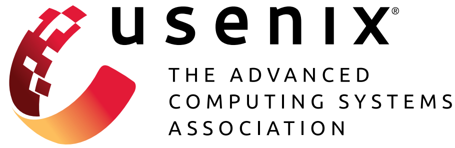


Deploying User-space TCP at Cloud Scale with Luna

Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu, *Alibaba Group*

https://www.usenix.org/conference/atc23/presentation/zhu-lingjun

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

Deploying User-space TCP at Cloud Scale with LUNA

Lingjun Zhu*, Yifan Shen*, Erci Xu†, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, Zhendan Yang, Zhongqing Chen, Wei Lin, Yijun Hou, Rong Liu, Chao Shi, Jiaji Zhu, and Jiesheng Wu

Alibaba Group

Abstract

The TCP remains the workhorse protocol for many modern large-scale data centers. However, the increasingly demanding performance expectations—led by advancements in both hardware (e.g., 100Gbps linkspeed network) and software (e.g., Intel DPDK support)—make the kernel-based TCP stack no longer a favorable option. Over the past decade, multiple parties have proposed various user-stack TCP stacks offering *things-as-usual* TCP support with significant performance improvement. Unfortunately, we find these proposals may not function well in the field, especially when subjected to large-scale deployments.

In this paper, we present LUNA, a user-space TCP stack widely deployed at Alibaba Cloud. We elaborate on the design tradeoffs, emphasizing three unique features in thread, memory, and traffic models. Further, we share our lessons and experiences learned from the field deployment. Extensive microbenchmark evaluations and performance statistics collected from the production systems indicate that LUNA outperforms kernel and other user-space solutions with up to 3.5× in throughput, and reduce up to 53% latency.

1 Introduction

At Alibaba Cloud, we follow a "compute-storage disaggregation" philosophy to enable the frontend computing servers (a.k.a Elastic Computing Service) and backend storage services (e.g., Elastic Block Storage, EBS) to evolve and scale separately. Initially, we adopted kernel TCP to connect computing servers to the storage servers for high compatibility and out-of-the-box usability.

However, advancement in hardware, including ultra-low latency (ULL) NVMe SSDs and high linkspeed networks (e.g., 100Gbps to 200Gbps), have significantly raised users' expectations for cloud storage systems. Kernel TCP is no longer a suitable option to deliver satisfactory performance, as it cannot fully leverage these benefits, and can lead to high tail latency and low single-core throughput. Moreover, kernel TCP can impose the well-known "data center tax" (e.g., consuming 70% of its CPU cycles in the kernel) [6, 20, 22].

Back in 2017, we started to notice such mismatches between the inefficient kernel TCP stack and the growing capabilities of new devices. We then began to look for an alternative solution to connect the frontend servers to backend storage systems. We explored the possibilities of replacing TCP with other protocols, such as Remote Direct Memory Access (RDMA), or leveraging hardware offloading (e.g., TCP Offload Engines). In fact, we have successfully deployed RDMA within our backend storage systems and achieved the expected performance gains [14]. Moreover, we have also designed a UDP-based protocol and leveraged Data Processing Units (DPUs) to accelerate one of our services, EBS [29].

Yet, interconnecting frontend and backend for all services is a different story. First, it requires inter-DC support to let storage services to be accessed from geo-distributed availability zones—not well supported by RDMA back then. Moreover, the interconnection network needs to provide compatibility and legacy support for various services, thereby prohibiting a complete overhaul with both the protocol altered and specialized hardware installed.

It is user-space TCP to the rescue. We noticed that a series of work, from both academia and industry, had demonstrated great performance potentials (e.g., saturating 40Gbps with IX [7]) by moving the TCP from the kernel to the user space [7, 9, 20, 21, 32]. More importantly, user-space TCP solutions provide a familiar programming model to the upper-level applications and offer legacy support by nature.

Unfortunately, we are unable to shoehorn existing user-space TCP solutions onto our production systems. First, these stacks normally use separate threads for application logic and the TCP processing (e.g., IX [7] and mTCP [20]), thereby incurring high communication overhead and impacting our Service Level Objectives (SLOs). Second, these solutions usually follow a copy-based memory model (e.g., mTCP and VPP [9]), aggravating memory bandwidth bottlenecks. Third, existing solutions require exclusively ownership of the NICs, thus preventing legacy support for kernel traffic.

In this paper, we present LUNA, a user-space TCP stack in Alibaba Cloud. We have successfully deployed LUNA in the field for more than 5 years and enable it to be the de-facto transport layer for all new servers in Alibaba Cloud since its release. Similar to previous practices (e.g., mTCP [20] and IX [7]), LUNA runs in a LibOS mode, operates in a shared-nothing architecture between threads and leverages DPDK user-space driver support.

*Equal contribution
†Corresponding author

Compared to previous practices, there are also three unique features in thread, memory and traffic models help LUNA successfully serve as an alternative to the kernel TCP. First, LUNA uses the run-to-completion (r2c) thread model. In each thread, LUNA packs both the application logic and TCP stack together, and process them with an event loop in each thread. Under this design, LUNA can significantly reducing the context switch overhead. Second, LUNA supports full data-path zero copy for both send and receive buffers based on a user-space slab subsystem. The zero-copy buffer effectively reduces the overhead from data movement. Third, LUNA can collaborate with kernel TCP stack to provide legacy support. We orchestrate the two types of traffic in the same NIC by utilizing Flow Bifurcation and SR-IOV to reserve certain ranges for user-space traffic.

We extensively evaluate LUNA against kernel TCP and two other user-space TCP implementations (mTCP [20] and VPP [9]) in a series of microbenchmarks. Results show that LUNA can outperform kernel and other user-space TCP stacks with up to 3.5× in throughput and reduce latency by up to 53%. Also, we compare performance between LUNA and kernel in the field across three representative scenarios. The field statistics show that LUNA could reduce latency by up to 50% and/or improve throughput by up to 50%.

The rest of the paper is organized as follows. We introduce the network architecture and the corresponding requirements at Alibaba Cloud (§2). We discuss the motivations behind LUNA in §3. We present the LUNA overview (§4) and three features in thread (§5), memory (§6) and traffic model (§7) designs. We further conduct series of evaluations on both microbenchmarks and field deployment (§8). We end this paper with several lessons we learned from deployment (§9) and a short conclusion(§10).

2 Background

2.1 Alibaba Cloud Storage Network Architecture

Alibaba Cloud offers various storage services, such as Elastic Block Storage (EBS), Object Storage Service (OSS), and Cloud Tablestore Service (OTS). In Figure 1, we illustrate the typical three layers of a service, including the interface, the function, and the persistence. In a nutshell, the interface layer comprises a set of servers to relay users' requests from the Internet (e.g., OSS and OTS) or the virtual stack within computing instances (e.g., vhost for EBS) to the function layer. The users' requests are further parsed and processed by the function layer (e.g., the BlockServers of EBS) before finally being sent to the persistence layer (i.e., Chunkstore Server, the storage engine of our distributed file system Pangu) for storage or retrieval. We use Pangu [25], an HDFS-like distributed file system developed by Alibaba Cloud, as the persistence layer.

Our cloud architecture adheres to a "compute-to-storage disaggregation" philosophy. This allows the computing servers—hosting the interface layer and virtual machine instances subscribed by the users—and backend storage systems (i.e., the function and persistence layers) to scale and evolve at different paces. In this case, we can divide our networks into two scopes, the frontend network and the backend network (see Figure 1). In the frontend network, we primarily use TCP. The backend network normally follows a two-layer Clos topology of Point of Delivery (PoD) and utilizes TCP or RDMA [14]. We focus on the frontend network in this paper.

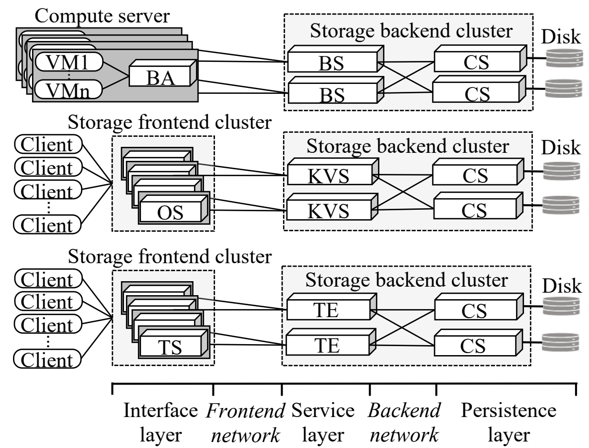


Figure 1: Alibaba Cloud storage network architecture. ***VM**: Virtual Machine; **BA**: Block storage interface Agent; **OS**: Object storage interface Server; **TS**: Tablestore interface Server; **BS**: Block storage function Server; **KVS**: OSS Key-Value Server; **TE**: Tablestore Engine; **CS**: Pangu distributed file system Chunkstore Server.*

2.2 Requirements for Our Networks

Following hardware evolution. With an ever-increasing user base, we are always in the process of expanding our fleet to accommodate more users and offering better performance. To achieve such goals, cloud vendors like us usually seek help from hardware advancement. For example, there are two aspects of recent development that fundamentally change the landscape of our data center networks. First, the network linkspeed has jumped from 10Gbps to 50Gbps and, more recently, 200Gbps. Moreover, high-throughput and low-latency storage devices become readily accessible. For example, off-the-shelf products, such as Intel P5800 [19] and Samsung Z-NAND SSD [36], can achieve up to 6GB/s throughput with around 10 microseconds latency. The combination of the two provides opportunities but also drastically raises the users' expectations of the cloud storage services.

Inter-DC access. A major difference between frontend and backend network is that the former needs to support computing servers accessing the storage servers from different clusters, data centers or even geographically far-apart availability zones. On the contrary, supporting the latter is relatively straightforward—normally a two-layer Clos of PoD.

Legacy support. Over the years, the sizes and types of our

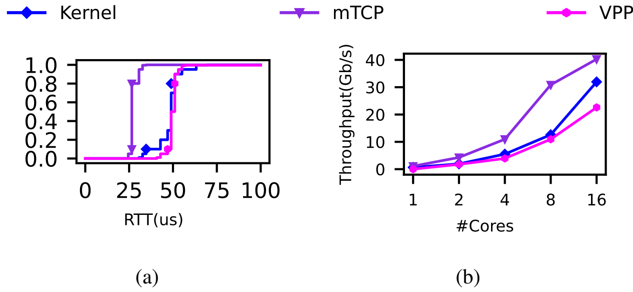


Figure 2: Different network stacks' performance in an offline RPC microbenchmark. a) RPC latency CDF with a single core; b) Maximal throughput and multi-core scalability.

storage services have been rapidly growing. Consequently, many outdated servers, while functioning, do not offer certain functionalities (e.g., DPDK-ready NICs), thereby may still require classic kernel traffic support.

3 Motivation & Related Work

In this section, we revisit the motivations behind LUNA. Although LUNA was initiated back in 2017, we believe the many observations and related work that inspired LUNA remain valid today and may even be strengthened over time.

3.1 Revisiting Kernel TCP

The kernel TCP was (in 2017) and may still remain to be a popular choice for large-scale clusters for its ease of usage and compatibility. However, it has become evident that the kernel TCP can no longer meet the performance demands of data centers (e.g., charging expensive data center tax due to kernel interruptions and memory copies) [6, 22, 33]. Here, we examines the kernel TCP performance via a microbenchmark.

We set up *knb* (Kuafu Network Benchmark)—an internal network stack benchmark that emulates the RPC services in the datacenters—to evaluate the kernel TCP latency and throughput. In this test, the client node sends a RPC request with 4KB messages to the server, measures the latency when receiving the RPC response with the same message size from the server, and then sends out the next request. (See § 8 for *knb* detailed usage.)

From Figure 2, we can see that the kernel TCP performance is far from our SLOs. Specifically, Figure 2(a) indicates the median basic RPC latency of kernel TCP has already reached 50μs. In stark contrast, our high-performance class EBS requires end-to-end response latency to be 100μs [3]. Figure 2(b) further shows that the kernel network stack on a single core could only provide a maximum of 600Mbps throughput. Moreover, Figure 2(b) also reveals that the performance issue of the kernel TCP cannot be resolved by allocating more cores. A possible reason is that the kernel overhead—such as inter-core competition—increases with the scaling of CPU cores.

3.2 Beyond Kernel TCP

Since the kernel TCP can be inefficient for the modern data center networks, it becomes urgent to explore possible alternatives. In general, there are three types of solutions: 1) developing new protocols, 2) moving TCP stack to user space, and 3) hardware offloading.

First, researchers have been proposing new transport layer protocols to replace TCP [4, 13, 16, 30]. pFabric [4] assigns priorities to packets based on the flow size, and tends to discard lower-priority packets in the switch when the buffer is full, thereby achieving both high throughput and low flow completion time. pHost [13], Homa [30] and NDP [16] leverage receiver-driven traffic control where the sender's sending rate is limited by the tokens sent from the receiver side. As each receiver has a global view of the incoming traffic, the receiver-driven protocols can achieve high bandwidth and low latency, and avoid the in-cast issue. For instance, Homa achieves less than 15μs for short messages on a 10 Gbps network running at 80% load. Moreover, these new protocols can eliminate the head-of-line blocking issue in TCP due to its byte-streaming nature.

Moreover, there are multiple work explore building a user-space TCP stack, such as mTCP [20], IX [7], ZygOS [35], TAS [23] and F-Stack [1]. These proposals usually leverage the user-space NIC drivers such as DPDK [2] to directly access packets from the NIC queues, and optimize the performance with techniques such as polling, batching [7,20], cache planning [23], lock-free [7], and zero-copy buffers [7,24]. The user-space network stacks minimize the overhead from the kernel, and demonstrate significant performance gain under the hardware advancement. For example, IX could achieve one-way latency of 5.7μs and fully utilize the 40Gbps bandwidth with 8 cores.

Third, there are several attempts aim to resolve the network performance issues with hardware assistance. For example, offloading TCP processing entirely [8,37,38] or partially [31] to specific devices can significantly improve packet processing performance and reduce the CPU overhead. RDMA [15] offloads traffic control and data movement to hardware, thus bypassing the CPU and achieving microsecond-level latency.

3.3 Our Choice

Among the available options, we chose to build a user-space TCP stack (i.e., LUNA) based on two aspects of reasons.

Inter-DC access and legacy support. Recall that, unlike the backend network, the frontend network needs to provide inter-DC access support (e.g., connecting computing servers from geo-distributed availability zones to storage servers). Therefore, we did not choose RDMA because it did not support inter-DC communication back then. Further, designing and deploying a new transport protocol (with hardware offloading) may also be rather challenging due to the required support for legacy software/hardware (e.g., sharing the NICs with kernel TCP traffic).

Table 1: Characteristics of existing user-space TCP

	kernel collaboration	zero-copy	high throughput	low latency
mTCP				
VPP			✓	
IX		✓	✓	✓
LUNA	✓	✓	✓	✓

Engineering effort. Note that enabling inter-DC access over RDMA or providing legacy support with the new protocol is still achievable but can be time-consuming. For example, the intra-region RDMA solution was not proposed until 2021 and a recent paper further discusses their strenuous effort on realizing wide-area RDMA accessing at Azure Cloud [5]. Back in 2017, we did have a rather tight timetable. Therefore, we chose user-space TCP stack solution to avoid designing/debugging the protocol and/or hacking support for legacy hardware. In fact, it only takes us 9 months to build LUNA from scratch to deployment. Later on, we have gradually added various optimizations and patches over the next five years (§ 9).

3.4 Why Not Just Use Existing Solutions?

Once we decided to use a user-space TCP stack for the frontend network, the next question is—"should we employ existing user-space TCP solutions or should we build our own?" Owning to the following reasons or concerns, we chose to develop a new user-space TCP stack, called LUNA.

High packet processing overhead. The microsecond-scale Service Level Objectives (SLOs) place significant pressures on packet processing speed within the network stack. Several existing user-space TCP solutions (e.g., mTCP and IX) delegate TCP protocol processing and application logic to separate threads for better portability. Meanwhile, others (e.g., VPP and TAS [23]) assign network and application processing to different cores for better scaling. Such partitioning could slow the processing speed due to context switch overhead or inter-core communication. For example, in Figure 2(a), we further profile the mTCP and VPP with the microbenchmark in §3.1. The results indicate that VPP suffers high latency, mainly introduced by the CPU cycles waste, and inter-core communication overheard.

Expensive memory copying. Data movement contributes a large proportion of datacenter tax. In a typical cloud storage service test on the 50Gbps network, memory copy can consume up to 12.5% CPU cycles, severely impacting the end-to-end latency (see §8). When the bandwidth grows to 100Gbps or more, the memory copy will take more than 40% CPU cycles, and further incur the memory bandwidth bottleneck problem. However, for user-space network stacks with a traditional IO path like mTCP [20] and VPP, there are two copy operations on the both receive and send paths (i.e., from user to TCP receive/send buffer and between TCP buffer to the packet). Figure 2(b) shows that, mTCP could not fully utilize the 50Gbps network bandwidth even with 16 cores. Our further analysis concludes that the memory copying does consequence with excessive overhead. One reason that most user-space TCP solutions do not support zero-copy buffer is that the traditional BSD-like socket interface would introduce inevitable memory copy between the application and TCP buffer for isolation.

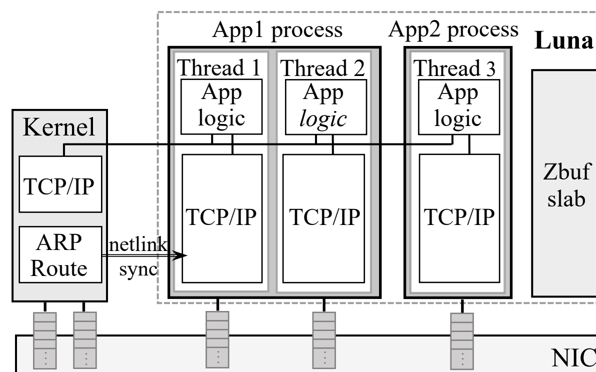


Figure 3: LUNA architecture

Supporting kernel traffic. Our storage services are deployed across multiple clusters consisting of several generations of machines. For many servers, their hardware are not capable of running user-space networks (e.g., lack of hugepages support). Moreover, many applications (such as monitoring agents) still rely on the kernel TCP. However, many user-space TCP stacks [1,7,20,23] demand to exclusively own the entire NIC, thus could not collaborate with applications relies on kernel network stack on the same machine.

Implementation quality. Many existing user-space TCP works are research-oriented and thus can have various compatibility or performance issues. For instance, VPP is not well-compatible with Mellanox NICs when applying flow director filters. IX requires a particular Linux kernel version to run as it relying on the Dune kernel module, and also only provides drivers to the Intel NICs of outdated versions. Hacking into these problems will take an unexpected amount of engineering effort with rather limited community support. Hence, building a new user-space TCP from scratch can be actually more time-saving.

4 LUNA

4.1 Overview

LUNA is a high-performance user-space TCP/IP network stack that powers the frontend network in nearly all Alibaba Cloud Storage Services. Figure 3 shows that LUNA supports both kernel and user-space IO paths. LUNA leverages NIC multi-queues to separate user-space traffic from the kernel's. The u/ser-space IO path bypasses the kernel by leveraging the DPDK's user-space driver [2] to poll packets from the NIC

queues, processes them with a customized TCP/IP network stack, and interacts with applications mainly through an RPC library. The user-space IO path follows a run-to-completion mode, which runs iterations to complete both packet processing and application logic in a single thread on each core, and shares nothing between cores. LUNA separates kernel traffic and user-space traffic with NIC's hardware support. In this case, the kernel traffic for traditional applications remains unaffected. LUNA leaves the control plane (i.e., ARP table and route table management) to the kernel, and uses the *netlink* interface to access the route information.

4.2 Similar Design Choices

LUNA shares several similarities with existing work in the data path and the architecture. Here, we will discuss our rationales behind these design choices.

LibOS mode. LUNA operates in a LibOS mode, similar to the mTCP [20] and F-stack [1]. In this setup, the application and LUNA run in the same process and share the memory address space. Alternatively, one can run the network stack in the separate process, and communicate via shared memory. In this case, if there is only one network process in every server to serve different application processes, it is called the *Microkernel* mode (e.g., Google Snap [27]). Another solution is to have one network stack process for each application process, known as the *Sidecar* mode.

Although microkernel-based solutions at the industry level were not widely popular back in 2017, we later—after the launch of LUNA—have observed several instances adopting this approach. One notable example is Snap [27]—a user-space network stack deployed in Google's datacenter. One prerequisite for Snap is to closely follow the weekly release cycle of the network stack in Google Cloud. As a result, to avoid service interferences, the support transparent/live upgrade becomes indispensable. For microkernel-based solutions, enabling transparent/live upgrade is rather straightforward as the network stack is an independent process. Further, to achieve high efficiency, Snap also adopts Google's Pony Express transport protocol and one-side operation.

LUNA does not adopt the Microkernel or Sidecar mode due to performance concerns because both modes require frequent inter-process communication, incurring high runtime overhead. Note that LUNA still uses TCP for the transportation layer (see§3) and thus can not fundamentally modify the protocol like Pony Express or adopt one-side operation for high performance. We are aware that LibOS mode lacks live/transparent upgrade support as the network stacks with LibOS mode have to be compiled with the application code together. From our perspective, this disadvantage is acceptable as the TCP protocol is rather mature (i.e., not requiring frequent changes) and our storage services have periodical upgrading schedules. Hence, LUNA can just follow storage service upgrading roadmaps.

User-space NIC driver. Like most kernel-bypass network systems [1, 7, 10, 20, 21, 23, 32, 35], we build LUNA with DPDK [2] for its rich development kits and active community support. LUNA leverages DPDK's PMDs (Poll Mode Drivers) to directly access packets from the NIC queues. We also utilize DPDK's hugepage management, and data structures like hash map and *mbuf*. Further, as there are several generations of NICs in our cloud, the user-space driver provides a convenient way to communicate with them in a uniform interface.

Share-nothing architecture. To exploit the parallel processing capability of multi-core systems, like many previous designs [7,20], LUNA runs the threads in a share-nothing mode. Each core processes its own traffic divided by the NIC's multi-queue technique, and finishes related application-layer processing on the same core. LUNA does not use a dispatcher mode (like TAS [23]) or load balancing (like task-stealing in Shenango [32]) due to cache efficiency and synchronization overhead (e.g., from lock and atomic operations) concerns. Note that there is already a service-level load balancing inside applications. The share-nothing architecture improves multi-core scalability, as the system can simply allocate more cores to applications to improve performance. Moreover, run-to-completion LibOS avoids the potential CPU cycles waste in dispateched mode as the user-space network stack has to keep polling the NIC queues.

TCP stack. As discussed in § 3, LUNA uses TCP as the transport layer protocol. LUNA implements TCP according to RFCs [28, 34], and supports congestion control, flow control, RTT estimation, and SACK. LUNA is compatible with other standard TCP stacks like Linux kernel network stack.

4.3 Unique Features

To better serve our systems in the field, LUNA also includes unique features from the following aspects:

Thread model. LUNA uses a run-to-completion thread model to run network and application-layer processing in the same thread. We use this design to improve the performance, and avoid risks in scheduling (e.g., thread-hang) with the characteristics of our storage service workloads (§ 5).

Memory model. LUNA supports a full data-path zero-copy buffer on receive and send end, aiming to minimize the data movement overhead. LUNA realizes its full-stack zero-copy with the aid of a user-space slab subsystem. This subsystem introduces little overhead and maintains the traditional programming model.

Traffic model. LUNA collaborates with kernel network stack, to offer the legacy applications with kernel TCP support, and to leverage kernel TCP for the control plane (e.g., ARP). LUNA uses the Flow Bifurcation and SR-IOV support to reserve a certain port range for user-space traffic, so that there is no interference with kernel traffic. The kernel network stack directly processes the control plane messages such as ARP requests and responses, and manages the control plane

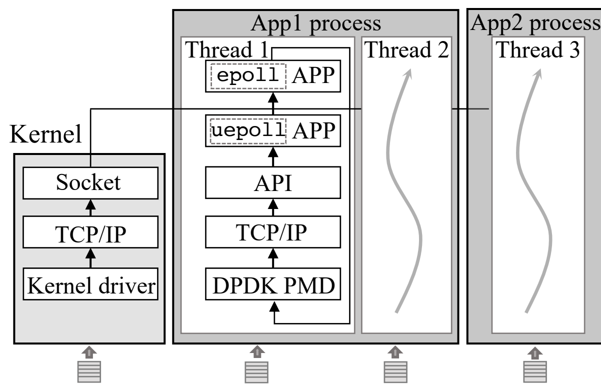


Figure 4: LUNA thread model

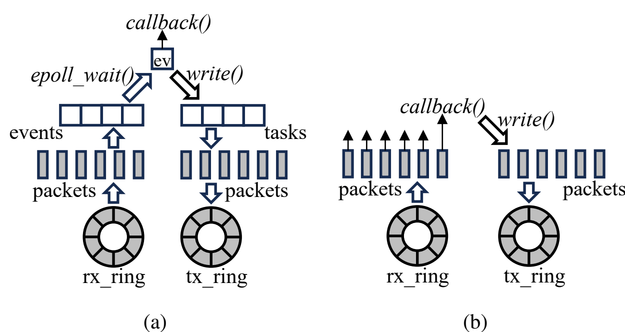


Figure 5: run-to-completion thread model in LUNA. a) batch-r2c mode; b) inline-r2c mode.

states. LUNA can obtain control plane information via *netlink* interface.

5 Thread Model

Figure 4 shows that LUNA uses a run-to-completion (r2c) thread model that encapsulates application logic and network stack processing in a single thread. During runtime, LUNA employs an event loop for each thread. Between threads, LUNA keeps a share-nothing isolation. Moreover, LUNA supports applications to use both kernel network stack and user-space IO path (§ 7), and the two types of traffic could be processed in the same thread.

5.1 Run-to-complete model

We assign varying numbers of cores to LUNA based on the service type. For instance, in a block server (i.e., the functional layer of EBS), LUNA can utilize 8 cores, while a computing server (from ECS) typically employs only 4 cores. Each core initiates one thread and shares nothing with the other. Moreover, each thread uses an event loop to manage data processing.

LUNA offers two distinct r2c modes——*inline-r2c* and *batch-r2c*——each tailored for different scenarios. In both r2c modes, LUNA starts the loop after receiving a fixed number of packets (called a batch) from the corresponding NIC Rx queue. Then, LUNA processes the packets based on the type of r2c mode.

For batch-r2c, Figure 5(a) shows that LUNA processes the received packets one at the time through the TCP/IP stack, and then adds a read event to the event queue for every packet with TCP payload. These read events would be immediately processed by application after the LUNA has processed all the received packets in this round. Then, the RPC framework invokes the callback functions registered to each event, generates the response messages, and sends the messages to the send buffer. After all the events are processed, LUNA adds the protocol headers for the messages in the send buffer, forwards them to the NIC, and starts the next round.

For inline-r2c, Figure 5(b) demonstrates that LUNA also processes the packets one by one. However, LUNA avoids adding event to the event queue, and instead immediately invokes the registered callback function, generates the response along with the protocol headers for the packets, and send them out. In short, inline-r2c will process every packet to completion.

Obviously, inline-r2c eliminates the overhead from event enqueue and dequeue, and improves the cache locality, thereby providing better performance. However, inline-r2c also requires a new programming model and forces the upper-layer application to use a zero-copy raw-packet-like read-/write interface. Moreover, inline-r2c is only available in LibOS model as the application-layer code has to co-locate with the network stack. In contrast, batch-r2c works in a more traditional epoll-like or libev-like programming model, and is compatible with a traditional BSD-Socket-like interface. In practice, we deploy inline-r2c on performance-oriented services like EBS, and use batch-r2c for services such as object store due to compatibility concerns.

The r2c design can significantly reduce the overhead and improve performance. First, there is no context switch between application and stack processing. Second, as the network stacks receive a fixed-sized batch of packets from NIC in each iteration, it allows the upper-layer application to handle them in a timely fashion (i.e., no need for buffering packets). Hence, CPU could get most data from the L1 and L2 cache directly, especially in inline-r2c. Moreover, as there are few buffered packets, the DDIO will not fill up the Last Level Cache (LLC), and further improves the cache hit rate [11].

5.2 Discussion

The risks of r2c model. R2c model could significantly improve performance. However, it is not favored by many user-space TCP stacks. A primary reason is that r2c model may be stuck at application level, causing severe tail latency and packet drop in NIC queues. LUNA adopts the r2c design because the logics in our storage services (i.e., applications) is rather simple and stable. Moreover, another safeguard is that our applications also adopt flow control and can avoid burst traffic by limiting the number of concurrent connections and in-flight requests. Hence, with relatively simple logic at

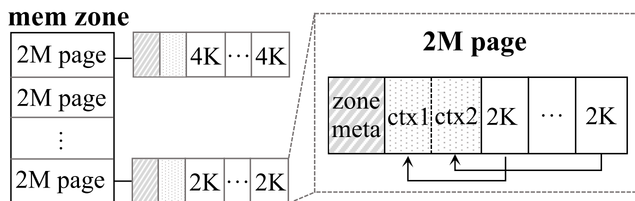


Figure 6: The structure of *Zbuf*.

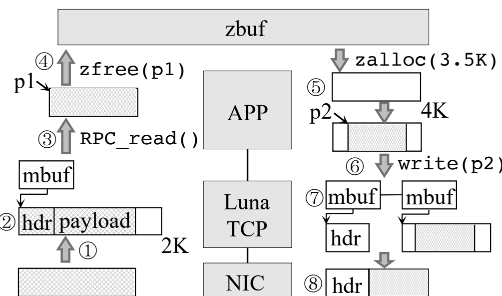


Figure 7: The LUNA receive and send path with zero-copy.

the application level and flows throttled, it is unlikely for the requests to be stuck in the application level and cause packet drop. Note that it is still possible for a thread in LUNA to get stuck by various exceptions (e.g., application bugs, burst incast load, etc.), resulting in dropping packets and TCP/RPC timeouts.

Unsuitable scenarios for r2c model. The LUNA's r2c design is not always suitable for all scenarios. In OSS service, we discover that dedicating multiple cores to polling for run-to-completion is not reliable because there are other services on the same machine that should be guaranteed with a large number of cores. Therefore, LUNA dedicates only one core for NIC IO and protocol processing, and places application logic to other threads on the other cores. The application thread will block the event-poll when there is no more events to avoid wasting CPU cycles on unnecessary polling.

6 Memory Model

LUNA achieves full-stack zero-copy to mitigate the overhead associated with frequent data movement. A straightforward approach to realize end-to-end (i.e., from NIC to TCP, and application) zero-copy is to only transfer the memory addresses of the read/write buffers. This is challenging for user-space TCP due to three factors. First, the lifecycle (and status) of a buffer is different from the application to the network. The application typically frees or reuses the buffer after dispatching it to the network stack, whereas the network stack must retain the buffer until it receives "acks". Second, the NIC requires the physical memory address while applications use the virtual address. Third, the traditional BSD-socket APIs and socket-oriented programming models are designed with copy semantics for the isolation between the user space and kernel space. To overcome these challenges, in LUNA, we build a user-space slab system, called *Zbuf*, to provide cross-layer memory lifecycle management and address translation.

6.1 *Zbuf*

User-space slab subsystem. *Zbuf* works as a user-space slab subsystem that pre-allocates memory chunks for users. Figure 6 shows the structure of *Zbuf*. We can see that *Zbuf* reserves several hugepages allocated from the DPDK's memory address space and divides them into multiple 2MB *memory zone*s. The header of each *memory zone* records the meta information such as the physical address. *memory zone* is further split into *obj*s, which could be directly allocated by the users (i.e., applications). The metadata of *obj*s (denoted as ctx in Figure 6) is co-located with *obj*s in the same *memory zone*, right after zone metadata. All *obj*s within the same *memory zone* share the same size but the sizes can be different from one *memory zone* to another (e.g., 2KB vs. 4KB in Figure 6).

***Obj* lifecycle management.** *Zbuf* uses a reference counter to manage the lifecycle of each *obj*. The counter is set to 1 after initialization. Afterward, the counter increases by 1 whenever the corresponding *obj* is replicated and decreases by 1 each time the *obj* is freed. The *obj* will be put back to the free-list of the *memory zone* once the count reaches 0.

Metadata and physical address translation. Translating the virtual address to the physical address and parsing the metadata are straightforward in *Zbuf*. For example, consider a user allocating a 4KB *obj*. The user then generates a 2KB string str within the *obj*, and the virtual address of str is $addr$. When the user sends str to the network stack, it would increase the reference count of the corresponding *obj*. *Zbuf* first compares $addr$ with the address range of contiguous *memory zone*. By getting the offset of $addr$ to the start of *memory zone* area and dividing the offset with 2MB, *Zbuf* could get the index of the *memory zone* which *obj* belongs to. As the *memory zone* metadata records the start address of contiguous *obj* and the size of each *obj*, the user could directly get the index of the *obj* containing the str, and get the *obj* meta data from the *obj* meta array. Therefore, the users could directly make replicas or free the objects inside the *obj*s, but do not need to manage the *obj*. Since the metadata of *memory zone* records the physical address of itself, the physical address of str could be calculated by adding the offset of str to the *memory zone*. When the str is going to be sent to the NIC, LUNA can calculate the physical address following the same procedures above.

6.2 Full-stack Zero-copy

With the support of *Zbuf*, LUNA provides full-stack zero-copy on data receiving and sending. Moreover, upper-layer applications could still use the traditional programming model except for a few minor changes. Now, we use Figure 7 to illustrate the procedures on both ends.

On the receiving end, LUNA registers *obj* addresses to NIC receive queue, so that the NIC will deliver the received

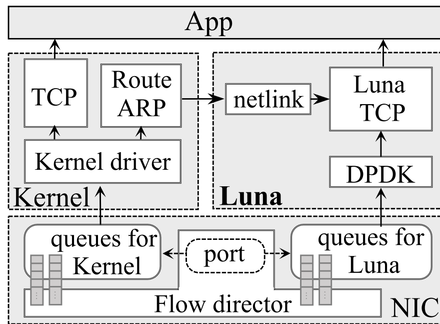


Figure 8: LUNA traffic model

packet data to the *obj* through DMA (①). The network stack processes the packet with network protocols (②) and delivers the pointer to the readable payload (marked as $p1$ in figure 7) to the upper-layer application with an RPC interface (③). The application could directly free the payload after finishing the message processing (④). *Zbuf* will locate the related *obj* and free it by decreasing the reference count.

On the send data path, application first allocates a writable buffer from the *Zbuf*, or reuses the memory space of the received data, and then directly writes data to the buffer (⑤). The content (marked as $p2$ in figure 7) is directly sent to LUNA's TCP send queue, and the send API automatically increases the reference count of the related *obj* (⑥), to avoid the *obj* to be freed by the application after sending. LUNA allocates a new segment to generate the packet header (⑦), and sends out the header and content as a whole packet together through the DPDK interface (⑧). Moreover, if the application wants to multicast the data (typical in cloud storage service), it just needs to send out the same data segment multiple times, and *Zbuf* will accordingly increase the reference count of the *obj*. And the *obj* stored in the LUNA's send buffer will be freed after receiving the acknowledge packet.

6.3 Discussion

Alternative solutions. There are also other design choices to achieve end-to-end zero copy. One is to pass the pointers of the read/write buffers between the application and network stack. However, adopting this practice requires the application to fundamentally alter the programming model and manage the buffer at application layer. For cloud services, it can be rather difficult as there are multiple applications developed by various developer teams. Another approach is to leverage *mmap* to avoid copying large size messages [24], which could maintain the programming model and standard BSD-socket API format. However, the *mmap* call can introduce considerable overhead of additional system calls [24].

7 Traffic Model

7.1 Traffic Split

LUNA uses Flow Bifurcation mechanism [18] (supported by NIC's flow-director) and SR-IOV functionalities to separate kernel network traffic from the user-space network traffic, thereby offering legacy support. LUNA establishes the hardware filtering by setting the mask to the destination port of TCP packets on each machine. LUNA routes the incoming TCP packets with certain destination ports to the specific virtual functions, which are then processed in user space. The TCP packets that do not align with the port filters and the not-TCP packets would still be accepted and processed by the kernel network stack.

LUNA uses different flow-director filter rules for clients and servers. On the client side, LUNA reserves the port from 61440 to 65535 for the user-space traffic, and allocates contiguous sub-ranges of the port number to different LUNA applications. The sub-range within an application are further divided into ranges for different Rx queues which are processed by the corresponding LUNA threads.

For instance, LUNA could reserve TCP port numbers between 61440 to 63487 to APP_A, and write the flow-direct filter rules to direct dest port between 61440 to 62463 to $Thread_1$ in APP_A. Then LUNA can direct TCP ports between 62464 to 63487 to $Thread_2$ in APP_A. On the server side, LUNA first uses flow-director rules to direct TCP packets with the same destination port as the application-listening port to the corresponding application. Moreover, the RPC layer over LUNA will establish full-mash connection for all thread peers between each client node and server node. Hence, LUNA on the server side can simply hash all connections to different server threads according to the source TCP port for load-balance scheduling. For example, the TCP destination port of 1234 is directed to APP_A, and uses the lowest 2 bits of the TCP source port to hash the packets to the 4 threads of APP_A. Each client thread initializes connections with typical host ports to establish a connection with every server thread, and selects a connection for each RPC request for load balancing.

Further, although LUNA is compatible with standard TCP stacks in the design and implementation, LUNA avoids directly communicating with the kernel network stack. In other words, LUNA only supports kernel-to-kernel and LUNA-to-LUNA traffic, and does not permit kernel-to-LUNA connection (or reverse). The reason for this choice is that LUNA implements a tailored TCP for the datacenter environment to optimize the performance (§ 9), and there are different versions of LUNA running in different datacenter applications. If we use LUNA to directly communicate with kernel network stack, we have to verify and evaluate the communication among all versions of kernels and the LUNA. Therefore, this introduces extra verification costs every time updating the kernel or modifying the kernel network configuration.

7.2 Thread Model Support

When the applications need to communicate with different clusters through both kernel and user-space IO paths, LUNA will process them in the same thread, so that the applications don't need to manage the requests separately. In every

iteration, after finishing the user-space run-to-completion processing, LUNA will get a batch of events from kernel's *epoll* framework by calling *epoll_wait()*, and handle the events by calling the callback function registered by the application. LUNA calls *epoll_wait()* with non-blocking, and limits the batch size of kernel events to prevent user-space IO path from starving.

7.3 Control Plane

As LUNA only processes the TCP packets, the control plane packets (e.g. ARP and ICMP requests) are sent to the kernel network stack, and the kernel also manages the control plane states (e.g. ARP table and route table).

LUNA gets the control plane information with *netlink*, a rather infrequent behavior. LUNA initializes a netlink socket which dumps the ARP table and route table in the kernel, and wait event through linux *epoll*. When there are any variations in these two tables, the kernel will send messages to the netlink socket and raise epoll events. Then this would invoke LUNA to receive the update message and update the control plane information managed in user space.

Delegating the control plane to the kernel brings two benefits. First, LUNA could focus on the transport layer and leverage the well-developed kernel control plane implementation. This also enables the LUNA to evolve independently without worrying about keeping up with changes in the control plane. Second, this allows LUNA to fast recover from the system failures, as the just-restarted LUNA could simply regain the control plane information stored in the kernel.

7.4 Discussion

Alternative solutions. There are several approaches to collocating user-space traffic within kernel network stack. The first one is to separate different NICs (or different ports of the same NIC) for user-space traffic and kernel network traffic. Unfortunately, this can severely waste bandwidth. Another approach is to receive all the packets using the user-space TCP and then re-dispatch certain filtered packets back to the kernel via KNI (Kernel NIC Interface). This solution can also impact the performance and crash the whole network service when the failure occurs in user-space network stack.

Complex filtering rules. LUNA splits the traffic according to the TCP ports to collaborate with the legacy applications and employs share-nothing design between cores for high performance. However, the traffic splitting is limited by the NIC hardware capabilities. The commodity NICs (e.g., Intel and Mellanox NIC cards) provide limited flow director support, i.e., setting masks to certain fields of the packet headers (e.g., IP address and TCP/UDP port number).

In LUNA design, one application keeps the same listening port for all server threads to cater the existing programming models. And the LUNA RPC framework establishes full-mash connection channels between every thread of each peer node for load balancing. Therefore, LUNA has to write multiple flow director filter rules to the NIC to spray the traffic to different Rx queues, and the number of rules increases dramatically when the number of LUNA threads is not a power of 2. For instance, when a machine running 12 LUNA threads communicate with a peer node running 6 LUNA threads, this will lead to 160 traffic filter rules on both nodes, imposing significant overhead onto the packet receiving, and resulting in packet drops.

One simple approach to solve this problem is to assign different listening ports for each LUNA thread. However, this also requires modification of the application logic. A more practical way is to reserve range of the port numbers with flow-director, and perform build-in hash-based RSS to establish the connections to different LUNA queues. Yet, this feature is not supported by the commodity NICs when the LUNA was developed. Once the flow bifurcation rule is deployed, we have to provide legacy support for previous versions for compatibility. As a result, LUNA still requires complex filtering rules at the moment.

8 Evaluation

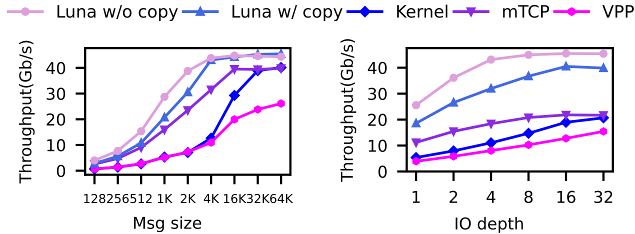
8.1 Microbenchmark Evaluation

8.1.1 Experiment Setup

We first evaluate with microbenchmarks in the emulated client-server environment to compare LUNA against other candidates including kernel TCP, mTCP and VPP. Both client and server machine are equipped with an Intel(R) Xeon(R) CPU E5-2682 v4 @ 2.50GHz CPU with 128GB DRAM each. We connect the client with server through a Mellanox ConnectX-4 Lx NIC with 2× 25Gbps network port, and utilize both the port for a total 50Gbps bandwidth.

For the kernel network stack, we use Linux 4.19, and bind all traffic to the certain CPU cores to optimize the performance. We download the latest versions of mTCP and VPP, and modify them to fit our environment. For VPP, we use half of the cores for the VPP threads processing network packets, and use the rest cores for application logics. As for the common hardware offloading, we enable both *tso* and *lro* for Linux kernel, and enable *tso* for LUNA. For mTCP, *tso* is not supported, and *lro* is not supported by default. For VPP, we failed to enable *lro* and *tso* on our cx4 NIC with default driver after a series of attempts. And the MTU is 1,500 bytes for all the systems.

We use *knb*, a datacenter network microbenchmark, to evaluate the performance of LUNA and the rest. *knb* emulates the RPC workloads in the datacenter, and evaluates the network stack performance at both client and server side. *knb* runs a configured number of threads at client and server, and builds long-lived TCP connection between every client and server thread, similar to most data center RPC frameworks. Then, the *knb* client will send requests with configurable message size to the server. Afterwards, the server send back the response with the same message size on each request, and the



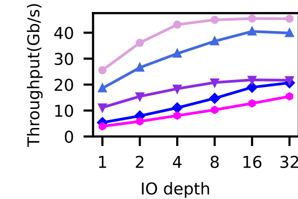
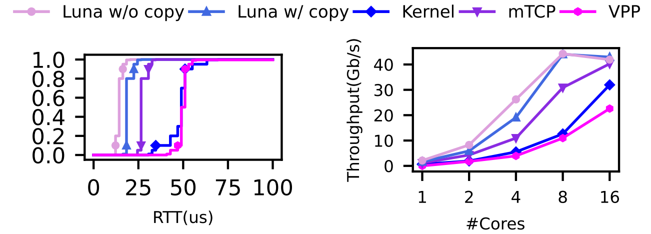


Figure 9: Different message sizes (*iodepth* =1, #core=8)

Figure 10: Different *iodepth* (msg_size=4KB, #core=4)



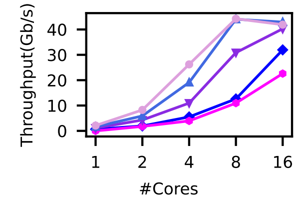


Figure 11: Response latency CDF (msg_size=4KB, *iodepth* =1, #core=1)

Figure 12: Multi-core scalability (msg_size=4KB, *iodepth* =1)

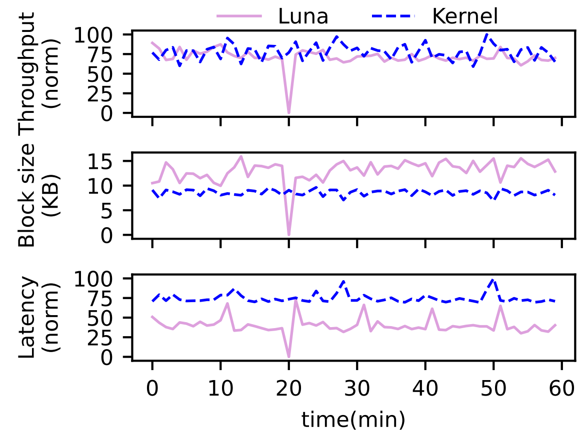


Figure 13: Performance of distributed-storage service for a public cloud OLTP service, collecting tracing data within 1 hour.

client will send another request when it receive a response. The number of in-flight requests on each connection could be controlled with the variable *iodepth*.

8.1.2 Experiment Results

Throughput We first evaluate the throughput under different message sizes. Here, we set LUNA with both the copy mode (backed by batch-r2c) and the zero-copy mode (backed by inline-r2c). In this experiment, we allocate 8 cores for each candidate, and limit iodepth to 1 (i.e., only one request on each connection, and do not send next request until receiving the response). The Figure 9 shows the result. LUNA with zero-copy can fully utilize the bandwidth when the message size grows to 4K, a typical size in our storage services. The throughput of LUNA is 3.5× of kernel, and outperform mTCP and VPP by 50%. The mTCP and VPP do not fully utilize the bandwidth in these experiments. For mTCP, its performance is limited by the heavy overhead from copy and context switch. As for VPP, when the packet size is small, it shows a similar performance with kernel network stack, mainly as a result of wasted CPU cycles on idle polling. When packet size grows larger, VPP shows a even worse performance than Linux kernel (possibly due to *lro* and *tso* are not enabled in VPP).

We also evaluate the throughput under different setups of iodepth as RPCs in datacenters are always concurrent on the same connection. In this experiment, we dedicate 4 cores for each candidate (commonly seen in servers of the interface layer), and set the message size to be 4KB. Figure 10 shows that the throughput grows with the increasing of iodepth. Moreover, the zero-copy version of LUNA can saturate the bandwidth with an iodepth of 16. LUNA provides 2× throughput than mTCP and kernel network stack.

Latency We then evaluate the latency of the network stacks, and show the results in Figure 11. In this experiment, we allocate one core for VPP network worker thread and one core for *knb* application thread. Then, we use a single core when test other network stacks. We set the iodepth as 1 (i.e., no backlog blocking latency) and set the message size to be 4KB. The result shows that, LUNA with zero-copy reduces the 99th percentile latency by 25% than LUNA with copy, and reduce 70% latency than the kernel network stack.

Multi-core scalability. In Figure 12, we evaluate the multi-core scalability of the tested network stacks. In this experiment, we limit the iodepth to be 1, and set the message size to be 4KB. We can see that LUNA shows a near linear multi-core scalability, and full-utilize the bandwidth with 8 cores with zero-copy. mTCP shows relatively good multi-core scalability as it also uses the share-nothing architecture. Yet, it still could not fully saturate the bandwidth due to the performance limitation. The kernel network stack and VPP show a similar multi-core scalability because there are extra inter-core communication overhead and contention over locks between different cores.

8.2 LUNA Performance in the Field

In this section, we will introduce the performance of LUNA with the datacenter storage services in the wild, and make comparison with the traditional kernel network stack. Since LUNA has been deployed in Alibaba Cloud storage service for more than 5 years, most servers that require high-performance are running on LUNA instead of kernel network stack. Therefor, we only show the performance comparisons in the services that still have legacy nodes running the kernel network stack.

EBS for OLTP. OLTP (On-Line Transaction Processing) ser-

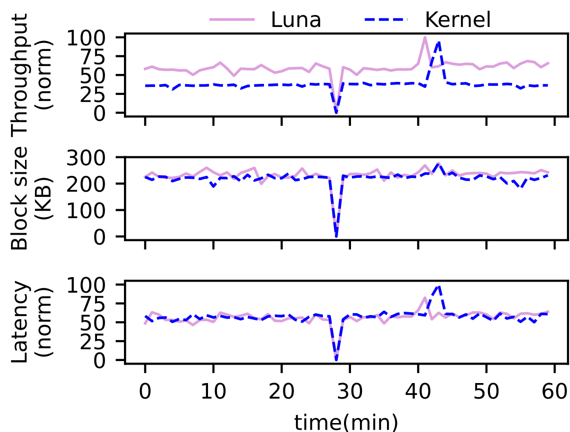


Figure 14: Performance data of distributed-storage service for a public cloud video transcoding service, collecting tracing data within 1 hour.

vice generate typical latency-sensitive workloads for the cloud storage system. Under such workloads, the IO throughput of the storage are usually not very high, but have rather demanding requirements on latency. Here, we show the LUNA deployment in a storage cluster serving a public cloud OLTP service. In these clusters, there are several generations of machines. The LUNA is only deployed to the server nodes that meet the hardware requirements, and still use the kernel network stack in the other legacy server nodes. Note that this cluster uses different network stack IO path only in the frontend network communicating with the block servers, but uses the same version of the backend network and the chunk servers.

In figure 13, we demonstrate the performance of the distributed-storage IO which is captured by the hypervisor-level monitoring during 1 hour. We can see that LUNA outperforms the kernel network stack with both 50% lower end-to-end average latency with similar end-to-end throughput. This gain mainly benefits from the thread model design and zero-copy support therefor reduces the processing and queueing delay. Note that the workload over LUNA has larger IO block sizes as they are from different customers, thereby higher pressure on the delay.

EBS for video transcoding. Here, we show LUNA performance on another EBS cluster which serves a public cloud video transcoding service. In this scenario, the datacenter application requires both high throughput and low latency. This cluster also uses both LUNA and kernel network stacks to communicate with blockserver agents (BAs) in the frontend network. Similarly, the backend network shares the same backend network and chunk server (CS) architecture. all the server nodes share the same workloads. We also collect and compare the user-layer performance data at the hypervisor. Figure 14 indicates that, when the service node with LUNA and kernel network stack have the similar user-level IO latency, LUNA provide about 50% higher IO throughput than the kernel network stack.

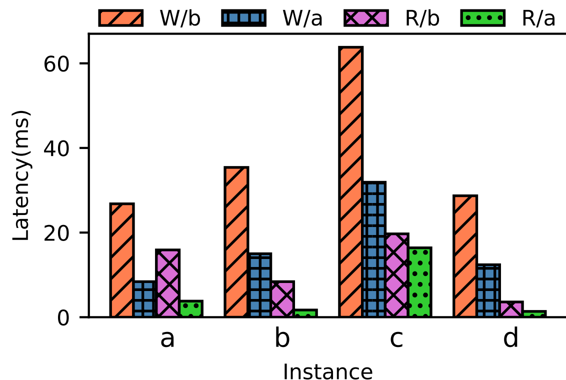


Figure 15: OTS response latency comparison before and after an architecture upgrade, collected from several typical instances.

TABLE STORE. LUNA is deployed in TABLE STORE, a NoSQL database service of Alibaba Cloud. TABLE STORE provides storage and real-time access to massive structured data. Here, we collect and compare the performance data between two generations of the TABLE STORE architecture, named V1.0 and V2.0. The workload remains unchanged after the architecture upgrade. The architecture evolution mainly includes using LUNA user-space network to replace the Linux kernel at both the frontend network and the backend network, and using the user-space file system to replace the Linux ext4. In our offline estimation, LUNA contributes 30%-50% to the total performance gain. Figure 15 shows that, TABLE STORE service instances upgraded to V2.0 reduce the end-to-end latency by 50% to 68%.

9 Lessons From Deployment

Portability. We believe there are four levels of portability.

- Kernel-based applications do not need any code changes to use the new stack (e.g., LOS [17]).
- The application needs to replace the APIs while keeping the same API formats and semantics.
- The API formats are different but the programming model and the API logic stay the same (e.g., change malloc/free to create/destroy).
- Redesign the entire programming model and the API logic.

Our lesson is that, to achieve extremely high performance, refactoring legacy application (i.e., programed with the kernel network stack) is inevitable. However, as a fundamental component of datacenter software codebase, the network stack is often widely used and serves various kinds of applications. Hence, changing the programming model is unacceptable. In this case, we would recommend a user-space network stack to obtain portability between 2) and 3).

In Alibaba Cloud storage, the applications attach to the network through an RPC framework which is also supported

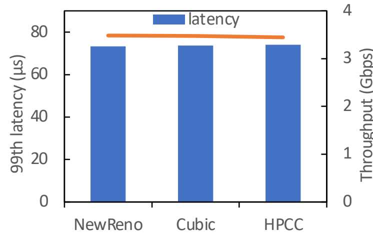


Figure 16: Performance microbenchmark of different congestion control algorithms tested in Alibaba Cloud cloud storage network.

by the network group, and programed in a *libevent-like* model, so adapting to the batch-r2c and inline-r2c does not require any change to the programming model. However, in order to achieve zero-copy at both receive and send side, LUNA provides io-vector-oriented receive/send APIs which directly describe the addresses and lengths of the readable/writable buffers of each RPC call. Therefore, the applications with zero-copy have to change the receive/send API formats. Nonetheless, the data buffers can be allocated or freed as if they are allocated from the heap through the *Zbuf* APIs with the same formats.

TCP tailoring. With a relatively simple environment as the datacenter network, we can tailor the implementation of TCP to achieve better performance. For example, LUNA implements a straightforward yet high-performance TCP fast path for the packets arriving in order. Note that out-of-order packets are not common in the datacenter. In our practice, we found that simply using NewReno [12] could deliver satisfying congestion controlling in many services and does not rely on any novel hardware features (Figure 16). Additionally, we also use HPCC [26] for the communication between VMs in the computing cluster to improve tail latency.

Fast recovery from the failures. For high availability, the network stack should fast recover from the failures such as switch flips and black holes. In our services, applications use LUNA via an RPC framework, which detects connection failure and tries to reconnect the peer node through different network links, e.g., another NIC port, or ToR switch. Further, LUNA also improves the failure recovery procedures based on the characteristics of the datacenter environment. For example, the network distance of every two nodes in our cluster is no more than 4 switch hops. As a result, LUNA could set a tighter timeout threshold, e.g. 4 milliseconds. Currently, the longest recover time for LUNA is guaranteed to be less than 2 seconds.

Alibaba Cloud storage network evolution. The evolution of LUNA is driven by service. Back in 2017, Alibaba Cloud planned to launch a high-performance ESSD service—the first elastic block storage service achieving both 1M IOPS and 100μs latency—around early 2018. With the release date within a year ahead, we therefore chose user-space TCP stack solution to avoid designing/debugging the protocol and hacking support for legacy hardware. It only takes us 9 months to build LUNA from scratch to deployment.

In the initial release, LUNA still uses socket-like APIs and requires data copy from the application to the network stack on the send path. Later, to support the bare-metal servers, LUNA needs to run on a Data Processing Unit (DPU) which has rather limited resources. Therefore, we designed a new RPC framework which removes the RPC serialization stage, supports the inline-r2c thread model (§ 5) and the zero-copy IO on both ends (§ 6). Note that this also requires the EBS to change the programming model to use io-vector-oriented APIs with *Zbuf* for zero-copy, and co-design with LUNA's flow control to adopt the inlined-r2c thread model.

The upper bound of LUNA. In this paper, we discussed that LUNA can efficiently utilize 50Gpbs NICs. However, for even higher bandwidth (e.g., 200 to 400Gbps), the LUNA's run-to-completion with a shallow buffer may lead to NIC queue overflow and packets dropping. Moreover, when the message size is 4KB, LUNA needs at least 8 cores to saturate the 100Gbps network bandwidth. Therefore, for adopting a high linkspeed network, we believe leveraging hardware acceleration becomes necessary. Additionally, while TCP can use multi-path transmission with Multipath TCP, the head-of-line blocking problem in TCP and its limitations in failure-recovery have still led us to design a new protocol specifically for high-performance cloud storage. Our recent effort, called Solar [29], which involves using a new transport layer protocol co-designed with the DPU exemplifies this point.

10 Conclusion

In this paper, we describe LUNA, a user-space TCP stack at Alibaba Cloud storage network. We discuss our efforts in building LUNA with a focus on the thread, memory and traffic model. Apart from introducing LUNA, we have also covered various design tradeoffs and lessons from the last five years of development. We hope the experiences shared in this paper shall benefit practitioners from both academia and industry.

Acknowledgments

The authors thank our shepherd Yizhou Shan and the anonymous reviewers for their feedback. We also thank the EBS, Pangu, AIS Fushionnet, OSS and OTS teams for their tremendous help on the LUNA project. This research was partly supported by Alibaba Innovation Research, Alibaba Research Fellow and NSFC(62102424) program.

References

[1] F-stack. http://www.f-stack.org/.

[2] Intel DPDK. https://www.dpdk.org/.

[3] Aliyun. EBS product. https://www.aliyun.com/product/disk.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: minimal near-optimal datacenter transport. In D. M. Chiu, J. Wang, P. Barford, and S. Seshan, editors, ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013, pages 435–446. ACM, 2013. https://doi.org/10.1145/2486001.2486031.

[5] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill. Empowering azure storage with RDMA. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 49–67, 2023.

[6] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan. Attack of the killer microseconds. Commun. ACM, 60(4):48–54, 2017. https://doi.org/10.1145/3015146.

[7] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014., pages 49–65, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[8] H.-C. Chiang, Y.-P. Dai, and C.-Y. Wang. Full hardware based tcp/ip traffic offload engine (toe) device and the method thereof, Jan. 12 2010. US Patent 7,647,416.

[9] Cisco. VPP. https://fd.io/gettingstarted/technology/.

[10] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021, pages 621–637, 2021. https://doi.org/10.1145/3477132.3483571.

[11] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostic. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In A. Gavrilovska and E. Zadok, editors, 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020, pages 673–689. USENIX Association, 2020. https://www.usenix.org/conference/atc20/presentation/farshin.

[12] S. Floyd, T. Henderson, and A. Gurtov. The newreno modification to tcp's fast recovery algorithm. Technical report, 2004.

[13] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. phost: distributed near-optimal datacenter transport over commodity network fabric. In F. Huici and G. Bianchi, editors, Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT 2015, Heidelberg, Germany, December 1-4, 2015, pages 1:1–1:12. ACM, 2015. https://doi.org/10.1145/2716281.2836086.

[14] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In J. Mickens and R. Teixeira, editors, 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021, pages 519–533. USENIX Association, 2021. https://www.usenix.org/conference/nsdi21/presentation/gao.

[15] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity ethernet at scale. In M. P. Barcellos, J. Crowcroft, A. Vahdat, and S. Katti, editors, Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016, pages 202–215. ACM, 2016. https://doi.org/10.1145/2934872.2934908.

[16] 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, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017, pages 29–42. ACM, 2017. https://doi.org/10.1145/3098822.3098825.

[17] Y. Huang, J. Geng, D. Lin, B. Wang, J. Li, R. Ling, and D. Li. LOS: A high performance and compatible user-level network operating system. In Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017, pages 50–56, 2017. https://doi.org/10.1145/3106989.3106997.

[18] Intel. Flow-bifurcation. https://doc.dpdk.org/guides-18.08/howto/flow_bifurcation.html.

[19] Intel. Intel 5800x. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[20] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014, pages 489–502, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[21] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In 16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019., pages 345–360, 2019. https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[22] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks. Profiling a warehouse-scale computer. In D. T. Marr and D. H. Albonesi, editors, Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015, pages 158–169. ACM, 2015. https://doi.org/10.1145/2749469.2750392.

[23] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. E. Anderson. TAS: TCP acceleration as an OS service. In Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019, pages 24:1–24:16, 2019. https://doi.org/10.1145/3302424.3303985.

[24] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socksdirect: datacenter sockets can be fast and compatible. In Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019, pages 90–103, 2019.

[25] Q. Li, Q. Xiang, Y. Wang, H. Song, R. Wen, W. Yao, Y. Dong, S. Zhao, S. Huang, Z. Zhu, H. Wang, S. Liu, L. Chen, Z. Wu, H. Qiu, D. Liu, G. Tian, C. Han, S. Liu, Y. Wu, Z. Luo, Y. Shao, J. Wu, Z. Cao, Z. Wu, J. Zhu, J. Wu, J. Shu, and J. Wu. More than capacity: Performance-oriented evolution of pangu in alibaba. In 21st USENIX Conference on File and Storage Technologies (FAST 23), pages 331–346, 2023.

[26] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: high precision congestion control. In J. Wu and W. Hall, editors, Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019, pages 44–58. ACM, 2019. https://doi.org/10.1145/3341302.3342085.

[27] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019, pages 399–413, 2019.

[28] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgment options. Technical report, 1996.

[29] R. Miao, L. Zhu, S. Ma, K. Qian, S. Zhuang, B. Li, S. Cheng, J. Gao, Y. Zhuang, P. Zhang, R. Liu, C. Shi, B. Fu, J. Zhu, J. Wu, D. Cai, and H. H. Liu. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In F. Kuipers and A. Orda, editors, SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022, pages 753–766. ACM, 2022. https://doi.org/10.1145/3544216.3544238.

[30] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In S. Gorinsky and J. Tapolcai, editors, Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018, pages 221–235. ACM, 2018. https://doi.org/10.1145/3230543.3230564.

[31] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. Acceltcp: Accelerating network applications with stateful TCP offloading. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 77–92, Santa Clara, CA, Feb. 2020. USENIX Association.

[32] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In 16th USENIX Symposium on Networked

Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019, pages 361–378, 2019. https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[33] J. Ousterhout. It's time to replace tcp in the datacenter. arXiv preprint arXiv:2210.00714, 2022.

[34] J. Postel et al. Transmission control protocol. 1981.

[35] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, pages 325–341, 2017. https://doi.org/10.1145/3132747.3132780.

[36] Samsung. Samsung z-nand ssd. https://semiconductor.samsung.com/ssd/z-ssd/.

[37] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter. Flextoe: Flexible TCP offload with fine-grained parallelism. In A. Phanishayee and V. Sekar, editors, 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022, pages 87–102. USENIX Association, 2022. https://www.usenix.org/conference/nsdi22/presentation/shashidhara.

[38] Z. Wu and H. Chen. Design and implementation of TCP/IP offload engine system over gigabit ethernet. In Proceedings of the 15th International Conference On Computer Communications and Networks, ICCCN 2006, October 9-11, 2006, Arlington, Virginia, USA, pages 245–250. IEEE, 2006. https://doi.org/10.1109/ICCCN.2006.286280.