SocksDirect: Datacenter Sockets can be Fast and Compatible

Bojie Li[1,2,†], Tianyi Cui[3,†], Zibo Wang[1,2], Wei Bai[1], Lintao Zhang[1]
[1]Microsoft Research [2]University of Science and Technology of China [3]University of Washington

ABSTRACT

Communication intensive applications in hosts with multi-core CPU and high speed networking hardware often put considerable stress on the native socket system in an OS. Existing socket replacements often leave significant performance on the table, as well have limitations on compatibility and isolation.

In this paper, we describe SocksDirect, a user-space high performance socket system. SocksDirect is fully compatible with Linux socket and can be used as a drop-in replacement with no modification to existing applications. To achieve high performance, SocksDirect leverages RDMA and shared memory (SHM) for inter-host and intra-host communication, respectively. To bridge the semantics gap between socket and RDMA / SHM, we optimize for the common cases while maintaining compatibility in general. SocksDirect achieves isolation by employing a trusted monitor daemon to handle control plane operations such as connection establishment and access control. The data plane is peer-to-peer between processes, in which we remove multi-thread synchronization, buffer management, large payload copy and process wakeup overheads in common cases. Experiments show that SocksDirect achieves 7~20x better message throughput and 17~35x better latency than Linux socket, and reduces Nginx HTTP latency to 1/5.5.

CCS CONCEPTS

• **Networks** → **Transport protocols**; • **Software and its engineering** → *Operating systems*;

KEYWORDS

Datacenter, Socket, RDMA, User-space

ACM Reference Format:
Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, Lintao Zhang. 2019. SocksDirect: Datacenter Sockets can be Fast and Compatible. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3341302.3342071

1 INTRODUCTION

Socket API is the most widely used communication primitive in modern applications. It is used universally for communications among processes, containers and hosts. Linux socket can only achieve latency and throughput numbers which are an order of magnitude worse than the raw capability of hardware.

A large body of work aims at improving socket performance. Existing approaches either optimize the kernel networking stack [35, 46, 73], or move TCP/IP stack to user-space [3, 11, 38, 50, 51], or offload transport to RDMA NICs [10, 58]. However, all these solutions have limitations on compatibility and performance. Most of them are not fully compatible with native socket in areas such as process fork, event polling, multiple application socket sharing and intra-host communication. Some of them [38] have isolation problems that do not allow multiple applications to share a NIC. Even in the performance front, there is still much room for improvement. None of existing work can achieve performance close to raw RDMA, because they fail to remove some important overheads such as multi-thread synchronization, buffer management and memory copy. For example, a socket is shared among threads in a process, so, many systems use locking to avoid race conditions.

Realizing these limitations, we design SocksDirect, a user-space socket system to achieve compatibility, isolation and high performance simultaneously.

- **Compatibility**. Applications can use SocksDirect as a drop-in replacement without modifications. It supports both intra-host and inter-host communication, and behaves correctly during process fork and thread creation.
- **Isolation**. First, SocksDirect preserves isolation among applications and containers, which means that no application can eavesdrop or interfere connections among other applications, and a malicious application cannot cause its peers to malfunction. Second, SocksDirect enforces access control policies to prevent unauthorized connections.
- **High Performance**. SocksDirect achieves high throughput and low latency that is comparable to raw RDMA and shared memory. The throughput is scalable with number of CPU cores.

To achieve high performance, SocksDirect fully exploits the capability of modern hardware. It leverages RDMA for inter-host communication and uses *shared memory* (SHM) for intra-host communication. However, translating socket functions to RDMA and SHM operations is non-trivial. Straightforward solutions may violate compatibility or leave much performance on the table. For example, after socket *send()* returns, the application may overwrite the buffer. In contrast, send in RDMA needs to pin and write-protect the buffer. Existing works [10] either provide a zero copy API incompatible with unmodified applications, or manage internal buffers and copy data from application buffer.

To achieve all the three goals simultaneously, we first need to understand how Linux socket provides compatibility and isolation. Linux socket provides a virtual file system (VFS) abstraction to applications. With this abstraction, application developers can do communication like operating files, without digging into network protocol details. This abstraction also provides good isolation among applications that share an address and port space. However, the

† Bojie Li and Tianyi Cui are co-first authors who finish this work during internship at Microsoft Research.

SIGCOMM '19, August 19–23, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342071

VFS abstraction is very complicated and many APIs are inherently not scalable [17, 20, 38].

We find that many commonly used socket operations are actually simple despite the generality and complexity of VFS. Therefore, our design principle is optimizing for the common situations while staying compatible in general.

To speedup data transmission while keeping isolation in connection management, SocksDirect separates control and data plane [57]. We design a *monitor* daemon in each host as the *control plane* to enforce access control policies, manage address/port resources, dispatch new connections, and establish transport channels between communication peers. The *data plane* is handled by a dynamically loaded user-space library LIBSD, which intercepts function calls to Linux *glibc*. LIBSD implements socket APIs in user space and forwards non-socket related APIs to the kernel.

In SocksDirect, data transmission and event polling is handled in a peer-to-peer fashion between processes. We exploit multiple techniques to effectively utilize hardware and improve system efficiency. In general, a socket connection is shared among threads and forked processes. To avoid race conditions in accessing socket metadata and buffers, synchronization is needed. Rather than locking on each operation, we design a token-based approach for sharing that remove synchronization overhead for the common case. To send and receive data from NICs, existing systems allocate buffers for each packet. To remove buffer management overhead, we design a per-connection ring buffer with two copies on both sender and receiver, then leverage RDMA and SHM to synchronize from sender ring buffer to receiver. To achieve zero copy for large messages, we leverage the virtual memory to remap pages.

SocksDirect achieves latency and throughput close to the raw performance achievable from the underlying SHM queue and RDMA. On the latency side, SocksDirect achieves 0.3μs RTT for intra-host socket, 1/35 of Linux and only 0.05μs higher than a bare-metal SHM queue. For inter-host socket, SocksDirect achieves 1.7μs RTT between RDMA hosts, almost the same as raw RDMA write and 1/17 of Linux. On the throughput side, a thread can send 23 M intra-host messages per second (20x of Linux) or 18 M inter-host (15x of Linux, 1.4x of raw RDMA write). The throughput is scalable with number of cores. For large messages, with zero copy, a single connection saturates NIC bandwidth. SocksDirect offers significant speedup for real-world applications. For example, HTTP request latency of Nginx [60] improves by 5.5x.

In summary, we make the following contributions: (1) An analysis of overheads in Linux socket. (2) Design and implementation of SocksDirect, a high performance user space socket system that is compatible with Linux and preserves isolation. (3) Techniques to support fork, token-based connection sharing, allocation-free ring buffer and zero copy that may be useful in many scenarios other than sockets. (4) Evaluations show that SocksDirect achieves performance that is comparable to raw RDMA and SHM queue.

2 BACKGROUND

2.1 Overheads in Linux Socket

Socket is the standard communication primitive among applications, containers and hosts. Modern datacenter networks have microseconds of base latency and tens of Gbps throughput. However, traditional Linux socket is implemented in the OS kernel space with shared data structures, making socket a well-known bottleneck for communication intensive distributed applications [15]. In addition, cloud applications and containers at the same host often communicate with each other, making intra-host socket communication increasingly important in the cloud era. Under stress tests, applications such as Nginx [60], Memcached [32] and Redis [18] consume 50% to 90% CPU time in the kernel, mostly dealing with TCP socket operations [38].

Type	Overhead	Our Solution
Per op	Kernel crossing (syscall)	User space library (§3)
Per op	Socket FD locks for concurrent threads & processes	Token-based socket sharing (§4.1)
Per packet	Transport protocol (TCP/IP)	Use RDMA / SHM (§4.2)
Per packet	Buffer management	New ring buffer (§4.2)
Per packet	I/O multiplexing	Use RDMA / SHM (§4.2)
Per packet	Interrupt handling	Event notification (§4.4)
Per packet	Process wakeup	Event notification (§4.4)
Per byte	Payload copy	Page remapping (§4.3)
Per conn.	Kernel FD allocation	FD remapping table (§4.5)
Per conn.	Locks in TCB management	Distribute to LIBSD (§4.5)
Per conn.	New connection dispatch	Monitor daemon (§4.5)

Table 1: Overheads in Linux socket.

Conceptually, the Linux networking stack consists of the following three layers. First, the Virtual File System (VFS) layer provides socket APIs (e.g., *send* and *epoll*) to applications. A socket connection is a bidirectional, reliable and ordered pipe, identified by an integer *file descriptor* (FD). Second, the transport layer, traditionally TCP/IP, provides I/O multiplexing, congestion control, loss recovery, routing and QoS functions. Third, the Network Interface Card (NIC) layer communicates with the NIC hardware (or the virtual interface for intra-host socket) to send and receive packets.

Among the three layers, it is well known that the VFS layer contributes a large portion of cost [17, 20]. This can be verified by a simple experiment: the latency and throughput of Linux TCP socket between two processes in a host is only *slightly* worse than those of pipe, FIFO and Unix domain socket (Table 2), which bypass the transport and NIC layers.

In the following, we classify socket overheads as Table 1 shows.

2.1.1 *Per Operation Overheads.*

Kernel crossing. Traditionally, socket APIs are implemented in kernel, thus require kernel crossing (system call) for each socket operation. To make it worse, the Kernel Page-Table Isolation (KPTI) patches [24] to protect against Meltdown [47] attacks make kernel crossings 4x expensive, as Table 2 shows. We aim to bypass kernel without compromising security (§3).

Socket FD locks. Multiple threads in a process share socket connections. In addition, after a process fork, both parent and child processes share existing sockets. Sockets can also be passed to another process through Unix domain socket. To protect concurrent operations, Linux kernel acquires a per-socket lock for each socket operation [17, 35, 46]. Table 2 shows that a SHM queue protected by atomic operations has 4x latency and 22% throughput of a lockless queue, even if there is no contention. We aim to remove synchronizations in frequently used socket operations (§4.1).

2.1.2 *Per Packet Overheads.*

Transport protocol (TCP/IP). Traditionally, TCP/IP is the defacto transport protocol in datacenters. TCP/IP protocol processing, congestion control and loss recovery consume CPU on every

Operation	Latency (μs)	Throughput (M op/s)
Inter-core cache migration	0.03	50
Poll 32 empty queues	0.04	24
System call (before KPTI)	0.05	21
Spinlock (no contention)	0.10	10
Allocate and deallocate a buffer	0.13	7.7
Spinlock (contended)	0.20	5
Lockless shared memory queue	0.25	27
Intra-host SocksDirect	0.30	22
System call (after KPTI)	0.20	5.0
Copy one page (4 KiB)	0.40	5.0
Cooperative context switch	0.52	2.0
Map one page (4 KiB)	0.78	1.3
NIC hairpin within a host	0.95	1.0
Atomic shared memory queue	1.0	6.1
Map 32 pages (128 KiB)	1.2	0.8
Open a socket FD	1.6	0.6
One-sided RDMA write	1.6	13
Two-sided RDMA send / recv	1.6	8
Inter-host SocksDirect	1.7	8
Process wakeup	2.8~5.5	0.2~0.4
Linux pipe / FIFO	8	1.2
Unix domain socket in Linux	9	0.9
Intra-host Linux TCP socket	11	0.9
Copy 32 pages (128 KiB)	13	0.08
Inter-host Linux TCP socket	30	0.3

Table 2: Round-trip latency and single-core throughput of operations (testbed settings in §5.1). Message size is 8 bytes if not specified.

sent and received packet. In addition, loss detection, rate-based congestion control and TCP state machine use timers, which is hard to achieve both microsecond-level granularity and low overhead [38]. Fortunately, recent years witnessed large scale deployment of RDMA in many datacenters [34]. RDMA bypasses kernel and offloads transport to the RDMA NIC. For inter-host sockets, we aim to use RDMA given its high throughput, low processing latency, and near zero CPU overhead (§4.2). For intra-host sockets, we aim to completely bypass the transport protocol.

Buffer management. Traditionally, CPU send and receive packets from NICs via *ring buffers*. A ring buffer is composed of a fixed number of fixed-length metadata entries. Each entry points to a buffer that stores the packet payload. To send or receive a packet, a buffer needs to be allocated and deallocated. The cost is shown in Table 2. Further, to ensure MTU-sized packets can be received, each receive buffer should be at least MTU-sized. However, many packets are smaller than MTU [70] in practice, so the internal fragmentation decreases memory utilization and locality. Although modern NICs support LSO and LRO [22] to batch multiple packets in a buffer, we aim to completely remove buffer management overhead.

I/O multiplexing. With legacy NICs, received packets from different connections are often mixed in the ring buffer, so the networking stack needs to sort the packets into corresponding socket buffers.This involves both additional data copy and CPU processing overhead. Modern NICs support Receive Packet Steering [4] that can map a specific connection to a dedicated ring buffer, which is used by high performance socket systems [38, 46, 51]. We aim to leverage a similar feature in RDMA NICs which de-multiplex received packets into Queue Pairs (QPs).

Interrupt handling. The Linux networking stack is separated into process contexts and interrupt contexts because it processes events from both applications and hardware devices. For example, when an application calls `send()`, the networking stack is in process context and sends out the packet (if has available window). When the NIC receives the packet, it sends an interrupt to the CPU, and the networking stack processes the packet in interrupt context. The ACK clocking mechanism [48] in TCP congestion control requires interrupts and timers to be processed timely. The interrupt context is not necessarily on the same core with application, resulting in poor core locality. However, with RDMA NICs, packet processing that require accurate timing is handled by the NIC hardware, so it can eliminate most interrupts on the data plane.

Process wakeup. When a process sends a request and waits for response (e.g., RPC), should it relinquish CPU to other processes that are ready to run? The default answer of Linux is yes, and waking up a sleeping process takes 3 to 5 μs as shown in Table 2. During the round-trip time of an intra-host RPC, two process wake-ups contribute more than half of the RPC round-trip time. For inter-host RPC via RDMA, the round-trip time for small messages (i.e. less than MTU size) is even lower than the wakeup cost. To this end, many distributed systems [29] and user-space network stacks [51] use polling to avoid wakeup cost. However, straightforward polling burns one CPU core for each thread, which is not scalable to many threads. To hide the microsecond scale RPC latency [15], we notice that cooperative context switch via `sched_yield` is much faster than process wakeup (Table 2). We aim to let threads share one CPU core to poll the received messages efficiently (§4.4).

2.1.3 Per Byte Overheads.

Payload copy. In most socket systems, the semantics of `send` and `recv` cause memory copies between application and network stack. The reader may wonder if we can achieve zero copy `send` and `recv` by using RDMA verbs `send` and `receive`. Unfortunately, this straightforward solution may violate application correctness due to semantic differences between RDMA and socket. For example, for RDMA send, the sender cannot reuse the send buffer just after send is called and it is required to receive the completion from the driver. However, for some web applications, the send buffer is immediately reused after socket send API is called since the semantics of the socket send guarantees that the data to send has already been copied to the kernel. We aim to allow zero copy for large transfers in *standard* socket applications.

2.1.4 Per Connection Overheads.

Kernel FD allocation. In Linux, each socket connection is essentially a file in VFS, so an integer file descriptor (FD) and an *inode* need to be allocated. The challenge for a user-space socket is that there are many APIs (e.g. open, close and epoll) that support both socket and non-socket FDs (e.g. files and devices), so we must distinguish socket FDs from others. Linux compatible sockets in user space [10, 51] typically open a file in kernel to obtain a dummy FD for each socket, so they still need kernel FD allocation. LOS [36] partitions the FD space to user and kernel portions, but violates the property that Linux allocates the smallest available FD. However, many applications such as Redis [8] and Memcached [32] rely on this property. We aim to bypass kernel in socket FD allocation while keeping compatibility (§4.5).

Locks in TCB management. During connection setup, Linux acquires several global locks to allocate the TCP control block (TCB).

	FastSocket	MegaPipe / StackMap	IX	Arrakis	SandStorm / mTCP	LibVMA	OpenOnload	Rsocket / SDP	FreeFlow	SocksDirect
Category	Kernel optimization		User-space TCP/IP stack					Offloading to RDMA NICs		
Compatibility										
Transparent to existing applications	✓		✓	✓		✓	✓	✓	✓	✓
`epoll` (Nginx, Memcached etc.)	✓	✓	✓	✓	✓	✓	✓			✓
Compatible with regular TCP peers	✓	✓	✓	✓	✓	✓	✓			✓
Intra-host communication	✓	✓		✓					✓	✓
Multiple applications listen a port	✓	✓						✓	✓	✓
Full fork support	✓	✓					✓			✓
Container live migration	✓	✓								✓
Isolation										
Access control policy	Kernel		Kernel	Kernel				Kernel	Daemon	Daemon
Isolation among containers (VMs)	✓		✓	✓				✓	✓	✓
QoS (performance isolation)	Kernel	Kernel	Kernel	NIC	NIC	NIC	NIC	NIC	Daemon	NIC
Removed Performance Overheads										
Kernel crossing on data plane			Batched	✓	Batched	✓	✓	✓	✓	<16KB msg
Socket FD locks										✓
Transport protocol								✓	✓	✓
Buffer management										✓
I/O multiplexing & Interrupt handling		Improved	✓	✓	Improved	✓	✓	✓	✓	✓
Process wakeup										✓
Payload copy		✓			✓					≥16KB msg
Kernel FD allocation		✓			✓					✓
TCB mgmt & Connection dispatch	✓	✓	✓	✓	✓					✓

Table 3: Comparison of high performance socket systems.

Recent works such as MegaPipe [35] and FastSocket [46] reduce lock contention by partitioning the global tables, but as Table 2 shows, non-contended spinlocks are still expensive. We distribute the work to the user-space library LIBSD in each process (§4.5).

New connection dispatch. Multiple processes and threads may listen on the same port to accept incoming connections. In Linux, cores handling `accept` calls contend on the queue of incoming connections. We notice that delegation is faster than locking [62] and use a monitor daemon to dispatch new connections (§4.5).

2.2 High Performance Socket Systems

Table 3 summarizes high performance socket systems.

Kernel network stack optimization: The first line of work optimizes the kernel TCP/IP stack. FastSocket [46], Affinity-Accept [56], FlexSC [65] and zero-copy socket [19, 25, 69] achieve good compatibility and isolation. MegaPipe [35] and StackMap [73] propose new APIs to achieve zero copy and improve I/O multiplexing, at the cost of requiring application modifications. However, the bulk of kernel overheads are still there. The challenge for supporting zero copy is the socket semantics (§2.1.3).

User-space TCP/IP stack: The second line of work completely bypasses kernel TCP/IP stack and implements TCP/IP in user space. In this category, IX [16] and Arrakis [57] are new OS architectures that use virtualization to ensure security and isolation. IX leverages LwIP [30] to implement TCP/IP in user space while using kernel to forward every packet for performance isolation and QoS. In contrast, Arrakis offloads QoS to NIC, therefore bypasses kernel for data plane. They use the NIC to forward packets between applications in a same host. As shown in Table 2, the hairpin latency from CPU to NIC is still much higher than inter-core cache migration delay. The throughput is also bounded by Memory-Mapped I/O (MMIO) doorbell latency and PCIe bandwidth [44, 54].

Apart from these new OS architectures, many user space sockets use high performance packet I/O frameworks on Linux, e.g., Netmap [61], Intel DPDK [37] and PF_RING [6], in order to directly access NIC queues in user space. SandStorm [50], mTCP [38], Seastar [11] and F-Stack [3] propose new APIs and thus need to modify applications. Most of the API changes aim to support zero copy, and the standard API still copies data. LibVMA [51], OpenOnload [59], DBL [5], LOS [36], and TAS [42].

The performance of user-space TCP/IP stacks is much better than Linux, but still lower than RDMA and SHM. Another important problem is that none of existing solutions supports sharing socket among threads and processes, thus causing compatibility problems in fork and container live migration, and multi-thread locking overhead as well. First, in LibVMA and RSocket, after fork, the child process either takes the ownership of all existing sockets away from the parent process, or have no access to any of the existing sockets. There is no means to control the ownership of each socket independently. However, many web services [1, 2, 7, 12, 60] and key-value stores [32] have a master process to accept a connection on a listen socket, then it may fork off a child process to handle the request, where the child needs to own the socket. At the same time, the parent process still needs to accept new connections via the listen socket. This makes such web services fail to work. A more tricky case is that parent and child processes may concurrently write to logging servers. Second, multi-threading is common in applications. Either the application takes the risk of race conditions, or a socket FD lock must be taken per operation. Locking guarantees correctness, but hurts performance.

Offloading transport to NICs: The third line of research uses hardware offloading. TCP Offload Engines (TOE) [27] offload part or full TCP/IP stack to the NIC, but they only achieve commercial success in specialized areas (e.g. iSCSI HBA [72]) and stateless offloads (e.g. checksum, flow steering and LSO/LRO [22]). The story of *stateful* offloads is different in recent years because of hardware trends and application demands in datacenters [33]. As a result, RDMA [14] becomes widely available in production datacenters [34]. RDMA provides two types of abstractions: one-sided verbs to read and write remote SHM, and two-sided verbs that resemble socket send and receive [40]. RDMA uses hardware offloading to provide ultra low latency and near zero CPU overhead.

To enable socket applications to use RDMA, RSocket [10], SDP [58], and UNH EXS [63] translate socket operations to two-sided RDMA verbs. They have similar designs, and RSocket is the most actively

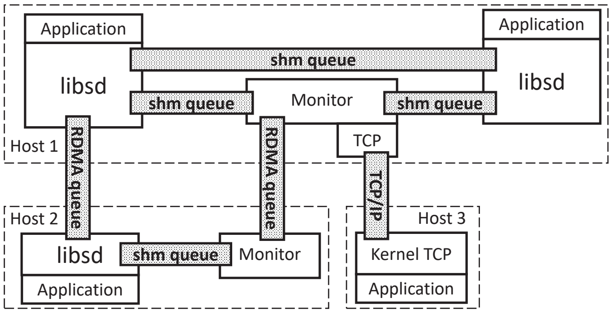


Figure 1: Architecture of SocksDirect. Host 1 and 2 are RDMA capable, while host 3 is RDMA incapable.

maintained one. FreeFlow [43] virtualizes an RDMA NIC for container overlay network, which uses SHM for intra-host and RDMA for inter-host communication. FreeFlow uses RSocket to translate socket to RDMA.

However, these works have limitations due to abstraction mismatch between socket functions and RDMA verbs. On the compatibility side, first, they lack support for several important APIs, e.g. *epoll*, so it is not compatible with many applications including Nginx, Memcached, Redis, etc. This is because RDMA only provides transport functions, while *epoll* is a file abstraction integrated with OS event notification. Second, RDMA QP does not support *fork* and container live migration [43], so RSocket has the same problems. Third, because RSocket uses RDMA as the wire protocol, it cannot connect to regular TCP/IP peers. This is a deployment challenge because it is unlikely that all the hosts in a datacenter can be switched to RSocket simultaneously. We aim to transparently detect whether the remote side supports Rsocket, and fall back to TCP/IP if not. On the performance side, they fail to remove payload copy, socket FD locks, buffer management, process wakeup and the per connection overheads. For example, RSocket allocates buffers and copies payload on both send and receive sides. Similar to Arrakis, RSocket uses the NIC for intra-host communication, thus incurring a performance bottleneck.

3 ARCHITECTURE OVERVIEW

To simplify deployment and development [28], as well remove kernel crossing overhead, we implement SocksDirect in user space rather than kernel space. To use SocksDirect, an application just needs to load a user-space library libsd by setting the `LD_PRELOAD` environment variable. libsd intercepts all Linux APIs in standard C library that are related to file descriptor (FD) operations. It uses a *FD remapping table* to distinguish socket FDs from kernel FDs (e.g. file and devices), implements socket functions in user space and forwards others to the kernel. However, from a security point of view, because libsd resides in the application address space, we cannot trust its behavior. For example, a malicious application may directly inject an arbitrary message into RDMA QPs and bypass the checks in libsd library. In addition, TCP port is a global resource that needs logically centralized allocation [43, 46]. Therefore, we need a trusted component outside libsd to enforce access control and manage global resources.

To this end, we design a *monitor* daemon at each host to coordinate control plane operations, e.g., connection creation. To ensure isolation, we consider all applications and the monitor as a shared-nothing distributed system, and use message passing as the exclusive communication mechanism. The monitor is a single thread that polls control messages from all applications. In each host, all the applications loading libsd must establish a SHM queue with the host's monitor daemon, forming the control plane. On the data plane, applications build peer-to-peer queues to communicate directly, thus relieving the burden of the monitor daemon. Figure 1 shows the architecture of SocksDirect.

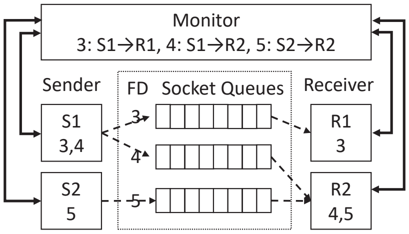


Figure 2: Token-based socket sharing with two sender and two receiver threads. Dashed arrows indicate the active sender and receiver of each socket. Each thread tracks its active sockets and communicates with the monitor via an exclusive queue.

To achieve low latency and high throughput, SocksDirect uses SHM for intra-host communication and RDMA for inter-host communication. Each socket connection is mapped to a SHM or RDMA QP. A SHM or RDMA QP is marked by a unique token, so other non-privileged processes cannot access it. A socket send operation is translated to a SHM or RDMA write operation to the socket buffer on the peer. We only use one-sided RDMA write verb rather than two-sided RDMA send/recv verbs.

For *intra-host* communication, the communication initiator first sends a request to the local monitor, then the monitor establishes a SHM queue between the two applications. Afterwards the two applications can communicate directly.

For *inter-host* communication, the monitors of two hosts are both involved. When an application connects to a remote host, its local monitor establishes a regular TCP connection and detects whether the remote host supports SocksDirect and RDMA. If both conditions are satisfied, it establishes an RDMA queue between the two monitors, so that future connections between the two hosts can be created faster. The monitor at the remote side dispatches the connection to the target and helps the two applications establish an RDMA queue, as between host 1 and 2 in Fig. 1. If the remote host does not support SocksDirect or RDMA, it keeps using the TCP connection, as between host 1 and 3 in Figure 1. The detailed connection management protocol is presented in §4.5.

To ensure thread safety and support fork and container live migration, we design a token-based socket sharing mechanism that optimize for the common cases (§4.1). To remove buffer management overheads, we design a per-socket ring buffer that only requires close to one RDMA write operation per inter-host message and one cache migration per intra-host message (§4.2). Zero copy and event notification are discussed in §4.3 and §4.4.

4 DESIGN

4.1 Token-based Socket Sharing

Most socket systems acquire a per-FD lock to enable threads and processes share a socket (Sec. 2.1.1). Previous work [17, 21] suggests that many socket operations are not commutable and synchronizations cannot always be avoided. We leverage the fact that

SHM message passing is much cheaper than locking [62], and use message passing as the exclusive synchronization mechanism.

Logically, a socket is composed of two FIFO *queues* in opposite directions, each with multiple concurrent senders and receivers. Our aim is to maximize the common-case performance while preserving the FIFO semantics. We make two observations: First, fork and thread creations are infrequent in high performance applications because of their high cost. Second, it is uncommon that several processes send or receive concurrently from a shared socket, because the byte-stream semantics of socket makes it hard to avoid receiving partial messages. The common case is that the application implicitly migrates a socket from one process to another, e.g. offload a transaction from master to a worker process.

Driven by the above two observations, our solution is to have a *send token* and a *receive token* per *socket queue* (one direction of a socket). Each token is held by an *active thread*, which has the permission to send or receive. So there is only one active sender thread and one active receiver thread at any time. The socket queue is shared among the threads and processes, which allows concurrent access from one sender and one receiver without locking (details will be discussed in Sec. 4.2). When another thread wants to send or receive, it should request to *take over* the token.

The details for each type of operations are as follows:

4.1.1 Send/Recv Operation.

When a thread does not have the send token but wants to send through the socket, the inactive thread needs to *take over* the token. If we create a direct communication channel between the inactive and active threads, there will either be peer-to-peer queues with number quadratic to the number of threads, or a shared queue with locking. To avoid both overheads, we use the monitor as a proxy during the *take over* process. Since take over is an infrequent operation, the monitor would unlikely be a bottleneck. This message passing design also has the benefit that sender processes can be on different hosts, which will be useful in container live migration.

The take over process is given as follows: The inactive sender puts a *take over* command into the SHM queue to the monitor. The monitor polls commands from SHM queues and adds the sender to a *waiting list*. If the waiting list is empty, the monitor notifies the active sender to return the send token to the monitor. The monitor grants the token to the first inactive sender in the waiting list. After receiving the token, the inactive sender becomes active and able to send. This mechanism is deadlock-free, because either a sender or the monitor holds the send token. It is also starvation-free, because each sender can appear in the waiting list at most once and served in FIFO order. The take over process on the receiver side is similar.

The take-over process takes 0.6 μs, so, if multiple processes concurrently send through one socket, the aggregated throughput may drop to 1.6 Mop/s. However, if we simply use locking, the normal case throughput would be 5 Mop/s, far lower than 27 Mop/s, the normal case throughput of token-based socket sharing.

4.1.2 Fork, Exec and Thread Creation.

Socket data sharing. The main challenge is to share socket metadata, buffers and underlying transports after `fork` and `exec`. The memory space becomes copy-on-write after fork and is wiped after exec, but the socket FDs should still be usable. We use SHM to store the socket metadata and buffers, so after fork, the data is still shared. After fork, because a socket created by the child process should not be visible by the parent, the child creates a new piece of SHM to store metadata and buffers of new sockets.

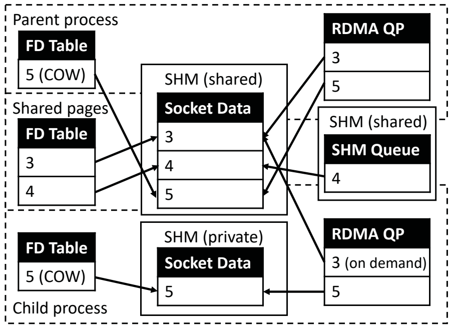


Figure 3: Memory layout after fork. FDs 3 and 4 are created before fork and thus shared. After fork, parent and child processes each creates a new FD 5, which is copied on write in FD table. Socket metadata and buffers of FDs 3 and 4 are in SHM and thus shared. The child process creates a new SHM to store socket metadata and buffers of FD 5, which will be shared to its child when it forks again.

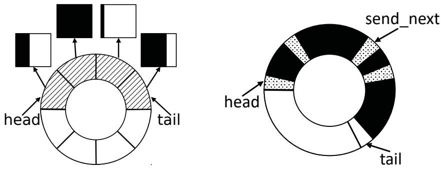
Now we need to share the underlying transports that are connected to peers. SHM transport does not need special care, because a SHM created before fork/exec is still shared after fork/exec. However, RDMA has problem with fork/exec because the DMA memory regions are not in SHM. They become copy-on-write after fork, while the NIC still DMAs from the original physical pages, so the child process cannot use existing RDMA resources. After exec, the entire RDMA context is wiped out. Our solution is to let the child process re-initialize RDMA resources, e.g., Protection Domain (PD), Memory Region (MR), etc., after fork/exec. When a child process uses a socket created before fork, it asks the monitor to re-establish an RDMA QP with the remote endpoint. So, the remote may see two or more QPs for one socket, but they link to the unique copy of socket metadata and buffer. As we will see in §4.2, we only use RDMA write verb, so using any of the QPs is equivalent. Figure 3 shows an example of fork.

FD space sharing. Different from socket data, the FD space becomes copy-on-write after fork: existing FDs are shared, but new FDs are exclusively owned by the creater process. So the FD remapping table resides in heap memory, which becomes copy-on-write after fork. To recover the FD remapping table after `exec`, it is copied to a SHM before `exec`. The SHM is attached to the new process during LIBSD initialization.

Security. Security is a concern because a malicious process may disguise itself as the child process of a privileged parent process. To identify the parent and child relationship in the monitor, when an application calls `fork`, `clone` or `pthread_create`, LIBSD first generates a secret for pairing and sends it to the monitor, then invokes the original function in *libc*. After fork, the child process creates a new SHM queue to the monitor and sends the secret. The monitor can thus pair the child process with the parent.

Monitor action. Upon fork, exec or thread creation, each existing socket needs to add a sender to the sending direction and a receiver to the receiving direction. The parent process inherits the token, so the child process is always inactive.

4.1.3 Container Live Migration.



(a) Traditional. (b) SocksDirect.

Figure 4: Ring buffer data structures for an intra-host socket. Shaded part is packet metadata, and black part is payload.

Migration of remaining data in socket queues. Because libsd runs in the same memory space with the application, it is migrated to the new host together with the application. The memory states include the socket queues, so the in-flight (sent but not received) data will not be lost.

Migration of monitor states. The monitor keeps track of listening socket information, active thread and waiting list of each connection, and SHM secrets. During container migration, monitor states are also migrated to the new monitor.

Establish new communication channels. After migration, all communication channels become obsolete because SHM is local on a host and RDMA does not support live migration [43, 75]. First, the migrated container on the new host needs to establish a connection to the local monitor. The local monitor directs the following process. An intra-host connection between two containers may become inter-host, so libsd creates an RDMA QP in this case. An inter-host connection between two containers may become intra-host, and libsd creates a SHM queue. Finally, libsd re-establishes remaining RDMA and SHM connections.

4.2 Per-socket Ring Buffer

Traditionally, the network stack send and receive packets from NICs using a ring buffer. As Figure 4a shows, it leads to buffer management overhead and internal fragmentation. Traditional NICs support a limited number of ring buffers, so multiple connections may share one ring buffer, and the networking stack needs to scatter messages from the ring buffer to multiple socket receive buffers. Fortunately, both RDMA write verb and SHM write operation allow the sender to specify the address on the receiver. So, we eliminate the traditional ring buffer, and use RDMA and SHM to send the per-socket send buffer directly. In addition, traditional send buffer is a linked list of messages and therefore need buffer allocation. To avoid this overhead, we organize the socket buffer as a ring buffer and store messages back-to-back, as Figure 4b shows. The sender determines the receive buffer address (i.e. *tail* pointer), then use RDMA write verb to write the message to the tail pointer in remote-side memory. During transmission, the receiver CPU is fully bypassed. When the receiver application calls `recv`, data is dequeued from the *head* pointer. The process is similar for SHM because both SHM and RDMA support write primitives.

To tell whether the ring buffer is full, the sender maintains a *queue credits* count, indicating the number of free bytes in ring buffer. When sender enqueues a packet, it consumes credits. When receiver dequeues a message, it increments a counter locally, and writes a *credit return flag* in sender's memory once the counter exceeds half the size of ring buffer. The sender regains queue credits upon detecting the flag.

Two copies of ring buffers on send and receive sides. The above mechanism still incurs buffer management on the send side because the sender needs to construct RDMA message in a buffer. Second, RDMA QP does not support container live migration. Third, we aim to batch small messages to improve throughput. To this end, we maintain a copy of ring buffers on both send and receive sides. The sender side writes its local ring buffer, and invokes RDMA to synchronize the sender to the receiver. We use an RDMA Reliable Connection (RC) QP for each ring buffer, and maintain a counter of in-flight RDMA messages. If the counter does not exceed the threshold, an RDMA message is sent for every socket `send` operation. Otherwise, the message is not sent, and *send_next* marks the first unsent message. Upon completion of an RDMA write, we send a message containing all unsent changes (*send_next* to *tail* in Figure 4b). This adaptive batching mechanism minimizes latency on idle links and maximizes throughput on busy links. For SHM, we have only one copy of ring buffer shared by two processes, and synchronization is done by cache coherence hardware.

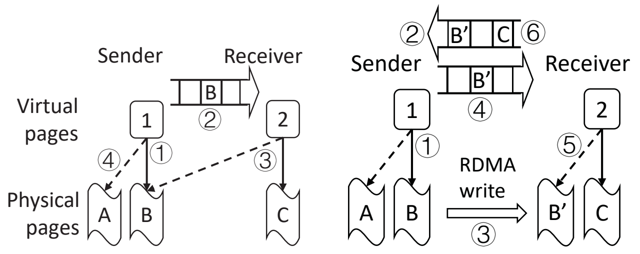
Consistency between payload and metadata. For SHM, X86 processors provide total store ordering [26, 64], which implies that two writes are observed by other cores in the same order as they were written. Because the receiver polls the metadata to find next message, the sender writes payload before metadata, so the receiver would not read the message before the payload is written. In addition, an 8-byte `MOV` instruction is atomic. The metadata of a message is 8 bytes, so the fields in metadata are consistent.

However, RDMA does not ensure write ordering within a message [14], so, we do need to make sure a message is completely arrived. Although message write ordering is observed in RDMA NICs that use go-back-0 or go-back-N loss recovery [29], it is not true for more advanced NICs with selective retransmission [48, 53]. In libsd, the sender uses *RDMA write with immediate* verb to generate completions on receiver. The receiver polls RDMA *completion queue* rather than the ring buffer. RDMA ensures cache consistency on receiver, and the completion message is guaranteed to be delivered after writing the data to libsd ring buffer [14].

Amortize polling overhead. Polling ring buffers wastes CPU cycles of the receiver when a socket is not used frequently. We amortize polling overhead using two techniques. First, for RDMA queues, we leverage the RDMA NIC to multiplex event notifications into a single queue. Each thread uses a *shared completion queue* for all RDMA QPs, so it only needs to poll one queue rather than all per-socket queues. Second, each queue can switch between *polling* and *interrupt* modes. The queue to the monitor is always in polling mode. Receiver of each queue maintains a counter of consecutive empty polls. When it exceeds a threshold, the receiver sends a message to sender notifying that the queue is entering interrupt mode, and stops polling after a short period. When sender writes to a queue in interrupt mode, it also notifies the monitor and the monitor will signal the receiver to resume polling.

4.3 Zero Copy

As Sec. 2.1.3 discussed, the main challenge for zero copy is to maintain the semantics of socket API. Fortunately, virtual memory provides a layer of indirection, and many works leverage this *page remapping* technique. Linux zero copy socket [25] only support send side, by setting the data pages as copy-on-write. However,



(a) Intra-host SHM. 1) Get physical pages and set copy-on-write; 2) send page addresses via SHM; 3) map received pages; 4) (optional) remap when sender write / memcpy / recv.

(b) Inter-host RDMA. 1) Get physical pages and set copy-on-write; 2) get free pages from pool; 3) send data via RDMA; 4) send page addresses via RDMA; 5) map received pages; 6) return unmapped pages.

Figure 5: The procedure to send a page with zero copy.

many applications overwrite the send buffer frequently, so the copy-on-write mechanism simply delays the copy from `send` time to first overwrite time. To achieve zero copy receive, 20 years ago, BSD [69] and Solaris [19] remap the virtual page of application buffer to the physical page of system buffer. However, as Table 2 shows, on modern CPUs, mapping one page has even higher cost than copying it, because of kernel crossing and TLB flush costs. Recently, many high performance TCP/IP stacks [35, 73] and socket-to-RDMA libraries [10, 58] provide both standard socket API and an alternative zero-copy API, but none of them achieves zero copy for the standard API. Further, no existing works support zero copy for intra-host sockets.

To enable zero-copy, we add a kernel module to expose several kernel functions related to page remapping. To amortize page remapping cost, we only use zero copy for `send` or `recv` with at least 16 KiB payload size. Smaller messages are copied instead.

Page alignment. Page remapping only works when the send and receive addresses are page aligned. We intercept `malloc` functions and allocate 4 KiB aligned addresses for multiple-of-4K sizes, so most buffers will align to page boundary, without wasting memory for small allocations. If the size of sent message is not a multiple of 4 KiB, the last chunk of data is copied on `send` and `recv`. However, some applications send and receive from the middle of buffer, e.g., HTTP body in Nginx follows HTTP headers and is not aligned. To optimize for the case that applications do not read the received data and simply send it to another connection, libsd can unmap the address range and record the page offset.

Minimize copy-on-write. When sender overwrites the buffer after `send`, existing designs use copy-on-write. Our observation is that most applications do not write send buffers byte-by-byte. Instead, they overwrite entire pages of the send buffer via `recv` or `memcpy`, so it is unnecessary to copy original data of the page. For `memcpy`, we invoke the kernel to remap new pages and disables copy-on-write, then do the actual copy. For `recv`, the old page mappings are replaced by the received pages.

Page allocation overhead. Page remapping requires the kernel to allocate and free pages for each zero copy `send` and `recv`. Page allocation in kernel uses a global lock, which is inefficient. libsd manages a pool of free pages in each process locally. libsd also tracks the origin of received zero-copy pages. When a page is unmapped, if it is from another process, libsd return the pages to the owner through a message.

Send page addresses securely via SHM. For intra-host socket, we send the physical page addresses in a message in user-space queues, as step 2 in Figure 5a. We must prevent unsolicited remapping of arbitrary pages. To this end, libsd invokes a modified NIC driver to get obfuscated physical page addresses of the send buffer and send the address to receiver via shared memory queue. On the receiving side, libsd invokes the kernel to remap the obfuscated physical pages to the application-provided receive buffer.

Zero Copy under RDMA. libsd initializes a pinned page pool on receiver and send the physical addresses of the pages to the sender. The sender manages the pool. On sender, libsd allocates pages from the remote receiver page pool to determine the remote address of RDMA write, as step 2 in Figure 5b. On receiver, when `recv` is called, libsd invokes the NIC driver to map pages in the pool to application buffer virtual address. After the remapped pages are freed (e.g. overwritten by another `recv`), libsd returns them to the pool manager in sender (step 6).

To send or receive pages via RDMA, they need to be pinned at the first time of use. When libsd gets the physical address of a page, the kernel pins the pages if it is not pinned. Because most applications maintain a pool of send and receive buffers, the pages are recycled frequently. So, after a while, most pages in send and receive buffers become pinned. Other pages of the application remain unpinned. If the OS runs out of memory, libsd unpins pages.

Hints from applications. The major reason for copy-on-write on both sender and receiver is that the sender may read the data in send buffer after `send()`. However, many applications do not do this. If applications can be modified to give some hints to libsd, we would like to add an option `O_DISCARD_SEND_BUF` to `send()` API. So, libsd can map new pages for the send buffer, and the sender do not need copy-on-write. Further, the ownership of pages in send buffer can be transferred to the receiver, and the receiver can read and write the pages directly.

4.4 Event Notification

Challenge 1: multiplex events between kernel and libsd. The application polls events from both sockets and other *kernel FDs* handled by Linux kernel. A naive way to poll kernel events is to invoke the syscall (e.g. *epoll_wait*) every time, which incurs high overhead because event polling is a frequent operation on virtually every send and receive. Differently, libsd creates a per-process *epoll thread* which invokes *epoll_wait* syscall to poll kernel events. Whenever epoll thread receives a kernel event, application threads will report the event together with user-space socket events.

Challenge 2: interrupt busy processes. The socket take-over mechanism (§4.1.1) requires a process to respond monitor requests promtly. However, processes may execute application code without calling libsd for a long time. To address this issue, we design a *signal* mechanism analogous to interrupts in OS. Event initiators first poll the receive queue for a period of time for ACK. If no reply, it sends a Linux *signal* to the receptor and wake up the process.

The signal handler, registered by libsd, first determines whether the process is executing application or libsd code. libsd sets and clears a flag at entry and exit of the library. If signal handler finds that the process is in libsd, it does nothing and libsd will process the event before returning control to the application. Otherwise,

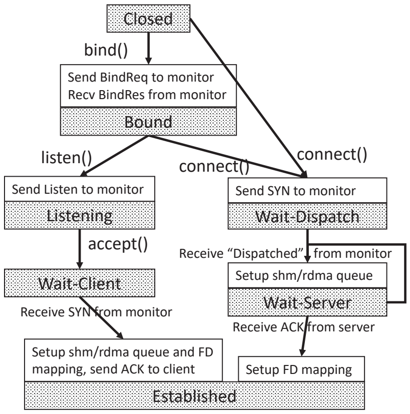


Figure 6: LIBSD maintains a TCP-like state machine for each connection. This figure shows connection setup states.

the signal handler immediately processes messages from the emergency queue to the monitor.

Challenge 3: enable multiple threads time-share a core. For blocking operations (e.g., blocking recv, connect and epoll_wait), LIBSD first polls the ring buffers once. If the operation is not completed, rather than polling infinitely, LIBSD calls *sched_yield* to yield to other processes on the same core. If LIBSD continues to yield for a certain number of rounds, it will put itself into sleep. Before sleeping, it sends a message to the monitor and all peers, so they can wake it up later through a message.

4.5 Connection Management

4.5.1 FD Remapping Table.

Socket FDs and other FDs (e.g. files) share a namespace and Linux always allocates the lowest available FD. To preserve this semantics without allocating dummy FDs in the kernel, LIBSD intercepts all FD-related Linux APIs and maintains an *FD remapping table* to map each application FD to a user-space socket object or a kernel FD. When an FD is closed, LIBSD put it to an *FD recycle pool.* Upon FD allocation, LIBSD first tries to obtain an FD from the pool. If the pool is empty, it allocates a new FD by incrementing an *FD allocation counter.* The FD recycle pool and allocation counter are shared among all threads in a process, protected by atomics.

4.5.2 Connection Establishment.

Bind. As Figure 6 shows, after socket creation, the application calls `bind` to allocate address and port. Because addresses and ports are global resources with permission protection, the allocation is coordinated by the monitor. As shown in Figure 6, LIBSD sends the request to monitor. To hide the latency of contacting monitor, LIBSD returns to application immediately if the bind request would not fail, e.g., when port is not specified for client-side sockets.

Listen. When a server application is ready to accept connections from clients, it calls `listen` and notifies the monitor. To dispatch new connections, the monitor maintains a list of listening processes on each address and port.

Connect. A client application calls `connect` and sends a SYN command to monitor via SHM queue. Now the monitor needs to dispatch the new connection to a listening application. In Linux, new connection requests are queued in a *backlog* in the kernel. Every time the server application calls `accept`, it accesses the kernel to dequeue from the backlog, which requires synchronization and adds latency. In contrast, we maintain a per-listener backlog for every thread that listens on the socket. The monitor distributes SYN to a listener thread in round-robin manner.

Dispatching connection to listeners may lead to starvation when a listener does not accept new connections. We devise a *work stealing* approach. When a listener invokes `accept` while the backlog is empty, it requests the monitor to steal from others' backlog. To avoid contention between a listener and monitor, the monitor sends a request to the listener to steal from the backlog.

Establish a peer-to-peer queue. The first time a client and a server application communicates, the server monitor helps them establish a direct connection. For intra-host, the monitor allocates a SHM queue and sends the SHM key to both client and server applications. For inter-host, the client and server monitors establish a new RDMA QP, and send the local and remote keys to the corresponding applications. To reduce latency, the peer-to-peer queue is established by monitor(s) when the SYN command is distributed into a listener's backlog. However, if the SYN is stolen by another listener, a new queue needs to be established between client and the new listener, as shown in the Wait-Server state of Figure 6.

Finalize connection setup. After the server sets up peer-to-peer queue, as the left side of Figure 6 shows, the server application sends an ACK to client. Similar to TCP handshake, the server application can send data to the queue after sending the ACK. When the client receives the ACK, as shown on the right side of Figure 6, it sets up the FD mapping and can start sending data.

4.5.3 Compatibility with Regular TCP/IP Peers.

To be compatible with peers that do not support SocksDirect and RDMA, we need to detect SocksDirect capability and fall back to TCP/IP. However, Linux does not support adding special options to TCP SYN and ACK packets. Using another port (e.g. LibVMA [51]) is also unreliable due to middleboxes and network reordering. To this end, we first use kernel raw socket to directly send SYN and ACK packets with a special option, then fall back to kernel TCP/IP socket if the special option is not present.

On client side, the monitor sends a TCP SYN packet with a special option over the network. If the peer is SocksDirect capable, its monitor would receive the special SYN and know the client is SocksDirect capable. The server then responds SYN+ACK with special option, including credentials to setup an RDMA connection, so that the two monitors can communicate through RDMA afterwards. If either the client or the server finds out that the peer is a regular TCP/IP host, it creates an established TCP connection in kernel using TCP connection repair [23]. Then the monitor sends the kernel FD to the application via Unix domain socket, and LIBSD can use the kernel FD for future socket operations.

A tricky thing is that received packets are delivered to both the raw socket and kernel networking stack, and the kernel will respond with RST because such connection does not exist. To filter out the RST packets, the monitor installs *iptables* rules.

4.5.4 Connection Teardown.

LIBSD deletes an FD from remapping table when the application calls `close`. However, the socket data may be still useful because the FD may be shared to other processes, and there may be unsent data in the buffer. We track a reference count for each socket, which is incremented on fork and decremented on close. To push

Type	Overhead	SocksDirect	LibVMA	RSocket	Linux
Per op	Total (not thread safe)	53	56	71	413
Per op	Total (thread safe)	53	177	209	413
Per op	C library shim	15	10	10	12
Per op	Kernel crossing (syscall)	N/A	N/A	N/A	205
Per op	Socket FD locking	N/A	121	138	160
Per packet	Total (inter-host)	850	2200	1700	15000
Per packet	Total (intra-host)	150	1300	1000	5800
Per packet	Buffer management	50	320	370	430
Per packet	Transport protocol	N/A	260	N/A	360
Per packet	Packet processing	N/A	200	N/A	500
Per packet	NIC doorbell and DMA	600	900	900	2100
Per packet	NIC processing & wire	200			
Per packet	Handling NIC interrupt	N/A	N/A	N/A	4000
Per packet	Process wakeup	N/A	N/A	N/A	5000
Per kbyte	Total (inter-host)	173	540	239	365
Per kbyte	Total (intra-host)	13	381	212	160
Per kbyte	Wire transfer	160			
Per conn.	Total (inter-host)	47000	18000	77000	47000
Per conn.	Total (intra-host)	700	3800	33000	14700
Per conn.	Initial TCP handshake	16000	16000	47000	N/A
Per conn.	Monitor processing	180	N/A	N/A	N/A
Per conn.	RDMA QP creation	30000	N/A	30000	N/A

Table 4: Latency breakdown in SocksDirect and other systems. Per-operation latency is measured with `fcntl()`, per-packet and per-kbyte mean the latency from `send()` to `recv()`, and per-connection latency means connection setup. Numbers are rough estimations in nanoseconds.

out unsent data, we require a handshake between peers, similar to TCP close. Because socket is bidirectional, `close` is equivalent to `shutdown` on both send and receive directions. When application shuts down one direction of a connection, it sends a *shutdown message* to the peer. The peer responds with a shutdown message. A socket is deleted when libsd receives shutdown messages in both directions. If an application fails, libsd in the peers will generate `SIGHUP`. Although RDMA does not have clear failure semantics, SocksDirect can handle failures correctly because the ring buffer has a copy on both send and receive sides.

5 EVALUATION

We implement SocksDirect in three components: a user-space library libsd, a monitor daemon, and a kernel module to support zero copy. They have 17K lines of C++ code. We aim to show that SocksDirect uses RDMA and SHM efficiently, scalable with multi-cores, and can speedup unmodified applications.

5.1 Methodology

We evaluate SocksDirect on servers with two Xeon E5-2698 v3 CPUs, 256 GiB memory and a Mellanox ConnectX-4 NIC. The servers are interconnected with an Arista 7060CX-32S 100G switch [13]. We use Ubuntu 16.04 with Linux 4.15, RoCEv2 for RDMA and poll completion queue every 64 messages. Each thread is pinned on a CPU core. We run tests for enough warm-up rounds before collecting data. For latency, we build a ping-pong application and report the mean round-trip time, with error bars representing 1% and 99% percentile. We compare with Linux, raw RDMA write verb, Rsocket [10], and LibVMA [51], a user-space TCP/IP stack optimized for Mellanox NICs. We did not evaluate mTCP [38] because the underlying DPDK library has limited support on our NIC. mTCP has much higher latency than RDMA due to batching, and the reported throughput was 1.7 M packets per second [39]. We also compare with SocksDirect without batching and zero copy, namely SD (unopt). This work does not raise any ethical issues.

5.2 Microbenchmarks

5.2.1 *Latency and Throughput.*

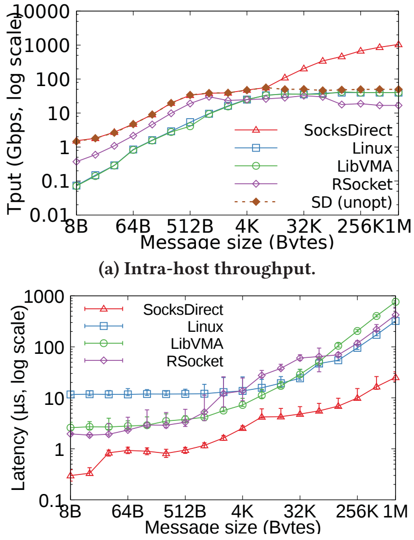
Figure 7 shows intra-host socket performance between a pair of sender and receiver threads. For 8-byte messages, SocksDirect achieves 0.3μs round-trip latency (1/35 of Linux) and 23 M messages per second throughput (20x of Linux). In comparison, a simple SHM queue has 0.25μs round trip latency and 27 M throughput, indicating that SocksDirect adds little overhead. SocksDirect does not use batching for intra-host sockets. RSocket has 6x latency and 1/4 throughput of SocksDirect, because it uses the NIC to forward intra-host packets, which incurs PCIe latency. LibVMA uses the NIC or a standard kernel socket to forward intra-host packets. The one-way delay of SocksDirect is 0.15μs, even lower than a kernel crossing (i.e. syscall, 0.2μs). Kernel-based sockets require a kernel crossing on both sender and receiver.

Due to memory copy, for 8 KiB messages, the throughput of SocksDirect is only 60% higher than Linux, and the latency is 1/4. For messages with at least 16 KiB size, SocksDirect achieves zero copy. For 1 MiB messages, SocksDirect achieves 1/13 latency and 26x throughput of Linux. The latency of RSocket is unstable and may be even higher than Linux in some cases.

Figure 8 shows inter-host socket performance between a pair of threads. For 8-byte messages, SocksDirect achieves 18M messages per second throughput (15x of Linux) and 1.7μs latency (1/17 of Linux). The latency is close to raw RDMA write operations (shown as dashed line), which does not have socket semantics. Batching does not affect latency in our benchmarks because an RDMA write is delayed only when the send queue is full. SocksDirect has even higher throughput than RDMA for 8-byte messages due to batching. Throughput of non-batched SocksDirect is between RSocket and RDMA. LibVMA also uses batching to achieve good throughput, but its latency is 7x of SocksDirect. For message sizes less than 8 KiB, the throughput of inter-host RDMA is slightly lower than intra-host SHM, because the ring buffer structure is shared. For 512B to 8 KiB messages, as well as larger messages without zero copy, SocksDirect is bounded by payload copy, but still faster than RSocket and LibVMA due to lower buffer management overheads. For zero copy messages (≥16 KiB), SocksDirect saturates the 100 Gbps link bandwidth, which has 3.5x throughput of all compared systems and 72% latency of RSocket.

5.2.2 *Latency Breakdown.*

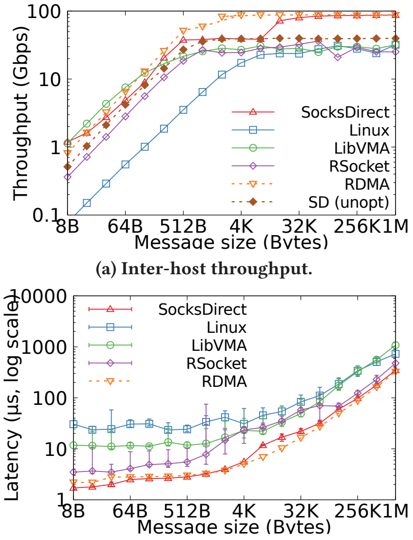
Table 4 shows why SocksDirect outperforms other socket systems. Per socket operation, Linux involves kernel crossing and all systems except SocksDirect involve locking in thread safe mode. Per packet, SocksDirect saves buffer management cost and offloads transport and packet processing to NIC. SocksDirect only needs two DMAs due to one-sided RDMA write. RSocket uses two-sided RDMA and LibVMA uses a similar packet interface, so the receive side need one more DMA. LibVMA and RSocket use the NIC to forward intra-host packets, while SocksDirect uses SHM. The high latency of Linux attributes to interrupt handling and process wakeup. For large messages, SocksDirect avoids payload copy and the page remapping cost is much lower. RSocket performs better than LibVMA and Linux because it overlaps sender copy, RDMA send and receiver copy. The connection setup latency of SocksDirect mainly comes from the initial handshake via Linux raw socket and creating RDMA QPs via `libibverbs`.



(a) Intra-host throughput.

(b) Intra-host latency.

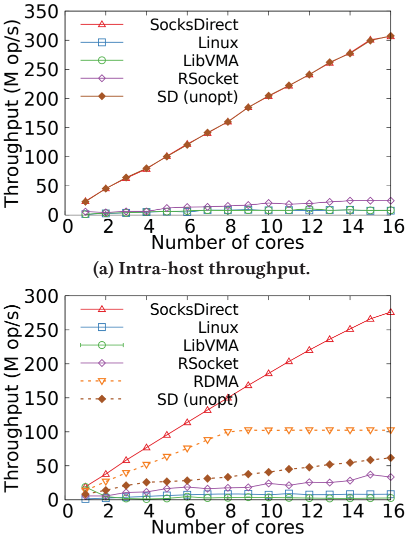
Figure 7: Single-core intra-host performance with message sizes.



(a) Inter-host throughput.

(b) Inter-host latency.

Figure 8: Single-core inter-host performance with message sizes.



(a) Intra-host throughput.

(b) Inter-host throughput.

Figure 9: 8-byte data transmission throughput with number of cores.

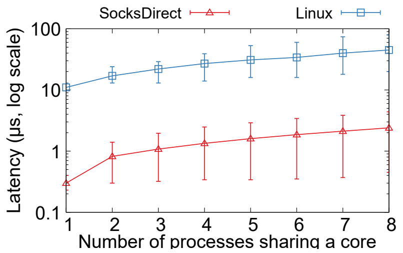


Figure 10: Message processing latency where processes share a core.

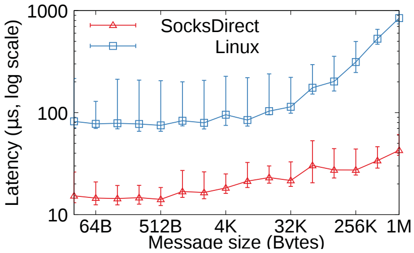


Figure 11: Nginx HTTP request end-to-end latency.

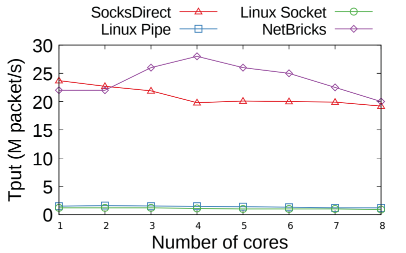


Figure 12: Throughput of network function pipeline.

5.2.3 *Multi-core Scalability.*

Figure 9 shows the throughput of 8-byte messages with different number of cores. Sender and receiver process each creates several threads, then each thread pins on a core and communicates with a peer thread. SocksDirect achieves almost linear scalability for both intra-host and inter-host sockets. For intra-host socket, SocksDirect provides 306 M message per second throughput between 16 pairs of sender and receiver cores, which is 40x of Linux and 30x of RSocket. LibVMA falls back to Linux for intra-host socket. Using RDMA for inter-host socket, SocksDirect uses batching to achieve 276 M messages per second throughput with 16 cores, which is 2.5x of the message throughput of our RDMA NIC and 8x of RSocket. Without batching, SocksDirect can only achieve 62 M throughput (60% of RDMA). RSocket peeks at 24 M for intra-host and 33 M for inter-host due to limited scalability of buffer management. Due to lock contention on shared NIC queues, compared to single thread, the throughput of LibVMA reduces to 1/4 with two threads, and 1/10 with three and more threads. The Linux throughput scales linearly from 1 to 7 cores and bottlenecks on the loopback or NIC queues with more cores. Although not evaluated, mTCP is supposed to have higher scalability with multiple cores.

When multiple threads share a core, each thread needs to wait for its turn to process messages. As Figure 10 shows, although the message processing latency increases almost linearly with number of active processes, it is still 1/20 to 1/30 of Linux.

5.3 Application Performance

In this section, we demonstrate that SocksDirect can significantly improve the performance of real-world applications without modifying the code. Rsocket [10] is not compatible with any of the following applications.

5.3.1 *Nginx HTTP Server.*

To test a typical Web service scenario where the clients come from the network and served within a host, we use Nginx [60] v1.10 as a reverse proxy between an HTTP request generator and an HTTP response generator. Nginx and the response generator are in a same host, while the request generator is in a different host. The generators use a keep-alive TCP connection to communicate with Nginx. LibVMA [51] does not work with unmodified Nginx due to fork. In Figure 11, the request generator measures the time from sending an HTTP request to receiving the whole response. For small HTTP response sizes, SocksDirect reduces latency by 5.5x compared to Linux. For large responses, due to zero copy, SocksDirect reduces latency by up to 20x.

5.3.2 *Redis Key-Value Store.*

We measure Redis [8] latency using redis-benchmark and 8-byte GET requests. Using Linux, the mean latency is 38.9 μs, with 1%

and 99% percentile 31.6 and 56.1 μs. With SocksDirect, the mean latency is 14.1 μs (64% lower than Linux), with 1% and 99% percentile 8.4 and 19.1 μs.

5.3.3 RPC Library.

We measure RPC round-trip time using RPClib [9]. We run the example 1 KiB RPC in RPClib among two processes inside a host, and it takes 45 μs. Across two hosts, the RPC takes 79 μs. Using SocksDirect, intra-host latency becomes 21 μs (53% reduction) and inter-host is 46 μs (42% reduction). However, SocksDirect is no panacea. The performance of RPClib and libsd is much lower than state-of-the-art RPC libraries, e.g., eRPC [39], because the overhead in RPClib dominates performance.

5.3.4 Network Function Pipeline.

64-byte packets in *pcap* format originate from an external packet generator, pass through the network function (NF) pipeline, and sends back to the packet generator. We implement each NF as a process that inputs packets from *stdin*, updates local counters, and outputs to *stdout*. For Linux, we use *pipe* and *TCP socket* to connect NF processes inside a host. Figure 12 shows that the throughput of SocksDirect is 15x and 20x of Linux pipe and TCP socket. It is even close to a state-of-the-art NF framework, NetBricks [55].

6 LIMITATIONS AND DISCUSSIONS

Scale to many connections. The scalability of SocksDirect to many connections is bounded by the underlying transports, i.e., SHM and RDMA, instead of libsd and monitor. To demonstrate this, we run a synthetic experiment that creates a lot of connections between two processes without creating new RDMA QPs and SHMs. An application thread with libsd can create 1.4 M new connections per second, which is 20x of Linux and 2x of mTCP [38]. The monitor can create 5.3 M connections per second.

Because the number of processes inside a host is limited, the number of SHM connections will probably not be very large. However, the scalability of RDMA becomes a concern, especially in large-scale RDMA deployments [34]. This boils down to two problems. First, RDMA NIC keeps per-connection states using on-NIC memory as cache. With thousands of concurrent connections, the performance suffers from frequent cache misses [39, 41, 48]. We note that most NIC vendors have realized this problem. As a result, recent commodity NICs have larger and larger memory, e.g., Mellanox ConnectX-5 [66], can store the states of thousands of connections [39], and SmartNICs even have gigabytes of DRAM on board [31, 67, 68]. We believe the NIC cache miss problem will be less a concern in future datacenters. The second problem is that RDMA connection establishment takes $\approx 30\mu s$ in our testbed, which is significant for short connections. However, this process only involves communication between local CPU and NIC, and we expect future works to improve.

Transport. We offload transport mechanisms to the RDMA NIC. The reader may have some concerns about RDMA NIC's transport mechanisms. For example, most commodity RDMA NICs rely on Priority-based Flow Control (PFC) to eliminate congestion losses in Ethernet networks. PFC brings many problems, such as head-of-the-line blocking, congestion spreading, and even deadlocks [34]. marking our networks hard to manage and understand. We note that many efforts have been made to improve RDMA transport performance. Emerging RDMA congestion control algorithms [45, 48, 52, 74] not only improve throughput and latency, but also reduce the number of PFC pause frames. Many advanced loss recovery mechanisms [49, 53] have also been proposed to allow RDMA deployments over lossy networks without PFC. Hence, we envision that future RDMA NICs can provide low latency and high throughput transport over lossy datacenter networks.

Idle thread wakeup overhead. §4.4 achieves time-sharing among threads via round-robin polling. However, with many threads and random arrival of messages, most threads are idle when waked up (Figure 10), and therefore waste CPU cycles. A fundamental solution would need to modify the kernel scheduling ordering without reintroducing the overheads. For intra-host communication, in single dispatcher and multiple worker communication pattern, the dispatcher can determine the scheduling ordering of workers. For inter-host communication, since the NIC is a centralized dispatcher, it would be interesting to use the NIC to determine the scheduling ordering of threads.

Compatibility limitations. SocksDirect has compatibility limitations that also exist in other user-space network stacks. The glibc shim using `LD_PRELOAD` cannot intercept direct syscalls, which can be found in statically linked applications. Sockets are invisible from `/proc` file system, so network monitoring utilities may not work. SocksDirect lacks several kernel stack functions, e.g., `netfilter` and traffic control. However, state-of-the-art NICs already support QoS and ACL offloading [4]. So these functions can be offloaded to hardware.

Monitor. Busy polling of the monitor uses a CPU core. Failure of the monitor daemon is hard to recover. These problems can be solved by implementing the monitor in the kernel. A kernel monitor will incur synchronization overhead, but the monitor is not involved for most data-plane operations.

Abstractions beyond socket. Realizing the limitations of RDMA, LITE [71] proposes an abstraction layer above RDMA, which includes a socket-like API and shares many techniques with SocksDirect. Different from RDMA transport offload, eRPC [39] proposes an *onload* approach to high performance communication. It consolidates RPC and transport into a user-space networking stack. Looking forward, we envision that new communication abstractions should be proposed to bridge the gap between emerging programmable hardware and high performance applications.

7 CONCLUSION

SocksDirect is a Linux compatible and high performance user-space socket system. We design a per-host monitor daemon for trusted control plane; a peer-to-peer synchronization-free data plane to fully support fork and multi-thread socket sharing; and a ring buffer that utilizes shared memory and RDMA efficiently. SocksDirect achieves performance close to hardware limits and improves end-to-end performance of real-world applications.

ACKNOWLEDGEMENTS

We would like to thank Yongqiang Xiong, Peng Cheng, Kun Tan, Daniel Firestone, Yuanwei Lu, Wencong Xiao, Zhixiong Niu, and Chuanxiong Guo for all technical discussions and suggestions on the project. We also thank our shepherd, Dan Li, and other anonymous reviewers for their valuable feedbacks and comments.

REFERENCES

[1] 2019. Apache HTTP Server. (2019). Available at https://httpd.apache.org/.

[2] 2019. FastCGI Process Manager for PHP. (2019). Available at https://php-fpm.org.

[3] 2019. High-Performance Network Framework Based on DPDK. (2019). Available at http://f-stack.org/.

[4] 2019. Mellanox Adapters Programmer's Reference Manual (PRM). (2019). Available at http://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf.

[5] 2019. Myricom DBL. (2019). Available at https://www.cspi.com/ethernet-products/software/dbl/.

[6] 2019. PF_RING. (2019). Available at http://www.ntop.org.

[7] 2019. Python WSGI HTTP Server for UNIX. (2019). Available at https://gunicorn.org.

[8] 2019. Redis. (2019). Available at https://redis.io.

[9] 2019. rpclib - modern msgpack-rpc for C++. (2019). Available at http://rpclib.net.

[10] 2019. rsocket(7) - linux man page. (2019). https://linux.die.net/man/7/

[11] 2019. Seastar: High-Performance Server-side Application Framework. (2019). Available at http://seastar.io/.

[12] 2019. vsftpd. (2019). Available at https://security.appspot.com/vsftpd.html.

[13] Arista. 2019. Arista 7060X Series. (2019). Available at https://www.arista.com/en/products/7060x-series.

[14] InfiniBand Trade Association. 2000. The Infiniband Architecture Specification. *http://www. infinibandta. org/specs/* (2000).

[15] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[16] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2017. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2017), 11.

[17] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores.. In *OSDI*, Vol. 10. 86–93.

[18] Josiah L. Carlson. 2013. *Redis in action.* Manning Publications Co.

[19] Hsiao-keng Jerry Chu. 1996. Zero-copy TCP in Solaris. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference.* Usenix Association, 21–21.

[20] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. 1989. An analysis of TCP processing overhead. *IEEE Communications magazine* 27, 6 (1989), 23–29.

[21] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)* 32, 4 (2015), 10.

[22] Jonathan Corbet. 2007. Large receive offload. (2007). Available at https://lwn.net/Articles/243949/.

[23] Jonathan Corbet. 2012. TCP connection repair. (2012). https://lwn.net/Articles/495304/

[24] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. (Nov. 2017). Available at https://lwn.net/Articles/738975/.

[25] Jonathan Corbet. 2017. Zero-copy networking. (2017). https://lwn.net/Articles/726917/

[26] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developer Manual, Volume 3. (2019).

[27] Microsoft Corporation. 2008. Information about the TCP Chimney Offload. (2008). Available at https://support.microsoft.com/en-us/help/951037/information-about-the-tcp-chimney-offload-receive-side-scaling-and-net.

[28] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 373–387.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation.* 401–414.

[30] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science* 2 (2001), 77.

[31] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 51–66.

[32] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[33] Chuanxiong Guo. 2017. RDMA in Data Centers: Looking Back and Looking Forward. (2017). Keynote on ACM SIGCOMM APNet 2017, available at http://sysnetome.com/Talks/chguo_apnet2017_keynote.pptx.

[34] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16).* ACM, New York, NY, USA, 202–215. https://doi.org/10.1145/2934872.2934908

[35] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI*, Vol. 12. 135–148.

[36] Yukai Huang, Jinkun Geng, Du Lin, Bin Wang, Junfeng Li, Ruilin Ling, and Dan Li. 2017. LOS: A High Performance and Compatible User-level Network Operating System. In *Proceedings of the First Asia-Pacific Workshop on Networking.* ACM, 50–56.

[37] Intel. 2014. Data plane development kit. (2014). Available at https://dpdk.org/.

[38] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.. In *NSDI*, Vol. 14. 489–502.

[39] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* USENIX Association, Boston, MA.

[40] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 295–306.

[41] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference.* 437.

[42] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19).* ACM, New York, NY, USA, Article 24, 16 pages. https://doi.org/10.1145/3302424.3303985

[43] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19).* USENIX Association, Boston, MA.

[44] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, 137–152.

[45] Yuliang Li, Rui Miao, Hongqiang Liu, Yan Zhuang, Fei Fang, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *2019 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '19).*

[46] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable kernel TCP design and implementation for short-lived connections. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 339–352.

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18).* 973–990.

[48] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 357–371.

[49] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2017. Memory Efficient Loss Recovery for Hardware-based Transport in Datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking.* ACM, 22–28.

[50] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network stack specialization for performance. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 175–186.

[51] Mellanox. 2019. Messaging Accelerator (VMA). (2019). Available at https://github.com/mellanox/libvma.

[52] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 537–550.

[53] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* ACM, 313–326.

[54] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication.* ACM, 327–341.

[55] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI.* 203–216.

[56] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 337–350.

[57] Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2016), 11.

[58] Jim Pinkerton. 2003. Sockets Direct Protocol V1.0. (2003).

[59] Steve Pope and David Riddoch. 2011. *Introduction to OpenOnloadBuilding Application Transparency and Protocol Conformance into Application Acceleration Middleware.* Technical Report.

[60] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal* 2008, 173 (2008), 2.

[61] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*. 101–112.

[62] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 342–358.

[63] Robert Russell. 2008. The extended sockets interface for accessing rdma hardware. In *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS '08)*. 279–284.

[64] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.

[65] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 33–46.

[66] Mellanox Technologies. 2018. ConnectX-5 Single/Dual-Port Adapter supporting 100Gb/s with VPI. (2018). Available at http://www.mellanox.com/page/products_dyn?product_family=258&mtag=connectx_5_vpi_card.

[67] Mellanox Technologies. 2019. Mellanox BlueField(TM) SmartNIC VPI. (2019). Available at http://www.mellanox.com/page/bluefield_smartnic_vpi?mtag=smartnic_vpi1.

[68] Mellanox Technologies. 2019. Mellanox Innova-2 Flex Open Programmable SmartNIC. (2019). Available at http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex.

[69] Moti N. Thadani and Yousef A. Khalidi. 1995. *An efficient zero-copy I/O framework for UNIX.* Sun Microsystems Laboratories.

[70] Kevin Thompson, Gregory J Miller, and Rick Wilder. 1997. Wide-area Internet traffic patterns and characteristics. *IEEE network* 11, 6 (1997), 10–23.

[71] Shin-Yeh Tsai and Yiying Zhang. 2017. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 306–324.

[72] Wikipedia. 2019. iSCSI Host Adapter (HBA). (2019). Available at https://en.wikipedia.org/wiki/Host_adapter.

[73] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs.. In *USENIX Annual Technical Conference*. 43–56.

[74] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 523–536.

[75] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA.