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]USTC [3]University of Washington

* Co-first authors

Presenter: Bojie Li

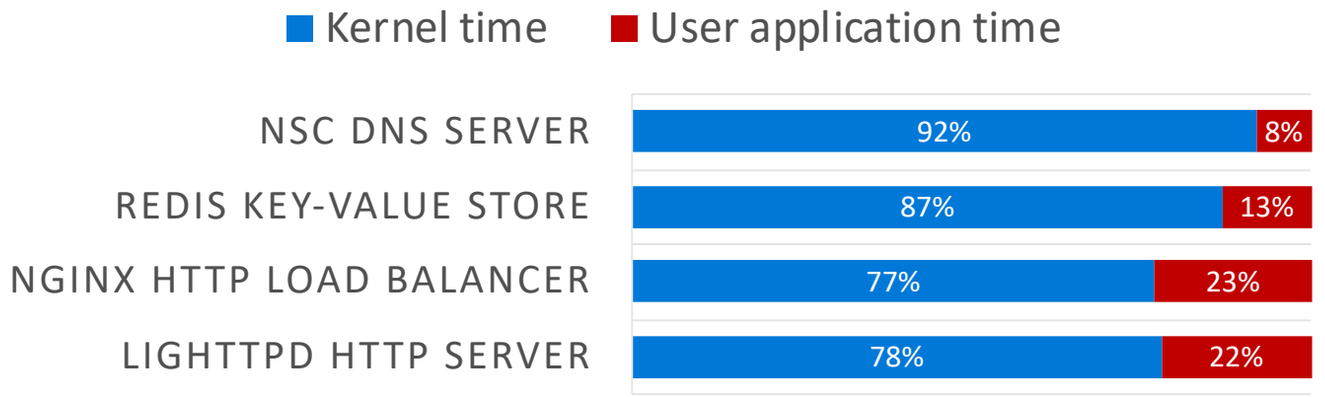
Distributed and Parallel Software Lab, Central Software Institute, Huawei 2012 Labs

The Socket Communication Primitive

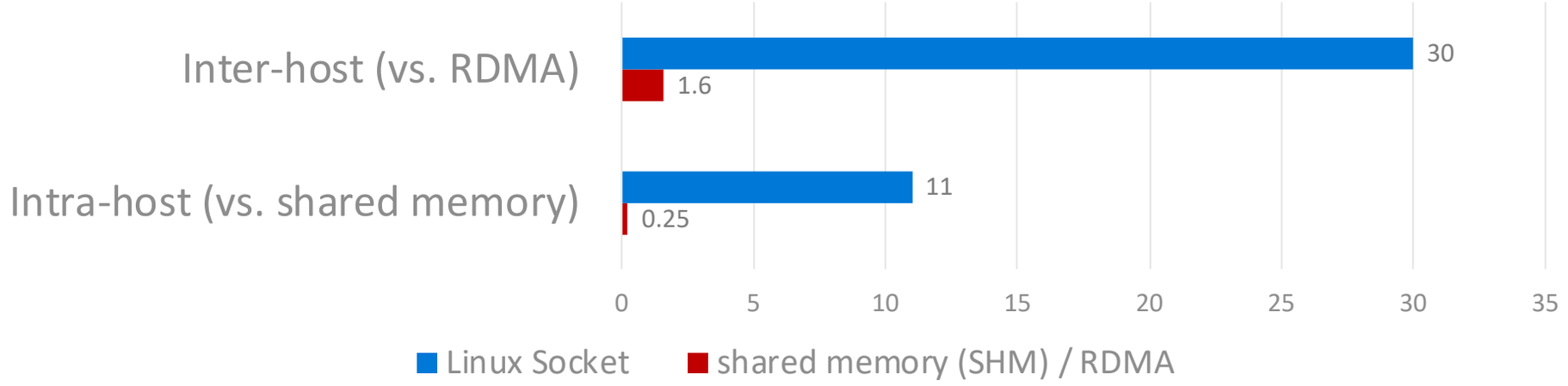
Server	Client
socket()	socket()
bind()	connect()
listen()	send()
accept()	recv()
recv()	close()
send()	
close()	

Socket: a Performance Bottleneck

Socket syscall time >> user application time



Socket latency >> hardware transport latency

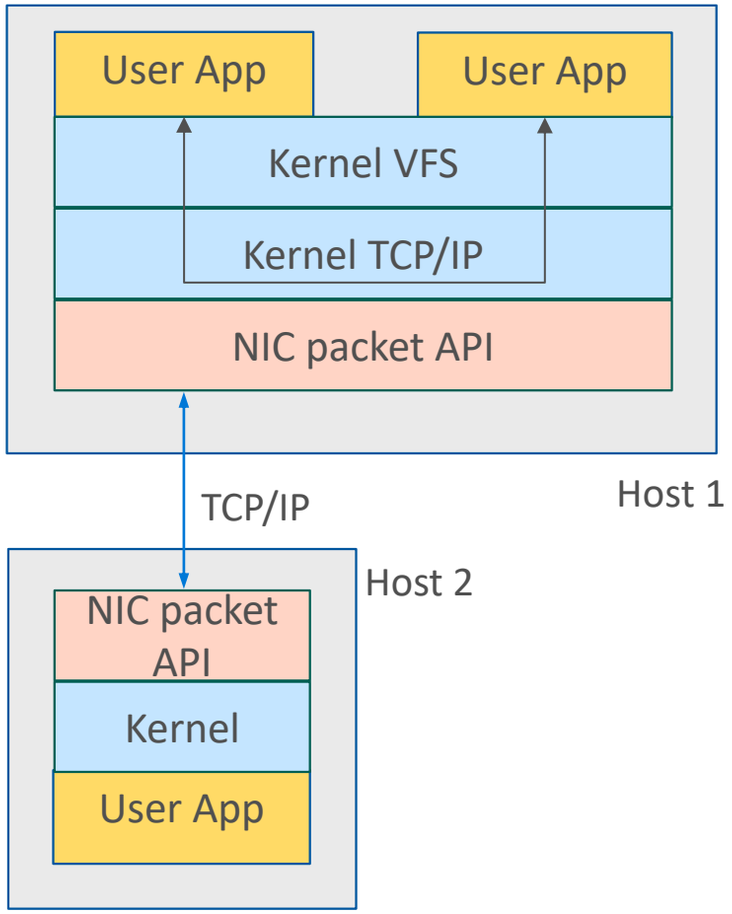


Timeline of the SocksDirect Project

2017/04	Start research on high-performance socket, IPC and container networking.
2017/07	IPC-Direct: Fast and Compatible Container Networking won the most impactful award and global 1st place in Cloud & Enterprise category in Microsoft Hackathon 2017.
2017/08	Submitted a poster to SOSP'17 Student Research Competition, admitted in final presentation rounds.
2018/04	First submission to OSDI'18 but rejected in first-round review. Major reasons: difference from related work; presentation is not clear.
2018/09	Second submission to NSDI'19 but rejected in first-round review. Major reasons: design is too complicated; evaluation not solid.
2019/01	Third submission to SIGCOMM'19 and accepted. Improve discussion on related work; remove unnecessary goals and simplify design; make the prototype work on real applications.

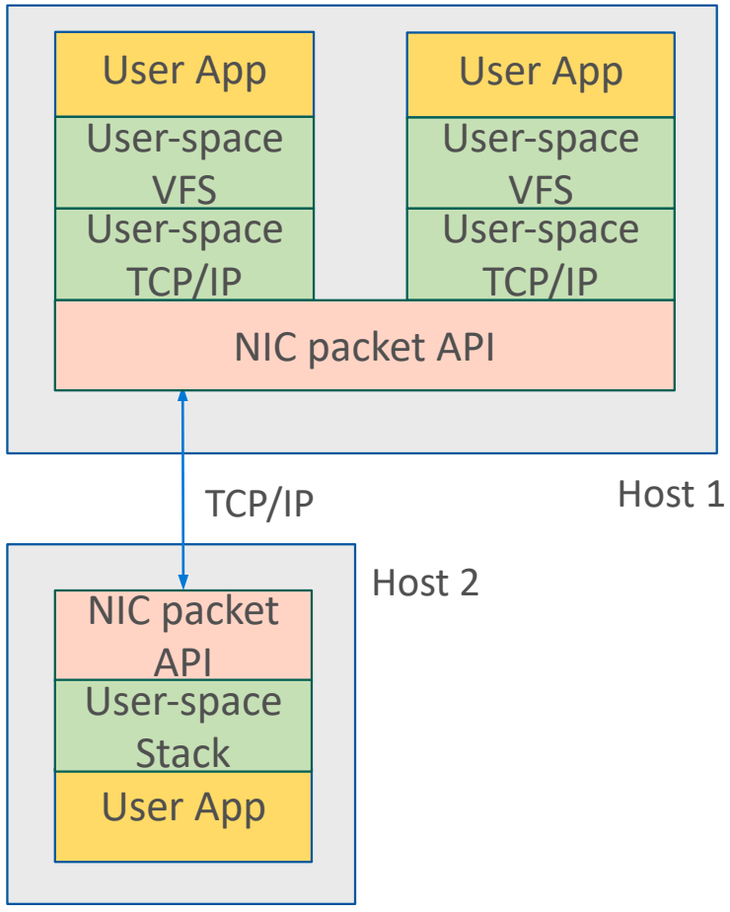
High Performance Socket Systems

Kernel Optimization



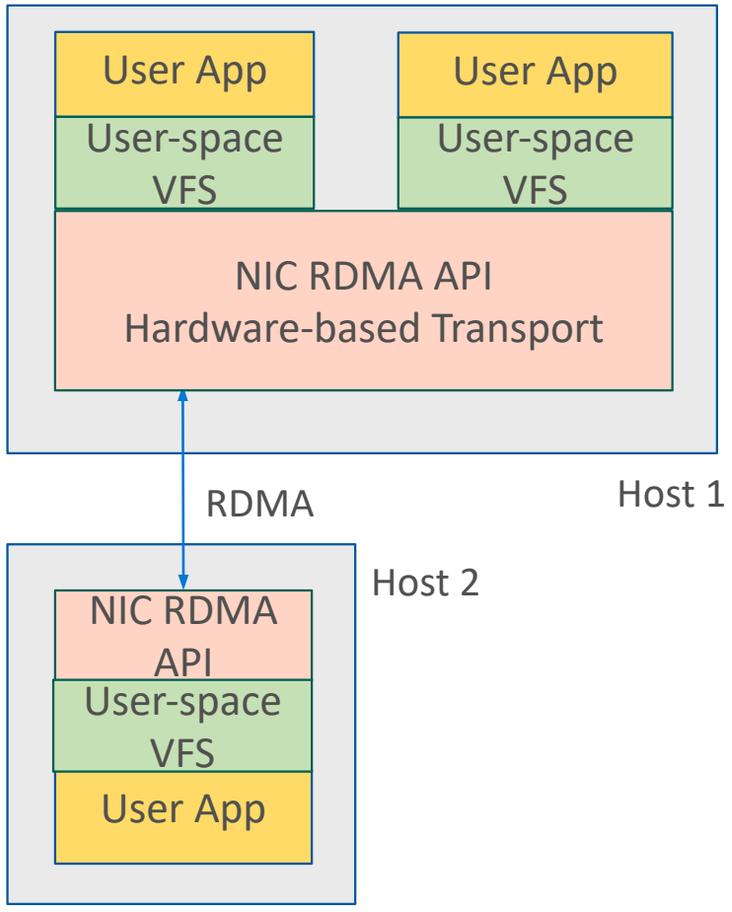
FastSocket, Megapipe, StackMap…

User-space TCP/IP



IX, Arrakis, SandStorm, mTCP, LibVMA, OpenOnload…

Offload to RDMA NIC



Rsocket, SDP, FreeFlow…

Takeaways: related work

- OSDI'18 Review: On the inter-server socket side, there have already been several proposals in socket over RDMA, including SDP, libvma, rsocket, OpenUCX, and UNH EXS. Among these, SDP, UNH EXS, rsocket all support zero copy (in some of their modes). There are no performance comparison with these solutions or related work discussion of SocksDirect's difference. There are also a bunch of user-level TCP/IP implementation.
- OSDI'18 Review: There has been a whole lot of work on high-performance stacks… The related work (and Table 3) would do better to contrast with these in terms of design rather than features… You mention highly similar systems like mTCP, but completely fail to make any characterization about what insights or lessons you've had that are different than theirs. SDP has highly similar goals to yours, and contrary to what you claim in the related work section was designed to be transparent to applications.
- OSDI'18 Review: Zero copy or shared memory communication isn't new. Note that even recent Linux TCP stacks support zero copy.
- SIGCOMM'19 PC Meeting Summary: We also feel that you need to relate your work to LITE (Tsai and Zhang, SOSP'17). While LITE does not allow transparent substitution, it has similar goals and techniques to your work.

Takeaways: related work / motivation

- NSDI’19 Review: Another doubt I have is about the need to transparently support socket functionality in this richer set of scenarios. The paper makes the point that previous approaches do not work across fork, or that they do not support intra-machine communication. … But a practical question here is whether this support for full semantics is actually needed. I didn't see the compelling applications that do not work on previous approaches because such support is lacking.
- NSDI’19 Review: Similarly, I strongly suspect that another reason that previous work has not focused on intra-machine performance is that the application scenarios haven't demanded it. … can you start the introduction by listing important applications that cannot use previous high-performance networking stacks because they lack support for fork, intra-machine communication, etc.?

Takeaways: related work / motivation

- OSDI'18 Review: I did not understand why we should use a TCP socket for intra-machine communication instead of a unix domain socket or a pipe. The latter performs much better on Linux, but evaluation seems to miss this point.
- NSDI'19 Review: I did not find the corresponding motivation for accelerating intra-host communication in this paper. You might want to show that (1) the performance of intra-host communication is critical for many applications and (2) what performance problems that existing kernel stacks have. The paper does not even show and compare the performance of unix domain sockets or pipes, and simply jumps on improving TCP sockets for intra-host communication.
- OSDI'18 Review: Why do you use TCP sockets for intra-machine connections? Why not use a unix domain socket or a pipe? The following page says that using unix domain sockets or pipe() brings 3x better latency and produces 7x better throughput than a TCP connection. …
- OSDI'18 Review: Using shared memory for IPC is common. The authors use it for socket communication across processes/threads.

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.

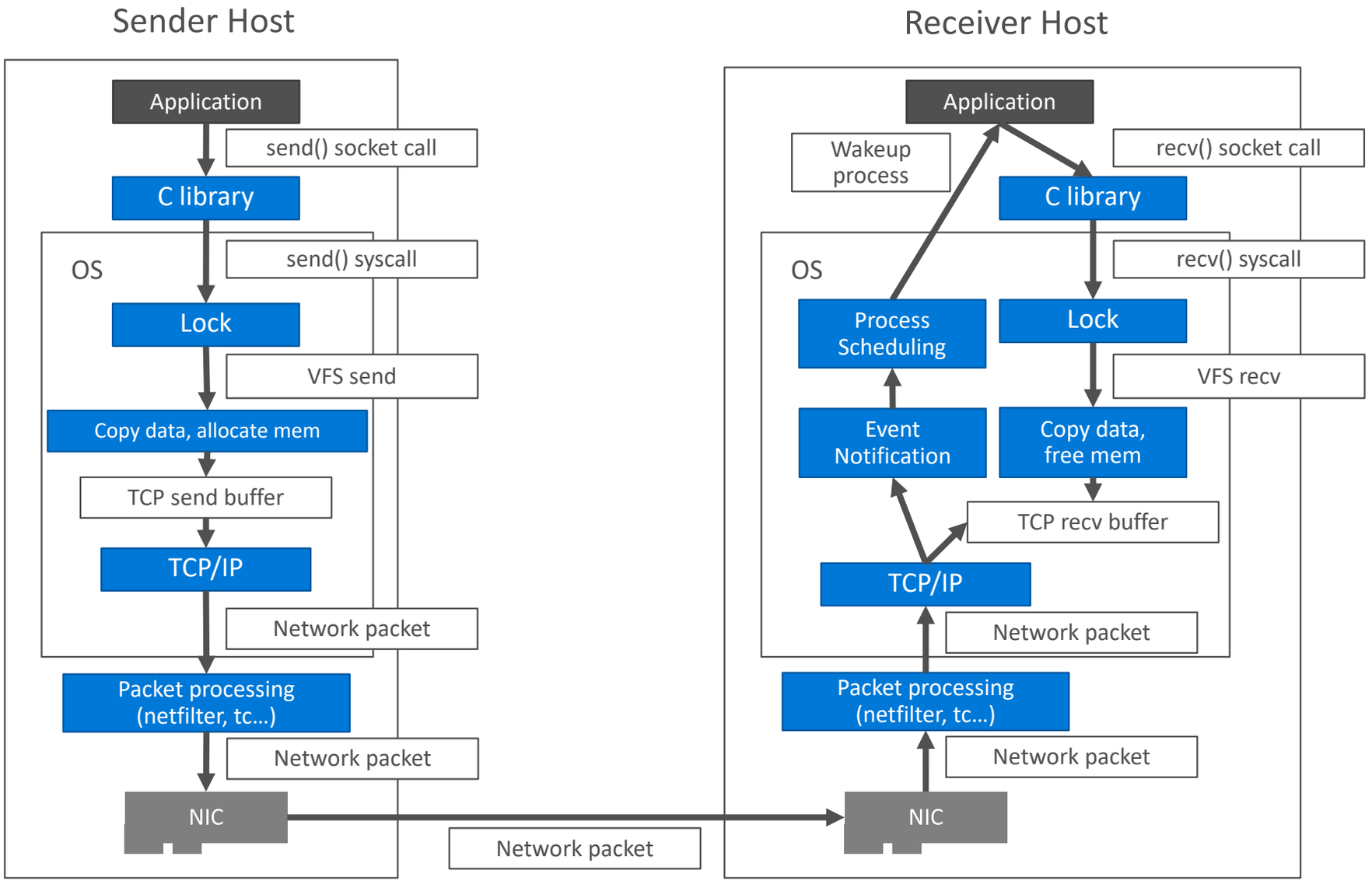
In the following, we classify socket overheads as Table 1 shows.

2.1.1 *Per Operation Overheads.*

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.

Linux Socket: from send() to recv()



Round-Trip Time Breakdown

Type	Overhead (ns)	Linux		LibVMA		RSocket		SocksDirect	
		Inter-host	Intra-host	Inter-host	Intra-host	Inter-host	Intra-host	Inter-host	Intra-host
Per operation	**Total**	**413**		**177**		**209**		**53**	
	C library shim	12		10		10		15	
	Kernel crossing (syscall)	205		N/A		N/A		N/A	
	Socket FD locking	160		121		138		N/A	
Per packet	**Total**	**15000**	**5800**	**2200**	**1300**	**1700**	**1000**	**850**	**150**
	Buffer management	430		320		370		50	
	TCP/IP protocol	360		260		N/A		N/A	
	Packet processing	500	N/A	130		N/A		N/A	
	NIC doorbell and DMA	2100	N/A	900	450	900	450	600	N/A
	NIC processing and wire	200	N/A	200	N/A	200	N/A	200	N/A
	Handling NIC interrupt	4000	N/A	N/A		N/A		N/A	
	Process wakeup	5000		N/A		N/A		N/A	
Per kbyte	**Total**	**365**	**160**	**540**	**381**	**239**	**212**	**173**	**13**
	Copy	160		320		160		13	
	Wire transfer	160	N/A	160	N/A	160	N/A	160	N/A

Takeaways: performance breakdown

- SIGCOMM’19 Review: I'd like to see experimental results that don't use the page-remapping technique, plus more details of how this can be managed securely.
- SIGCOMM’19 Review: … it fails to highlight how the various pieces to the solution contribute to the performance gains. I would have liked to see the performance gains dissected in more details.
- SIGCOMM’19 Review: I would expect that there are some performance vs. compatibility trade-offs here but the paper fails to highlight them and quantify where certain performance gaps existing.
- SIGCOMM’19 PC Meeting Summary: There's a lot of engineering work in your solution, but the evaluation doesn't let us separate the relative costs and benefits of the individual choices.

SocksDirect Design Goals

Compatibility

- Drop-in replacement, no application modification

Isolation

- Security isolation among containers and applications
- Enforce access control policies

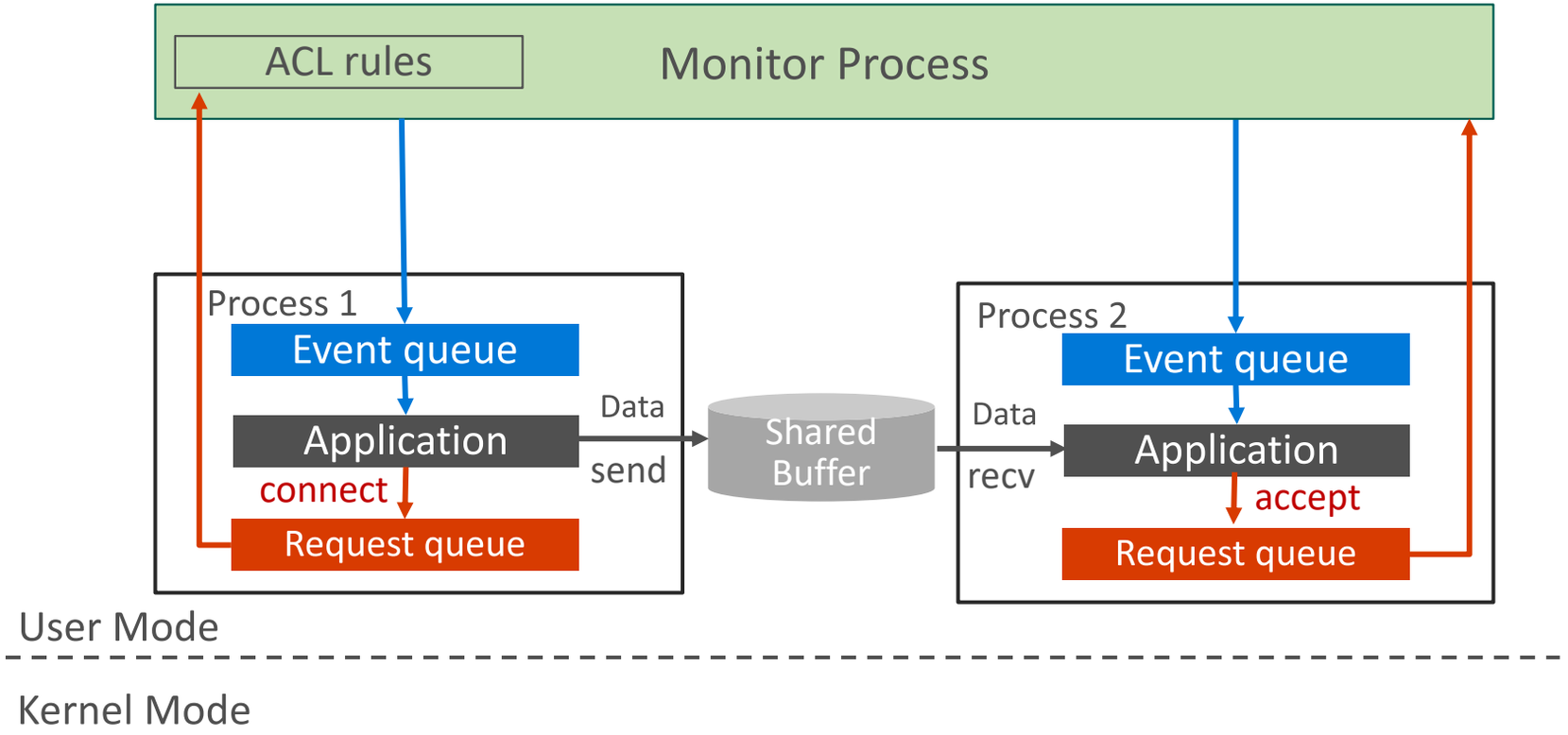
High Performance

- High throughput
- Low latency
- Scalable with number of CPU cores
- ~~Scalable with number of concurrent connections~~

Takeaways: remove unnecessary design goals which lead to a complicated design

- NSDI'19 Review: Misleading claim. The intro says that the proposed system maintains good performance even with millions of concurrent connections, which sounds very nice. I later find out that this is for supporting millions of connections between TWO processes/threads as they multiplex a single queue for communication. But why is that important? What matters in practice is to support a large number of concurrent connections with millions of remote clients, where you can't share a queue.
- NSDI'19 Review: What is the usage scenario for supporting a million simultaneous connections between two processes over RDMA? The million connections scenario is a web server scenario, which is going to have a (relatively) small number of server processes and a huge number of external clients, and they will be using TCP since it's not internal to a data center.
- The real problem in datacenter that SocksDirect did not solve: one storage/DB server with thousands of storage/DB clients, where the traffic is bursty.

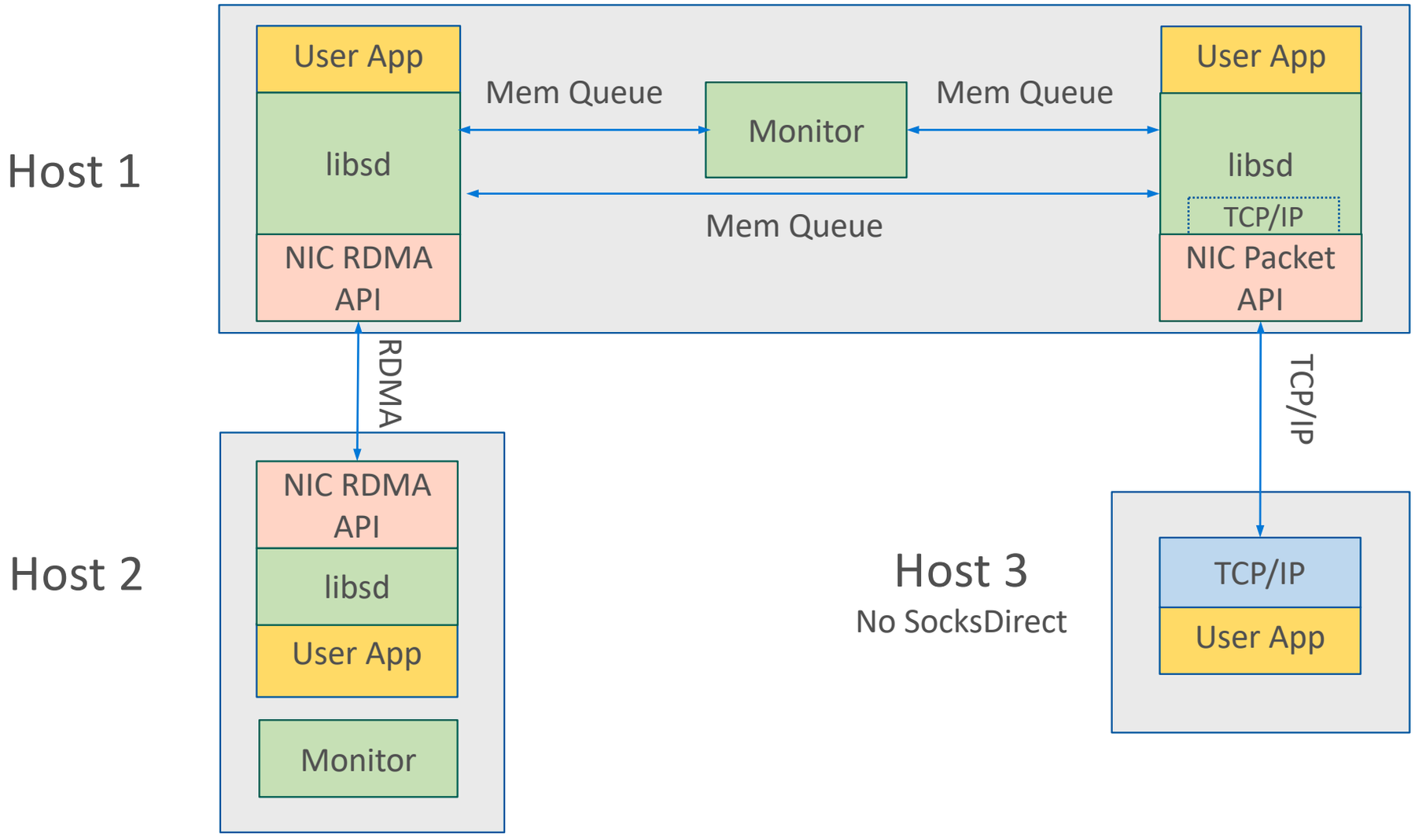
SocksDirect: Fast and Compatible Socket in User Space



Monitor: user-space daemon process to coordinate global resources and enforce ACL rules.

Processes as a shared-nothing distributed system: use message passing over shared-memory queues.

SocksDirect Supports Different Transports for Data



Remove the Overheads (1)

Type	Overhead	Linux RTT (ns)		SocksDirect RTT (ns)	
		Inter-host	Intra-host	Inter-host	Intra-host
Per operation	**Total**	**413**		**53**	
	C library shim	15		15	
	Kernel crossing (syscall)	205		N/A	
	Socket FD locking	160		N/A	
Per packet	**Total**	**15000**	**5800**	**850**	**150**
	Buffer management	430		50	
	TCP/IP protocol	360		N/A	
	Packet processing	500	N/A	N/A	
	NIC doorbell and DMA	2100	N/A	600	N/A
	NIC processing and wire	200	N/A	200	N/A
	Handling NIC interrupt	4000	N/A	N/A	
	Process wakeup	5000		N/A	
Per kbyte	**Total**	**365**	**160**	**173**	**13**
	Copy	160		13	
	Wire transfer	160	N/A	160	N/A

Remove the Overheads (2)

Type	Overhead	Linux RTT (ns)		SocksDirect RTT (ns)	
		Inter-host	Intra-host	Inter-host	Intra-host
Per operation	**Total**	**413**		**53**	
	C library shim	15		15	
	Kernel crossing (syscall)	205		N/A	
	Socket FD locking	160		N/A	
Per packet	**Total**	**15000**	**5800**	**850**	**150**
	Buffer management	430		50	
	TCP/IP protocol	360		N/A	
	Packet processing	500	N/A	N/A	
	NIC doorbell and DMA	2100	N/A	600	N/A
	NIC processing and wire	200	N/A	200	N/A
	Handling NIC interrupt	4000	N/A	N/A	
	Process wakeup	5000		N/A	
Per kbyte	**Total**	**365**	**160**	**173**	**13**
	Copy	160		13	
	Wire transfer	160	N/A	160	N/A

Takeaways: explain application scenario

Example: why need locking?

- NSDI'19 Review: It is nice to handle sharing a socket with multiple senders or multiple receivers. However, while required for correct operation, such communication pattern is pretty unusual. Except for a popular scenario that shares a listening socket with multiple processes/threads, I think it would be rare for multiple senders to share a socket to send messages concurrently or the other way around (multiple receivers contend to receive messages simultaneously). If not, please explain which application does this.

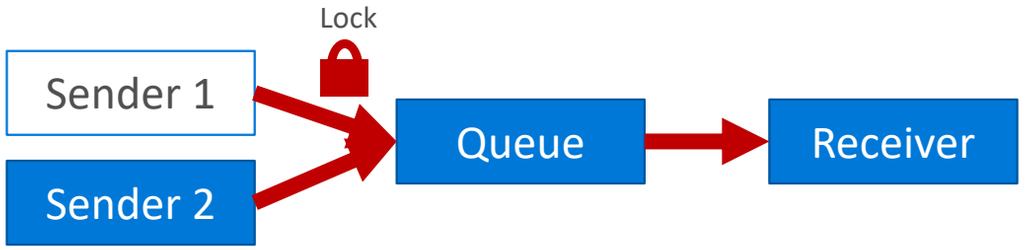
Token-based Socket Sharing

Socket is shared among threads
and forked processes.

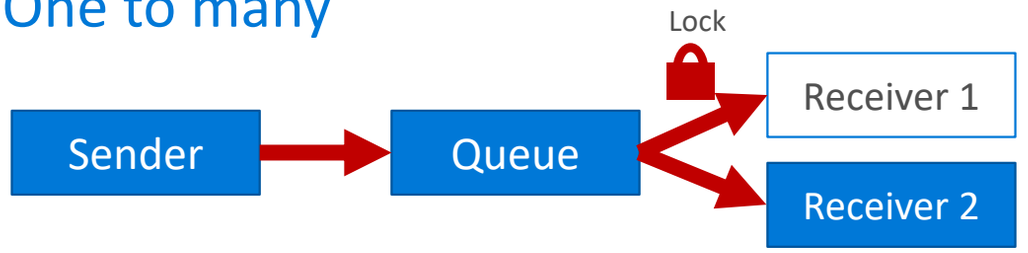
Optimize for common cases.

Be correct for all cases.

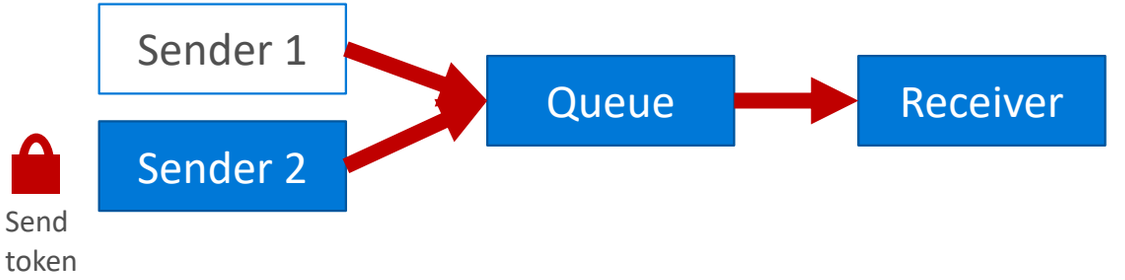
Many to one



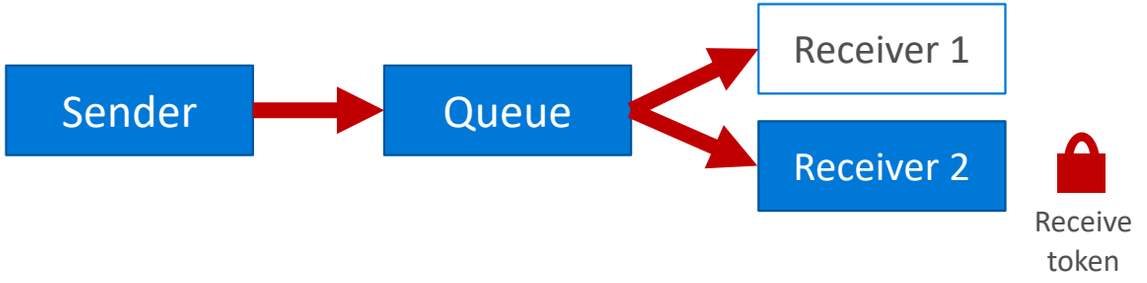
One to many



Many to one

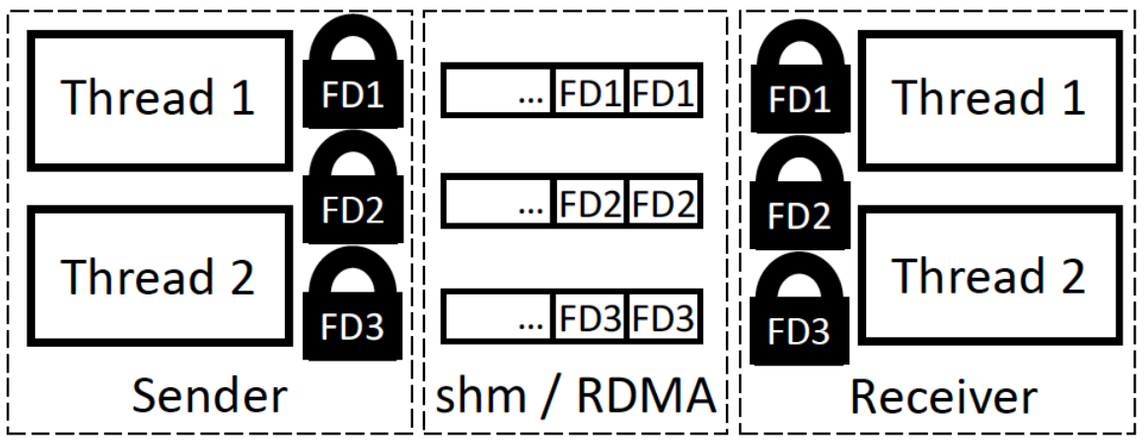


One to many



Transfer ownership of tokens via monitor.

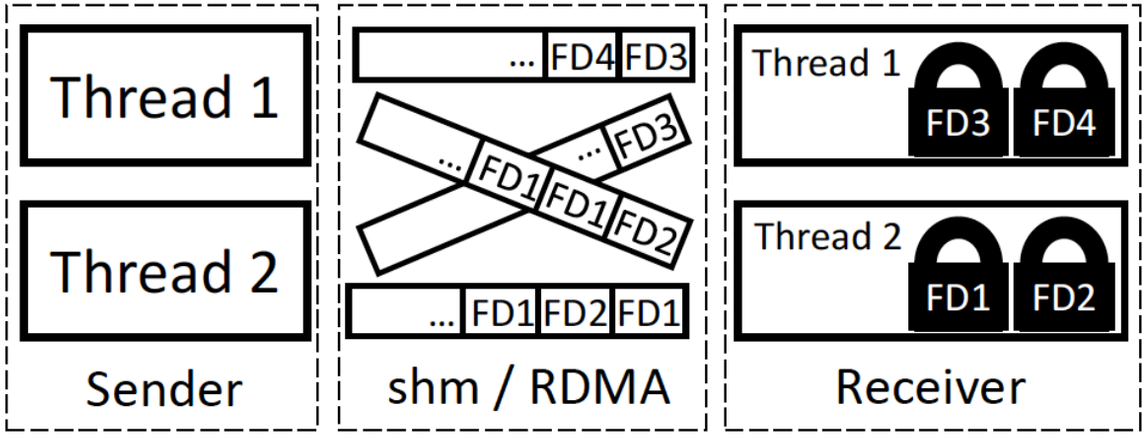
Takeaways: remove unnecessary design goals which lead to a complicated design



Traditional queue

1 FD = 1 queue

All threads share a queue



Multiplexed queue in previous submissions

N FDs share 1 queue

1 queue per send/recv thread pair

- OSDI'18 Review: Not sure what's the benefit of a single queue for all connections of two communicating machines? I think such a design choice makes the implementation unnecessarily complex (e.g., picking from middle of queue). You might want to compare the performance of a single aggregate queue vs. a per-connection queue.

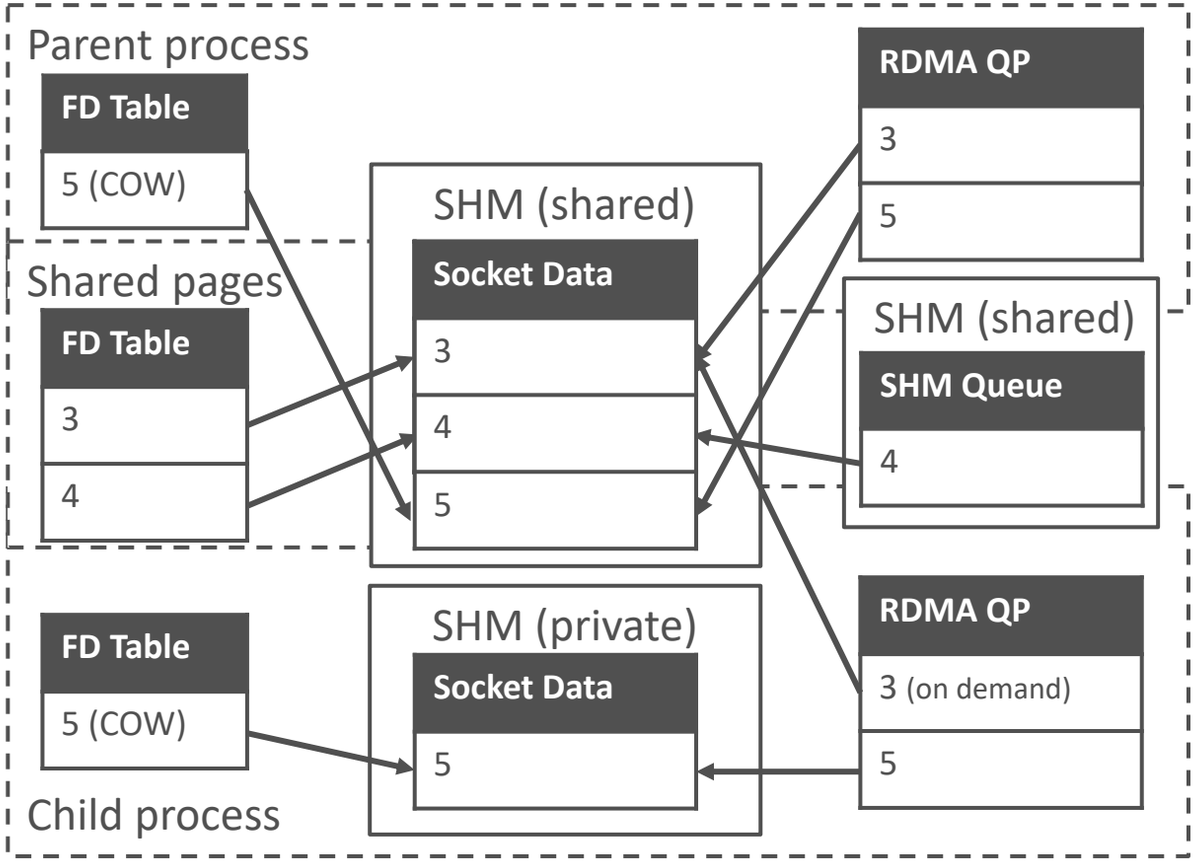
Handling Fork

Linux semantics requirement:

- File descriptors need to be sequential (1,2,3,4,5…).
- Sockets are shared for parent and child processes.

Transport limitations:

- RDMA QP cannot be shared among processes.



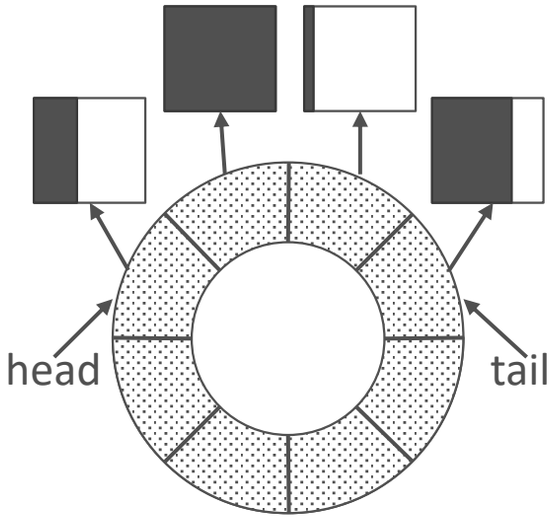
Takeaways: make trade-offs

- OSDI'18 Review: The Linux socket semantics is that "The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process." (http://man7.org/linux/man-pages/man2/socket.2.html), which would no longer be true. This change would break (potentially buggy) applications that rely on low-numbered socket descriptors as an index to an array.
- In NSDI'19 submission, we found that Redis and Memcached rely on the lowest FD behavior.
- NSDI'19 Review: In section 3.2, your system maintains an FD translation table to be compatible with the POSIX standard that returns the lowest unused FD for a new FD. While this maintains the compatibility with existing applications, it incurs a high overhead on a busy server in practice. … Note that other high-performance TCP/IP stacks like MegaPipe and mTCP intentionally avoid this overhead by not conforming to the requirement.
- In SIGCOMM'19 submission, we no longer claim socket establishment throughput because it is not important at all in datacenters! (Remove unnecessary design goals)

Remove the Overheads (3)

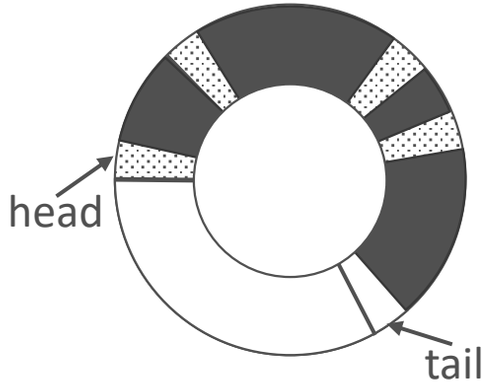
Type	Overhead	Linux RTT (ns)		SocksDirect RTT (ns)	
		Inter-host	Intra-host	Inter-host	Intra-host
Per operation	**Total**	**413**		**53**	
	C library shim	15		15	
	Kernel crossing (syscall)	205		N/A	
	Socket FD locking	160		N/A	
Per packet	**Total**	**15000**	**5800**	**850**	**150**
	Buffer management	430		50	
	TCP/IP protocol	360		N/A	
	Packet processing	500	N/A	N/A	
	NIC doorbell and DMA	2100	N/A	600	N/A
	NIC processing and wire	200	N/A	200	N/A
	Handling NIC interrupt	4000	N/A	N/A	
	Process wakeup	5000		N/A	
Per kbyte	**Total**	**365**	**160**	**173**	**13**
	Copy	160		13	
	Wire transfer	160	N/A	160	N/A

Per-socket Ring Buffer



Traditional ring buffer

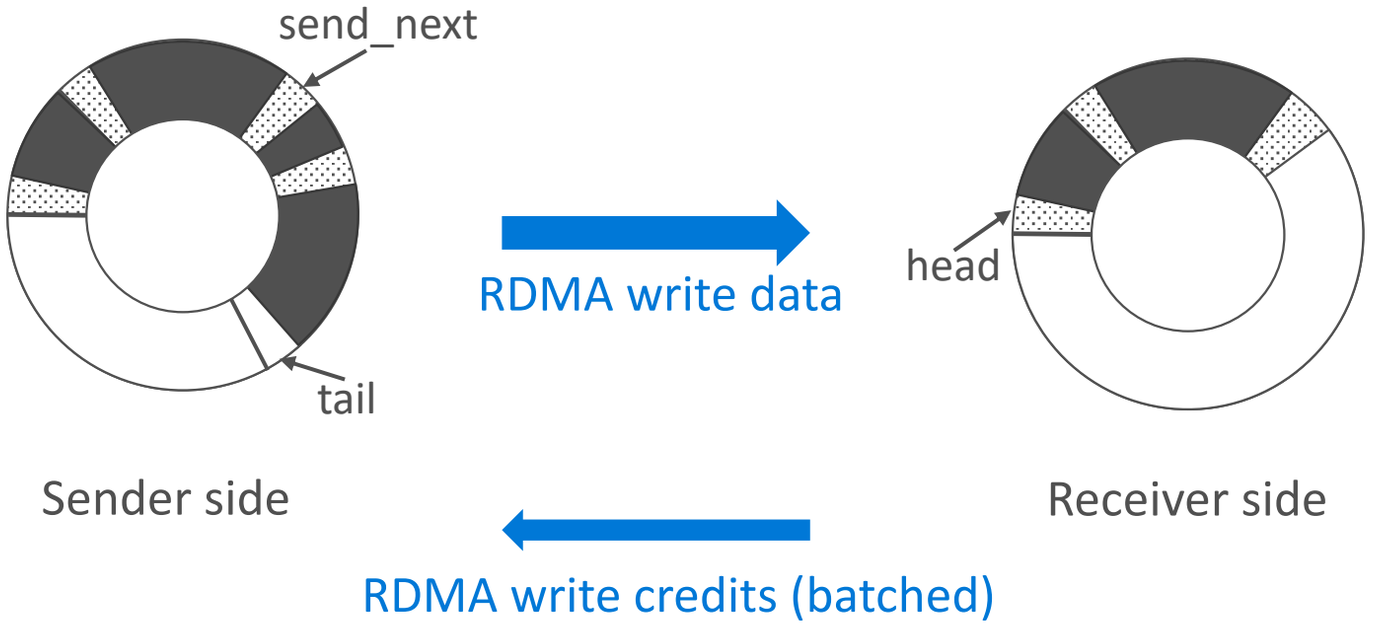
- Many sockets share a ring buffer
- Receiver segregates packets from the NIC
- Buffer allocation overhead
- Internal fragmentation



SocksDirect per-socket ring buffer

- One ring buffer per socket
- Sender segregates packets via RDMA or SHM address
- Back-to-back packet placement
- Minimize buffer mgmt. overhead

Per-socket Ring Buffer



Two copies of ring buffers on both sender and receiver.

Use one-sided RDMA write to synchronize data from sender to receiver, and return credits (i.e. free buffer size) in batches.

Use RDMA write with immediate verb to ensure ordering and use a shared completion queue to amortize polling overhead.

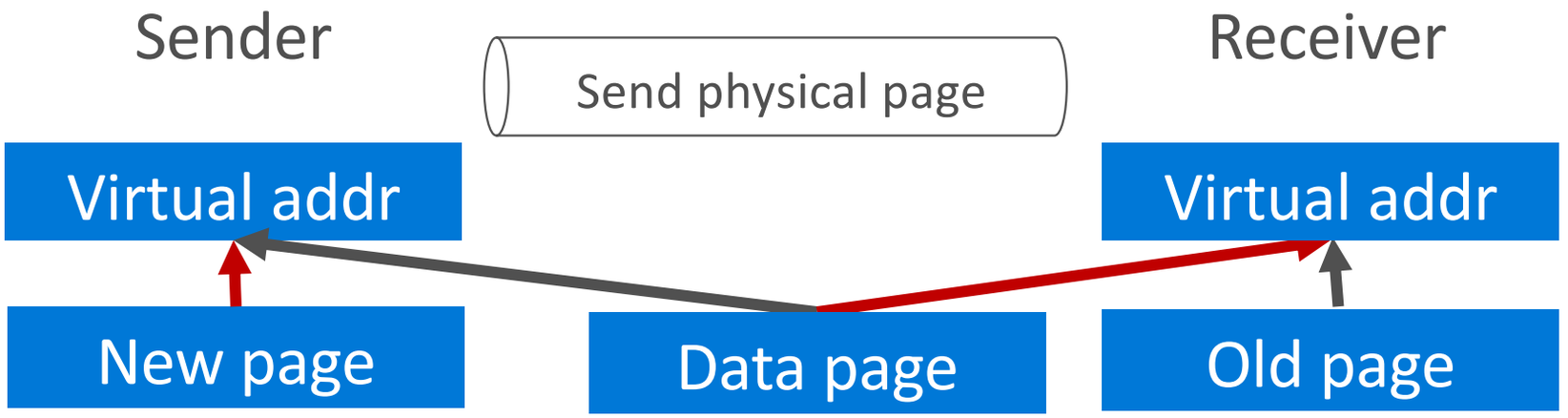
Takeaways: refine design according to reviews

- NSDI'19 Review: How does SocksDirect recover when the other endpoint fails?
- Our approach: two copies of ring buffers on both sender and receiver. (It also enables batching.)

Remove the Overheads (4)

Type	Overhead	Linux RTT (ns)		SocksDirect RTT (ns)	
		Inter-host	Intra-host	Inter-host	Intra-host
Per operation	**Total**	**413**		**53**	
	C library shim	15		15	
	Kernel crossing (syscall)	205		N/A	
	Socket FD locking	160		N/A	
Per packet	**Total**	**15000**	**5800**	**850**	**150**
	Buffer management	430		50	
	TCP/IP protocol	360		N/A	
	Packet processing	500	N/A	N/A	
	NIC doorbell and DMA	2100	N/A	600	N/A
	NIC processing and wire	200	N/A	200	N/A
	Handling NIC interrupt	4000	N/A	N/A	
	Process wakeup	5000		N/A	
Per kbyte	**Total**	**365**	**160**	**173**	**13**
	Copy	160		13	
	Wire transfer	160	N/A	160	N/A

Zero Copy via Page Remapping



Problem: page remapping needs syscalls!

- Map 1 page: 0.78 us
- Copy 1 page: 0.40 us

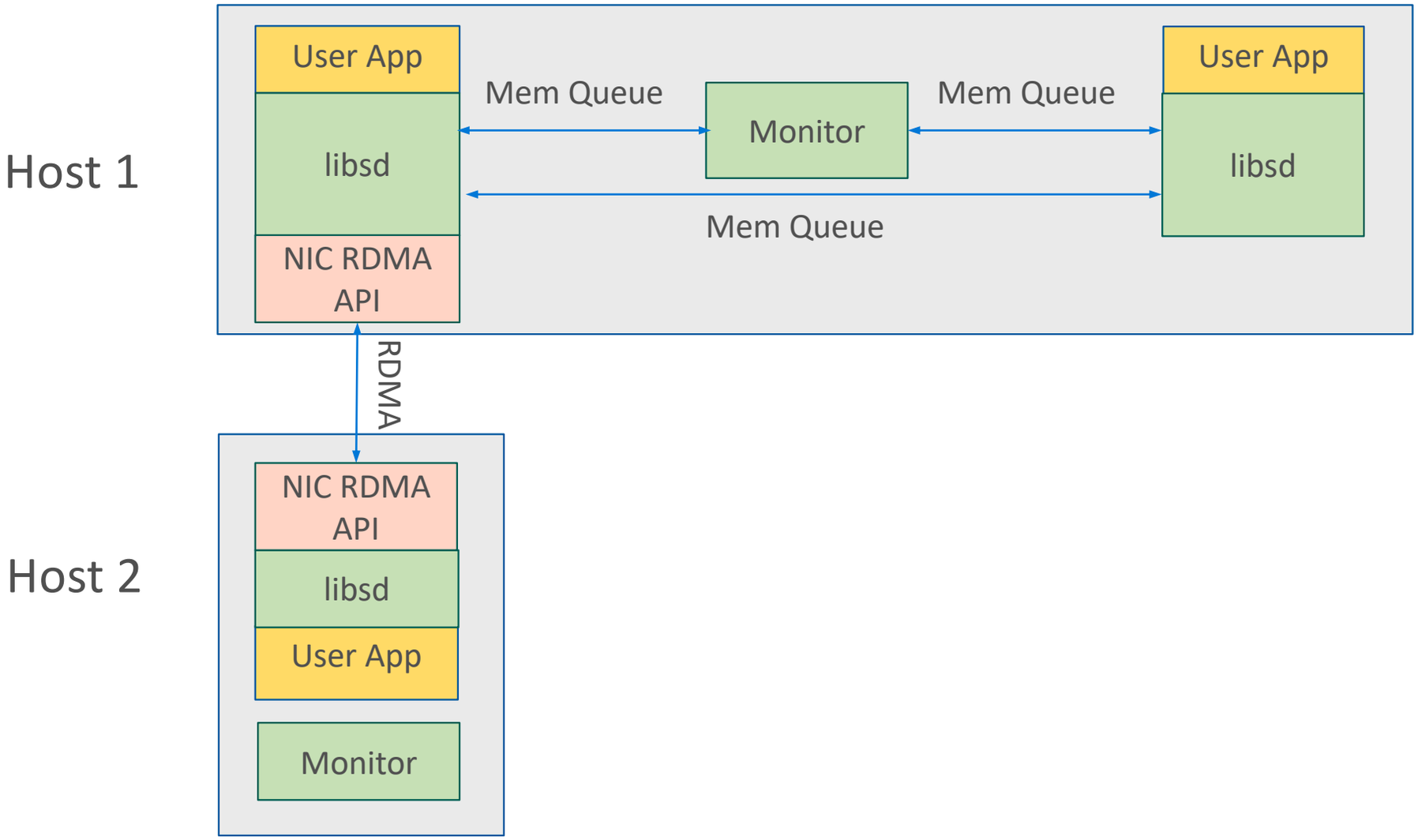
Solution: batch page remapping for large messages

- Map 32 pages: 1.2 us
- Copy 32 pages: 13.0 us

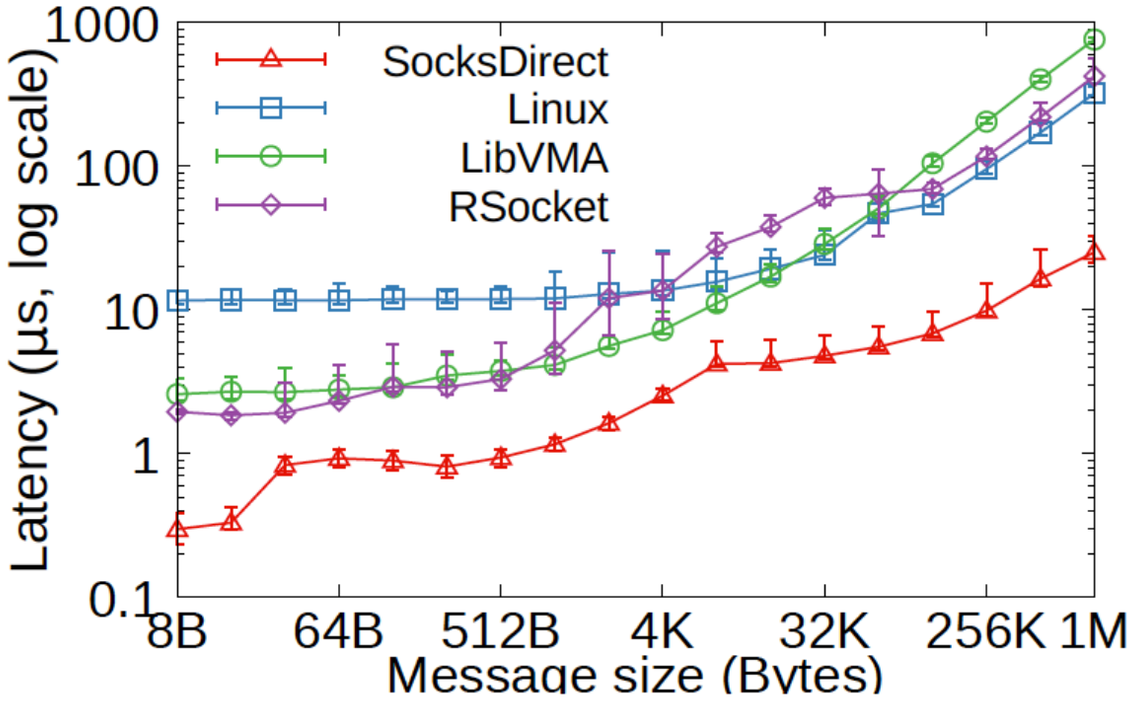
Summary: Overheads & Techniques

1. Kernel crossing (syscall)
   Monitor and library in user space.
   Shared-nothing, use message passing for communication.
2. TCP/IP protocol
   Use hardware-based transports: RDMA / SHM.
3. Locking of socket FDs
   Token-based socket sharing.
   Optimize common cases, prepare for all cases.
4. Buffer management
   Per-socket ring buffer.
5. Payload copy
   Batch page remapping.
6. Process wakeup
   Cooperative context switch.

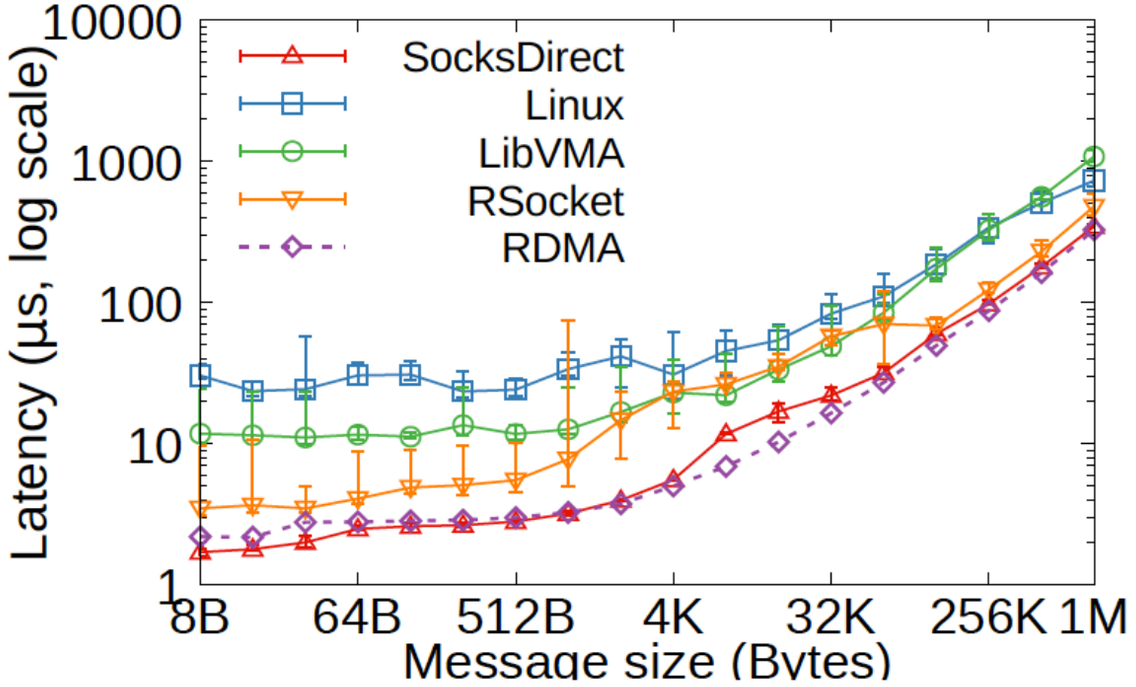
Evaluation Setting



Latency

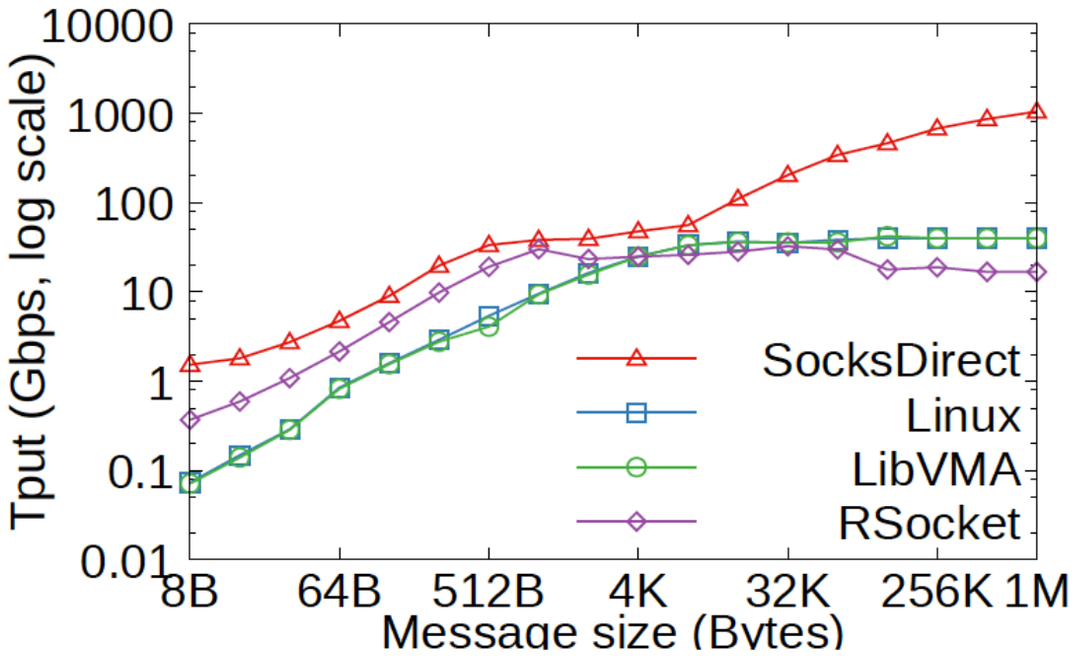


Intra-host

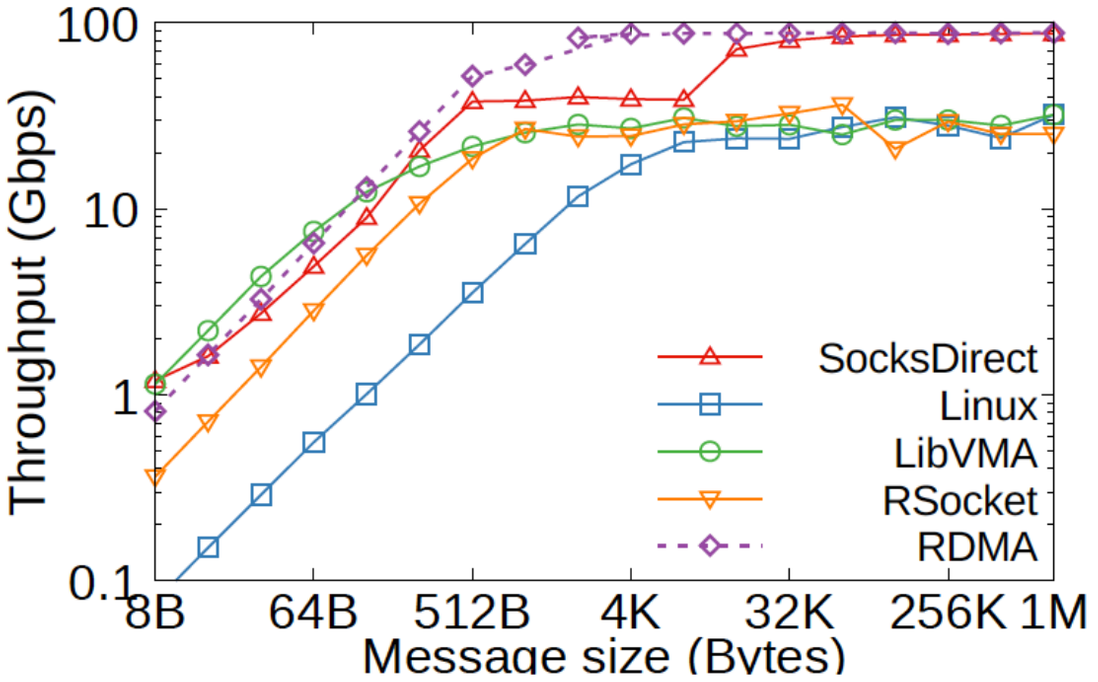


Inter-host

Throughput

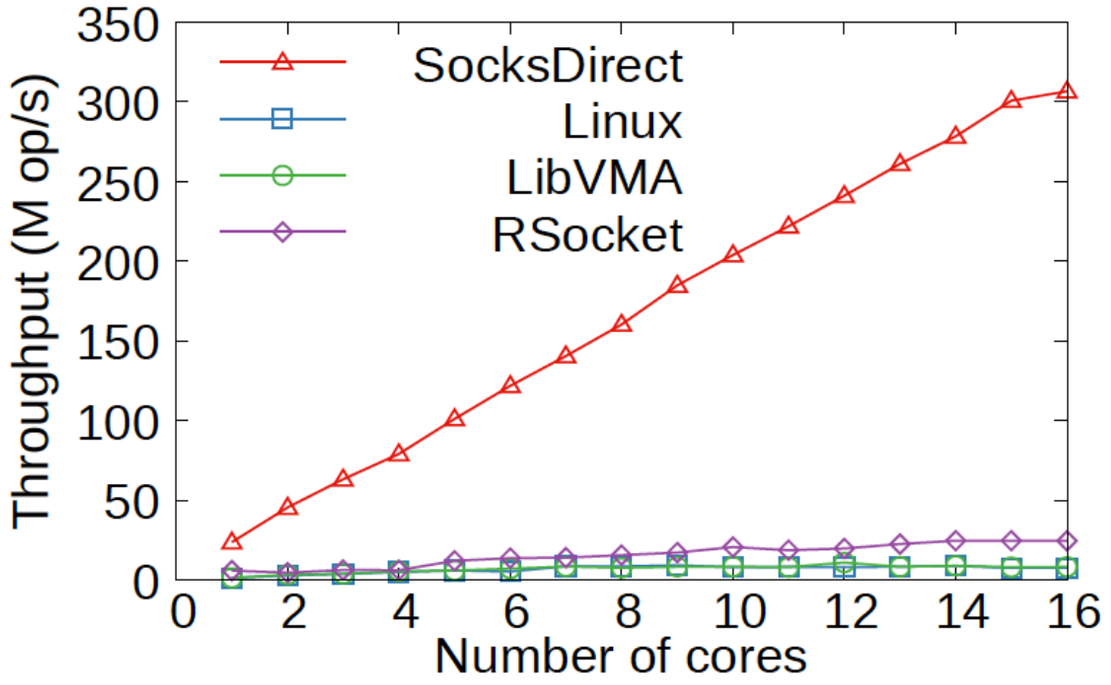


Intra-host

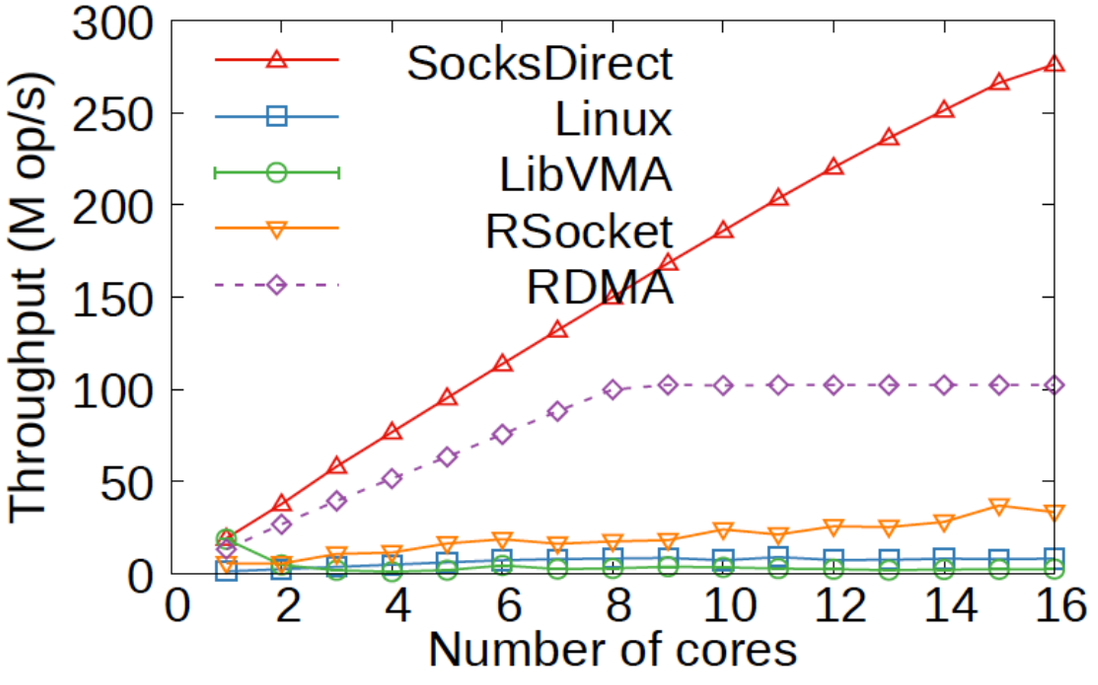


Inter-host

Multi-core Scalability



Intra-host

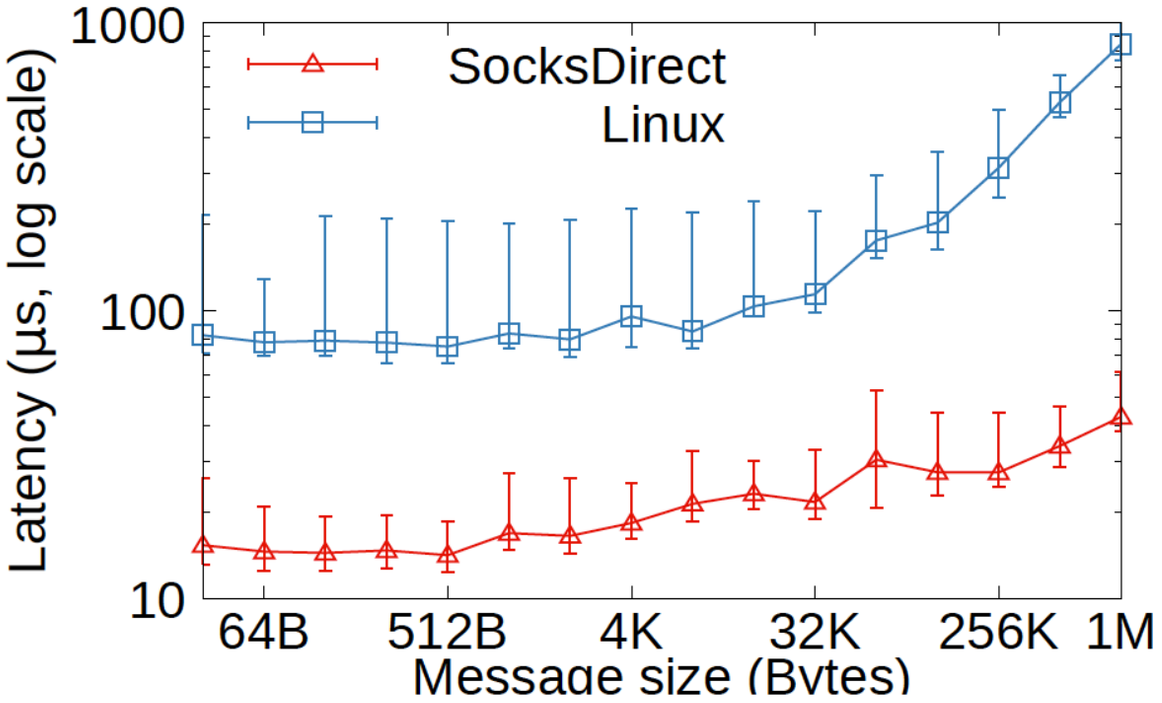


Inter-host

Takeaways: evaluate not only the best cases

- SIGCOMM'19 Review: It also does not help that the evaluation presents results that appear to have been cherry-picked to put SocksDirect in the best scenario.
- SIGCOMM'19 Review: I would have liked to see a more thorough evaluation and I would have expected that the paper highlights where SocksDirect is not ideal. Currently my feeling is that these results were chosen as they show the best scenarios.
- OSDI'18 Review: Really importantly, what are the edges of your system that don't perform well, or are hard to deploy and maintain? What are the important tradeoffs?

Application Performance



Nginx HTTP server request latency

Takeaways: evaluate with practical applications

- NSDI'19 Review: Evaluation setup (in Section 6) is weird. Running a load balancer, web service, and key-value store on the same machine is pretty unconvincing. This seems to emphasize the performance of your intra-host communication, but I don't think people would provision the services on a single machine.
- OSDI'18 Review: What is the experimental setup for the applications in 6.2? What are the load balancer, the web service, and the KV store? Are they located on the same or different machines? If they are on different machines are they communicating using TCP or RDMA?

Conclusion

Contributions of this work:

- An analysis of performance overheads in Linux socket.
- Design and implementation of SocksDirect, a high performance user space socket system that is compatible with Linux and preserves isolation among applications.
- 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.
- Evaluations show that SocksDirect can achieve performance that is comparable with RDMA and SHM queue, and significantly speedup existing applications.

Closing Thoughts

1. Rejections are common.
2. Rejections are constructive.
3. Published papers are not perfect.
4. Talk with experts both inside and outside your institution.
5. There is no new thing under the sun. The solution to your (sub-)problem may be already in a paper twenty years ago, or in a paper of another field.
6. Think why many old, failed techniques are reinvented 30 years later and become successful.

Thank you!

Wish your papers get accepted by the next major conference.

Ad: Welcome to Parallel and Distributed Software Lab, Central Software Institute, Huawei 2012 Labs.