorithm delivers the optimal performance within restricted environment of netmap platform. Fig. 11 shows the basic execution diagram of *nm_tx* with the Lopt algorithm. For each time interval T_i, Poisson random number (variable *RN*) was generated using the Lopt algorithm. Afterwards, *nm_tx* sends packets using the netmap API, while incrementing *number of packets* (*NoP*) variable until NoP reaches RN. Next, *nm_tx* sleeps for the rest of time interval ($T_i - T_g$, where T_g is the time elapsed during transmission of *RN* packets).

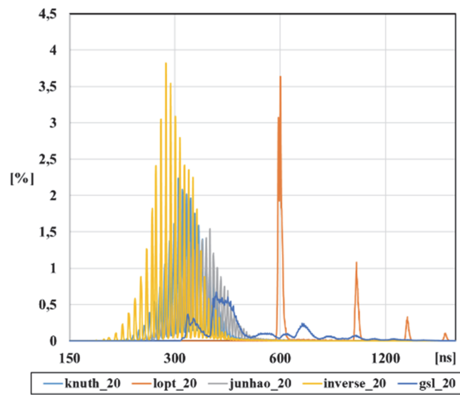

Fig. 7. Execution time distribution of all analyzed algorithms for $\lambda = 20$.

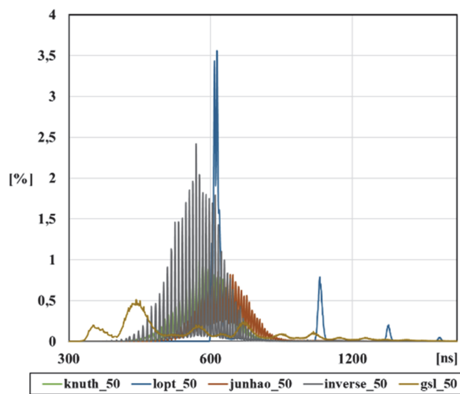

Fig. 8. Execution time distribution of all analyzed algorithms for $\lambda = 50$.

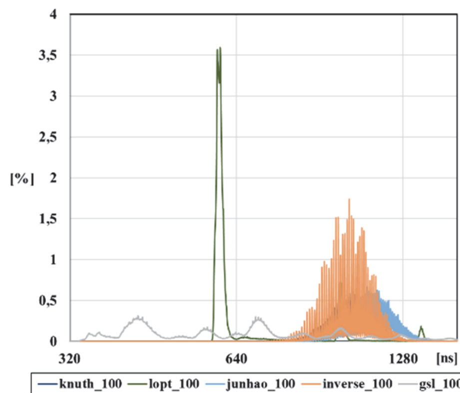

Fig. 9. Execution time distribution of all analyzed algorithms for $\lambda = 100$.

Fig. 12 shows workflow diagram of *nm_tx* with the *on-off* model. In contrast to the Poisson model, simplicity of *on-off* model results in implementation that has an elementary programing code and a negligible execution time. The *on-off* model depicted in Fig. 12 determines active or idle state (based on parameters p and q) in each iteration. If the state is active, then, *nm_tx* generates and sends a *predefined number of packets* (*PNoP*). Afterwards, *nm_tx* sleeps during the remaining part of the time interval, $T_i - T_g$, where T_g is the time elapsed during transmission of *PNoP* packets. If the state is idle, *nm_tx* just sleeps for the time interval with the duration T_i. Parameter *PNoP* is usually set to 64 or more, which will guarantee utilization of packet batching optimization method.

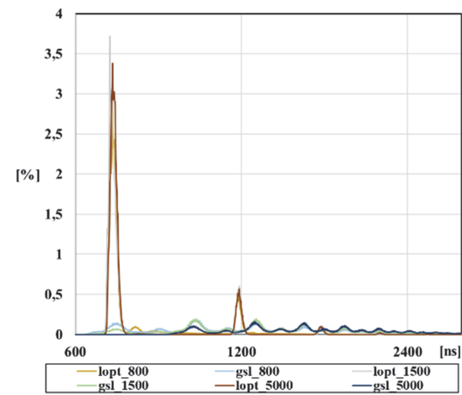

Fig. 10. Execution time distribution of Lopt algorithm and GSL based algorithm for λ values: 800, 1500 and 5000.

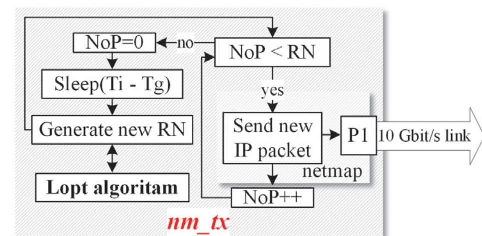

Fig. 11. Basic workflow diagram of *nm_tx* with Lopt algorithm.

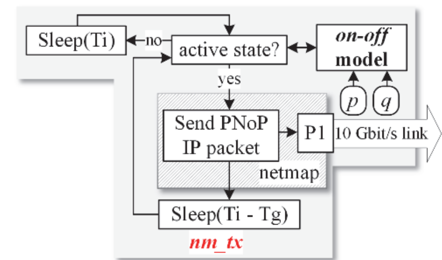

Fig. 12. Basic workflow diagram of *nm_tx* with the *on-off* model.

Let us summarize *nm_tx* capabilities. In the case of the Poisson model, as long as parameter is $\lambda \geq 1024$, *nm_tx* will not be limited by packet batching optimization, and, it will provide any desired packet rate on 10 Gbit/s links. In the case of *on-off* model, avoidance of values for parameters p and q that cause very fast state switching and generating a group of packets (e.g. 64 packets) per iteration will guarantee that *nm_tx* can provide any desired packet rate on 10 Gbit/s links. These mentioned limitations of netmap framework affect transmitting (packet generation) side, as well as the receiving side. For example, application that receives packets through a netmap framework will struggle to measure the distribution of received packets on a relatively small scale – that is smaller than the optimal batch size. Users define different formats of IP packets (packet headers and payloads), after which *nm_tx* creates packets and stores them in memory. Afterwards, packets are copied to corresponding Tx netmap rings. Only if users define a large number of packets (e.g. a sequence that has more than 2000 different packets), *nm_tx* might struggle with performance.

VII. Testing nm_tx with Poisson and *on-off* Models

Testing of *nm_tx* was performed by using two machines M1 and M2 connected with 10 Gbit/s link. M1 was configured to run nm_tx, which can generate and send emulated IP traffic towards M2 using ether the Poisson or *on-off* model. M2 was configured to run another netmap-based application (*nm_rx*) that can receive and analyze packets with a high packet throughput. Two verification tests were conducted - one for each traffic model. In the first test, *nm_tx* was configured to use the Poisson model. Parameters were set to λ = 7000 and T_i = 1 ms, which allowed *nm_tx* to generate on average 7 Mpps. The length of generated packets was 64 bytes.

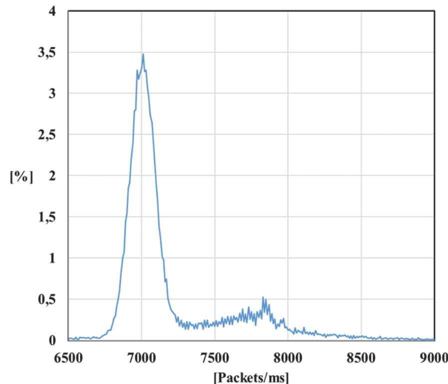

Fig. 13. Time distribution of received packets on M2.

Fig. 13 shows the time distribution of received packets at M2. It roughly matches the Poisson distribution, which confirms that *nm_tx* is generating packets correctly. The distortion in Fig. 13 is caused by packet batching that limits *nm_rx* capability to accurately measure time intervals of received packets.

Next, we conducted a test with *nm_tx* (machine M1) configured to use the *on-off* mode. As the implementation must include batching, it disrupts the *on-off* distribution. Application *nm_rx* (machine M2) was set to measure the number of packets received in the netmap ring after each synchronization, termed as the batch size. We will compare the batch size versatility in the case of the *on-off* model, with the batch size versatility in the case of the fixed packet rate. In particular, for the fixed packet rate, *nm_tx* was set to generate a fixed number of packets in each time interval. The fixed number of packets was 7000 per time interval of 1 ms, which results in the throughput of 7 Mpps. Application *nm_rx* only records if a specified batch size appears during the testing period, which was 30 seconds for both tests. Fig. 14-a shows results in the case when *nm_tx* was not using the *on-off* model. Possible batch sizes at the reception were between 0 and 1024, and Fig. 14-a shows that about 50 % of these numbers appear during the measurement at *nm_rx*. The second subtest was conducted with *nm_tx* set to use the *on-off* model. Parameters *p* and *q* were set to the value of 0.3 and *PNoP* was set to 64. Application *nm_tx* was generating on average 7 Mpps – the same packet throughput as in the case of the first subtest. Fig. 14-b shows results, which demonstrate a significant increase of different batch sizes appearing during the measurement. This result verifies that traffic distribution diversity increases when *nm_tx* is using the *on-off* model.

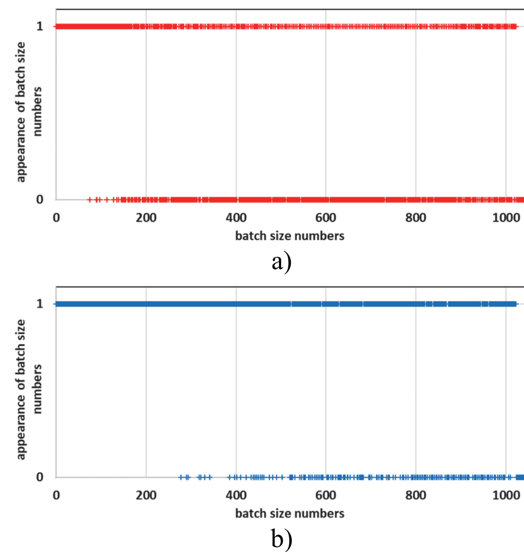

Fig. 14. The batch sizes appearance in two cases: a) *nm_tx* with fixed packet rate; b) nm_tx with *on-off* model.

VIII. Conclusion

In this paper, we analyze the advantages and disadvantages of netmap platform for IP traffic emulation. We describe the characteristics of the *on-off* and Poisson model. The *on-off* model requires an increase in the number of packets sent per iteration in order to utilize packet-batching optimization of the netmap platform. Different algorithms for the generation of the Poisson traffic were described and analyzed. The focus of algorithm analysis was the execution time and ability to support high values of λ. We concluded that the Lopt algorithm is optimal for creating a netmap traffic generator (*nm_tx*) that can emulate IP traffic with the Poisson distribution at high packet rates. We tested *nm_tx*, and, confirmed that it worked correctly. In future work, we plan to implement other IP traffic models within *nm_tx*.

References

[1] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *USENIX, the Advanced Computing Systems Association*, Bellevue, 2012.

[2] H. J. Chao and B. Liu, *High Performance Switches and Routers.* Wiley-Interscience, 2007

[3] D. E. Knuth, *The Art of Computer Programming*. Addison Wesley, 1969.

[4] P. McQuighan, “Simulating Poisson Process,” Department of Mathematics - University of Chicago, 23 July 2010. [Online]. Available: https://goo.gl/h383DS. [Accessed 2018].

[5] K. K. Karakacha, “Exponential Distribution: Its Constructions, Characterizations And Related Distributions,” 2009. [Online]. Available: https://goo.gl/yGYTrK. [Accessed 2018].

[6] A. C. Atkinson, “The Computer Generation of Poisson Random Variables,” *Journal of the Royal Statistical Society. Series C (Applied Statistics),* vol. 28, no. 1, pp. 29-35, 1979.

[7] H. Redzovic, A. Smiljanic and Z. Cica, “Traffic modelling challenges on 10 Gbit/s links using netmap platform,” *Proceedings of Telfor 2018.*, Belgrade, Serbia, November 2018.