High-Performance Packet Processing and Measurements

Sebastian Gallenmüller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich
Munich, Germany
{gallenmu | scholz | wohlfart | scheitle | emmericp | carle}@net.in.tum.de

***Abstract*—Networks and network architectures are constantly evolving, manifesting in new developments such as SDN, P4, and 400G Ethernet. These novel paradigms and technologies require network researchers to investigate and to adapt their measurement facilities. We present two tools which can foster this process.**

The first tool, pos, supports a fully automated workflow for performing and evaluating network experiments. One of its key features is the testbed orchestration to maintain and recreate a specified network test environment gaining reproducible experiment results. The second tool, libmoon, is a user-friendly userspace packet processing framework based on DPDK. Among many other projects, libmoon powers MoonGen, a dedicated packet generator in broad use by the community. MoonGen's hardware-supported generation and measurement capabilities are central to our network experiments to reliably recreate measurements. Further, a survey of scientific publications and applications presents projects based on both libmoon and MoonGen.

We argue that combining pos and libmoon/MoonGen creates an ideal platform for network experiments. This platform offers an affordable price, high flexibility, ease of use and generation of reproducible experiments.

***Index Terms*—reproducible network measurements, high-performance packet processing, libmoon, MoonGen**

I. Introduction

Understanding the behavior of single network nodes is essential for researching complex network structures especially in the context of developments such as SDN, P4, or 400G Ethernet. The most basic experiment setup for testing such nodes is a two-machine network like it is required for the well-known network benchmark described in RFC 2544 [1]. Therein the first machine acts as a device under test (DuT) and the second as a packet generator representing the traffic source and sink.

Despite the simple network setup, this configuration already involves significant complexity when it comes to the internal hardware architectures and software stacks. Multiple bus systems are involved in packet processing, for instance, the NIC itself and connections offered by the CPU, such as PCIe, cache, memory, or CPU interconnects like QPI and Hypertransport. Depending on the ingress port and the kind of network traffic, different paths through such a system are traversed by the packet flow on its way to the egress port. Load-dependent limits of the interconnects may be reached, additionally influencing the performance. The software of such systems can be equally complex. Different isolation technologies such as lightweight Linux containers, realized with cgroups and namespaces, or full-fledged hypervisors, such as Xen or KVM, influence packet processing performance. Nowadays, applications employ userspace packet processing frameworks such as DPDK [2] or netmap [3]. These frameworks bypass the traditional kernel-based network stack, for improving performance properties including jitter compared to kernel-based approaches. With every single hardware and software component offering adjustment possibilities, the overall configurability of such systems is highly complex. This makes identifying the settings with best processing performance a highly complex task. Identifying suitable configurations relies on the ability to reliably measure such systems, which in turn requires the recreation of the exact configuration of a DuT.

Small variances in the traffic pattern, such as microbursts [4], can have an influence on the measured outcome. Therefore, reliably generating packet streams according to a specified pattern is crucial for recreating the outcome of an experiment. However, software-based traffic generators often fail at this task [4], [5]. Generation of traffic is only one part of the problem, with measurement capabilities being equally important. Testing high-performance network devices requires means to quantify several performance properties. Precise latency measurements are hard to perform [6], as hardware-assisted technologies are needed to determine latency in the nanosecond range necessary for network bandwidths of 10 Gbit/s and beyond.

This paper presents our approach for dealing with reliable state or traffic pattern recreation and high-performance measurements:

- We define our measurement goals and describe the means to achieve them involving our testbed and the toolchain, called *pos*, used to execute our network experiments.
- We present our main tools for conducting network experiments: *MoonGen*, the underlying framework *libmoon*, and typical use cases for both tools.

The paper is structured as follows. Section II describes the goal for general reproducibility of network experiments we aim for. In Section III, we introduce our experiment

methodology and our testbed that support the creation of reproducible experiments. Subsequently, we present our packet generator MoonGen as our main tool for creating experiments in Section IV, and typical experiments and tools built on libmoon, our general purpose packet processing framework, in Section V. After discussing our current work in progress in Section VI, Section VII concludes the paper.

II. Towards Reproducible Experiments

Independent reproduction of results is vital to understanding and validating scientific results. A full reproduction is aided not only by resulting measurement data, which is occasionally published along scientific publications, but also by meta-data such as full measurement and configuration descriptions. This includes, for example, tools and scripts used to configure, run, and evaluate the experiment. An ACM policy [7] considers reproducibility as a three-stage process, with full experiment reproduction being the final stage:

1) **Repeatability** is the minimum requirement for scientific publications. It can be achieved if experimenters can recreate their previous results by using the same hardware and tools as before.
2) **Replicability** is accomplished if other researchers can recreate the results using the same tools as the authors of the initial experiment. This requires all experiment artifacts, including measurement data, as well as tools and scripts to run and evaluate the measurement, to be made accessible.
3) **Reproducibility** requires other research groups to invest time and resources not only to understand our approach but also, to come up and implement their own. While we always encourage others to reproduce our results, the incentives for doing so are still low.

Unfortunately, reproducing results is rarely being done in computer science. Amongst other reasons, this methodology is not yet appropriately incentivized for publications, and as a consequence, rarely being done by research groups, as it causes overhead with little reward. Our group has long-term dedication to support reproducible research in computer networking [8], [9].

Through the tools and methods presented in this paper, we aim to accomplish replicable and repeatable experiments, showing a way towards more reproducible research.

III. Our Methodology

For our network experiments, we set up a testbed with the focus on testing SDN. This testbed consists of 15 different servers equipped with commodity hardware and 1G, 10G, and 40G Intel or Mellanox NICs (e.g., I350, X520, X540, X710, and ConnectX-4 Lx EN respectively). A typical experiment configuration and workflow in our testbed is depicted in Figure 1, with two experiment hosts (DuT, LoadGen) and an orchestrating server (pos). The entire experiment workflow is controlled by our own tool called pos (plain orchestrating service) running on a separate server which deploys (①),

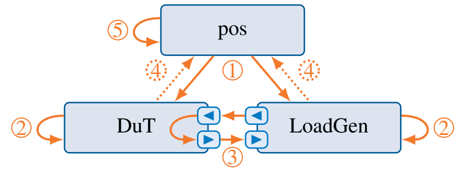

Figure 1. Typical experiment setup & workflow

configures (②), executes (③), and collects (④) the measurement artifacts of network experiments, before automatically evaluating (⑤) the measured data. As we use live images for our experiments, configuration state is lost after a reboot of the system. This enforces testbed users to use scripts for configuring experiment servers and the subsequent evaluation. On first glance this burdens users and might be seen as a disadvantage, however, it has three major benefits:

1) The testbed operators profit as they can reuse the testbed infrastructure for different projects and experiments by simply rebooting the servers and deploying the necessary configuration scripts. There is no need to maintain separate testbeds for different projects.
2) Experiment designers profit as configuration scripts enforce consistent configuration over all experiment repetitions. This lowers the possibility of accidental misconfiguration by missing or wrong configuration commands that influence the behavior of the DuT in a non-repeatable way.
3) The entire research community profits from this approach. By forcing experiment designers to use configuration scripts, pos creates inherently repeatable experiments, as the experimenter can easily rerun the experiment and repeat the results.

While our testbed can be accessed remotely, we restricted access to known and trusted persons (members of our research group and research collaboration partners). We need to rely on trust as we provide root access to all testbed machines, which is often required for network experiments. The remote access enables others to replicate our testbed experiments. Thereby, our experiments reach the second stage for reproducibility. Achieving the third stage, reproducibility, cannot be achieved by relying on the testbed and its processes but rather needs other scientists to create the same results utilizing their tools and test equipment. However, releasing testbed artifacts, including configuration scripts, test results, and evaluation scripts, can foster the reproduction of results by other research groups.

Our test setup allows for both black box tests and white box tests. For typical black box tests, data is collected on the egress and ingress ports of the load generator and used to determine metrics such as throughput and latency. White box tests are also possible by recording the behavior on the DuT itself, for instance by profiling the interrupt rate of NICs or the cache load caused by applications. Our automated testbed can record many features in parallel, leading to gigabytes of data for tests running only a few minutes. The data generated in the

testbed can be processed, plotted, and used to derive models of high precision. However, such models may be complex and difficult to handle or apply. Therefore, an important process is the selection of the crucial measurement data for explaining the relevant observed behavior, and to derive models requiring a minimal amount of data to describe the observed behavior. While in general having a lower accuracy, such models are highly valuable as they can be applied more easily and foster the basic understanding of complex systems (such as the combined HW and SW of the DuT).

We support the selection process by choosing Jupyter notebooks [10] as the backend for the evaluation (see ⑤) in Figure 1) and provide pre-built scripts for standardized evaluations, e.g., for visualizing throughput or latency. The experiment designer can extend the functionality of the Jupyter notebooks to support more specific visualizations. By using Jupyter notebooks we gain two important benefits: first, the process is automated and makes the evaluation replicable and repeatable, and second, notebooks support the interactive evaluation and subsequently the selection process for the most specific and relevant parameters.

Network experiments always require some kind of traffic source. For our tests, a traffic source is needed which must be able to fully load several 10G NICs. Small differences in the traffic source can have an influence on the observed behavior of a DuT [4], where we identified the burstiness of the traffic as the root cause. Therefore, we need the means to reliably and precisely create and measure traffic. While hardware traffic generators by Ixia [11] or Spirent [12] would be suitable devices, they are expensive and difficult to use when aiming for replication and reproduction. Another hardware-based platform is the NetFPGA with the Open Source Network Tester (OSNT) by Antichi et al. [13]. NetFPGA, and OSNT in particular, target network researchers by allowing network measurements with high precision and accuracy, utilizing programmable hardware at affordable prices. However, the hardware is still more expensive than commercial NICs, and dealing with programmable hardware is more complex than dealing with commercial NICs. Therefore, we developed our own software packet generator MoonGen, which uses commercial-off-the-shelf hardware and is easily extensible.

IV. MoonGen/Libmoon

Since publishing MoonGen in 2015, we have continued improving and adding features to it. As it matured, we deemed refactoring a necessary step. This resulted in the formation of our general purpose high-performance packet processing framework libmoon. In the following, we will explain the differences between MoonGen and libmoon, as well as the provided protocol stack.

A. MoonGen

MoonGen is a high-performance, open-source software packet generator based on the high-speed packet processing framework DPDK [2]. MoonGen can generate minimum sized packet at 10 Gbit/s (14.88 Mpps) using a single core with packets generated by user-defined Lua scripts. One important feature is the hardware-assisted timestamping of packets, which allows delay measurements in the sub-microsecond range. The timestamping feature misuses registers on modern Intel 10G NICs originally intended for the precision time protocol (PTP). Due to hardware limitations, not the entire packet flow can be timestamped but only a few thousand packets per second. MoonGen also offers precise pattern generation. By filling gaps in the packet stream with corrupt packets arbitrary traffic patterns can be created. This allows configuring the length of the gaps down to the byte level. The corrupt data typically does not influence the packet processing software of the DuT as corrupt frames are silently dropped in hardware. Our approach allows patterns to be replayed with higher precision than with other available software tools relying on waiting to introduce inter packet gaps. Features and implementation details of MoonGen are explained in more detail in the following publications [4], [14].

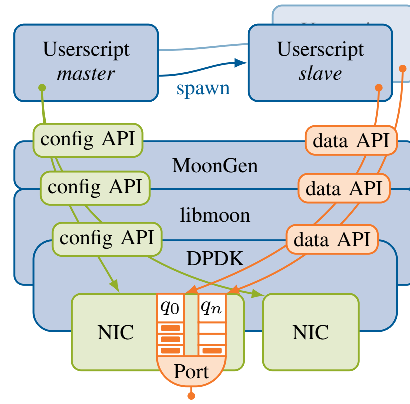

Figure 2. Architecture of MoonGen/libmoon

B. libmoon

Since the initial publication of MoonGen, multiple applications were realized on top of it (see Section V). Investigating the purpose of these applications we identified two fields of application for MoonGen. Some applications use MoonGen as it was originally intended – as a tool to generate realistic traffic to benchmark and validate their prototype [15]–[19], to simulate DDoS attacks [20], or as basis for benchmarking suites [19], [21]–[23] to compare different third-party implementations.

However, some applications relied on MoonGen because of its simple and convenient way to use DPDK [24]–[26]. Therefore, we decided to split MoonGen in two parts – a packet processing framework called libmoon and a packet generator called MoonGen. libmoon provides a simple API for DPDK. Figure 2 shows the new architecture with the former monolithic MoonGen split into libmoon and MoonGen. There, MoonGen does not use DPDK directly anymore but instead is an application based on libmoon, consisting of only a few hundred additional lines of code. Programmers can now

decide to either modify a packet generator, or to build entire applications based on libmoon.

C. MoonStack

libmoon provides the protocol stack API MoonStack that combines performance and usability, as an extension to mere byte-level operations in DPDK. MoonStack is a novel dynamic protocol stack developed to define arbitrary packet headers to conveniently implement new protocols for simple and rapid prototyping. It was designed to perform well even for packet rates beyond 10 Gbit/s while modifying packet headers at line rate.

The architecture of the stack is influenced by the general structure of libmoon user tasks offering two kinds of utility functions. Slow functions can be used to fill one or all headers of a packet buffer using only a single function call. All header fields are filled with sensible default values or user-defined values when using labels to reference specific header fields. These slow functions should be used during start-up of the application, essentially creating packet templates for the kind of traffic that the application later uses or modifies. During runtime of the application, all operations are performance-critical. Therefore, the stack provides dedicated functions that allow manipulating each header field of the stack separately. This additional layer offers data type abstractions and performs bit and byte conversions for the user. As these are lightweight, LuaJIT-compiled wrapper functions, performance is not impaired in comparison with directly performing byte-level operations on the underlying C structure. Micro-benchmarks have shown that in both cases memory locality of the sequential packet data is exploited, only causing performance penalties when requiring to load a new cache line.

MoonStack can be extended easily to support new or even custom-made protocol headers. This is essential as new protocols emerge rapidly, for instance for optimized usage in data centers [27], [28] or to tunnel packets in environments with specific requirements [29], [30]. To add a new protocol header, the format of the underlying C structure of the header must be defined. Wrapper functions are created automatically for standard integer data types. For complex data types like addresses, these must be created manually. Optionally, a developer can define default values per field. Default values can also depend on the next header in a protocol stack or the length of the total packet. This allows for the previously mentioned reasonable default values, which can change depending on the current stack, e.g., different EtherType based on the used Internet Protocol version.

Based on the different headers available – at the time of writing 16 different protocol headers are implemented – complex protocol stacks can be created using MoonStack's own simple embedded description language. Thereby, merely the order of the headers within the total stack must be defined. Including one protocol header multiple times (e.g., for tunneling IP over IP) is possible. Additional options allow to customize the subtype of the header (e.g., Ethernet with or without VLAN tag) and even to adjust the length of variably sized fields (e.g., TCP options). This design allows the protocol stack to be highly flexible while requiring low effort and no overhead when adding a new header to libmoon or creating a new stack that is required by the user's application. For instance, with 10 to 20 lines of code, a complex operation like VXLAN en- or decapsulation can be accomplished [31]. Even more complex encapsulation protocols like IPsec are simple to implement [25].

It is important to note that MoonStack only offers support for the syntax of the packet stack, however, not the semantic implementation for most protocols. In other words, MoonStack creates packets as a sequence of bytes with specific values based on an order of protocol headers and their fields. However, the logic behind protocols like TCP using sequence numbers, retransmissions, or even general state keeping is not handled by MoonStack. This functionality must be implemented by the user for the specific application for most protocols, as only utility protocols like ARP, ICMP, and LACP are implemented as part of MoonStack. MoonStack serves two purposes: it can either be used for packet generation in measurement experiments, or it can be used as a library realizing protocols for general packet processing applications.

V. Experiments and Use Cases

In the following, we present selected applications running on top of MoonGen/libmoon. We categorized these programs into four different areas of application:

A. High-Performance Applications

We selected two high-performance applications, which were originally written on top of MoonGen. However, they only use MoonGen because of its convenient interface to DPDK – the functionality which was taken out of MoonGen and shifted to libmoon since then. Nowadays, these applications would rather be realized on top of libmoon.

a) FlowScope [32]: is a tool to record and analyze packet dumps for network debugging and network forensics. Established tools fail at recording or analyzing bandwidths of 100 Gbit/s. FlowScope is able to capture at this rate even with 128 Byte packets when using multiple threads. Compared to Bro Time Machine [33], a factor 50 performance increase can be achieved.

FlowScope utilizes receive side scaling (RSS) and multi-queues leading to an efficient multi-threaded design. However, the key to its performance is founded in the novel in-memory data structure, which is organized as a ring buffer, illustrated in Figure 3. The elements of this ring buffer (or outer queue) are queues which contain the actual packets. This structure of queues within queues coins QQ, the data structure's name. QQ supports the multi-producer/multi-consumer scheme. Figure 3 depicts the producers on the left and the consumers on the right-hand side. Every producer has exclusive access to one of these inner queues for recording incoming packets, which renders explicit synchronization redundant. After an adjustable amount of recorded data or a specified timeout the producer stops filling its inner queue. This inner queue is handed over

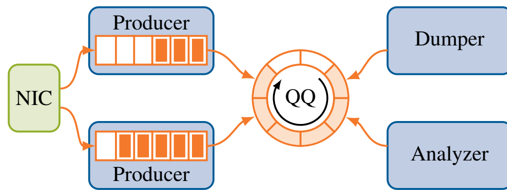

Figure 3. Architecture of QQ

to the outer queue and the producer begins filling a new inner queue. After releasing an inner queue, it can be consumed by one of two different processes - the analyzer or the dumper. The analyzer allows peeking at packets without removing them from the queue. Using libpcap based filter expressions, an analyzer can trigger a dumper process, which dumps selected packets to disk. This operation removes the packet from the queue. Access to the outer queue is rare and hence handled with locks to facilitate multiple accessors. A lock-based outer queue allows implementing special features such as the time-traveling dumper process that would be hard to implement in a fully lock-free queue. An inner queue is only handed out to a single producer or consumer and does not need locks, this allows us to maintain a high performance despite using locks at the high-level interface which is accessed only rarely.

Whereas QQ offers unprecedented throughput figures, it introduces a considerable amount of latency. The latency originates from the desired size or specified timeout of the inner queues. Packets can only be read after adding the inner queue to the outer ring of queues.

b) MoonRoute [24]: is a software router built on top of MoonGen/libmoon. This router preserves the main features of the underlying framework, i.e., high performance and a flexible architecture.

High performance is achieved by several methods. NIC features such as multiple queues and RSS allow distributing traffic in hardware to CPU cores at almost no overhead. This allows for efficient utilization of modern multi-core and future many-core architectures. At the same time the individual worker threads of the router are kept independent to avoid processing delays introduced by synchronization. We avoid information sharing wherever possible, and utilize lock-free data structures for processing. For shared information, e.g., the routing table, the router maintains two copies of a data structure utilized in a double buffering approach. The active copy is read-only, therefore lock-free, and shared across all workers. The inactive copy is updated by a single worker. After an update the copies are swapped, i.e., all workers switch to the former inactive copy and the previously active copy becomes the inactive one and can be updated. There are two different kinds of threads: dedicated forwarding threads which are kept intentionally simple but fast and slow but powerful worker threads which perform additional tasks such as error handling (ICMP) or address resolution (ARP). Employing different workers forms two different packet processing paths, a *fast path* and a *slow path*. MoonRoute also heavily relies on batching between its internal processing modules to improve the performance.

Flexibility is achieved by composing MoonRoute into different reusable modules. The full functionality of the router is formed by chaining different modules sharing a common API. New functionality is realized by implementing a new module. Modules can either be written in Lua for quick and dirty prototyping or, for better performance, written in C and imported utilizing LuaJIT's foreign function interface (FFI). The modularized approach also allows exchanging the implementation of a module with another algorithm, if the API remains the same. For instance, the double buffering approach used for the implemented routing table can easily be transferred to any given data structure, such as other routing tables or prototypes.

B. Benchmarking Tools

Benchmarking tools in this area of application are based on MoonGen, which was enhanced to perform automated measurements.

a) RFC 2544 benchmarking tool: We implemented a prototype benchmarking tool for performing RFC 2544 compliant tests [21], which defines basic testing procedures for benchmarking network interconnect devices, e.g., throughput or latency tests. We transferred the principles successfully applied in our testbed to our benchmarking tool, i.e., we aim for reproducibility and automated configuration. To make device configuration reproducible, we integrated methods to configure different devices into our tool. The full RFC compliant benchmarking procedure involves several benchmarks, e.g., for measuring throughput and latency. Our benchmarking tool can load an individual configuration for each of the devices under test automatically.

We also enhanced the benchmarks themselves, where the specified tests, dating back to 1999, no longer seemed appropriate. For instance, the latency test suggests a single timestamped packet after a 60-second warm-up and cool-down phase of the DuT. This test should be repeated 20 times and as result, the average latency is reported. MoonGen is able to timestamp several thousand packets per second. These thousands of individual measurements allow creating a latency distribution, from which latency percentiles can be deducted. This allows a more detailed assessment of the service quality provided by a DuT. We also identified that all our investigated devices did not change the latency distributions after a warm-up phase of 60 s, therefore we reduced the warm-up phase to a few seconds and the overall benchmarking time to 30 s for latency measurements.

The architecture of the benchmarking tool is modular. Each benchmark, such as throughput, latency, back-to-back, and frame loss benchmarks, is realized as its own module. Unfortunately, there is no unified way to reliably configure different devices. Device configuration is therefore handled in different modules to reflect the varying configuration possibilities like SSH or SNMP. In case none of the configuration possibilities are available, configuration can be done manually by the user.

Even if SSH or SNMP are used, there are differences in device configuration. Subsequently adding support for a new device comes down to implementing a separate module for this device. At the end of the benchmark the tool creates a final report as LaTeX document, not only containing the measurements required by RFC 2544 but also additional graphs, i.e., latency distribution graphs.

b) OPNFV benchmark: The OPNFV project is an open source platform for building, deploying, and testing network functions. They maintain their own automated testing tool called VSPERF [22], which supports several packet generator backends, such as Spirent, IXIA, or MoonGen.

c) FLOWer [34]: Regular server hardware, offering a limited number of NIC ports and bandwidth, is restricted when it comes to high-performance packet generation. Therefore, testing a switch with 10 or more ports becomes infeasible. We demonstrated that MoonGen can surpass these limitations when combined with an OpenFlow hardware switch. This hardware switch replicates the packets generated by MoonGen on all its ports, creating a high-bandwidth traffic generator, which allows testing another switch on all available ports. The usage of OpenFlow features such as groups, meters, and installed flow table entries to count statistics allows us to generate diverse flows beyond mere replication of incoming traffic.

C. Traffic & Packet Generation

MoonGen/libmoon was already used by various scientific publications for its packet generation capabilities. Rincón et al. [20] implemented a DNS flood attack tool on top of MoonGen. Due to ethical considerations, Rincón et al. did not release the full tool but only released the nonhazardous parts of their DNS query generator.

Kulkarni et al. [16] benchmarked their framework for network functions, and Zaostrovnykh et al. [17] measured the performance and latency of their verified NAT implementation.

Shahbaz et al. [15] load tested their P4 software switch. They provide a repository [35] including the code of the switch and VM setups for the DuT and the load generator. Based on these scripts we were able to replicate the published results in our own testbed.

D. MoonGen/Libmoon under Test

MoonGen itself was subject to an investigation performed by Primorac et al. [6]. They compared the hardware and software timestamping capabilities of different tools. The differences between the hardware timestamps reported by MoonGen and by a Spirent hardware packet generator only diverged beyond the 99.99th percentile. We believe this to be an effect of different traffic patterns. Software-based timestamping showed worse results over all tested tools and scenarios. We also performed a study on multiple software packet generators [4] that shows the limits of software timestamping compared to hardware timestamping. Additionally, we investigated the influence of traffic patterns on the latency response of the tested device. We then showed that most packet generators fail to follow the requested distribution as they generate undesired bursts in the traffic. Instead of using constant bitrate traffic we suggest using traffic distributed according to a Poisson process, which is easier to generate in a reliable manner and at the same time resembles real traffic patterns more closely.

E. Collection of MoonGen/Libmoon Examples

Table I contains an overview of the publications mentioned before. The table contains their individual usage scenario and references the publications or the code artifacts if available.

VI. Future Considerations

Encouraged by the adoption of MoonGen in the research community and beyond we constantly improve and add new functionality to MoonGen and libmoon.

A. Overcoming Challenges of TCP

Although new protocols that aim to (partially) replace TCP have emerged or are being standardized [44], [45], TCP is and seems to remain the most used and important transport protocol in the future Internet because of its widespread enrollment. Consequently, it is an important capability being able to benchmark TCP applications or build new applications based on TCP using open source frameworks. For this, a fully functional TCP protocol stack must be implemented, combining both the functionality of the protocol with the performance required to satisfy packet rates of 10 Gbit/s or more preferably on a single core.

The implementations of TCP in the kernels of Linux or FreeBSD reveal several problems. Implementations residing in the kernel of an OS are optimized for high compatibility and reliability rather than performance. The complexity of the TCP protocol itself poses a challenge for the maintainer of a specific implementation. Due to increasing bandwidth demands and stricter upper bounds for the latency of modern applications and network characteristics, TCP is constantly being developed. Introducing new extensions like TCP Fast Open or the BBR congestion control algorithm into a stack focused on reliability takes time. Slow development and release cycles led to the implementation of QUIC as a userspace library [44]. Looking at these problems, we think that libmoon requires a userspace TCP stack not bound by the limitations of a kernel stack to enable high performance and quick prototyping.

Userspace TCP stacks have already been realized. Projects like mTCP [46] for netmap or DPDK and MultiStack [47] for netmap move the stack to userspace, completely bypassing the kernel. The disadvantage is apparent: not using the kernel functionalities requires the TCP stack to be implemented from scratch. This conflicts with the aforementioned problem of TCP being complex. Therefore, this approach risks that only a subset of TCP's functionality is implemented, or that the project cannot keep up with updates and new extensions to TCP, resulting in not being maintained or compatible.

Instead of developing a new TCP stack from scratch, trying to integrate an existing fully functional kernel or userspace stack into a packet processing framework is another solution

Table I
PROJECTS USING MOONGEN/LIBMOON

Name	Usage scenario	Publication	Code
High-performance applications			
FlowScope	Tool for high-performance flow capture and analysis	[32]	[26]
MoonRoute	Extensible high-performance router	[24]	[36]
Benchmarking tools			
RFC 2544	Modular benchmarking tool	[21]	[37]
OPNFV VSPERF	Automated NFV testing framework	[22]	[38]
FLOWer	High-performance switch benchmarking	[34]	[39]
Traffic & packet generation			
DNS flood query generator	DNS implementation and flooding attack tool	[20]	[40]
NFVnice	Throughput and latency measurements	[16]	-
Verified NAT	Throughput and latency measurements	[17]	-
PISCES	Throughput measurements	[15]	[35]
MoonGen / libmoon under test			
MoonGen investigation	Precise and accurate rate control and timestamping	[4]	[41], [42]
MoonGen timestamping	Investigation of timestamping in MoonGen and other packet generators	[6]	-
Additions to MoonGen / libmoon			
MoonStack	Easy-to-use and efficient packet creation	-	[43]

pursued. StackMap [48] is an interface which does not bypass the kernel. It incorporates the existing stack of the Linux kernel in an optimized way benefiting from the mature TCP/IP kernel implementation. The authors show that the observed performance limitations of the Linux TCP/IP stack are not due to its processing of the different protocol related layers, but rather because of I/O bottlenecks at layer two and at the Socket API.

The third type of TCP stacks can be found in embedded systems, for example, the stack lwIP [49] is commonly used in resource-constraint environments on microcontrollers. Unlike typical academic userspace stacks, it is mature software, deployed in Internet-of-things applications, with a fully featured TCP implementation. A core feature of lwIP is that it supports a wide range of CPU architectures, network chips, and operating systems. Its external interface can be adapted for a library such as DPDK, and can be used in a normal userspace process.

Thus, we decided to integrate the userspace TCP stack into libmoon. We currently evaluate if a dedicated high-performance stack such as mTCP or a mature, feature-rich stack such as lwIP is the best fit for libmoon regarding the API, the usability, and the performance. Integration of an existing TCP stack provides libmoon with the desired functionality, while avoiding the time-consuming task of developing our own TCP stack. With the function-rich TCP protocol integrated into libmoon, a basis for testing webservers and firewalls is given.

VII. CONCLUSION

We believe MoonGen and the underlying framework libmoon to be two highly relevant tools for the research community as the selected examples in Table I indicate. The latter offers a convenient platform to build modular high-performance applications or other tools. MoonGen provides means to reliably and precisely control the generated traffic as well as to reliably measure latency and throughput of the received traffic – two essential requirements when performing network experiments and making them reproducible.

However, utilizing MoonGen only gives control over half the setup, i.e., the load generator side. Full reproducibility also requires control over the DuT. Therefore, we present our measurement orchestrating tool pos, which provides the facilities to recreate configuration state on the DuT making our experiments repeatable. The testbed was used in several student projects successfully replicating results. We plan to release pos in a more mature version, which will enable replication on a wider scale. However, the larger problem of reproducibility in the computer networking community will require further work on incentives and ecosystems, as discussed in a recent SIGCOMM workshop [9], [50], [51].

ACKNOWLEDGMENT

This work was supported by the High-Performance Center for Secure Networked Systems, the DFG Priority Programme 1914 „Cyber-Physical Networking“, and the German BMBF project SENDATE-PLANETS (16KIS0472).

REFERENCES

[1] S. Bradner and J. McQuaid, “RFC 2544: Benchmarking Methodology for Network Interconnect Devices,” Tech. Rep., 1999, https://tools.ietf.org/html/rfc2544.

[2] “DPDK,” https://www.dpdk.org/.

[3] L. Rizzo, “netmap: a novel framework for fast packet I/O,” in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[4] P. Emmerich, S. Gallenmüller, G. Antichi, A. W. Moore, and G. Carle, “Mind the Gap: A Comparison of Software Packet Generators,” in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 2017, pp. 191–203.

[5] A. Botta, A. Dainotti, and A. Pescapé, "Do ou trust your software-based traffic generator?" *IEEE Communications Magazine*, vol. 48, no. 9, 2010.

[6] M. Primorac, E. Bugnion, and K. Argyraki, "How to Measure the Killer Microsecond," in *Proceedings of the Workshop on Kernel-Bypass Networks*. ACM, 2017, pp. 37–42.

[7] ACM, "Artifact Review and Badging," http://www.acm.org/publications/policies/artifact-review-badging.

[8] ACM, "Workshop on Models, Methods and Tools for Reproducible Network Research," http://conferences.sigcomm.org/sigcomm/2003/workshop/mometools/, 2003.

[9] Q. Scheitle, M. Wählisch, O. Gasser, T. C. Schmidt, and G. Carle, "Towards an Ecosystem for Reproducible Research in Computer Networking," in *Proc. of ACM SIGCOMM Reproducibility Workshop*. ACM, 2017, pp. 5–8.

[10] "Jupyter," http://jupyter.org/.

[11] "Ixia," https://www.ixiacom.com/products-services/test-hardware.

[12] "Spirent," https://www.spirent.com/Products/TestCenter/Platforms/Appliances.

[13] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott *et al.*, "OSNT: Open Source Network Tester," *IEEE Network*, vol. 28, no. 5, pp. 6–12, 2014.

[14] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[15] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 525–538.

[16] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 71–84.

[17] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A Formally Verified NAT," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 141–154.

[18] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood, "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16. New York, NY, USA: ACM, 2016, pp. 3–17.

[19] Y. Yiakoumis, S. Katti, and N. McKeown, "Neutral Net Neutrality," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 483–496.

[20] S. R. Rincón, S. Vaton, and S. Bortzmeyer, "Reproducing DNS 10Gbps flooding attacks with commodity-hardware," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2016 International*. IEEE, 2016, pp. 510–515.

[21] D. Raumer, S. Gallenmüller, F. Wohlfart, P. Emmerich, P. Werneck, and G. Carle, "Revisiting Benchmarking Methodology for Interconnect Devices," in *The Applied Networking Research Workshop 2016 (ANRW '16)*, Berlin, Germany, 2016.

[22] "OPNFV VSPERF," http://artifacts.opnfv.org/vswitchperf/docs/index.html.

[23] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A P4 Language Benchmark Suite," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 95–101.

[24] S. Gallenmüller, P. Emmerich, R. Schönberger, D. Raumer, and G. Carle, "Building Fast but Flexible Software Routers," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 2017, pp. 101–102.

[25] D. Raumer, S. Gallenmüller, P. Emmerich, L. Märdian, and G. Carle, "Efficient Serving of VPN Endpoints on COTS Server Hardware," in *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*. IEEE, 2016, pp. 164–169.

[26] "FlowScope," https://github.com/emmericp/FlowScope.

[27] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.

[28] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 29–42.

[29] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "RFC 7348: Virtual eXtensible local area network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," Tech. Rep., 2014, https://tools.ietf.org/html/rfc7348.

[30] P. Garg and Y.-S. Wang, "RFC 7637: NVGRE: Network Virtualization Using Generic Routing Encapsulation," Tech. Rep., 2015, https://tools.ietf.org/html/rfc7637.

[31] "MoonGen VXLAN example," https://github.com/emmericp/MoonGen/blob/5388192fa1dc016797fabe8912bbcdfc7713756e/examples/vxlan-example.lua.

[32] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "FlowScope: Efficient Packet Capture and Storage in 100 Gbit/s Networks," in *Proceedings of the 16th International IFIP TC6 Networking Conference*. IEEE, 2017.

[33] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," in *Internet Measurement Conference 2005 (IMC'5)*, 2005.

[34] P. Emmerich, S. Gallenmüller, and G. Carle, "FLOWer – Device Benchmarking Beyond 100 Gbit/s," in *IFIP Networking Conference (IFIP Networking) and Workshops, 2016*. IEEE, 2016, pp. 109–116.

[35] "PISCES experiment code," https://github.com/P4-vSwitch/vagrant/tree/ef197c77505252d1b255c3f1c83b976aae5d7fb7.

[36] "MoonRoute," https://github.com/emmericp/MoonRoute-data/tree/51333dc648ca42f3740f6d09895e1ad4a9f67d69.

[37] "RFC 2544 benchmark tool," Tech. Rep., https://github.com/emmericp/MoonGen/pull/98.

[38] "VSPERF code repository," https://git.opnfv.org/vswitchperf.

[39] "FLOWer code," https://github.com/emmericp/FLOWer-scripts/tree/c5bd7cb25c3da1537dad7a44db84d043b442bbb9.

[40] "MoonGen DNS code," https://github.com/emmericp/MoonGen/pull/118.

[41] "MoonGen rate control methods," https://github.com/emmericp/MoonGen/blob/5388192fa1dc016797fabe8912bbcdfc7713756e/examples/rate-control-methods.lua.

[42] "MoonGen timestamping," https://github.com/emmericp/MoonGen/blob/5388192fa1dc016797fabe8912bbcdfc7713756e/examples/timestamping-tests/.

[43] "MoonStack," https://github.com/libmoon/libmoon/tree/f3013c9ced5d0e4f5344451d2d269233852eb96c/lua/proto.

[44] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 183–196.

[45] R. Stewart, "RFC 4960: Stream Control Transmission Protocol," Tech. Rep., 2007, https://tools.ietf.org/html/rfc4960.

[46] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 489–502.

[47] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling Network Protocol Innovation with User-Level Stacks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 52–58, 2014.

[48] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[49] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.

[50] V. Bajpai, M. Kühlewind, J. Ott, J. Schönwälder, A. Sperotto, and B. Trammell, "Challenges with Reproducibility," in *Proc. of ACM SIGCOMM Reproducibility Workshop*, 2017.

[51] M. Canini and J. Crowcroft, "Learning Reproducibility with a Yearly Networking Contest," in *Proc. of ACM SIGCOMM Reproducibility Workshop*. ACM, 2017, pp. 9–13.