LOS: A High Performance and Compatible User-level Network Operating System

Yukai Huang
hykmail@gmail.com

Jinkun Geng
steam1994@163.com

Du Lin
lind15@mails.tsinghua.edu.cn

Bin Wang
wangbin@bjtu.edu.cn

Junfeng Li
hotjunfeng@163.com

Ruilin Ling
buptlrl@163.com

Dan Li
tolidan@tsinghua.edu.cn
Tsinghua University

ABSTRACT

With the ever growing speed of Ethernet NIC and more and more CPU cores on commodity X86 servers, the processing capability of the network stack in Linux kernel has become the bottleneck. Recently there is a trend on moving the network stack up to user level and bypassing the kernel. However, most of these stacks require changing the APIs or modifying the source code of applications, and hence are difficult to support legacy applications. In this work, we design and develop *LOS*, a user-level network operating system that not only gains high throughput and low latency by kernel-bypass technologies but also achieves compatibility with legacy applications. We successfully run Nginx and NetPIPE on top of *LOS* without touching the source code, and the experimental results show that *LOS* achieves significant throughput and latency gains compared with Linux kernel.

CCS CONCEPTS

• **Networks** → **Network architectures**; **Network design principles**; **Network components**; *Network performance evaluation*;

KEYWORDS

user level, network operating system, compatability, high througput, low latency

APNET '17, August 3–4, 2017, Hong Kong, China

ACM ISBN 978-1-4503-5244-4/17/08.
https://doi.org/10.1145/3106989.3106997

ACM Reference format:
Yukai Huang, Jinkun Geng, Du Lin, Bin Wang, Junfeng Li, Ruilin Ling, and Dan Li. 2017. LOS: A High Performance and Compatible User-level Network Operating System. In *Proceedings of APNET '17, Hong Kong, China, August 3–4, 2017,* 7 pages.
https://doi.org/10.1145/3106989.3106997

1 INTRODUCTION

The hardware capability of commodity X86 platform is ever increasing. Servers with tens of CPU cores and 100Gb Ethernet NICs are widely commercialized. However, there is a considerable mismatch between the high-performance hardware of servers and the processing capability of the network stack in Linux kernel. Therefore, in recent years a variety of high-performance network stacks have been designed and released [2, 7, 9, 10, 13, 14, 17–19]. Most of these stacks employ the idea of kernel-bypass networking, and move the network stack up to user level, by leveraging I/O libraries such as DPDK or Netmap. According to the benefits from technologies such as polling-mode network I/O, huge page memory, zero-copy, etc., these stacks show significantly better performance than Linux kernel.

However, the network stacks mentioned above are usually not fully compatible with legacy applications. Some stacks change the network APIs (so as to provide new functionalities such as zero-copy), which requires applications to rewrite the interfaces, such as IX [2]. Some stacks retain similar APIs as Linux kernel, but run the stack as a thread of the main application, such as mTCP [9]. In either way, the source codes of applications have to be modified. It not only causes much effort on porting legacy applications, but also requires developers to learn the new API semantics. What's more, most of these stacks manage a single application well, but fail to support multiple applications.

In this work, we seek to design a kernel-bypass network stack which not only gains high performance, but is also

fully compatible with legacy applications. Here compatibility means that not only the applications do not need to modify their codes to run on top of the user-level stack, but also multiple independent applications can share the stack. Hence, the stack is actually a user-level network operating system. By doing so, we want to decouple the network-related operations from the Linux kernel, and leave them on the user-level network operating system. Designing such a network operating system imposes several technical challenges, such as automatically taking over the network-related APIs, distinguishing the access of two FD (file descriptor) spaces, new event notification scheme, detecting application crashes and recycling the resources, etc. We employ a variety of technologies to address these challenges. Besides, in order to keep high performance of the network operating system, we not only use similar mechanisms as in existing works, such as polling-mode I/O, zero-copy in the stack and FD space separation, but also introduce new methods. We use lockless shared-queue for IPC between the networking operating system and applications, and run the user-level operating system and applications on different CPU cores, both of which show significant performance gains.

We have developed the single-core version of the user-level network operating system, which is called *LOS*. We successfully run Nginx and NetPIPE on top of *LOS* without touching the source code, and the experimental results show that *LOS* achieves 17~41% higher throughput and 20~55% lower latency compared with Linux kernel.

The remaining part of this paper is organized as follows. Section II introduces the related works. Section III describes the design details. Section IV presents the evaluation results. Finally, Section V concludes this paper.

2 RELATED WORK

High-performance packet processing libraries. So far, there has been much effort devoted to the development of high-performance packet processing libraries [6, 10, 15, 17]. These libraries aim to accelerate network I/O by completely decoupling the network data path from the kernel, thus eliminating overheads imposed by heavy-weight kernel network stacks. Our work in this paper is built on top of DPDK [6], which becomes an official open-source project of the Linux Foundation very recently. DPDK provides fast packet processing techniques such as polling mode driver (PMD) instead of per-packet interrupt, zero-copy packet sending and receiving in the user space, pre-allocating rings and memory pools to avoid per-packet memory allocation, using hugepage memory to reduce TLB misses, etc. Similar techniques can also be found in other libraries.

User-space TCP stacks. The main problem with Intel DPDK and other libraries is that they directly manipulate raw packets, rather than network connections and flows. In order to leverage these libraries to handle higher-level protocols such as TCP and UDP, developers have to implement a TCP/IP stack from scratch for their own applications, which not only requires a thorough understanding of TCP/IP protocols, but also considerably limits code reuse and maintenance. Therefore, it is reasonable to develop a user-level high-performance TCP/IP stack as a general-purpose library to benefit different applications. The representative works in the literature include mTCP [9] based on Packet I/O Engine (PSIO) [10], IX [2] and Seastar [19] based on DPDK, UTCP [5] and Sandstorm [14] based on netmap [17], etc. However, all these works require updating applications at some extent, which raises the barrier for wide deployment. Moreover, they usually well support a single application instead of multiple applications.

Kernel stack optimizations. On another theme, the literature never stops improving the performance of the network performance of Linux kernel. The SO_REUSEPORT option [11] has been added into the Linux kernel since version 3.9 to make multi-thread server scale better on multicore systems. The basic idea is to reduce the contention on shared data structures (e.g. the socket data structure and the accept queue) by allowing multiple sockets on the same host to bind to the same port. For passive connections, Affinity-Accept [16] and MegaPipe [18] guarantee connection locality and improve load balance between cores by providing a listening socket with multiple per-core accept queues instead of a shared one. In addition, MegaPipe also proposed *system call batching* to amortize system call overhead and lightweight sockets to avoid VFS overhead. In a recent work, Fastsocket [21] achieves connection locality and scalability for both active and passive connections, and avoids lock contention by partitioning the global Listen Table/Established Table into local ones. However, the aforementioned optimizations fail to fully address the overhead incurred in kernel stack, such as the overhead of interrupt handling on receiving packets, which can greatly affect the overall performance under the scenario of high network load.

3 SYSTEM DESIGN

We design *LOS*, a user-level network operating system which not only gains higher throughput and lower latency by exploiting kernel-bypass technologies, but also achieves full compatibility with legacy applications. We expect that by our work more applications and more users can benefit from kernel-bypass networking to improve their performance.

3.1 Performance Improvement

At first we describe our design on how to improve the network performance of *LOS*.

Typical optimization methods as in existing works. In the scenario of high-speed network, it has been proved that polling-based I/O gains distinctive advantages over interrupt-based mechanisms [4]. Considering that DPDK is a widely-used polling-mode I/O library in the industry, in this work we build *LOS* on top of DPDK. DPDK also enables zero copy in the user-level network stack. We also decouple the socket FD assignment from the heavy VFS structure in the Linux operating system. Therefore, when allocating FDs for new sockets, we do not need to visit VFS and thus many overheads are eliminated. It is especially helpful when serving highly concurrent short-lived connections.

The above technologies, namely, polling-based I/O, zero-copy in the stack, and separating socket FD from VFS, are already widely used in existing works [2, 9, 10, 14, 17–19]. Since they have demonstrated considerable performance gains compared with traditional approaches, we introduce them in *LOS*. Besides, in *LOS* we adopt the following mechanisms to further improve the performance.

Lockless shared-queue for IPC between *LOS* and applications. *LOS* runs as a user-mode process on top of Linux operating system, and thus shared-memory IPC is a natural choice to exchange information between *LOS* and applications. Compared with traditional network stacks in the kernel, it can avoid the high overhead of system calls. However, since we design *LOS* as a general network operating system to support multiple applications, we need an efficient way to manage the message queue between *LOS* and applications. If we build a single message queue to serve multiple applications, we have to use lock/unlock operations to deal with resource competition, which will cause significant overhead. Otherwise, if we build different message queues for different applications, for each message the stack has to traverse all the message queues to get the right one; the cost is also very high, as shown by previous works [13].

Fortunately, DPDK *rte_ring* provides a way to allow multiple producers (applications) to directly access a shared-queue via CAS operation [6], which thus enables lockless visit to a shared message queue and considerably improves the IPC performance. We build multiple message queues between *LOS* and applications, some for commands and some for socket information, but each queue is shared by all the applications on top of *LOS*. The details are described in Section 3.3.

Running *LOS* and applications on separate cores. One feature of polling-based I/O is that the CPU core will be always fully utilized by polling. If we put the *LOS* and applications on the same core, the application performance will be adversely affected. Although people may think that it will reduce caches misses if we put the network stack and application on the same core, our experiment shows that the negative impact caused by polling is much higher. When we run the *LOS* and a Nginx worker together in one CPU core, the http responses from the Nginx worker degrades in a disastrous way compared with running them in separate cores.

Therefore, in our design we run *LOS* and applications on separate cores. *LOS* runs as a single process in a dedicated core, so it also avoids the cost of context switching. The separation is very easy to achieve in practice. Since the *LOS* core is fully loaded by polling, the Linux operating system will automatically assign the application processes to other cores. Actually, this kind of core separation is also suggested by previous works [13]. However, in this paper we validate this approach from a different motivation, and accordingly design a network operating system to decouple the network functionalities from the kernel.

3.2 Compatibility Design

The compatibility in *LOS* considers two aspects. First, applications do not need to change their APIs or modify their source codes. Hence, we need a way to automatically redirect the network-related APIs to *LOS*, while retaining the other APIs to the Linux kernel. Given that *LOS* and Linux kernel have two FD spaces without the awareness of applications, we also need to differentiate applications access to the two FD spaces. Second, as a network operating system, *LOS* should be able to manage the status of applications. Blocking APIs and event notification should be realized without the participation of Linux kernel. To support multiple applications, *LOS* needs to not only allocate network-related resource for new applications, but also recycle these resources for exited applications. When applications gracefully exit, it is easy for *LOS* to recycle the resources. However, given application crashes or abnormal exits, *LOS* requires a way to detect.

Taking over network-related APIs. In order to support legacy applications without touching the source codes, *LOS* needs to take over network-related APIs, while leaving other APIs to Linux kernel. To take over network-related APIs, *LOS* provides these APIs with the same function signature and complies these functions as a dynamic library (*.so in Linux). During the startup, we link the dynamic library to every application on *LOS* by setting the environment variable $LD_PRELOAD. In this way, we are able to intercept those APIs in *glibc*, including almost all network-related APIs. However, if applications call other APIs than networking, we have to switch the functions back to Linux kernel. Thanks to the *dlsym*[12] tool, *LOS* is able to capture the handlers in *glibc* and conduct the switching[1]. In essence, the *LD_PRELOAD* mechanism and *dlsym* tool are used for the redirection of function calls from applications and there are no extra system calls involved. Therefore, it is expected that

[1] *dlsym* is used to obtain the addresses of symbols in *glibc*.

the trival overheads caused by *LD_PRELOAD* and *dlsym* will not damage the high performance of *LOS* .

Distinguishing the access of two FD spaces. Some APIs, such as *read*, *write*, can access both FD spaces in *LOS* and Linux kernel. We need a way to distinguish the two cases and direct the API to the correct operating system. Note that although the FD space in *LOS* is for network sockets and that in Linux kernel is for other file systems, the two FD spaces are within the single FD space from the application's view. We note that Linux kernel allocates FD number in a bottom-up way, i.e., from 0,1, and upwards. Therefore, in *LOS* we allocate FD number in a top-down way, i.e., from $2^{31} - 1$ downwards. Given that the FD numbers in the two spaces do not collide with each other, we can use a simple way to distinguish the two FD spaces and the system will work well. In the current implementation, we split the space into two halves and set the FD number $2^{30} - 1$ as the threshold. If the FD number is less than the threshold, the API will be directed to the Linux kernel; while if the FD number is higher than the threshold, the API will be directed to *LOS*. Considering that the FD numbers are also recycled when an application exits, we believe each half of the FD space is large enough to hold all the network-related or unrelated FDs in practice.

Event notification. To support blocking APIs, such as *send*, *recv*, *accept*, *connect*, *epoll_wait*, we have several options. As a possible solution, we can translate these blocking APIs into a non-blocking way and thus work in a polling mode. However, since we might run multiple applications on one CPU core, this mechanism may cause interference and unfairness between applications. Therefore, in *LOS*, we still seek to maintain the *blocking* semantics for blocking APIs. Compared with other APIs, *epoll_wait* is special, since it also relates to the FD space in Linux kernel. So we first discuss how to realize *epoll_wait* in *LOS*.

When an application calls *epoll_wait*, it should be notified when there is either a network-related event or a network-unrelated event. However, *LOS* only cares for the network-related events. The network-unrelated events generated by Linux kernel may come without the awareness of *LOS*. To take both events into consideration, our approach leverages the *epoll_wait* syscall to realize blocking semantics and aggregates all the network-related events as one kernel event for *epoll_wait* syscall. The working process is as follows: When an application calls *epoll_create*, a *kernel epoll* [2] will be created firstly, then a FIFO (named pipe) is built between *LOS* and the application, and the FIFO's FD is registered to the *kernel epoll*. Via *epoll_ctl*, non-network FDs will be directly registered to the *kernel epoll* and network FDs will be stored by *LOS* in its own data structure. When the application calls *epoll_wait*, it will be blocked with the *epoll_wait* syscall. The blockage will surely be broken when the network-unrelated event (i.e. kernel event) comes. When there is a network-related event in *LOS*, a kernel event is generated for the FIFO [3], and the *epoll_wait* syscall will return. Then the application will also be unblocked. In this way, we integrate the *epoll_wait* operation of the two FD spaces together, without touching the applications. The cost is affordable, since when the network load is high, a single FIFO event will cause the application to process a large number of network events, and hence the number of invoked *epoll_wait* syscalls is limited.

For the other blocking APIs, including *send*, *recv*, *accept*, and *connect*, we can simulate the process as follows: *LOS* will let the application go to sleep when there is no event, and an efficient event notification mechanism is required to notify the blocked application. We have tried *semaphore* and *spin lock*, but the performance is bad, probably due to their high overheads. An alternative choice is reliable signal[4]. However, such mechanism is not feasible due to the well-known *thundering herd* problem [3, 8]: Considering those multi-threaded applications, which will create multiple threads to conduct network processing and each thread may call blocking APIs to sleep, when a signal is sent to wake up one of the threads, other threads will also be waken up because they belong to the same process and the signal is directed by the process number (*pid*). Although the *thundering herd* problem can be mitigated with some recent techniques, we find that the cost is still higher than our current approach. Currently we use the same approach above, namely, building FIFO between *LOS* and application and using *epoll_wait* syscall to realize the blocking semantics.

Resource recycle and fault detection. When an application gracefully exits, *LOS* will recycle the allocated resources, such as socket FD, FIFO FD, ring buffer, etc. But if an application crashes or exits in an abnormal way, *LOS* should be able to detect the status of the application and recycle its resource. Otherwise, the resource might be used up by this kind of abnormal applications. Our design works as follows. During the startup of *LOS*, a *kernel epoll* will be created and a *UNIX domain socket* will be registered to it. When an application is launched, it will establish a connection with the *UNIX domain socket* and *LOS* will generate another *UNIX domain socket* (via *accept* API) to handle the connection. The fresh *UNIX domain socket* generated by *LOS* will also be registered to the *kernel epoll* to help monitor the connection. Note that *LOS* runs in a polling mode, periodically it checks the states of these *UNIX domain socket*s after a certain number of loops by *epoll_wait* syscall in a non-block mode. If all the applications work well, *epoll_wait* will return 0. A

[2] *kernel epoll* refers to the epoll implemented by Linux kernel.

[3] *LOS* will *write* the FIFO FD, thus changing the state of the FIFO FD so that the blockage caused by *epoll_wait* syscall will be broken.

[4] Note that we need reliable signal instead of common signal, since a signal loss will make *LOS* crash.

non-zero return value indicates there is some fault with the application, since some FDs in the *kernel epoll* have changed their states [5]. By this way, *LOS* can detect the application fault and accordingly recycle the resources. One thing to notice is, the fault detection mechanism does not hamper the compatibility of *LOS* because the mechanism is completely implemented inside *LOS* [6] and those legacy applications do not need to separately call relevant APIs more than their own intentions.

3.3 Architecture Design

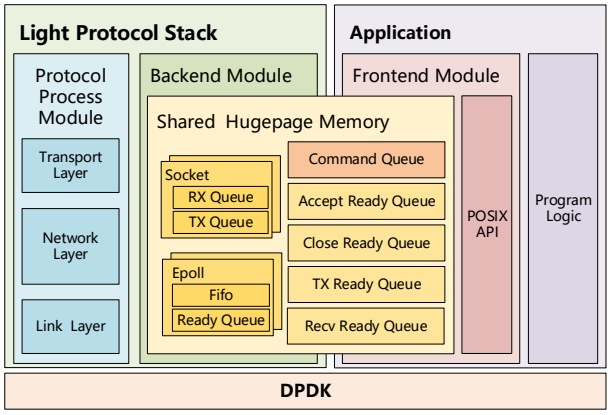

Figure 1: Architecture of *LOS*

Figure 1 shows an overview of the architecture design for *LOS*. Generally, *LOS* can be divided into three main components: *Frontend Module* (FM), *Backend Module* (BM), and *Protocol Process Module* (PPM).

PPM is the core component, which undertakes the major process in networking. We have implemented the process logic of the middle three layers of TCP/IP Five-Layer Model (i.e. Data Link Layer, Network Layer and Transport Layer). Data packets are captured from the NIC with DPDK in a polling mode, and then decomposed in PPM and transmitted to BM. Correspondingly, the transmitted packets are delivered from BM to PPM for packet encapsulation and final transmission to NIC.

BM maintains all the message queues, which are shared by BM and FM for communication. As we presented above, each message queue is shared by all the applications running on top of *LOS*, instead of for a single application. The *Command Queue* is responsible for the communication of command operations (particularly, all the API calls by the applications). Other message queues, such as *Accept Ready Queue*, *Close Ready Queue*, *Tx Ready Queue* and *Rx Ready Queue*, maintain the data information for the commands. Due to page length limit, we omit the details of these queues in this paper.

FM serves as a middleware to bind applications with *LOS*. FM provides POSIX API for applications and mask the original APIs provided by the kernel stack. When applications call these APIs, FM will encapsulate the command information into a specific data structure and insert it into the *command queue*. BM keeps polling the *Command Queue* and processes the commands sequentially. There is a field in the command data structure which maintains the return value of the operation. The return value field is set by BM and can also be accessed by FM to make applications aware of the status of the operation.

4 IMPLEMENTATION

So far we have implemented the single-core version of *LOS*, with a complete TCP/UDP/IP stack. It runs stably on Ubuntu 16.04 with the kernel vesion of 4.4.0 and DPDK 17.02. Most network APIs have been completed. Excluding the codes from DPDK I/O library and the codes for protocol stack (most codes for the protocol stack are copied from the Linux kernel), our current implementation of *LOS* contains 16702 lines of C codes. We will continue to support a complete set of APIs.

5 EXPERIMENT AND EVALUATION

To validate the compatibility and performance of *LOS*, we conduct the following experiments to answer the following questions.

Compatibility. *Can applications work properly on LOS without any modification of their source codes*? Based on the network APIs we current implemented, in the experiments we can successfully run two applications, Nginx and NetPIPE [7], on top of *LOS* without any code modification.

High throughput. *To what extent can LOS improve application's throughput, especially in the scenario of short-lived connections*? As prior work has shown [21], the kernel stack gains poor performance when handling heavy workload of short-lived connections. Therefore, we intend to compare *LOS* and Linux kernel 4.4.0 under the scenario of short-lived connections in Section 5.1.

Low latency *To what extent can LOS reduce application's end-to-end latency, when confronted with both high workload and low workload*? End-to-end latency under high workload is considered to be an important metric of user experience. In Section 5.1 and 5.2, we will demonstrate the capability of *LOS* to reduce latency under high load and low load, respectively.

[5] For those applications which have exited abnormally, the corresponding *UNIX domain sockets* will be collected by the operating system, thus the states of these sockets are changed.

[6] In other words, the *UNIX Sockets* are invisible to applications and *LOS* is responsible for creating and managing them.

5.1 Nginx

Nginx is one of the most widely deployed web servers and reverse proxy servers. In this experiment, we focus on measuring its throughput and latency as a web server under heavy workload of short-lived connections.

The evaluation environment consists of three machines connected by one switch. Two servers, equipped with two 16-core Intel Xeon CPU E5-2683 v4, 128GB memory and a dual-port intel 82599ES 10G NIC, work as clients. The remaining one, with one 2-core Intel Core i7-4790K 4.00Ghz CPU, 32GB and a dual-port intel 82599ES 10G NIC, works as the server. The operating systems for three machines are all Ubuntu 16.04 with kernel version of 4.4.0. We have confirmed that client side is not the bottleneck and try to explore the upper-bound performance of the network stack in the server. The version of Nginx used in our experiment is 1.9.5.

To create such a high workload, we set *keepalive_timeout* option to 0 in *nginx.conf* and use wrk [1] to generate http requests on the client machines, each of which fetches a simple 64B html. We use RPS (i.e. requests per second) as a measurement of the network performance.

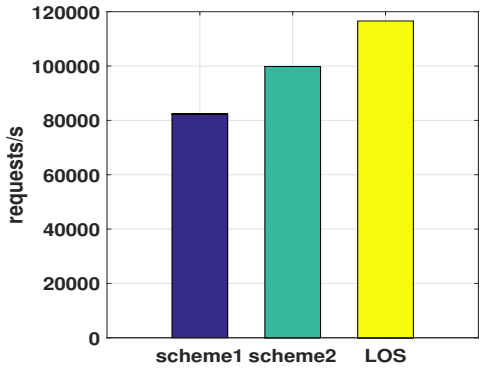

Figure 2: Comparison of RPS

Since currently *LOS* only has a single-core version, we use one core on the 2-core server machine to run *LOS* and the other to run Nginx in a daemon mode. For Linux kernel stack, we have designed two different schemes which we name as *scheme1* and *scheme2*: *scheme1* disables *master_process*, whereas *scheme2* enables *master_process* with two worker processes. As for *scheme2*, The *worker_cpu_affinity* is set thus each worker process is assigned to a different core. Meanwhile we have disabled the *reuseport* option and *accept_mutex* option[7]. We conduct experiments to compare the RPS of the three schemes. The result of the experiment is illustrated in Figure 2 and Figure 3.

[7]The second scheme is the common configuration for Nginx on multicore machines.

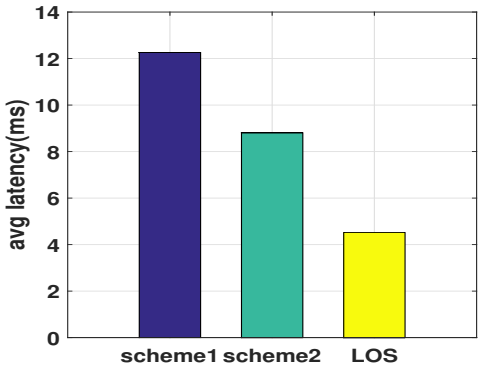

Figure 3: Comparison of Latency

For throughput, it is shown in Figure 2 that *LOS* outperforms *scheme1* by 41% and *scheme2* by over 17%. As for latency, Figure 3 shows that *LOS* reduces the average latency by 63% compared to *scheme1*, and by 49% compared to *scheme2*. Furthermore, for the worst 10% cases, *LOS* completes requests with a latency of 4.59ms, which is only slightly higher than its average latency of 4.52ms, whereas the latency of *scheme1* and *scheme2* acutely rises to 186.43ms and 188.74ms respectively. Such kind of phenomenon suggests that *LOS* not only reduces latency but also maintains a good stability and predictability.

5.2 NetPIPE

NetPIPE [7] is a commonly used network performance evaluator. For different message sizes, NetPIPE performs the standard Ping-Pong test for repeated times and calculate the average latency [20]. The TCP module in NetPIPE version 3.7.2 is used in our experiment. We run NetPIPE transmitter on the kernel network stack, and NetPIPE receiver on both *LOS* and kernel stack. The transmitter machine and receiver machine are directly connected.

The transmitter machine is installed with Ubuntu 14.04 (kernel version 3.16), equipped with Intel i7-4790K CPU Processor and Intel 82599ES 10G NIC; while the operating system on the receiver machine is Ubuntu 16.04 (kernel version 4.8), with Intel i7-7700 CPU Processor and Intel X250 10G NIC. We have measured the latency with typical message sizes for common applications and the results are shown in Figure 4.

From Figure 4, we see that the measured latency is reduced by more than 20% for all the message sizes. When the message size is 256 bytes, the latency reduction is more than 55%. This result indicates that, even under low network load, *LOS* significantly reduces the processing delay of each single packet, which will be important for delay-sensitive applications.

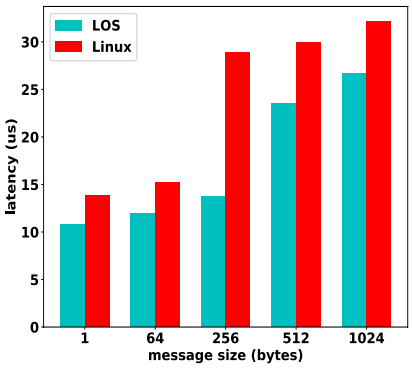

Figure 4: Comparison of Latencies

6 CONCLUSION AND FUTURE WORK

In this paper, we propose *LOS*, a high performance and compatible user-level network operating system. To achieve high performance, *LOS* not only uses proven technologies in existing works, but also exploits novel methods such as lockless shared-queue IPC between the network stack and applications, running *LOS* and applications on separate cores. To be compatible with legacy applications, *LOS* takes over network-related APIs of applications and distinguishes the access of two FD spaces (network sockets and kernel file system). Moreover, *LOS* employs an efficient event notification mechanism between the network stack and applications to support blocking APIs and resource recycling for faulted applications.

Currently we have developed the single-core version of *LOS* and supported most of network APIs. It runs well under Nginx and NetPIPE without touching their source codes, and achieves higher throughput and lower lantency than Linux kernel stack. We are now developing the multi-core version and in the future we will conduct a series of comparative work with existing works. Meanwhile, we will also strengthen the compatibility to support more applications.

ACKNOWLEDGMENTS

This work was supported by the National Key Basic Research Program of China (973 program) under Grant 2014CB347800, the National Key Research and Development Program of China under Grant 2016YFB1000200, and the NSFC funding under grant 61432002, 61522205.

REFERENCES

[1] [n. d.]. *wrk: Modern HTTP benchmarking tool* . https://github.com/wg/wrk.

[2] Belay Adam, Prekas George, Klimovic Ana, Grossman Samuel, Kozyrakis Christos, and Bugnion Edouard. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the USENIX OSDI 2014 Conference.*

[3] Jonathan Corbet, Greg Kroah-Hartman, and Alessandro Rubini. 2005. *Linux Device Drivers.* O'Reilly, Sebastopol, CA.

[4] Denys Haryachyy. 2015. *Ostinato and Intel DPDK 10G Data Rates Report on Intel CPU.* PLVision. http://www.slideshare.net/plvision/10-g-packet-processing-with-dpdk.

[5] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. 2014. Rekindling Network Protocol Innovation with User-level Stacks. *SIGCOMM Comput. Commun. Rev.* (2014).

[6] Intel 2016. *Intel DPDK: Data Plane Development Kit.* Intel. http://dpdk.org.

[7] Roberto De Ioris. 2009. *NetPIPE: A Network Protocol Independent Performance Evaluator.* http://bitspjoule.org/netpipe/.

[8] Roberto De Ioris. 2016. *Serializing accept(), AKA Thundering Herd, AKA the Zeeg Problem.* http://uwsgi-docs.readthedocs.io/en/latest/articles/SerializingAccept.html.

[9] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).*

[10] KAIST 2012. *Packet I/O Engine.* KAIST. http://shader.kaist.edu/packetshader/io_engine/.

[11] Michael Kerrisk. 2013. *The SO_REUSEPORT socket option.* http://https://lwn.net/Articles/542629/.

[12] Michael Kerrisk. 2017. *Linux Programmer's Manual-DLSYM.* http://www.man7.org/linux/man-pages/man3/dlsym.3.html.

[13] Shalev Leah, Satran Julian, Borovik Eran, and Ben-Yehuda Muli. 2010. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the USENIX ATC 2010 Conference.*

[14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM.*

[15] NTOP 2017. *PF_RING: High-speed packet capture, filtering and analysis.* NTOP. http://www.ntop.org/products/packet-capture/pf_ring/.

[16] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys 12).*

[17] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12).*

[18] Han Sangjin, Marshall Scott, Chun Byung-Gon, and Ratnasamy Sylvia. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the USENIX OSDI 2012 Conference.*

[19] ScyllaDB 2017. *Seastar.* ScyllaDB. http://http://www.seastar-project.org.

[20] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. 1996. NetPIPE: A Network Protocol Independent Performance Evaluator. In *IASTED international conference on intelligent information management and systems.*

[21] Lin Xiaofeng, Chen Yu, Li Xiaodong, Mao Junjie, He Jiaquan, Xu Wei, and Shi Yuanchun. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proceedings of the ACM ASPLOS 2016 Conference.*