李博杰 BA16011029

本学期在科研方面，进行了数据中心系统加速方面的研究，撰写了两篇学术论文（在投），申请了三项国际专利。

一、用户态套接字通信技术

为了尽可能降低操作系统内核中的通信开销，我们把同一个内核上运行的诸进程看作一个分布式系统，进程之间采用基于消息传递的共享内存通信。为了解决多进程并发同步的协调问题，以及进行特权操作，我们引入一个用户态的管程来处理多进程并发同步。通过完全绕过内核，我们消除了系统调用的开销。为了保持原有的接口抽象，我们设计实现了一个用户态的网络协议栈，应用程序不再调用标准运行库，而是调用用户态网络协议栈。多进程并发同步中的一部分操作也可以在用户态网络协议栈中完成，只有不可划分的操作和特权操作放在管程中完成。最后，为了消除可靠传输协议的开销，我们利用 RDMA 网卡来实现可靠传输协议。

二、可靠有序消息散播技术

在分布式系统中，应用程序需要解决并发数据访问的一致性问题。传统方案一般使用加锁或乐观并发控制的方法，在强竞争情况下可扩放性不佳。为了解决远程键值访问的并发控制问题，我们提出了可靠有序消息散射。发送端给数据包打上递增的时间戳，接收端保证按照时间戳递增的顺序投递来自所有发送端的数据包。我们利用可编程交换机的数据包处理功能，设计了一个终端主机和可编程交换机相结合的可靠有序消息散射方案，实现网络规模增加的情况下每台主机的处理开销、每条链路的网络带宽开销保持恒定，数据包投递的延迟也只随网络层数线性增长。

主要收获与思考

两篇在投论文都被顶级会议分别拒稿两次，目前正在修改，重新投稿。第一篇论文的主要问题是应用场景不够明确，假设不够合理（假设两个应用之间建立多个连接，但实际系统中一般只建立一个连接），导致系统设计过于复杂。最近咨询了一些工业界的大佬，了解了更多实际系统对套接字的用法和挑战。目前正在从实际应用场景出发，简化系统设计。

第二篇论文的主要问题是设计过于复杂，正确性难以保证。第二篇论文第一次投稿只解决了有序传输的问题，但缺少应用场景，从而被拒稿；第二次投稿加入了可靠传输的保证，但同时做到有序和可靠传输的系统设计过于复杂，审稿人认为正确性没有保证。目前我们发现一些应用场景只需要有序传输而不需要可靠传输；而且发现一些现有的成熟方案可以解决可靠传输问题，可以添加到有序传输的基础上。因此我们仍然专注于解决有序传输问题，采用现有的其他方案解决可靠传输问题，从而简化系统设计。

附件

两篇在投论文（为保护知识产权，分别截取了前 5 页）

SocksDirect: Push-Button Socket Acceleration in User Space

NSDI'19 submission #171

Abstract

Linux socket is implemented in the kernel space with shared data structures that needs concurrency protection, which incurs significant overhead. Communication intensive applications in hosts with multi-core CPU and high-speed networking hardware often put considerable stress on the socket system. Recent work on user-space sockets either does not support intra-host communication among containers and applications, or has limitations on compatibility, isolation and multi-thread scalability.

In this paper, we describe SOCKSDIRECT, a high performance socket system. SOCKSDIRECT is implemented in user space to avoid kernel crossing cost. It achieves security and isolation by employing a trusted monitor daemon to handle control plane operations such as connection establishment and access control. SOCKSDIRECT is fully compatible with Linux socket and can be used as a drop-in replacement with no modification to existing applications. The design fully handles Linux fork semantics, and can handle both intra- and inter-host communications with hosts equipped with SOCKSDIRECT as well as those without. Last but not least, SOCKSDIRECT is performant. SOCKSDIRECT uses shared memory queue and modern RDMA transport for intra- and inter-host communication. It removes multi-thread synchronization in common cases and improves memory efficiency with many concurrent connections. It leverages techniques such as cooperative multitasking and page-remapping based zero-copy to remove many overheads of existing socket systems. Experiment shows that SOCKSDIRECT achieves 7 to 20x better message throughput, 17 to 35x better latency, and 20x connection setup throughput compared with Linux socket.

1 Introduction

Socket API is the most widely used communication primitive in modern OS. It is used universally for communications between processes, containers and hosts. Traditional Linux socket implementation is not optimal. It can only achieve latency and throughput numbers an order of magnitude worse than what the raw hardware is capable of. Communication intensive applications such as distributed key-value stores and web servers could spend 50%∼90% of CPU time in the OS kernel, mostly processing socket operations.

There has been extensive work aiming at reducing the socket overhead, but existing approaches are not satisfactory. First, many user-space sockets are not fully compatible with native socket in areas such as when a process forks, when multiple processes listen to the same port, and when communication is intra-host. Second, some user-space sockets have security issues because applications can directly access shared NIC queues, thus violating process isolation and access control policies. Those solutions that do preserve security often incur the overhead of kernel crossing or virtual switch. Third, even in the performance front, there are still much room for improvement. For example, none of existing works can achieve performance close to raw RDMA, because they fail to remove some important overheads such as multi-thread synchronization, memory copy and TCP/IP packet processing.

We design SOCKSDIRECT, a user-space socket architecture with compatibility, security and performance in mind.

- **Compatibility**. Existing applications can use SOCKSDIRECT as a drop-in replacement with no modification. It supports both intra-host and inter-host communication, and behaves correctly during process fork and thread creation. If a remote peer does not support SOCKSDIRECT, the system falls back to TCP transparently.
- **Security**. SOCKSDIRECT preserves isolation among applications and containers, and it enforces firewall rules and access control policies.
- **High Performance**. SOCKSDIRECT delivers consistent high throughput and low latency that is scalable with number of CPU cores, and the performance does not degrade significantly with vast number of concurrent connections.

At its heart, SOCKSDIRECT follows the principle of separating control and data plane to achieve both security and performance. We treat processes as a shared-nothing distributed system that communicates via peer-to-peer message queues. We design a per-host *monitor* daemon as the control plane to enforce access control policies, dispatch new connections, perform address translation for overlay networks and establish transport channel between communication peers. The data plane is handled by a dynamically loaded user-space library LIBSD, which implements data transmission and event polling in a peer-to-peer fashion between processes while delegates connection creation to the local monitor. LIBSD intercepts Linux *glibc*, implements all socket-related functions in user space and forwards the other APIs to the kernel.

We face several challenges designing SOCKSDIRECT to achieve high performance while maintaining compatibility. One challenge is to share a socket among threads and pro-

cesses without locking. A socket connection is a FIFO channel. The most straight-forward approach is to implement each connection as a single queue. However, a socket may be shared by multiple senders and receivers, locking is thus needed to protect the shared queue, which significantly slows down the system. To avoid locking overhead, we treat each thread as a separate process and create peer-to-peer queues between each pair of communicating threads. In our design, a single socket connection may correspond to multiple queues, and we take special care to preserve FIFO semantics during fork and thread creation. Another challenge is to maintain performance with vast number of concurrent connections. To handle many concurrent connections efficiently, we need to reduce memory footprint and improve memory access locality. Rather than maintaining a separate queue for each connection, SOCKSDIRECT multiplexes socket connections through a single queue for each pair of communicating threads. We design the queue carefully to enable fetching from the middle of a queue and solve the head-of-line blocking problem.

We exploit multiple techniques to effectively utilize hardware and improve system efficiency. For communication within a same host, we design a high performance user space shared memory queue. For communication among hosts in an RDMA enabled data center, we use RDMA NIC hardware for transport. To remove memory copy cost for large messages, we take advantage of *page remapping* to achieve transparent zero copy. To share a CPU core efficiently among multiple active threads without thread wakeup overhead, we leverage *cooperative multitasking*.

SOCKSDIRECT achieves latency and throughput close to the raw performance achievable from the underlying shared memory queue and RDMA. On the latency side, SOCKSDIRECT achieves 0.3μs RTT for intra-host socket, 35x lower than Linux and only 0.05μs higher than a bare-metal shared memory queue. For inter-host socket, SOCKSDIRECT achieves 1.7μs RTT between RDMA hosts, almost the same as raw RDMA verbs and 17x lower than Linux. On the throughput side, a single thread can send 23 M intra-host messages per second (20x Linux) or 8 M inter-host (7x Linux). For large messages, with zero copy, a single connection saturates NIC bandwidth. Each thread can establish 1.4 M new connections per second (20x Linux). The performance above is scalable with number of cores, and do not degrade significantly with millions of concurrent connections.

2 Background

2.1 Socket

Socket is the standard communication primitive among applications and containers. Modern data center networks have microsecond-level 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 applications running on multiple hosts [9]. In addition to inter-host communications, 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, we observe applications such as Nginx [55], memcached [24] and Redis [13] consume 50% to 90% CPU time in the kernel, mostly dealing with TCP socket operations. This agrees with previous results [31].

TCP socket in a modern OS typically has three functions:

- Address, locate and connect to another application;
- Provide a reliable and ordered communication channel, identified by an integer *file descriptor* (FD);
- Multiplex events from multiple channels e.g., poll and epoll. Most Linux applications use a readiness-driven I/O multiplexing model. The OS tells application which FDs are ready to receive or send, then the application may prepare buffers and issue receive or send operations.

2.2 High Performance Socket Systems

Many high performance socket systems have been proposed from both academia and industry. Table 1 compares several of them from three aspects: compatibility, security and performance.

Kernel network stack optimization: The first line of work optimizes the kernel TCP/IP stack. FastSocket [39], Affinity-Accept [49], FlexSC [59] and zero-copy socket [60, 14, 18] achieve good compatibility and security, but leave some performance optimizations on the table.

MegaPipe [26] and StackMap [61] propose new interfaces to achieve zero copy and improve I/O multiplexing, at the cost of requiring application modifications.

User-level network stack: The second line of work completely bypasses kernel TCP/IP stack and implements TCP/IP in user space. In this category, IX [10] and Arrakis [50] are new OS architectures that uses virtualization to ensure security and isolation. IX leverages user space network stack [22] 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.

Apart from these new OS architectures, many recent efforts use high performance packet I/O frameworks on Linux, e.g., Netmap [56], Intel DPDK [29] and PF_RING [5]), to directly access NIC queues in user space. SandStorm [43], mTCP [31], Seastar [7] and F-Stack [2] propose new interfaces and thus need to modify applications. LibVMA [45], OpenOnload [54], DBL [4] and LOS [27] are designed to be compatible with existing applications. These designs often sacrifice security, because multiple applications of a packet I/O framework share a NIC. To receive packets of listening ports and established connections from the NIC directly, each application inserts flow steering rules to the NIC independently, which is insecure and leads to conflicts when

	FastSocket	MegaPipe / StackMap	IX	Arrakis	SandStorm / mTCP	LibVMA	OpenOnload	Rsocket / SDP	FreeFlow	SocksDirect
Reliable transport location	Kernel		User space					NIC hardware		
Wire protocol	TCP/IP							RDMA		
Changes needed for deployment	New kernel	New kernel	New kernel	New kernel	Lib+driver	Lib+driver	Lib+driver	Lib+driver	Lib+driver +daemon	Lib+driver +daemon
Compatibility										
Transparent to existing applications	✓		✓	✓		✓	✓	✓	✓	✓
Intra-host communication	✓	✓		✓					✓	✓
Container overlay network									✓	✓
Multiple applications listen a port	✓	✓						✓	✓	✓
Full fork support	✓	✓					✓			✓
Compatible with regular TCP peers	✓	✓	✓	✓	✓	✓	✓			✓
Security and 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
Performance										
Offload transport to RDMA NIC								✓	✓	✓
Many concurrent connections	✓	✓	✓	✓	✓					✓
Kernel bypass				✓	✓	✓	✓	Partial	✓	✓
Scale to multiple cores	✓	✓	✓	✓	✓	✓	✓	✓		✓
Multiple threads share a connection										✓
Multiple threads share a CPU core										✓
Socket-compatible zero copy										✓

Table 1: Comparison of high performance socket systems.

multiple applications listen on a same port. An application may send arbitrary packets that violate access control policy. Moreover, most user-space stacks are not designed to accelerate intra-host connections and have limited fork support.

RDMA offloading: The third line of research utilizes RDMA NICs [3] that are widely available in production data centers [25] and translate socket operations to RDMA verbs. RDMA uses hardware offloading to provide ultra low latency and near zero CPU utilization compared to software-based TCP/IP network stacks. RSocket [6] and SDP [53] convert each socket connection to an RDMA connection (RC QP). However, such simple translations suffer from throughput degradation with a large number of concurrent connections. This is because RDMA NIC keeps per-connection states using a ≈2 MB [33] on-NIC memory as cache. With hundreds of concurrent connections, we will suffer from frequent cache misses, resulting in serious throughput degradation [41, 34]. FreeFlow [62, 36] is a software switch to virtualize an RDMA NIC for container overlay network and enables intra-host RDMA communication. It uses RSocket to translate socket to RDMA, thus also suffering from cache miss problem.

2.3 Performance Challenges

We aim to deliver persistent high throughput and low latency regardless of the number of concurrent connections. In addition, the performance should scale linearly with the number of CPU cores. This subsection analyzes challenges to achieve above goals.

Intra-host communication. Most existing approaches for intra-host socket either use kernel network stack or NIC loopback. The kernel network stack has evolved to become quite complicated over the years [61], which is an overkill for intra-host communication.

Arrakis uses the NIC to forward packets from one application to another. As shown in Table 2, the hairpin latency from CPU to NIC is still 25x higher than inter-core cache migration delay (∼30 ns). The throughput is also limited by Memory-Mapped I/O (MMIO) doorbell latency and PCIe bandwidth [47, 37].

We aim to leverage user-space shared memory for intra-host socket communication.

Container overlay network. Many container deployments use isolated network namespaces for containers, which communicate via a virtual overlay network. In Linux, a virtual switch [51] forwards packets among host NIC and virtual NICs in containers. This architecture incurs the overhead of multiple context switches and memory copies on each packet, and the virtual switch becomes the bottleneck [52].

We aim to use the virtual switch for only control plane and implement peer-to-peer data plane operations.

RDMA offloading. The main challenge for leverage RDMA for inter-host socket communication is to bridge the semantics of socket and RDMA [21]. For example, RDMA preserves messages boundaries while TCP does not. For I/O multiplexing, RDMA provides a completion notification model while event polling in Linux socket requires a readiness model [26]. Further, one-sided and two-sided RDMA verbs have different efficiency and overheads [32, 34].

We aim to use RDMA efficiently for inter-host socket communication, while falling back to TCP transparently in case of non-RDMA peers.

Many concurrent connections. Internet facing applications often need to serve millions of concurrent connections efficiently [31, 39, 10]. Moreover, it is also common for two backend applications to create large number of connections

between them, where each connection handles a concurrent task [28, 30, 48]. In Linux, a socket connection has dedicated send and receive buffers, each is at least one page (4 KB) in size [20]. With millions of concurrent connections, the socket buffers can consume gigabytes of memory, most of which is empty. Random accesses to a large number of buffers also cause CPU cache misses and TLB misses. Similar issue exists in RDMA NICs with limited on-chip memory for caching connection states [41, 34].

We aim to minimize memory cache misses per data transmission by multiplexing socket connections.

Kernel bypassing. Traditionally, socket APIs are implemented in kernel, thus requiring kernel crossing for each socket operation. To make it worse, the Kernel Page-Table Isolation (KPTI) patches [17] to protect against Meltdown [40] attacks make kernel crossings 4x expensive.

We aim to bypass kernel without compromising security.

Scaling to multiple cores. Linux kernel acquires several global locks during connection creation [11, 12]. MegaPipe [26] and FastSocket [39] remove these bottlenecks, but leave other kernel overheads on the table [31]. Several efforts [44, 57, 27, 62] delegate all operations to a virtual switch running as a daemon process, which is a scalability bottleneck in modern servers with tens of cores.

We aim to minimize centralized coordinations while preserving isolation among applications and containers.

Multiple threads sharing a connection. Multiple threads in a process share socket connections. For example, after a process forks, 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 [12, 26, 39]. Table 2 shows that a shared memory queue protected by atomic operations has 4x latency and 22% throughput of a lockless queue, even if there is no contention. Previous work [12, 15] suggests that many socket operations are not commutable and synchronizations cannot always be avoided.

We aim to minimize synchronization overhead in two ways. First, we optimize for the common cases and remove synchronizations in frequently used socket operations. Second, we leverage the fact that shared memory message passing is much cheaper than locking [57], and use message passing as the exclusive synchronization mechanism.

Multiple threads sharing a CPU core. Most user-space network stacks use polling to avoid the overhead of interrupt and thread wakeup in the kernel. However, polling does not work when multiple threads have to share a core. To switch thread contexts, as Table 2 shows, using semaphore, mutex or `futex` to wake up a sleeping thread is 5x∼10x slower than cooperative context switch via `sched_yield`.

We aim to use cooperative multitasking instead of kernel thread wakeup mechanisms.

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
CPU L3 cache miss	0.07	14
Spinlock (atomic instruction)	0.10	5∼10
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
NIC cache miss	0.45	2.2
Cooperative context switch	0.52	2.0
Map 1 page (4 KiB)	0.78	1.3
CPU to NIC hairpin RDMA	1.0	14
Atomic shared memory queue	1.0	6.1
Map 32 pages (128 KiB)	1.2	0.8
Two-sided inter-host RDMA	1.6	8
One-sided inter-host RDMA	1.6	13
SOCKSDIRECT via RDMA	1.7	8
Semaphore, mutex, futex	2.8∼5.5	0.2∼0.4
Intra-host Linux TCP	11	0.9
Copy 32 pages (128 KiB)	13	0.08
Inter-host Linux TCP	30	0.3

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

Zero copy. The semantics of `send` and `recv` cause memory copies between application and network stack. For non-blocking `send`, the system needs to copy data out of the buffer because the application may overwrite the buffer right after `send` returns. For `recv`, application provides a buffer as the data destination. System needs to copy data received into the buffer. Many user-space TCP/IP stacks and socket-to-RDMA libraries provide both standard socket API and an alternative zero-copy API, but none of them achieves zero copy for the standard API on both send and receive paths.

We aim to allow transparent zero copy for large transfers.

3 Design

3.1 Architecture Overview

Figure 1 gives the architecture of SOCKSDIRECT. To use SOCKSDIRECT, an application loads a user-space library LIBSD by setting the `LD_PRELOAD` environment variable. LIBSD intercepts all Linux APIs in `glibc` that are related to file descriptor operations. It implements socket APIs in user space and forwards the other APIs to the kernel. From a security point of view, because LIBSD resides in the application address space, we cannot trust its behavior. Therefore, we need a trusted component outside LIBSD to enforce access control and support overlay networks.

To this end, we design a *monitor* daemon at each host to coordinate control plane operations, e.g., connection creation. The *monitor* daemon is started at OS initialization time. In each host, all the applications loading LIBSD must establish a shared memory queue with the host's monitor daemon, forming the control plane. On the data plane, appli-

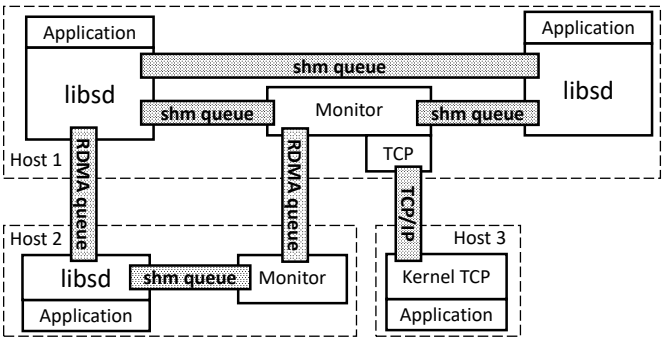


Figure 1: Architecture of SOCKSDIRECT. Host 1 and 2 are RDMA capable, while host 3 is RDMA incapable.

cations build peer-to-peer queues to communicate directly, thus relieving the burden of the monitor daemon.

To achieve low latency and high throughput, SOCKSDIRECT uses shared memory for intra-host and RDMA for inter-host communication. We now describe the procedure of intra-host communication. The communication initiator first sends a request to the local monitor, then the monitor establishes a shared memory queue between the two applications (possibly in different containers). 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. If so, 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 Figure 1. If the remote host does not support SOCKSDIRECT, it keeps using the TCP connection, as between host 1 and 3 in Figure 1. The detailed connection management protocol is presented in Sec. 3.2.

In order to remove synchronization overhead for multi-threaded applications, we treat each thread as a separate process. For two communicating applications, we create peer-to-peer queues between each pair of sender and receiver threads to avoid synchronization cost of contending on the same queue. In Sec. 3.3, we present the peer-to-peer queue design that preserves FIFO ordering semantics and avoids starvation, especially when a process forks or creates a new thread. To maintain performance with many concurrent connections, rather than creating separate queues for each connection, we multiplex data from all connections through one queue. In Sec. 3.4, we present the design of multiplexed queue that avoids head-of-line blocking and supports fetching data from any multiplexed connections.

3.2 Connection Management

Before designing the connection management protocol, we keep the following requirements in mind: 1) The applications and LIBSD are not trusted because they are in the same memory address space. We must enforce access control policies outside LIBSD to prevent access to restricted resources. 2) Each address and port may be listened by multiple processes, which needs load balancing while avoid starvation. 3) The applications may be in an overlay network and thus needs address translation. 4) A client should be able to connect to SOCKSDIRECT and regular TCP/IP hosts transparently, and a server should accept connections from all hosts.

These design requirements lead to a *monitor* service running as a daemon process in each host. Rather than delegating all operations to the monitor, we only delegate connection creation, which forms the control plane. From the application's perspective, connection creation is similar to TCP handshake. Monitor(s) on the path between client and server applications proxy the handshake commands and help them establish a peer-to-peer queue via shared memory or RDMA. If the remote peer does not support SOCKSDIRECT, all future operations with it will be delegated to the local monitor. The detailed procedure is as follows.

Initialization. During initialization, LIBSD connects to the monitor in local host via *bootstrap socket* (a Unix domain socket or kernel TCP socket on localhost or overlay network) and establishes a shared memory queue between them. After that, communication between the application and monitor goes through the shared memory queue.

Socket creation. An application first creates a socket identified by an integer *file descriptor* (FD). Socket FDs and other FDs (e.g. disk 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 a FD translation table to map each application FD to a user-space socket FD or a kernel FD. When an FD is closed, LIBSD put it to a *FD recycle pool*. Upon FD allocation, LIBSD first tries to obtain an FD from the recycle pool. If the pool is empty, it allocates a new FD by incrementing a *FD allocation counter*. The FD recycle pool and allocation counter are shared among all threads in a process.

Bind. 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 2, LIBSD sends the request to monitor and the monitor sets up an address translation rule between physical and overlay network. LIBSD employs an optimization to return success speculatively 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. The monitor maintains a list of listening processes on each address and port to dispatch new connections. The monitor also uses a user-space networking stack (modified Lib-

Scalable and Efficient Reliable Ordered Message Scattering in Data Center Networks

Eurosys'19 submission #23

Abstract

The ability to have reliable and ordered delivery of a group of messages can facilitate and simplify many distributed applications. Existing approaches either employ centralized sequencers or tokens, thus suffering from limited scalability, or use distributed consensus protocols, which incurs high overhead in bandwidth and delay, as well being faulty with malicious hosts.

This paper proposes Reliable Ordered Message Scattering (ROMS), a scalable and efficient method to deliver groups of messages reliably in order inside data centers. To achieve ordered delivery in a scalable manner, ROMS separates the bookkeeping of order information from message forwarding, and distributes the work to each switch and host. ROMS aggregates order information using in-network computation at switches. This forms the "control plane" of the system. On the "data plane", ROMS forwards messages in the network as usual and reorders them at the receiver based on the order information. To achieve reliability, switches detect packet losses and senders recover losses. To achieve atomicity, switches detect and notify Byzantine host failures and the SDN controller handles network partition.

We build two ROMS prototypes using Barefoot and Arista switches. Our evaluation shows that ROMS achieves high performance and fault tolerance with low CPU and network overheads. As a case study, ROMS achieves externally consistent independent transactions with latency and throughput close to a non-transactional, non-replicated system in YCSB+T and TPC-C benchmarks.

1 Introduction

In a network with arbitrary network delays, packet losses and failures, it is challenging to guarantee the reliability and ordering of message delivery. For example, multiple clients updates an object simultaneously in a distributed storage, where each client needs to update both metadata and data on different shards. The shards storing metadata and data may receive the updates in different orders, and some shards may fail to receive the updates, thus violating data consistency. Solutions that mitigate this problem often introduce synchronization overhead, and often complicate distributed system design.

Ordered communication provides an abstraction where different receivers process messages from senders in a consistent order. This abstraction, sometimes called Causally and Totally Ordered Communication Support (CATOCS) [19], is an important building block for both strongly consistent and eventually consistent systems. Ordered communication provides the guarantee that messages are delivered obeying the *causal order* in the Lamport logic clock sense [49]. This also imply *FIFO*, i.e., if a message is sent before the other from a same sender, it will also be delivered before the other. Moreover, messages are *total order*, ensuring that they are delivered in the same order to all participants.

Besides ordering, another desirable property is reliability in the presence of failure: either all the messages in a group are delivered to the desired targets, or none is delivered if any of them fails. Many efforts have been made to achieve reliable ordered group communication, *e.g.*, atomic broadcast [27] and consensus [51]. Most approaches are designed for a small-to-medium group of trusted hosts, where a client sends the same message to every participant. However, distributed systems in data centers scale to thousands of hosts, where each client communicate with a different and non-predetermined subset of hosts [69]. Existing solutions suffer from scalability, efficiency and security limitations.

In this work, we propose Reliable Ordered Message Scattering (ROMS), an efficient and scalable method to scatter groups of messages. We generalize multicast to *message scattering* [48], a communication pattern where a host sends a group of (potentially different) messages to multiple hosts simultaneously. ROMS strictly preserves the following two properties in an efficient and scalable manner:

- **Ordering**: Each receiver delivers messages from different senders in the ascending wall clock order of message sent time. Wall clock is a clock on each host that obeys monotonicity and causality, that is (not strictly) synchronized. This implies CATOCS.
- **Reliability**: It means both (1) *atomicity*, i.e., either all or none receivers deliver messages in a scattering group, and (2) *exactly once*, i.e., a scattering is guaranteed to be delivered exactly once if both the sender and all receivers of the scattering are not faulty, and the network is not partitioned. ROMS is fault tolerant under Byzantine failure of hosts and crash failure of network components.

We aim to implement ROMS in a data center network, where the topology is loop-free [35, 55], switches have generally good programmability, and switches are managed by a logically centralized SDN controller. We also rely on reasonable synchrony among the host and switch clocks, i.e. clock skew, drift and jitter on non-faulty hosts and switches are bounded. The ROMS primitive can facilitate the design and implementation of many canonical distributed systems in a data center, such as externally consistent distributed

transactions [40, 74], log replication in eventually consistent systems [84], state machine replication [50, 83] and improve consistency in distributed shared memory.

To achieve ordering in network with variable delay, we aggregate wall clock timestamps in network. The sender attaches a non-decreasing timestamp to each message. Messages sent by a host with the same timestamp are considered to form a scattering. Each receiver delivers messages in non-decreasing timestamp order.

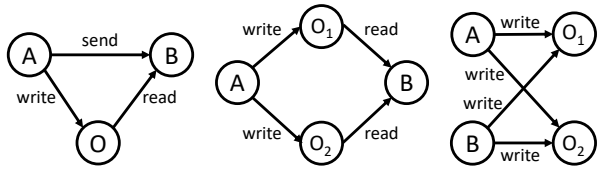
Our principle is to co-design end hosts with underlying data center networks. At its core, ROMS separates the bookkeeping of order information from message forwarding. ROMS forwards timestamped messages as usual in the network, and buffers them at the receiver side. The switch aggregates timestamp information of all messages to derive the *loss-free barrier* for each receiver. The barrier is essentially the *lower bound* of the timestamps of all future arrival packets. With this information, the receiver can *ordered deliver* the messages with timestamps below the loss-free barrier in order. Ordered delivered messages may be recalled if a failure occurs.

In order to derive the loss-free barrier, we generalize timestamp barriers from end hosts to every link in the network. Each switch keeps per-link barrier information and updates it for each packet. We merge barriers hierarchically at switches to reduce communication overhead. If some hosts or links are temporarily idle, we periodically generate beacons carrying barrier information.

To achieve reliability in case of packet loss and failure, rather than quorum-based voting, ROMS detects packet loss and failure in network, and relies on end hosts to retransmit lost packets and recall failed scatterings. To minimize the delay from scattering to delivery, we take advantage of the fact that data centers typically have very low packet loss rates [74]. When packet loss does not occur, a host can *reliably deliver* a message when it knows that all hosts have received messages with lower timestamps. The top-of-rack switches detect crash and Byzantine host failures with little overhead. Failure of network components are handled by a logically centralized SDN controller.

We implement three incarnations of ROMS on network devices with different programming capabilities: reconfigurable switching chips [3, 5] that can support flexible stateful per-packet processing, processors on the switches, and host CPUs if the switches are not programmable or the vendors do not expose accesses to switch CPUs.

We evaluate the performance of ROMS using both small-scale testbed experiments and large-scale simulation. Define *network diameter* D as the maximum one-way delay between two hosts. In normal cases, ROMS can deliver messages ordered in D and reliably in $2D$. ROMS also achieves low network bandwidth and CPU processing overhead in both small and large system scales. As a case study, ROMS supports externally consistent [21] independent transactions in one round-trip, close to a non-transactional, non-replicated system. ROMS achieves 6.4 million transactions and 40 μs latency in YCSB+T [28], which is 10x more efficient than MVCC concurrency control and Paxos-based replication. Moreover, the performance scales linearly with number of hosts. For New-Order and Payment transactions in TPC-C [23] with 4 warehouses, ROMS scales to thousands of concurrent clients, while MVCC only scales to tens.



(a) *Write after write.* ***(b)*** *Independent read, independent write.* ***(c)*** *Independent multi-write.*

Figure 1. *Ordering hazards in a distributed system.*

2 Motivation and Related Work

Ordering and reliability are two fundamental problems in distributed systems. Sec.2.1 and Sec.2.2 discuss the two problems and existing approaches. Sec.2.3 shows how ROMS can simplify distributed systems design.

2.1 Ordering

Ordering Hazards. Message scattering is a common communication pattern in distributed systems, where one end-host sender sends a group of messages to one or more end-host receivers. With variable network delays, four categories of ordering hazards [33, 76] may take place.

- *Reordering*. Host A writes data to O, then reads from O, but may not get the data due to in-network reordering.
- *Write after write (WAW)*. As Figure 1 shows, host A writes data to another host O, then sends a notification to host B. Send can be considered a write operation. When B receives the notification, it issues read command to O, but may not get the data due to message delays.
- *Independent read, independent write (IRIW)*. Host A writes O_1 and O_2 at the same time, while B reads O_1 and O_2 simultaneously. B may get an inconsistent state where only one of O_1 and O_2 is written.
- *Independent multi-write (IMW)*. Host A scatters writes to both O_1 and O_2. Concurrently, B also scatters writes. The ordering of A's and B's writes at O_1 and O_2 may be different, causing inconsistency.

Ordering hazards affect system performance. To avoid Reordering and WAW hazards, A needs to wait for the first write operation to complete (an RTT to O) before reading from O or sending to B, thus increasing latency and degrading throughput. To avoid IRIW and IMW hazards, application needs locks, centralized sequencers [44] or explicit coordination via logical timestamps [49].

ROMS removes all four hazards above. ROMS ensures a group of messages to be delivered in causal, FIFO and total order. The FIFO property removes Reordering hazard. In WAW hazard, the three messages $A \rightarrow O$, $A \rightarrow B$ and $B \rightarrow O$ are three message scatterings. By FIFO order, $A \rightarrow O$ is ordered before $A \rightarrow B$. By causality, $A \rightarrow B$ is ordered before $B \rightarrow O$. Consequently, $A \rightarrow O$ is before $B \rightarrow O$. Therefore, the write operation is delivered before the read operation, thus avoiding WAW hazard.

In IRIW and IMW hazards, messages $A \rightarrow O_1$ and $A \rightarrow O_2$ belong to scattering S_1, and $B \rightarrow O_1$ and $B \rightarrow O_2$ belong to scattering S_2. By total order property, O_1 delivers $A \rightarrow O_1$ before $B \rightarrow O_1$ if and only if O_2 delivers $A \rightarrow O_2$ before $B \rightarrow O_2$. Therefore, ROMS ensures consistency between metadata and data (IRIW hazard) and consistent history among replicas (IMW hazard).

Total-Order Multicast. Since the dawn of distributed system research [49], there has been extensive research in ordered communication, mostly providing a multicast or broadcast primitive [27]. One line of work leverages logically centralized coordination, *e.g.*, centralized sequencers [40] or a token passed among senders or receivers [30, 46, 75]. As a result, it is challenging to scale the system. Another line of work uses fully distributed coordination, *e.g.*, exchange timestamps among receivers before they start to process messages [49], or aggregate history during message delivery [17]. This causes extra network communication overhead and delay, thus degrading system efficiency. A third line of work assumes a synchronous network, *e.g.*, the block generation interval in Bitcoin blockchain [68] is designed to be higher than the maximum delay among hosts. However, in data center systems, waiting for worst-case delay leads to unacceptable latency.

Critics and proponents of causal and totally ordered communication (CATOCS) have long discussed the pros and cons of such a primitive [10, 19, 81]. ROMS achieves scalability with in-network computation and incurs little overhead, thus removing one of the biggest criticisms of this primitive.

2.2 Reliability

Atomic Multicast and Consensus. The presence of packet loss and failure adds complexity. In Figure 1a, if the write message to O is lost, B will not get the data. In Figure 1b and 1c, if O_2 fails for a short period, updates to O_1 may cause data inconsistency. A fault-tolerant distributed database uses atomic commitment for failure atomicity among shards and consensus for consistency among replicas. 2PC [9] is blocking on coordinator failure, while 3PC [77] cannot recover from network partition. Atomic multicast [11, 42] provides an all-or-nothing guarantee under failures, which is equivalent to consensus [16]. Although atomic multicast and consensus imply total order, they are orthogonal to FIFO and causal order. For example, Zab [42] is stronger than Multi-Paxos [51] because Zab ensures FIFO ordering when coordinator fails, thus removes Reordering hazards. Causal ordering in Figure 1a is also desirable.

Due to the lack of a reliable failure detector [17], most atomic multicast and consensus protocols use voting to guarantee that a quorum of receivers agree on the sequence of messages to be delivered. The votes need to be broadcast to and collected from all hosts, incurring significant communication overhead for hundreds of thousands of hosts in a data center. In addition, Paxos does not allow Byzantine failure of hosts, which is insecure for a multi-tenant data center service with potentially malicious hosts. Achieving Byzantine fault tolerance [15, 47, 53] in an asynchronous network is notably complicated and inefficient [65].

Fast Consensus with Improved Ordering. Recent years witness a trend of co-designing consensus protocols with data center network. The key idea is to provide FIFO and total ordering in the network, going a step further than best-effort ordering in Fast Paxos [45, 52, 66, 71]. Speculative Paxos [74] and NOPaxos [57] use a switch as a centralized sequencer or serialization point. NetPaxos [25, 26] and [24] implement Paxos in switches. Eris [40] proposes in-network concurrency control using switch as a centralized sequencer, which is a scalability bottleneck. To enable cross-shard packet loss recovery, Eris multicasts all operations to all participant shards, introducing network overhead. In addition, when all hosts are correct, an Eris transaction may fail due to packet loss, while subsequent transactions from the same client may succeed, violating exactly once or FIFO property of ROMS. Further, none of these works tolerate Byzantine host failure. NetChain [41] is a strongly-consistent fault-tolerant key-value store in switches, but the switch has very limited storage capacity.

2.3 Use Cases

In this section, we discuss three use cases of ROMS. These cases are used to illustrate how ROMS can help simplify distributed systems construction, and they are by no means the only usage scenarios where ROMS can be useful.

Distributed Independent Transactions. We consider *independent transactions* or *one-shot transactions* [43], in which the transaction involves multiple shards but the input of each shard does not depend on output of other shards. For example, YCSB+T workload for transactional key-value store [28] and the two most frequent transactions in TPC-C benchmark (New-Order and Payment) [23] are independent transactions. General transactions can be divided into independent transactions exactly as in Eris [40]. ROMS enables us to use active replication [82] and implement each independent transaction as a message scattering to involved shards and replicas. Independent transactions can complete in a single round-trip, close to a non-transactional, non-replicated system.

Serializable Log Replication. A canonical application of ROMS is *log replication* [7, 12, 73]. In both strongly [21, 43, 56]

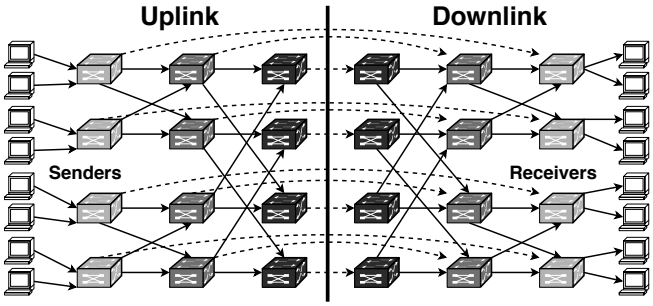


Figure 2. *Routing topology of a typical data center network. Each physical switch is split into two logical switches, one for uplink and one for downlink. The dashed virtual link between corresponding uplink and downlink switch indicates "hairpin" traffic from a lower-layer switch or host to another one.*

and eventually [58, 59, 79] consistent systems, when multiple replicas process write operations in parallel, the consistency among replicas is a paramount challenge due to scalability problem of log serialization [84].

ROMS ensures that each replica receives an identical sequence of write operations from all other replicas. If write operations are blocked until receiving potentially conflicting writes, *i.e.* insert a memory barrier after each write, the system is sequentially consistent because each operation can be serialized at its beginning time [61]. If write operations are returned immediately while changes propagate, because local read operations may occur during propagation, the system is not sequentially consistent, but causality is preserved [79]. Further, applying the Thomas write rule [80] on the timestamps of local and remote updates achieves eventual consistency.

Total Store Ordering in DSM. Recent x86 processors provides a *total store ordering* (TSO) memory model [76] in which each core observes a consistent ordering of writes from all other cores, thus being causally and eventually consistent. TSO reduces synchronization in concurrent programming [67, 78]. Obviously, a distributed shared memory (DSM) system built upon ROMS achieves TSO.

In addition, ROMS enables an efficient and scalable implementation of *memory barrier*. The caller blocks itself and sends a null message to itself via ROMS with timestamp T. Then it unblocks upon receiving the message at time T'. The causality property of ROMS ensures that all messages sent before T are received at T'.

3 Background

3.1 Data Center Network

Modern data centers typically adopt multi-rooted tree topologies [35, 55] to interconnect hundreds of thousands of hosts. In a multi-rooted tree topology, the shortest path between two hosts first goes up to one of the lowest common ancestor switches, then goes down to the destination. Therefore, the routing topology form a directed acyclic graph

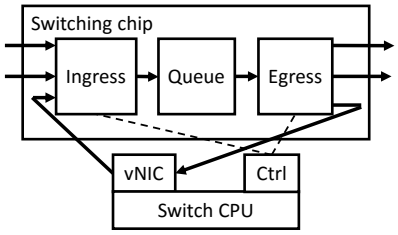


Figure 3. *Architecture of a typical network switch.*

(DAG), as shown in Figure 2. This loop-free topology enables hierarchical aggregation of barrier timestamps.

Apart from the production network, data centers have an isolated management network that interconnects switches. A logically centralized SDN controller [39] runs on the management network to detect failures of switches and links, then reconfigure routing tables on failure [85]. The SDN controller is replicated using Zookeeper [37].

Recent years saw progress in zero queue [20, 31, 72] and lossless [14, 18, 36] data center networks, making packet loss a rare case [74].

3.2 Programmable Switches

A switch is consisted of a switching chip [3, 4] and a CPU (Figure 3). The switch operating system [2] runs on the CPU to control the switching chip (*e.g.* configure chip registers). The switching chip forwards selected traffic (typically control plane traffic, *e.g.*, DHCP and BGP) to the CPU for processing via a virtual NIC.

The switching chip is composed of an ingress pipeline, multiple FIFO queues and an egress pipeline. When a packet is received from an *input link*, it first goes through the ingress pipeline to determine the output link and queueing priority, then get buffered into the corresponding FIFO queue. The egress pipeline pulls packets from queues according to priority, applies header modifications and sends them to *output links*. One key property of the switch is that the queuing model ensures FIFO property for packets with the same ingress and egress port, if the packets have the same priority. In addition, packet corruption on input links and buffer overflow inside switches can be detected, and a packet loss counter is incremented per lost packet.

Data center switches can provide good programmability. Often, the CPU can be used to process (a small amount of) data plane packets [60]. Moreover, the switching chip is becoming more and more reconfigurable in recent years. For example, Tofino chip from Barefoot networks [3] supports flexible packet parsers, reconfigurable pipelines and stateful memory. Users can program Tofino using P4 [13] to achieve *customized per-packet processing*.

Despite good programmability, the data center switch typically has limited buffer memory. The average per-port on-chip buffer is typically hundreds of kilobytes [6] in size. As a result, it is challenging to buffer many packets at the switches in a data center.

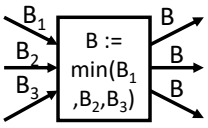


Figure 4. *Illustration of barrier timestamp aggregation.*

4 Reliable Ordering under No Failures

In this section, we introduce how ROMS achieves reliable ordered message scattering in the absence of packet losses and failures. We will introduce how ROMS achieves reliability under packet losses and failures in Sec.5.

To achieve ordered message scattering in an efficient and scalable manner, ROMS exploits the power of programmable switches and separates control plane and data plane. ROMS leverages *every* switch to aggregate order information and attach this information into packets, thus forming a *scalable* control plane. On the data plane, ROMS forwards messages in the network as usual and reorders them at the receiver, thus *efficiently* achieving ordered message delivery.

4.1 Message and Barrier Timestamp

Message timestamp. The ROMS sender[1] assigns a non-decreasing timestamp for each message. A group of messages with an identical timestamp forms a scattering, while different scattering are assigned with different timestamps. Message timestamp determines the delivery order at the receiver. Notice that we distinguish *receiving* a message from *delivering* a message. A message is received by a host and held in a reordering buffer. It is delivered to the application only when ROMS can be sure of the ordering and reliability properties discussed in Sec.1 are satisfied. To achieve ordered message scattering, the receivers should deliver arrival messages in the ascending order of their timestamps (ties are broken through host ID).

Loss-free Barrier timestamp. When a receiver wants to deliver a message with timestamp t, it must be sure that it has received and delivered all the messages whose timestamps are smaller than t. This is challenging since different network paths have different propagation and queuing delays.

To address this issue, we introduce the concept of *loss-free barrier timestamp*. For a link in the routing graph, its barrier timestamp is the *lower bound* of message timestamps of all *possible future arrival messages* from the link. We also generalize the barrier timestamp notion to a node (i.e. a host or a switch), which is just the minimum value of all the barrier timestamps of its incoming links. A receiver should maintain its barrier timestamp and only deliver the messages whose timestamps are smaller than the barrier timestamp.

Due to the non-decreasing timestamp assignment, in a FIFO network (such as by using TCP transport), a receiver can easily figure out the loss-free barrier if it has received messages from *all* the possible senders. The loss-free barrier of the receiver is the minimum timestamp of the latest messages from each sender to the receiver. Therefore, a naïve solution would be for every sender to send periodic beacons to every receiver so that the receivers can figure out the loss-free barrier and deliver messages. Unfortunately, such a solution requires quadratic beacons, thus introducing high bandwidth overhead in production data centers.

Hierarchical barrier timestamp aggregation. To scale in production data centers, ROMS leverages programmable switches to *aggregate* the loss-free barrier timestamp information. Given the limited switch buffer resource, ROMS does not buffer and reorder messages in the network. Instead, ROMS forwards messages in the network as usual, but reorders them at the receiver side based on the loss-free barrier provided by the switch.

In ROMS, we attach two timestamp fields to each message packet. The first is *message timestamp* field, which is set by the sender and will not be modified. The second field is *loss-free barrier timestamp*, which is initialized by the sender but will be modified by switches along the network path. The property of the loss-free barrier field is:

When a switch or a host receives a packet with loss-free barrier timestamp B from a network link L, this indicates that the message timestamp and loss-free barrier timestamp of future arrival packets from link L will be larger than B.

A sender initializes both fields of all packets in a message with the non-decreasing wall clock timestamp. To derive loss-free barrier timestamps, a switch maintains a register R_i for each input link $i \in \mathcal{I}$, where $\mathcal{I}$ is the set of all input links. After forwarding a packet with barrier timestamp B from input link i to output link o, the switch performs two updates. First, it updates register $R_i := B$. Second, it modifies the barrier timestamp of the packet to $B_{new} := \min\{R_i | i \in \mathcal{I}\}$.

Each switch independently derives the loss-free barrier timestamp based on its input links, as Figure 4 shows. Barrier timestamps are aggregated hierarchically through layers of switches hop-by-hop, and finally the receiver gets the barrier of all reachable hosts and links in the network. This algorithm maintains the property of barrier timestamps, given the FIFO property of each network link and switch.

The receiver buffers received packets in a priority queue that sorts packets based on the message timestamp. When a receiver receives a packet with loss-free barrier B, it knows that the message timestamp of all future arrival packets will be larger than B. Hence, it delivers all buffered packets with message timestamp below B for processing.

4.2 Beacons on Idle Links

As shown before, at each hop, the per-packet loss-free barrier is updated to the minimum loss-free barrier value of *all possible* input links. As a result, an idle link can keep the per-packet loss-free barrier stalled, thus throttling the whole system. To avoid idle links, we send *beacons* periodically on every link.

[1]The sender/receiver is an application running on a host.