PacketUsher: a DPDK-Based Packet I/O Engine for Commodity PC

Zhijun Xu*, Liang Zhou†, Li Feng‡, Yujun Zhang†, Jun Zhang§ and Huadong Ma*
*Beijing University of Post and Telecommunications, Beijing, CHINA
Email: {xuzhijun, mhd}@bupt.edu.cn
†Institute of Computing Technology, Chinese Academy of Sciences, Beijing, CHINA
Email: {zhouliang, zhmj}@ict.ac.cn
‡Faculty of Information Technology, Macau University of Science and Technology, Macau, CHINA
Email: lfeng@must.edu.mo
§Inner Mongolia University, Hohhot, CHINA
Email: zhangjun@imu.edu.cn

***Abstract*—Deploying network applications on commodity PC is increasingly important because of its flexibility and cheapness. Due to high packet I/O overheads, the performance of these applications are low. In this paper, we present PacketUsher, an efficient packet I/O engine based on the libraries of DPDK. By replacing standard I/O routine with PacketUsher, we can remarkably accelerate both I/O-intensive and compute-intensive applications on commodity PC. As a case study of I/O-intensive application, our RFC 2544 benchmark over PacketUsher achieves same testing results as dedicated commercial device. For compute-intensive application, the performance of our application-layer traffic generator over PacketUsher is more than 4 times of the original value and outperforms existing frameworks by about 3 times.**

***Index Terms*—network application; packet I/O; DPDK; I/O-intensive; compute-intensive**

I. Introduction

Software packet processing on commodity PC is an ideal choice to deploy network applications, especially after the thriving of Network Function Virtualization (NFV) [1]. It is inexpensive to operate, easy to switch between vendors and perfect to accommodate future software innovations [2]. While a significant step forward in some respects, it was a step backwards in others. Flexibility on commodity PC is at the cost of discouraging low performance, which is mainly restricted by packet I/O overheads. For example, the sendto() system call of FreeBSD averagely takes 942ns to transmit packets, and RouteBricks (a software router) reports that 66% CPU cycles are spent on packet I/O [3].

To address the issue of costly packet I/O, current works anticipate to bypass Operating System and design novel packet I/O frameworks to take direct control of hardware. Research [4] demonstrates that replacing raw packet I/O APIs in general purpose OS with novel packet I/O frameworks like Netmap [5] can transparently accelerate software routers, including Open vSwitch [6] and Click [7].

PF_RING [8] is a novel packet I/O framework on commodity PC. Its zero copy version can achieve line rate (14.881 Mpps) packet I/O on 10 Gbit/s link [9]. However, this version is not free for commercial companies or common users. The open-source Netmap usually takes 90 CPU cycles to send or receive packets [5]. But it is not convenient to deploy (sometimes need to re-compile Linux kernel) and suffers packet loss at high frame rate. Intel DPDK [10] is a set of open-source libraries for high-performance packet processing. It reduces the cost of packet I/O to less than 80 CPU cycles [10]. Many companies (Intel, 6WIND, Radisys, etc.) have already supported DPDK within their products. In this paper, we use the libraries of DPDK to design an efficient packet I/O engine for common users.

We argue that the packet I/O engine for commodity PC should have four properties: low coupling with user applications, multi-thread safe, simple packet I/O API and high-speed packet I/O performance. Such design goal motives us to implement the packet I/O engine of PacketUsher on commodity PC. PacketUsher brings noticeable performance improvement for both I/O-intensive and compute-intensive applications. For example, it makes our RFC 2544 benchmark (I/O-intensive) have same testing results as dedicated hardware, and makes our application-layer traffic generator (compute-intensive) gain more than 4 times of performance improvement.

The remainder of this paper is organized as follows. In Section II, we introduce some background knowledge. Section III shows PacketUsher and its performance evaluation. Section IV presents two case studies.

II. Background Knowledge

A. Overheads of Standard Packet I/O

Standard packet I/O mechanism in general purpose OS is interrupt-driven. It has three overheads: interrupt handling, buffer allocation and memory copy.

Interrupt handling: At high frame rate, interrupt-driven mechanism suffers the problem of receive livelock [12]. Previous works [3] [4] [13] utilize batch processing to mitigate receive livelock. However, some received packets may be dropped if the OS fails to handle interrupt requests timely. Another possible method is replacing interrupt-driven mechanism with polling which periodically checks the arrival of packets on NICs. Its drawback is that we must use custom drivers instead of standard ones. Compromised method is Linux NAPI [14] which uses interrupt to notify the arrival of packets and then uses polling to receive batch of packets.

Buffer allocation: Buffer allocation is another time-consuming action. Allocating buffers for transmitted or received packets costs much system resources. Previous works

(DPDK, Netmap, PF_RING and PacketShader [13]) all pre-allocate pool of fix-size packet buffers to accelerate this procedure.

Memory copy: Memory copy is the last overhead in the procedure of moving packets between physical NICs and user applications. For the reason of abstraction and deferred processing [5], whole packets are usually copied in system calls. To reduce this overhead, shared memory region is a good choice.

B. DPDK

Intel Data Plane Development Kit (DPDK) is a set of open-source libraries and drivers aiming at high-speed packet I/O on commodity PC. Currently, it has already supported many PCI NICs and paravirtualization interfaces including e1000 family, ixgbe, virtio-net and etc. DPDK can be applied to many network applications such as OpenFlow switch [15], load balancer and traffic QoS control, to name a few.

DPDK leverages many effective methods to reduce overheads of packet I/O. For interrupt handling, it utilizes polling to avoid the problem of receive livelock. For memory copy, batch of packets are processed in system calls to reduce amortized per packet costs. In DPDK, memory alignment, Hugepage and memory pool are all used to reduce overheads of buffer allocation. It is easy for DPDK to achieve line rate packet I/O on 1..10 Gbit/s links. Some packet processing functions have been benchmarked up to 160 Mpps [10].

III. PacketUsher

DPDK provides mechanisms and libraries to remove packet I/O related overheads. However, it is the programmers responsibility to build a high-performance and safe packet I/O engine based on these libraries, and sometimes that is not an easy task for freshman in this field. There are four obstacles for common users:

Complicated mechanism. DPDK takes direct control of hardware and system resources. Programmers should be familiar with the packet I/O mechanisms in DPDK, otherwise writing correct program is a tough work.

Multi-thread unsafe. Many system resources in DPDK (Hugepage, TX/RX queues of Ethernet devices, etc.) are not multi-thread safe. Programmers should carefully write their codes to avoid system crash.

Fake low performance. Common users usually lack experience in utilizing libraries of DPDK to achieve high performance. They also do not know how to properly configure parameters.

Different I/O APIs. Because DPDK operates directly on underlying system resources, its packet I/O APIs are not the same as the ones in standard Linux. User applications usually have high coupling with libraries of DPDK.

The above four problems of DPDK motivates us to design a simple, safe, low-coupling and high-performance packet I/O engine named PacketUsher. Normal users just need to run PacketUsher and replace their raw packet I/O APIs with our similar ones to enjoy high-performance packet I/O.

A. Architecture of PacketUsher

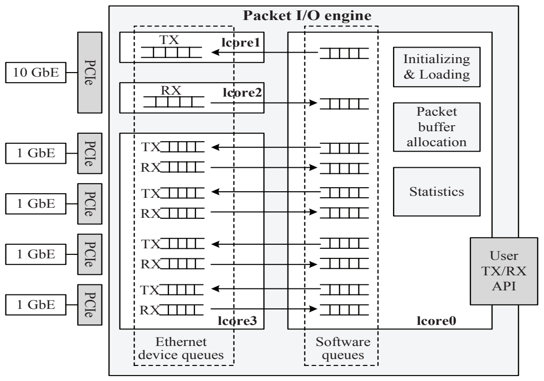


Fig. 1. Architecture of PacketUsher

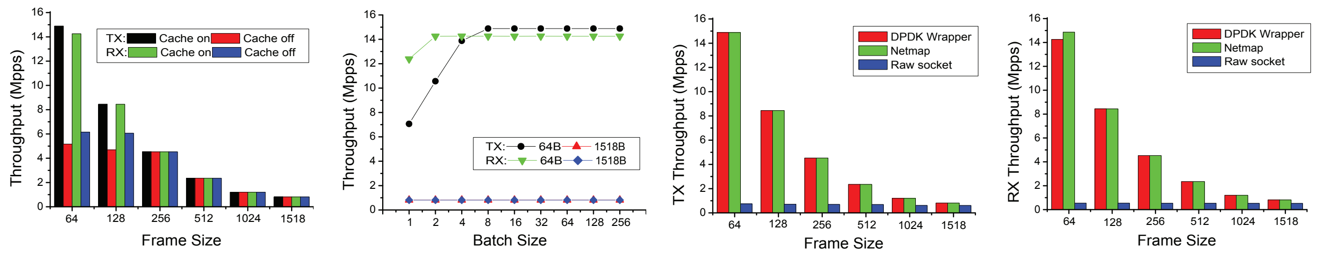
Our practical experience shows that some resources and functions in DPDK are not multi-thread safe. For example, it may result in segment fault or network subsystem crash when more than one threads call the TX/RX functions of Ethernet device queue. The reason is that these multi-thread unsafe functions may cause data inconsistency (wrong descriptor value) in drivers. In order to provide a safer packet I/O environment for applications, only one thread in PacketUsher has the privilege to initialize and configure system resources. Additionally, we add a layer of multi-producer, multi-consumer software queues between unsafe Ethernet device queues and user applications. Figure 1 depicts the simplified architecture of PacketUsher in the case of having four 1G NICs and one 10G NICs. In such architecture, lcore0 responds to the request of packet I/O from applications, and executes dequeue or enqueue actions on corresponding software queues. It also continuously moves packets between Ethernet device queues and software queues.

For high-performance packet I/O, we have already accumulated some experience in how to schedule system resource and configure parameters. In PacketUsher, both 1G and 10G NICs have only one pair of TX/RX device queues, and such design avoids the problem of synchronization between multiple queues. For 1G NICs, one CPU core could support four NICs to undertake line rate (1.488 Mpps) packet I/O. For 10G NICs, PacketUsher needs two separate CPU cores to achieve line rate (14.881 Mpps) packet transmission or reception. The line rate throughput of PacketUsher demonstrates that added software queues do not bring performance penalty.

Finally, showing users only familiar I/O APIs makes application migration easier. All the details of DPDK are limited in PacketUsher, and similar I/O APIs bring low coupling between user applications and underlying packet I/O engine.

B. Parameter Configuration

For programmers who want to leverage the libraries of DPDK to build high-performance packet I/O engine, it is necessary to configure these libraries with appropriate parameters. These parameters include cache option in memory pool, packet batch size, TX/RX queue size of Ethernet devices, size of



(a) Throughput versus cache (batch size = 16)

(b) Throughput versus batch size (cache on)

(c) TX throughputs

(d) RX throughputs

Fig. 2. Performance evaluation of PacketUsher

memory pool and TX/RX Prefetch, Host, Write-back threshold values. In order to quantify their influence on packet I/O performance, we implement a simple program that repeatedly transmits and receives packets via the packet I/O API of PacketUsher. For different configurations, we measure the TX and RX throughput of PacketUsher. In this paper, all experiments are conducted on the system that equipped with Intel Xeon E5-2620 2.0 GHz CPU and 1333 MHz 8G memory. The Ethernet interfaces are eight Intel I350 GbE NICs and two Intel 82599EB 10 GbE NICs. Our experiment results show cache option in memory pool and packet batch size significantly affect packet I/O throughput, and other parameters have little impact on the throughput. In PacketUsher, we configure the above two parameters with optimal values.

The first parameter is cache option in memory pool. The cost of multiple cores accessing the ring of free buffers (with locks) in memory pool is high. Programmers could configure memory pool to maintain a per-core buffer cache at creation time. Allocating buffers from per-core cache and doing bulk requests to ring of free buffers could reduce lock overhead therefore gaining better performance. As illustrated on Figure 2(a), the cache option in memory pool has notable influence on both TX and RX throughput of PacketUsher for all frame sizes. For example, the TX throughput with cache off (5.168 Mpps for 64 bytes packet) is about 34.7% of that with cache on (14.881 Mpps for 64 bytes packet). In the case of cache on, the TX throughput (14.881 Mpps for 64 bytes packet) is line rate on 10 Gbit/s link, and the RX throughput (14.255 Mpps for 64 bytes packet) is close to line rate. In PacketUsher, we turn the cache option on.

The second parameter is packet batch size. In DPDK, enqueue and dequeue operations process a batch of packets per function call to reduce the amortized costs of every packet. Figure 2(b) shows the TX and RX throughput of PacketUsher on different batch sizes. For 1518 bytes packet, we see that both TX and RX throughput (0.813 Mpps) are line rate on 10 Gbit/s link. For 64 bytes packet, both TX and RX throughput improve when the batch size increases. While TX throughput in packet-by-packet approach (batch size = 1) is 7.064 Mpps, the TX throughput (14.881 Mpps) reaches line rate with batch size of 16 and RX throughput (14.255 Mpps) is close to line rate as well. In PacketUsher, we set packet batch size to 16.

C. *Performance Evaluation and Comparison*

As comparison, we measure the TX and RX throughput of Netmap and Linux raw socket as well. Figure 2(c) and Figure 2(d) show that both PacketUsher and Netmap have much better TX/RX throughput than Linux raw socket. The TX throughput of PacketUsher and Netmap are line rate (14.881 Mpps for 64 bytes packet) on different frame sizes. The RX throughput of PacketUsher (14.255 Mpps) and Netmap (14.863 Mpps) come close to line rate as well. All the experiment results of PacketUsher are steady values, and the number of lost packets is zero.

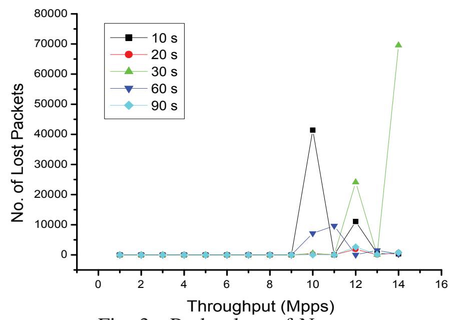


Fig. 3. Packet loss of Netmap

However, the high-performance of Netmap is at the cost of packet loss. For the evaluation of packet loss in Netmap, we send a stream of packets (64 bytes) at specific throughput, and count the number of received packets at destination NIC. For different run time, the numbers of lost packets are calculated and depicted on Figure 3. We find that packet loss occurs randomly when the throughput is larger than 9 Mpps. For example, the numbers of lost packets between 10 Mpps and 14 Mpps are 508, 47, 24104, 10 and 69549 when our program runs 30 second. While the throughput increases, the numbers of lost packets do not increase as well. From the aspect of running time, the number of lost packets in 90s (794) is not larger than that in 30s (69549) at throughput of 14 Mpps. In Netmap, batch of packets are received in per network interrupt. If the OS fails to handle this interrupt timely, this batch of packets would be dropped.

In PacketUsher, we have to point out that the performance gap between RX throughput (14.255 Mpps) and line rate (14.881 Mpps) is not caused by added software queues. The RX queues of Ethernet devices are never full. This means that all received packets in queues are passed to user applications. The real reason is that packet I/O cores do not run fast enough to move all packets from physical NICs to RX queues of Ethernet devices successfully.

IV. Case Study

For the evaluation of PacketUsher, we implement both I/O-intensive and compute-intensive network applications on top of it. I/O-intensive applications spend most of time on packet I/O, and undertake simple actions on packet headers. Compute-intensive applications have complicated and time-consuming actions on both packet headers and payload. Experiment results show that leveraging PacketUsher to transmit or receive packets brings encouraging performance gain for both of them.

A. *I/O-Intensive Application*

We select RFC 2544 benchmark as the example of I/O-intensive application. RFC 2544 defines the methodology to test performance (such as throughput, packet loss, back-to-back value, etc.) of network interconnected devices [16]. Its benchmark transmits packets (with timestamp) at high frame rate, and then receives these packets at another physical NIC. The key factor for this application is high-speed packet I/O. In commercial products, such as Spirent TestCenter [11] and IXIA FireStorm [17], such application is built on dedicated hardware. It can also be implemented on programmable hardware including NetFPGA [18] and Network Processor [19].

TABLE I
RFC 2544 testing results: benchmark over PacketUsher(PU) and TestCenter(TC)

Frame size (Byte)		Throughput (Kpps)	Loss rate (%)	Back-to-back (20s)
64	PU	1488	0	29,761,904
	TC	1488	0	29,761,905
128	PU	844	0	16,891,888
	TC	844	0	16,891,892
256	PU	452	0	9,057,968
	TC	452	0	9,057,972
512	PU	234	0	4,699,248
	TC	234	0	4,699,249
1024	PU	119	0	2,394,624
	TC	119	0	2,394,637
1518	PU	81	0	1,625,472
	TC	81	0	1,625,488

On commodity PC, this is not an easy task because of low packet I/O performance on Linux. According to RFC 2544, we write a benchmark on general purpose OS, which utilizes the APIs of PacketUsher to transmit and receive packets. In order to evaluate its performance, we use both Spirent TestCenter and our benchmark to test the performance of a Gigabits Switch (H3C S5024PV2-EI) [20]. As presented on Table 1, the testing results of our benchmark and TestCenter are the same.

B. *Compute-Intensive Application*

Application-layer traffic generator is widely used to test the performance of application-layer devices, including web

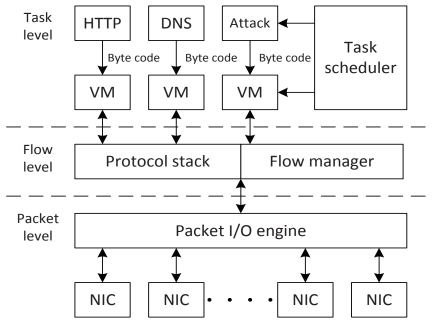


Fig. 4. Architecture of application-layer traffic generator

servers, application-layer firewalls, Intrusion Detection Systems, and etc. It could emulate both clients and servers to generate mixture of application and attack flows. Similar to the one presented in research [21], we design and implement an application-layer traffic generator which has already been used in some institutes. Its architecture is showed on Figure 4. In this system, we design one generic language to describe mixture of application and attack flows, and then its compiled byte codes are executed in virtual machines to generate realistic and responsive flows. Obviously, such application is compute-intensive.

FPS value refers to the number of flows generated per second by the application-layer traffic generator. It reflects the ability of system to simulate both clients and servers. In this paper, we use the FPS value to evaluate the performance of our application-layer traffic generator. Our system is configured to generate FTP flows and simulate both clients and servers. To minimize the impact of middlebox on generated flows, the system is connected to a switch. All generated flows are complete and steady.

For comparison, we implement raw socket, PF_RING, Netmap and PacketUsher on the application-layer traffic generator. Because PF_RING zero copy version is not free, we implement its normal version on our system. The normal version PF_RING utilizes buffer rings in kernel space and polling to speed up packet I/O. Netmap accelerates packet I/O mainly through light weight metadata representation, pre-allocated packet buffers, direct and protected packet buffer access.

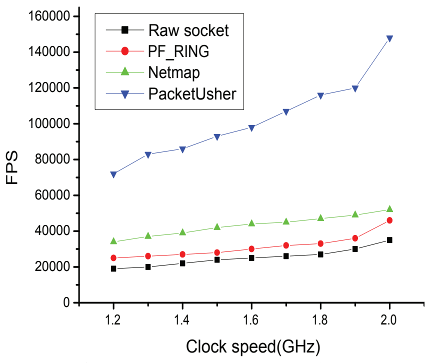


Fig. 5. FPS value versus CPU frequency

Figure 5 presents the FPS values of our application-layer traffic generator over raw socket, PF_RING, Netmap and PacketUsher. The original FPS value of our system over raw socket is 35,000 flow/s (CPU speed 2.0 GHz). For novel packet I/O framework of PF_RING, the FPS value is 46,000 flow/s (CPU speed 2.0 GHz). The performance of our traffic generator over Netmap is better than that over PF_RING. It could generate 52,000 flows per second at CPU speed 2.0 GHz. From Figure 5, we observe that both PF_RING and Netmap could not improve the performance significantly. On the same platform, PacketUsher improves the performance of our application-layer traffic generator encouragingly, and the FPS value is 148,000 flow/s (CPU speed 2.0 GHz) which is more than 4 times of that over raw socket. Additionally, the FPS value of our system over PacketUsher is about 3 times of that over Netmap and PF_RING.

Costly packet I/O in raw socket and PF_RING results in the low performance of our application-layer traffic generator. Efficient Netmap removes the packet I/O bottleneck in our system, and then the FPS values are better. However, Netmap suffers the problem of packet loss. Every flow generated by the application-layer traffic generator contains some sessions, and a flow is successfully simulated only when all the sessions are complete. Packet loss in Netmap causes many flows to be connection failure. For PacketUsher, the performance bottleneck of our system comes from low payload processing speed in user applications.

V. Conclusion

In this paper, we present PacketUsher, a multi-thread safe and high-performance packet I/O engine for any applications deployed on commodity PC. End users just need to run our program and utilize the packet I/O APIs of PacketUsher to transmit or receive packets. With the help of both I/O-intensive and compute-intensive experiments, we find that PacketUsher achieves high frame rate on commodity PC and remarkably accelerates network applications. Particularly, it is suitable for applications which have strict requirements on packet loss, while Netmap suffers unpredictable packet loss at high frame rate.

VI. Acknowledgments

This research is supported by the National Program on Key Basic Research Project (2012CB315804), the National Natural Science Foundation of China (61402446 and 61572474), the National Sciences and Technology Supporting Program of China (2012BAH45B01), the Instrument Developing Project of CAS (YZ201426), the Macao Science and Technology Development Fund (013/2014/A1) and the Natural Science Foundation of Inner Mongolia (2015MS0601).

References

[1] Network Functions Virtualization - Introductory White Paper. In *ETSI* (2012).

[2] Casado, M., Koponen, T., Shenker, S., Tootoonchian, A. Fabric: a retrospective on evolving sdn. In *the first workshop on Hot topics in software defined networks* (2012).

[3] Dobrescu, M., Egi, N., Argyraki, K., Chun, B., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. Routebricks: exploiting parallelism to scale software routers. In *SOSP* (2009), pp. 15–28.

[4] Rizzo, L., Carbone, M., Catalli, G. Transparent acceleration of software packet forwarding using netmap. In *IEEE INFOCOM* (2012).

[5] Rizzo, L. Netmap: a novel framework for fast packet i/o. In *2012 USENIX conference on Annual Technical Conference* (2012), pp. 101–112.

[6] Pfaff, B., Pettit, J., Koponen, T, Amidon, K., Casado, M., and Shenkerz, S. Extending networking into the virtualization layer. In *ACM SIGCOMM HotNets* (2009).

[7] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. The click modular router. In *ACM Transactions on Computer Systems (TOCS)* (2000), 18(3):263C–297.

[8] Deri, L. Improving passive packet capture: beyond device polling. In *SANE* (2004).

[9] pfring. http://www.ntop.org/products/pf_ring/.

[10] DPDK Web Site. http://www.dpdk.org/.

[11] TestCenter. http://www.spirent.com/Ethernet_Testing/Software/TestCenter/.

[12] Mogul, J., and Ramarkishnan, K. Eliminating receive livelock in an interrupt-driven kernel. In *ACM TOCS* (2000), 15(3):217–252.

[13] Han, S., Jang, K., Park, K., and Moon, S. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM Computer Communication Review* (2010), 40(4):195C–206.

[14] Linux NAPI. http://www.linuxfoundation.org/collaborate/workgroups/networking/napi/.

[15] Pongracz. Removing roadblocks from sdn: Openflow software switch performance on intel dpdk. In *2013 Second European Workshop on Software Defined Networks (EWSDN)* (2013).

[16] Bradner, S., McQuaid, J. Benchmarking methodology for network interconnect devices rfc2544[s].

[17] FireStorm. http://www.ixiacom.com/products/firestorm.

[18] Covington, G., Gibb, G., Lockwood, J., Mckeown, N. A packet generator on the netfpga platformk. In *17th IEEE Symposium on Field Programmable Custom Computing Machines* (2009), pp. 235–238.

[19] Antichi, G., Pietro, A., Ficara, D., Giordano, S., Procissi, G., Vitucci, F. Design of a high performance traffic generator on network processor. In *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools* (2008), pp. 438–441.

[20] H3C S5024P-EI Gigabits Switch. http://www.h3c.com.cn/Products___Technology/Products/Switches/Park_switch/S5000/S5024P-EI/.

[21] Vishwanath, Venkatesh, K., and Vahdat, A. Realistic and responsive network traffic generation[c]. In *conference on applications, technologies, architectures, and protocols for computer communications, SIGCOMM 06. ACM*.