Z-stack: A High-performance DPDK-based Zero-copy TCP/IP Protocol Stack

Anvaya B. Narappa*, Federico Parola†, Shixiong Qi*, K. K. Ramakrishnan*
*University of California, Riverside †Politechnico of Torino

***Abstract*—Data centers require high-performance and efficient networking for fast and reliable communication between applications. TCP/IP-based networking still plays a dominant role in data center networking to support a wide range of Layer-4 and Layer-7 applications, such as middleboxes and cloud-based microservices. However, traditional kernel-based TCP/IP stacks face performance challenges due to overheads such as context switching, interrupts, and copying.**

We present Z-stack, a high-performance userspace TCP/IP stack with a zero-copy design. Utilizing DPDK's Poll Mode Driver, Z-stack bypasses the kernel and moves packets between the NIC and the protocol stack in userspace, eliminating the overhead associated with kernel-based processing. Z-stack employs polling-based packet processing that improves performance under high loads, and eliminates receive livelocks compared to interrupt-driven packet processing. With its zero-copy socket design, Z-stack eliminates copies when moving data between the user application and the protocol stack, which further minimizes latency and improves throughput. In addition, Z-stack seamlessly integrates with shared memory processing within the node, eliminating duplicate protocol processing and serialization/deserialization overheads for intra-node communication. Z-stack uses F-stack as the starting point which integrates the proven TCP/IP stack from FreeBSD, providing a versatile solution for a variety of cloud use cases and improving performance of data center networking.

***Index Terms*—DPDK, zero-copy, TCP/IP protocol stack, shared memory**

I. Introduction

Data centers now serve as the backbone of almost all modern digital operations, hosting critical applications and services in the cloud. The key to their efficiency lies in robust, high-performance networking that ensures fast and reliable communication between applications. With the widespread adoption of microservice software architecture, applications are being built as a set of loosely coupled functions. This also results in frequent communication between application components which can contribute to increased latency. All of this requires the networking support to be fast and efficient, handling large volumes of data, while ensuring quality of service. In a cloud environment with multiple tenants and with virtualization (e.g., especially containerization), it is also important to consider the benefits of a user-space protocol implementation that can potentially avoid interactions across different tenant flows.

Most application designs still depend on the in-kernel implementation for protocol processing (the kernel TCP/IP stack), using the traditional POSIX sockets. They seek to leverage the full functionality and reliability of a kernel TCP/IP protocol stack. However, the kernel protocol stack has a number of significant performance challenges [1]. Its design, built for generality, was not originally focused on the current high-speed, low-latency requirements of today's data center networking. It has a number of sources of overhead: (1) *Context switches*: Caused by the need to switch between user and kernel mode during packet processing, it adds considerable latency [2]. (2) *Interrupts*: Handling network packet interrupts can be resource-intensive [1]. (3) *Data copies*: Moving data between the kernel and userspace results in additional processing time [1]. (4) *Protocol processing*: Each layer of the TCP/IP stack adds its own headers and performs its checks and computations, which can be resource-intensive. This includes tasks like computing checksums, segmenting/reassembling data packets, and handling retransmissions [3]. (5) *Serialization/deserialization*: The need to serialize and deserialize data for transmission over the network adds to the processing burden [2].

The overhead of kernel-based TCP/IP processing significantly impacts the performance of a variety of network functions and applications in the cloud and leads to additional CPU consumption [2], [4]. Existing solutions [2], [4], [5] seek to leverage shared memory processing to bypass the heavyweight kernel protocol stack and achieve considerable performance improvement in various use cases in the cloud, e.g., serverless computing [2], 5G core network [5], Network Function Virtualization and Middleboxes [4]. However, shared memory processing is limited to a single node and existing solutions [2], [4], [5] still rely on the kernel protocol stack for inter-node communication, leading to performance limitations.

Existing work explores high-performance inter-node communication approaches, such as Remote Direct Memory Access (RDMA [6]) or userspace TCP/IP stack with kernel bypass [7], [8]. However, RDMA does not accelerate communication with external clients outside the data center. TCP/IP stacks using kernel bypass, such as mTCP [7] and F-stack [8], have a POSIX-compatible API design and can accelerate data center networking. It can also improve communication with external clients through the integration of a cluster-wide ingress gateway at the edge of the data center cloud. However, the existing userspace TCP/IP stack solutions often introduce an additional copy when moving data between the user application and the protocol stack. This is particularly expensive when handling large messages (see §IV).

We describe Z-stack, which is a user-space high-performance TCP/IP protocol stack with a zero-copy design.

Z-stack leverages DPDK's Poll Mode Driver (PMD [9]) to bypass the kernel and move packets between the NIC and the protocol stack in userspace. We choose to use polling-based packet processing, as it yields better performance under high load and can eliminate receive livelock [10] compared to interrupt-driven packet processing [11]. Moving protocol processing to the userspace helps Z-stack eliminate a number of kernel-related overheads. such as context switching and interrupts. Further, Z-stack employs a more efficient, zero-copy protocol processing that minimizes latency and maximizes throughput. We achieve this in Z-stack by eliminating the data copy caused by the POSIX-style socket APIs, as in [7], [8]. The data copy at the socket interface is a natural fit with the userspace-kernel separation when the application works with a kernel-based protocol stack. It requires isolating the buffers between the user application and the kernel. However, as we move the protocol stack into userspace, the boundary crossing between the userspace and kernel is no longer needed. Thus, it facilitates the elimination of the data copy at the socket interface to deliver better performance. With its zero-copy design, Z-stack can seamlessly work with the shared memory processing within the node to eliminate duplicate protocol processing and serialization/deserialization overheads when moving the data across multiple components of a user application [2] (e.g., with microservices) or network functions [4].

We implement Z-stack on top of F-stack [8]. F-stack integrates the TCP/IP stack from FreeBSD, which is proven to be fully functional and robust, compared to other TCP/IP stacks such as mTCP [7] and Microboxes [12]. *Z-stack is available at https://github.com/anvayabn/Z-stack.*

II. Related work

Given the challenges with the traditional kernel-based TCP/IP stack, several inter-node communication alternatives have been explored in the past: **(1)** POSIX-compatible userspace TCP/IP stacks: mTCP [7] is a high-performance userspace TCP/IP stack with multicore scalability. It addresses some of the TCP/IP stack's limitations by using kernel bypass and operates fully in userspace. This helps mTCP avoid the overhead of kernel operations. mTCP also supports the efficient handling of network traffic across multiple cores. However, mTCP is not a complete TCP/IP stack replacement [4]. F-stack [8] is a production-level userspace TCP/IP stack developed by Tencent Cloud. It provides a high-performance TCP/IP stack with the help of DPDK's kernel bypass capability, which uses DPDK Poll Mode Driver (PMD [9]) to process network packets in userspace. In particular, F-stack's TCP/IP stack is migrated from FreeBSD, which is fully functional and robust compared to TCP. **(2)** Demikernel [13] is another example. It is a library operating system, in Rust, that leverages kernel-bypass technologies like DPDK and RDMA. It provides memory safety and efficient concurrency management through co-routines instead of POSIX threads. However, Demikernel requires applications to pre-segment their data into MTU-sized segments before being given for transmission. In general, Demikernel does not provide POSIX-like API, thus introducing additional overheads on the application development. **(3)** TAS (TCP Acceleration as an OS Service) [14] uses a different approach, replacing the data path of the TCP stack, with the control remaining in the kernel. Data path features such as congestion control, loss recovery, and packet filtering are implemented in TAS. TAS has userspace threads that execute packet IO using DPDK and post the data to the application, which runs on a separate thread. The partial reliance on the kernel can limit the overall performance gains from kernel-bypass as the control path can become a bottleneck under high network loads. **(4)** LUNA [15], is another approach that uses SR-IOV to split traffic between the kernel and userspace applications. However, LUNA uses a custom TCP/IP stack, unlike the more robust FreeBSD TCP/IP that Z-stack builds on. **(5)** RDMA allows direct memory access from the memory of one computer to another without involving either one's operating system. By bypassing the OS, RDMA reduces latency and improves throughput significantly [6]. It enables faster data transfer rates, crucial for data-intensive applications. In addition, RDMA offloads work from the CPU, freeing it up for other tasks, thereby reducing CPU overhead [6]. However, RDMA is not POSIX-compatible, requiring additional effort to port legacy applications that rely on POSIX sockets. Although there is POSIX-socket-style API support for RDMA such as *rsocket* [16] to simplify application porting, *rsocket* incurs an additional data copy between the application and RDMA transport, which may reduce the benefit of RDMA.

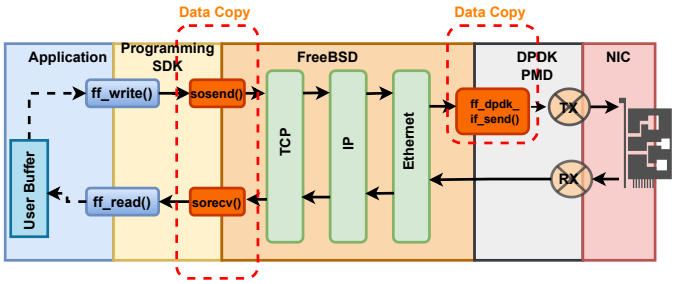

Fig. 1: Data copies in F-stack.

A. Anatomy of data copies in F-stack

We focus on the design of F-stack [8], a userspace TCP/IP protocol stack implementation here, as it aims to streamline network data handling while also ensuring POSIX API compatibility to ease application portability. As a consequence of needing to maintain compatibility with the POSIX API, user applications that utilize F-stack have to move (*copy*) the data into a socket buffer to interact with the FreeBSD TCP/IP implementation in F-stack, as shown in Fig. 1. There is an additional data copy incurred within F-stack as part of protocol processing, occurring at the interface between the protocol stack and the DPDK's PMD. This is caused by the distinct memory management systems (i.e., memory allocator) used by the protocol stack and the DPDK's PMD.

Copy in transmit path: Examining the packet transmission processing in F-stack in more detail (Fig. 1), we see that the data (payload) is moved between the user application and the socket buffer through F-stack's POSIX-style APIs (ff_write()), which introduces an explicit data copy (via sosend()). Subsequent to the protocol processing within F-stack, data payload and protocol headers (TCP/IP) are copied (via ff_dpdk_if_send()) to a contiguous buffer owned by the DPDK's PMD.

Copy in receive path: When examining the receive direction, the NIC DMAs the received data packets to the socket buffer in F-stack, working with DPDK's PMD. Subsequent to the receive-side protocol processing in the userspace F-stack, the data is copied (via sorecv()) from the socket buffer to the receive buffer provided by the user application. The user application consumes the data using the ff_read() provided by F-stack's programming SDK.

Although F-stack provides relatively better performance and is more comprehensive compared to other kernel-based or (even) userspace protocol stacks, it does not eliminate the overhead of data copying, which can be quite significant as we evaluate in §IV. This is true when processing large messages, causing up to 2× throughput degradation for F-stack.

III. Design of Z-stack

A. Overview of Z-stack

Fig. 2 is an overview of the architecture of Z-stack. Z-stack fully operates in userspace to avoid context switches between the kernel and the userspace. It leverages DPDK's PMD to move packets between the NIC and the userspace protocol stack, bypassing the kernel to eliminate the interrupt overhead and the potential receive livelock [10]. The implementation of the protocol stack is ported from FreeBSD. This gave us a thoroughly tested and reliable TCP/IP stack that is responsible for communication between network devices, ensuring that packets are sent, routed, and received correctly. The user application utilizes *zero-copy* socket APIs (`z_read()` and `z_write()`), which use the memory location of the data directly to interact with Z-stack. Z-stack retains the location in the memory of the data and constructs the necessary headers. At the interface, the network headers (TCP/IP) are prepended to the data instead of performing a full copy to DPDK buffers.

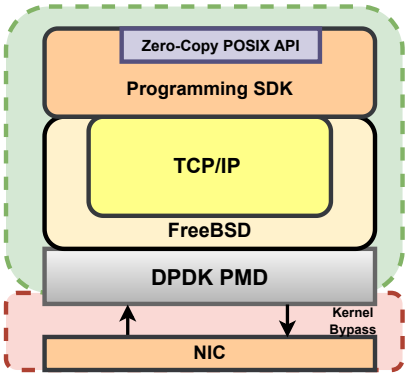

Fig. 2: An architectural overview of Z-stack.

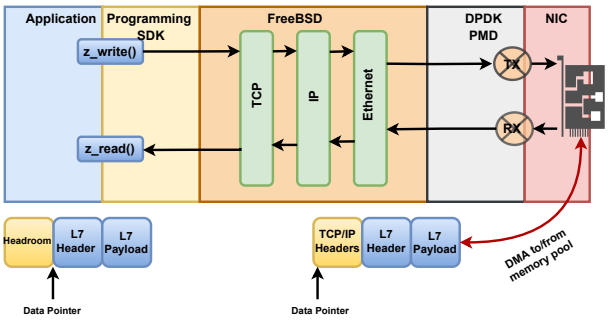

Fig. 3: Zero-copy design in Z-stack.

This process creates a contiguous data block that is then transferred via DMA by the NIC. While we recognize that Z-stack's socket APIs are not fully POSIX compliant, requiring applications to adapt their implementation to support these APIs, the adaptation to Z-stack is straightforward, facilitated by Z-stack's Software Development Kit (SDK). Applications need to prepare the buffer before using the write() API, and provide a memory location when using the read() API. To facilitate application development and interface with the protocol stack, our SDK (slightly enhanced from F-stack's SDK) in Z-stack includes all the necessary coroutines and socket management features such as epoll and kqueues.

B. Zero-copy socket APIs

Fig. 3 depicts Z-stack's zero-copy design. Z-stack removes the data copies by manipulating the buffers in the user application and providing to the customized socket APIs (`z_write()` and `z_read()`) a pointer to the data (i.e., descriptor). This eliminates the memory-memory copies when data is moved between the user application and the protocol stack. The API requires the buffer to be prepared with sufficient headroom to accommodate the TCP/IP header. The header is prepended to the payload as it traverses through the TCP/IP stack.

```
/* Event loop to process network events */
int loop(void *arg)
{
    struct kevent events[MAX_EVENTS];
    int nevents = ff_kevent(kq, NULL, 0, events,
    MAX_EVENTS, NULL);

    for (int i = 0; i < nevents; ++i){
        struct kevent event = events[i];
        int clientfd = (int)event.ident;

       if (event.filter == EVFILT_READ) {
            void *mb;
            ssize_t readlen = z_read(clientfd, &mb
    ,4096);

            if (readlen > 0) {
                char * data = ff_mbuf_mtod(mb);
                ff_mbuf_free(mb);
            }
        }
    }

    return 0;
}
```

Listing 1: Example using `z_read()`.

Zero-copy Read API (`z_read`()): For a read operation, `z_read`() is designed to provide a user application with the pointer to the data in the receive buffer avoiding the need to copy data into a buffer in the user application. As shown in Listing 1, the `z_read`() API is invoked by the application when there is data on the socket. The epoll notification mechanism is used to notify the application of any READ events on the socket. The `z_read`() is called by the application with a pointer (`*mb` in Listing 1) to a memory location for the received data. The `z_read`() API returns the memory location containing the reference to the L7 data buffer. The ownership of the buffer is transferred to the application. Z-stack puts the responsibility on the user application to free the buffer (via `ff_mbuf_free`()) once it completes the processing of data and no longer needs it.

Zero-copy Write API (`z_write`()): The `z_write`() API is called with a descriptor to the payload directly in the user application that it had created (via `rte_pktmbuf_alloc`() in Listing 2). It is typically used in conjunction with a notification mechanism (e.g., epoll) to send data once the socket is writable. The data buffer is manipulated while residing in the user application space, knowing the size of the data to be transmitted. The buffers here are allocated from the DPDK's memory allocator maintaining a headroom for the packet header. The `z_write`() API takes a socket descriptor, a data buffer containing the data to be sent, and the size of the data. Z-stack also eliminates the data copy when moving the data from protocol stack to DPDK's PMD (with F-stack, there is a copy introduced in ff_dpdk_if_send() because data is copied from the application onto a FreeBSD buffer, and then after protocol processing, the data is copied into a 2KBytes DPDK buffers at the interface between FreeBSD and DPDK's PMD). Since the data buffer is managed by DPDK's memory allocator throughout the data path, Z-stack can eliminate this data copy because the user application copies the data directly to the DPDK buffer, and Z-stack can manipulate the same buffer by prepending the TCP/IP header. The data is then handed to the NIC to be transmitted, utilizing the scatter-gather capability of NIC's DMA engine (see §III-E). This direct data path from the user application to the NIC ensures that the payload remains intact.

```
struct rte_mempool *mbuf_pool =
                  pktmbuf_pool[lcore_conf.socket_id];
rte_mb = rte_pktmbuf_alloc(mbuf_pool);
data = rte_pktmbuf_mtod(rte_mb, char *);
// Copy data into buffer
memcpy(data, "Your data here", data_size);
rte_mb->data_len = data_size;
rte_mb->pkt_len = rte_mb->data_len;
// Get a message buffer for sending
mb = ff_mbuf_get(NULL, rte_mb, data, rte_mb->
    data_len);
// Send data over the socket
z_write(clientfd, mb, rte_mb->data_len);
```

Listing 2: Example using `z_write`().

C. Managing Multiple Concurrent Connections

Concurrent connection management is a critical component of the protocol stack to enable the support of multiple user sessions, each of which is distinguished by its own connection. Z-stack is designed to handle multiple connections simultaneously as shown in Fig. 4. The key to implementing concurrent connection management is to bind distinct sockets ("Socket1" to "SocketN" in Fig. 4) to different connections, with a listening socket used to setup the connection between the read/write sockets of the client and server. Multiple connections of a user application share the same listening socket for connection establishment, as is done with typical socket programming. Z-stack's protocol processing demultiplexes packets to connected sockets based on IP tuples (i.e., source/destination IP addresses and source/destination port numbers).

As shown in Fig. 4, the server using Z-stack opens a listening socket (bound to a specific port), which is the primary endpoint to listen to incoming connection establishment requests from clients. The listening socket subscribes to a notification mechanism (e.g., kqueue/epoll). The notification mechanism monitors multiple file descriptors (i.e., sockets) to see if they are ready for I/O operations, thus enabling the server to manage multiple connections concurrently.

Each new client socket is dedicated to a particular client, created by the listening socket. The client socket also subscribes to the notification queue. This allows the server to be notified when there is activity on these client sockets. E.g., when a client sends data, the corresponding client socket in the server becomes "ready for reading," and this state change is reported as an event in the queue. The server's event loop continuously polls the notification queue for new events. When an event is detected on a client socket, the server reads the incoming data from that socket for processing based on the application's logic. If the server needs to send data back to the client, it writes the response to the client's socket.

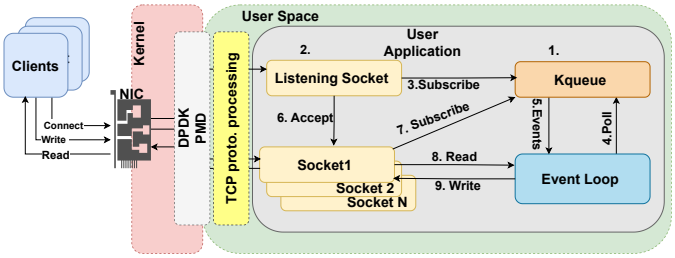

Fig. 4: Concurrent Connection Management in Z-stack.

D. Non-blocking I/O

Z-stack implements non-blocking I/O to manage network communication efficiently. Socket operations such as ff_accept(), ff_connect(), ff_read(), and ff_write() are executed in a non-blocking manner, i.e., when invoked, these operations return control to the application rather than waiting for data to be ready, or for the write to complete. Z-stack also incorporates an event-driven model by using kqueue/epoll-based notification mechanism (Fig. 4). This event-driven notification mechanism

monitors multiple sockets and alerts the application when a socket is ready for further action without a blocking wait. This method of I/O operations reduces the idle time of the application and uses system resources efficiently.

E. Hardware Offloading in Z-stack (TSO/LRO)

Modern NICs typically support TCP Segmentation Offload (TSO [17]) and Large Receive Offload (LRO [17]) to accelerate TCP/IP processing. Z-stack also takes advantage of these features. TSO allows the protocol stack to pass a large buffer as a single unit to the NIC. The NIC hardware takes care of segmenting this large buffer into smaller MTU-sized packets. This reduces CPU utilization and improves overall system performance. In Z-stack, TSO becomes particularly relevant as it uses a single buffer per transmission, avoiding buffer chaining. When a large message is ready for transmission, Z-stack relies on the NIC's TSO capability to handle the segmentation.

IV. Evaluation

We compare the performance of Z-stack against F-stack [8] and Linux kernel protocol stack, using an echo server application. The comparison between Z-stack and F-stack allows us to quantify the performance improvement of Z-stack's zero-copy protocol processing and understand in-depth the overhead of copying data. The comparison against the Linux kernel protocol stack helps us understand the difference in the performance of protocol processing in userspace and the overheads introduced with context switching between userspace and kernel space.

Testbed setup: We set up our experiments on NSF CloudLab [18]. We use two `r650` nodes, each with a 100Gbps NIC. We use Ubuntu 20.04 with kernel version 5.15. We use ***wrk*** [19] — an HTTP load generation tool to send HTTP requests to the echo server application. The echo server application returns the HTTP response to ***wrk***. We focus on the inter-node latency and throughput: the ***wrk*** and echo server applications are placed on different nodes. Latency and throughput are measured for different representative message sizes, ranging from small requests (64KBytes, 1KBytes) to large requests (4KBytes, 8KBytes). We also vary the concurrency and observe how the server handles a large number of connections (up to 200 concurrent connections).

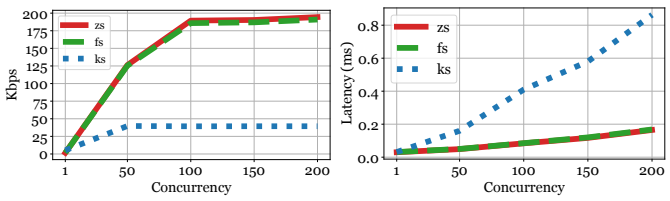

Fig. 5: Throughput (left) and latency (right) (64Byte messages). "zs": Z-stack; "fs": F-stack; "ks": Kernel stack.

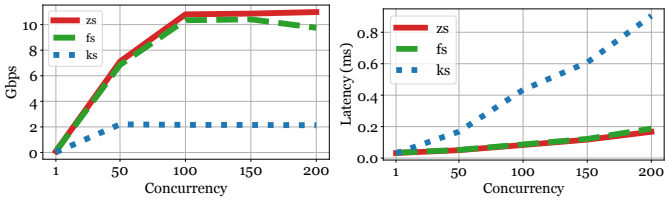

Fig. 6: Throughput (left) and latency (right) with 1KBytes message. "zs": Z-stack; "fs": F-stack; "ks": Kernel stack.

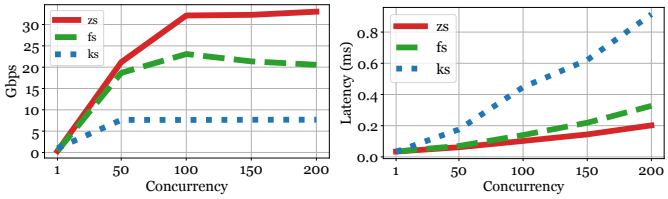

Fig. 7: Throughput (left) and latency (right) with 4KBytes message. "zs": Z-stack; "fs": F-stack; "ks": Kernel stack.

A. Throughput and latency performance

For small messages (64Bytes, 1KBytes), Z-stack has similar throughput and latency performance compared to F-stack (see Fig. 5 and Fig. 6). This is consistent with the observation in [1], in that the data copy is not the dominant networking overhead when message sizes are small. However, even going from 64Bytes to 1KBytes, we still observe that there is a performance difference between Z-stack and F-stack. The data copy incurred in F-stack accounts introduces a small performance loss, both in throughput and latency compared to Z-stack. When compared to the kernel protocol stack, Z-stack's zero-copy design (§III-B) substantially reduces the time spent on protocol processing. With 64Bytes (Fig. 5) and 1KBytes (Fig. 6) message sizes, Z-stack shows a significant throughput improvement and latency reduction (up to 5×), underscoring the efficiency gains from its design optimizations, including polling-based packet processing and bypassing the kernel.

As shown in Fig. 7 and Fig. 8, the performance improvement of Z-stack becomes even more significant with larger message sizes (4KBytes and 8KBytes). Z-stack achieves ∼2× the throughput and reduced latency when compared against F-stack. Against the kernel protocol stack, Z-stack's throughput showed an even more significant improvement, reaching around 4× higher throughput.

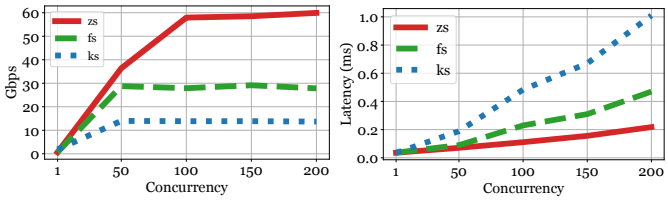

Fig. 8: Throughput (left) and latency (right) with 8KBytes message. "zs": Z-stack; "fs": F-stack; "ks": Kernel stack.

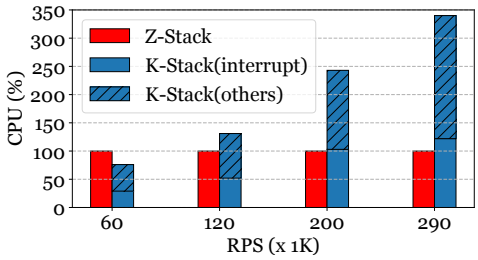

Fig. 9: CPU overhead."K-stack": Kernel stack.

B. CPU Overhead

Z-stack utilizes DPDK's PMD to move packets between the userspace application and the NIC. This requires a designated CPU core confined solely to the PMD, which constantly consumes CPU cycles as the CPU core is engaged in polling activity even in the absence of packets to process. Fig. 9 shows the comparison of the CPU overhead between Z-stack and kernel protocol stack.

Under light loads (up to about 60K requests per second), the CPU core utilization for the kernel protocol stack is 76%, while Z-stack's CPU utilization due to DPDK's PMD is at 100%. However, as we keep increasing the load, the kernel protocol stack shows significant CPU inefficiency, which is caused by its interrupt-based packet handling. For instance, when the load is 290K requests per second, the kernel protocol stack spends 122% CPU core on handling interrupts from the NIC, which leads to a total CPU consumption of 340% (3.4 CPU cores used). On the other hand, Z-stack's CPU utilization is still 100%, which is 3.4× less than the kernel protocol stack. This empirical evidence suggests that in comparison to the conventional kernel protocol stack, which incurs elevated CPU consumption stemming from NIC interrupt handling, Z-stack's polling approach proves to be more efficient for even slightly heavier traffic loads (e.g., even at 120K requests/sec). In addition, we allow multiple user applications to share the Z-stack to amortize the polling overhead under light load.

V. Conclusion

This paper described Z-stack, a high-performance userspace TCP/IP protocol stack that offers true *zero-copy* data movement between the user application and NIC. Z-stack uses DPDK's poll mode to move data between userspace and the NIC, outperforming the kernel-based approach that inevitably incurs context switches and interrupts. This increases throughput and reduces latency up to a factor of 5. Z-stack further adopts a zero-copy socket interface to move data between the protocol stack and the user application. This yields overall a 2× throughput and latency improvement when handling large messages. The zero-copy design of Z-stack makes it suitable for local shared-memory processing, which helps improve data plane performance when distributed applications are organized in a complex chain.

Acknowledgment

We thank the US NSF for their generous support through grants CRI-1823270, CNS-1818971.

References

[1] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 65–77.

[2] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, "Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 780–794.

[3] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 656–671, 2021.

[4] S. Qi, Z. Zeng, L. Monis, and K. K. Ramakrishnan, "Middlenet: A unified, high-performance nfv and middlebox framework with ebpf and dpdk," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 3950–3967, 2023.

[5] V. Jain, H.-T. Chu, S. Qi, C.-A. Lee, H.-C. Chang, C.-Y. Hsieh, K. K. Ramakrishnan, and J.-C. Chen, "L25gc: a low latency 5g core network based on high-performance nfv platforms," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 143–157.

[6] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be general and fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 1–16.

[7] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: a highly scalable user-level TCP stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502.

[8] "F-Stack," https://github.com/F-Stack/f-stack, 2024, [ONLINE].

[9] "Poll Mode Driver," https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2024, [ONLINE].

[10] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[11] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 101–112.

[12] G. Liu, Y. Ren, M. Yurchenko, K. K. Ramakrishnan, and T. Wood, "Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 504–517.

[13] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 195–211.

[14] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, "Tas: Tcp acceleration as an os service," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019.

[15] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, Z. Yang, Z. Chen, W. Lin, Y. Hou, R. Liu, C. Shi, J. Zhu, and J. Wu, "Deploying user-space TCP at cloud scale with LUNA," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 673–687.

[16] "rsocket," https://linux.die.net/man/7/rsocket, 2024, [ONLINE].

[17] "TCP offload engine," https://en.wikipedia.org/wiki/TCP_offload_engine, 2024, [ONLINE].

[18] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14.

[19] "wrk," https://github.com/wg/wrk, 2024, [ONLINE].