SRoCE: Software RDMA over Commodity Ethernet

Shailesh Mani Pandey
The University of Texas at Austin
shailesh.pandey@utexas.edu

Rajath Shashidhara
The University of Texas at Austin
rajaths@cs.utexas.edu

ABSTRACT

RDMA networks are used in datacenters and high performance computing clusters to support high-throughput, low-latency networking by allowing specialized hardware to directly copy to and from the application memory and network. We propose a software-based flexible RDMA verbs implementation that uses TAS - high-performance user-space TCP stack - as the underlying transport layer to allow similar semantics without hardware and network requirements. We discuss the design space that we explore and evaluate our prototype. The single-connection throughput and latency achieved by our implementation indicate not only that providing RDMA interface over TCP using commodity NICs is feasible but also comparable to the hardware implementation.

1 INTRODUCTION

Remote Direct Memory Access (RDMA) networks are used in datacenters and high-performance computing clusters to support high-throughput, low-latency networking. This is achieved by allowing the RDMA hardware to bypass the operating system and to directly copy from the memory of one computer to another computer. This approach reduces the overheads involved with context switches, packet processing in the kernel stack and frees up the CPUs from being directly involved in data transfer. Furthermore, RDMA programming model supports one-sided communication primitives with asynchronous data transfer semantics which enables the application to hide the latency of I/O by overlapping computation with communication. Several RDMA based products have been developed and marketed around the RDMA programming model to accelerate supercomputers, cloud storage, machine learning applications and datacenter networks.

In order for this to be efficient, typically the protocol stack (Transport/Network/Link layer) is implemented in hardware rNICs. There are several issues with this approach:

(1) Specialized hardware is expensive. Commodity hardware is preferred for large-scale deployments such as datacenters. Commodity hardware is easier to acquire in bulk, have more frequent refresh cycles and are easier to maintain - do not require specialized training for system administrators. In addition, older versions of RDMA were incompatible with Ethernet networks and required specialized Infiniband transport. Common traffic monitoring and management tools are incompatible with specialized NICs.
(2) Protocol stack is ossified in hardware and is difficult to customize. This restricts the evolution of network protocols and hinders deployment of recent advancements such as software-defined networking [8] and programmable dataplanes [2]. Furthermore, hardware bugs in rNICs have caused significant deployment issues as seen in [3].
(3) RDMA deployments require advanced network configuration and have severe limitations. For example, earliest version of RDMA-over-Converged Ethernet (RoCE) did not support L3-level routing. Although RoCEv2 relaxed this requirement, it needs lossless link-layer communication to function. To achieve this, Data Center Bridging (DCB) enhancements such as Priority Flow Control (PFC), Explicit Congestion Notification (ECN) and Enhanced Transmission Selection (ETS) must be enabled on top of Ethernet. Large scale deployment of RDMA is technically challenging and critical problems such as Head-of-Line (HOL) blocking delays, network deadlocks and livelocks have been reported by datacenter operators such as Microsoft [3]. Newer version of RDMA called Resilient RoCE has been marketed by vendors such as Mellanox which is designed to run on lossy Ethernet topologies. However, as shown by [10], Resilient RoCE is not successful in avoiding packet losses across all realistic scenarios with dynamic traffic patterns.

We propose a software-based RDMA verbs [11] implementation that uses TAS - high-performance user-space TCP stack - as the underlying transport layer [5]. This approach eliminates the kernel crossing overhead associated with other software-based implementations such as SoftiWarp[9], and potentially enables zero-copy network transfer to user applications. Another advantage with our approach is that it can work on top of existing network infrastructure. Specifically, our goals are:

(1) Design a software layer to support RDMA programming model on top of TAS
(2) Evaluate if a purely software-based approach can achieve near-hardware performance

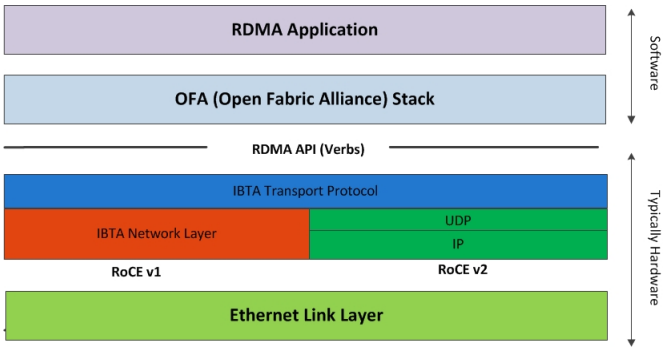

Figure 1: RDMA stack. RDMA allows devices to perform direct memory to memory transfers at the application level without involving the host CPU. Both the transport processing and the memory translation and placement are performed by hardware resulting in lower latency and higher throughput. Figure credits: [1]

2 BACKGROUND

Remote Direct Memory Access (RDMA) allows application on one machine to access the memory of an application on a remote machine without involving the kernel on either side. This avoids the overheads associated with kernel crossings. Current implementations of RDMA use specialized RDMA Network Interface Controllers (RNIC) to directly write data from the wire to the application memory (and vice versa). By offloading to the hardware and thus removing CPU from the data transfer path, this enables high-throughput, low-latency networking. RDMA enables access to a previously registered application memory region without kernel or application involvement. These memory registrations are stored in memory translation and protection tables (MTPT). The RNICs use these tables to directly access the application memory for writing the incoming data or reading the outgoing data.[12]

The RDMA Protocol Verbs Specification [4] lists the operations supported by the RDMA protocol, which include RDMA Read, RDMA Write, Send and Receive. In this work, we focus on the RDMA Write, and RDMA Read operations. The consumer that intends to use these RDMA operations registers a memory region with the RNIC and sets up *work queues*, and *completion queues*. The consumer then posts *Work Requests* on the work queues which reference a local and remote memory region and request reading data from the remote to local memory region (RDMA Read) or writing data to the remote from local memory region (RDMA Write). RNIC then completes the requested operation without the involvement of the OS or the application and posts a *Completion Queue Entry* to the completion queue. To access the memory on the remote side, the consumer has to provide a remote access key which is shared out of band on connection setup.

RDMA over Converged Ethernet (RoCE) is a network protocol frequently used to provide RDMA capabilities. As originally implemented and standardized by the InfiniBand Trade Association (IBTA) RoCE was a layer 2 protocol. RoCE v2 is an extension of the RoCE framework that allows it to be readily transported across layer 3 networks. As shown in figure 1, a layer 3 capable RoCE v2 protocol simply continues up the stack and adds a UDP header as a stateless encapsulation of the layer 4 payload. RoCE v2 needs lossless link-layer communication to function. To achieve this, Data Center Bridging (DCB) enhancements such as Priority Flow Control (PFC), Explicit Congestion Notification (ECN) and Enhanced Transmission Selection (ETS) should be enabled on top of Ethernet. Resilient RoCE is a version of RoCEv2 designed to run on lossy Ethernet topologies using DCQCN congestion control algorithm.

To support RDMA verbs in software while keeping the OS out of the dataplane operations, we use TAS, TCP Acceleration as a Software service. [5] TAS is a user space networking stack that handles common-case TCP operation in an isolated *fast-path* service which runs on dedicated CPUs, while handling corner cases in a *slow-path*. To be workload proportional, TAS dynamically allocates the appropriate amount of CPUs to accommodate the fast-path, depending on the traffic load. As shown in figure 2, TAS uses Receive-Side Scaling (RSS) to get incoming data packets delivered to a fast-path core. These messages are then written to the RX payload buffer and the app is notified of the availability of more data. Similarly, to send data across the network, the app writes the message in the TX payload buffer and notifies fast-path through a shared queue. Fast-path then reads these packets from the buffer, adds TCP segment and packet headers, and sends them over the wire.

For the control plane operations, such as connection setup, the app directly coordinates with the slow-path. Slow-path is also responsible for monitoring each flow, handling exceptions, and notifying the fast-path to adjust the flow rate. By subdividing the TCP stack data plane into fast-path and slow-path and dedicating separate threads to each, TAS achieves throughput up to 7× that of Linux. Moreover, this approach does not require new hardware and retains the flexibility of a software implementation which is crucial for us. This flexibility enables us to extend TAS to support RDMA verbs over commodity hardware without the involvement of OS in the data flow path.

3 DESIGN

This section elaborates on the system architecture, design trade-offs and limitations of our implementation.

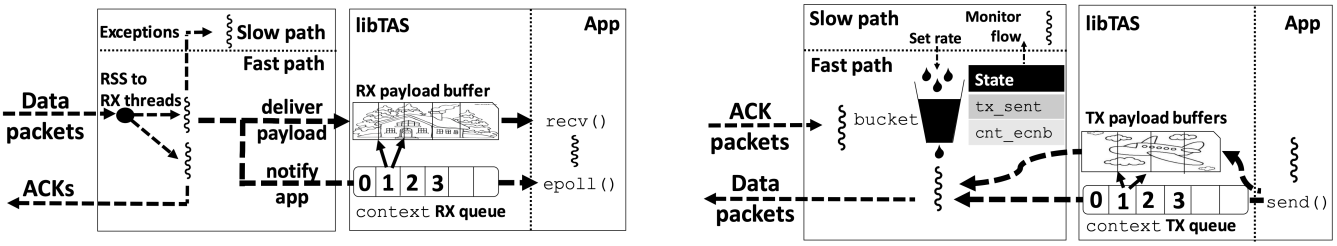

Figure 2: TAS receive and send flow. The common case dataplane operations are handled by *fast-path* in user space while the exceptions and other control plane operations are monitored by the *slow-path*. Figure credits: [5]

API	Operation	Behavior
`rdma_init()`	Initialize application library Setup IPC and application context with TAS *Must be called before using any other RDMA functions*	-
`rdma_listen(localaddr, backlog)`	Listen to RDMA connections on localaddr (ip, port) Queue up to backlog pending SYN *Analogous to a wrapper around socket(), bind() and listen()*	Asynchronous
`rdma_accept(socket, *remoteaddr, *mr_base, *mr_len)`	Accept an incoming RDMA connection on listener socket Setup Memory Region (MR) for connection *Analogous to accept()*	Blocking
`rdma_connect(remoteaddr, *mr_base, *mr_len)`	Connect to remote server at remoteaddr (ip, port) Setup Memory Region (MR) for connection *Analogous to connect()*	Blocking
`rdma_read(socket, len, loffset, roffset)`	Read len bytes of data from remote memory region at roffset and copy to local memory region at loffset	Asynchronous
`rdma_write(socket, len, loffset, roffset)`	Write len bytes of data from local memory region at loffset and copy to remote memory region at roffset	Asynchronous
`rdma_completion_poll(socket, *events, num)`	Poll for completion of at most num read/write work requests	Non-blocking Asynchronous

Table 1: API definition

API

As RDMA verbs is a standardized API, aligning our implementation to remain compatible with the standards is advantageous as existing RDMA verbs applications need not be modified. However, we choose to simplify implementation complexity by giving up application compatibility. We have designed APIs which retain the essence of the RDMA programming model but also allows us to work on top of the TCP interface with minimal changes. Implementation complexity is a major deal breaker in our design choices as we had to tackle the challenge of implementing this system under severe time constraints.

Table 1 lists the APIs and their semantics. To support RDMA connection establishment, we augment the FreeBSD `sockets` API. We modify the `accept(), connect()` to receive the memory region address and length. Moreover, our APIs operate only in blocking mode and advanced socket options are not supported. These modifications allow us to retain the existing slow path implementation in TAS with minimal changes such as allocating memory region and work queues.

RDMA verbs provides a rich API with several variants of `read()/write()` operations with both one-sided and two-sided communication primitives, atomic operations, etc. We only provide one-sided communication primitives in our API. We have limited our problem scope as the other operations are not fundamentally different and are trivial extensions to the existing design. Finally, data transfer operations are guaranteed to complete in-order of requests like RDMA verbs operation.

Data transfer operations (`read(), write(), poll()`) are *always enabled* i.e., they do not block for any reason. Read and Write operations return immediately after the work request is enqueued and return with an error code `EAGAIN` if work queue is full. Similarly, poll dequeues elements present in the completion queue without waiting for the requested number of events to be completed. With this design, we lay the responsibility on the user to poll for data transfer and completion notifications. This design choice also allows full flexibility to the application to overlap communication with computation as it desires.

Architecture

Supporting RDMA abstraction over TCP requires 3 major components:

- **Framing:** Work queue requests from the application must be converted into messages transmitted on TCP stream abstraction. We define a header structure to frame the messages. On the transmission entity, header is added along with the message into the TCP byte stream. On the reception side, byte stream is parsed to RDMA requests/responses to take appropriate actions. We rely on the in-order lossless delivery properties of TCP to ensure that messages do not get reordered by the transport layer. In addition, RDMA request/response may be much larger than the TCP transmission/reception buffer size. Framing entity must take care to copy the data in chunks ensuring that the buffer is never overrun.
- **Request processor:** With one-sided communication primitives, application on the receiving side is passive and is completely oblivious to the ongoing data transfer. We need an entity on the receiving side to process the requests received from the remote peer. This processing element also performs the copy to/from the shared memory region into the TCP stream.
- **Application Interface:** Pending work queue requests from the application must be transferred to TAS for asynchronous processing. Similarly, application must be notified of the completed requests.

We considered various interposition options as seen in Figure 3. Option (a) shows the existing TAS architecture. TAS is divided into slow path and multiple fast path cores. Application interfaces with TAS through `libTAS`.

In option (b), TAS is treated as a black box TCP service and RDMA processing is purely implemented as a part of the application library. This option is the easiest to implement as it requires no modifications to TAS. In addition, `libRDMA` has access to full application memory and it is not necessary to setup dedicated memory regions. However, due to the passive nature of the receiver, an application thread within the library is required to busy poll for RDMA requests from the peer. Furthermore, an additional `memcpy()` is required to transfer the data from application memory to TCP transmit buffers in this design.

Option (c) represents the other end of the design spectrum where the TCP protocol and implementation inside TAS are fused to support message-oriented communication and scatter-gather transmission/reception directly from application memory without maintaining explicit TCP transmission/reception buffers. Although this option has the least overhead, protocol design and implementation is extremely complex.

In option (d), dedicated cores are reserved for RDMA processing in TAS. *rDMA* cores translate the RDMA work requests from the application into TCP stream by interfacing with the fast path cores. This design requires minimal modifications as there is a clean separation between the core TCP implementation and rDMA processing. Moreover, we can achieve pipeline parallelism and have minimum impact on the performance of the TAS stack. However, additional cores must be dedicated in this design. Moreover, an extra hop between rDMA and fast path cores and `memcpy()` introduces performance overheads.

In option (e), we add the rDMA processing functionality within the fast path core. With this option, we can choose to integrate the rDMA operation with fast path without disrupting the core TAS design. For example, by adding minimal RDMA state to the flow state structure, we can gain performance advantages due to cache efficiency of TAS. Furthermore, we also retain the auto-scaling feature of TAS - growing the number of fast path cores in response to increased workload. We choose to keep the TCP buffers separate from the RDMA work queue requests introducing additional overhead due to an extra `memcpy()`. This option retains the core benefits of TAS, has reasonable implementation complexity and introduces small performance overheads. Therefore, we choose this architecture for our implementation.

4 IMPLEMENTATION

As described in the previous section, our implementation is divided into two parts, `libRDMA` and *fast-path*. To simplify our implementation and have a working prototype in a limited time, we make the following simplifying decisions:

(1) Instead of dynamic memory registration, each app gets a pointer to a fixed length shared (with TAS fast-path) memory region for each established connection.
(2) Applications that have established a connection have access to each other's shared memory region. Thus, we do not implement any additional access control mechanism.

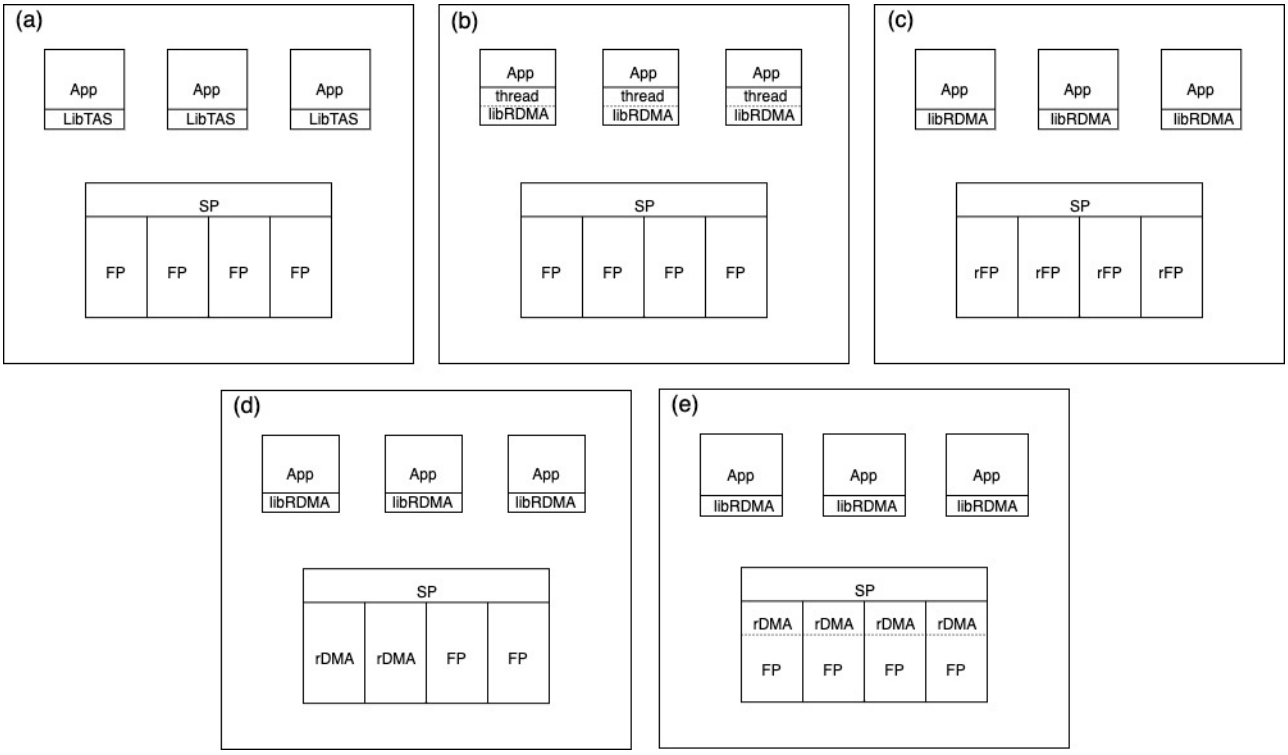

Figure 3: Design choices (a) Existing TAS design (b) Each app is linked with `libRDMA` which has a separate thread to process RDMA (c) *fastpath* code is modified to integrate TCP and RDMA operations (d) App is linked with `libRDMA` and separate *rDMA* cores are used to implement RDMA operations (e) Each app is linked with `libRDMA` and *fastpath* is modified to perform RDMA functions.

(3) We only implement the RDMA write operation. We believe that our design enables a simple extension to other operations, such as RDMA read.
(4) We only support reliable connection mode (RC) and do not implement the unreliable datagram (UD) operations.

`libRDMA`

First, we describe the implementation of `libRDMA` which provides the APIs to initialize and establish connection, and perform RDMA operations. The application must make a call to `rdma_init()` before starting any RDMA operations. In this call, we establish connection with TAS, register the app context and initialize the internal data structures. To start listening on a particular port, the application should make a call to `rdma_listen()`. This is just a wrapper around the listen API originally provided by TAS. For `rdma_connect()` and `rdma_accept()`, we make a call to the slow-path which is modified to allocate the shared memory region and work / completion queue in addition to the connection establishment.

For the dataplane operations in the library, we first validate the arguments passed to the API (e.g. the local and remote offsets should be in the valid shared memory region). Then we create a *work queue entry* for the request and write it in the shared work queue. Once the entry is successfully written in the work queue, we notify the fast path about the presence of an additional work request. The API returns only after both these operations complete. It is the application's responsibility to call `rdma_completion_poll` to check for the completion of a previously submitted request. In this call, we check for any notifications from the fast path about completion of a request. If a notification is pending, we read that and copy the completed work queue entries to the buffer provided by application. This call only checks once for the presence of a notification and thus gives the application the ability to check occasionally and continue with some other useful work in the meantime. This also enables applications

to be in control of the number of total operations in progress at any moment.

Fast-path

Originally, fast-path expects data to be present in the TX buffer on receiving a notification from the application. We modified the fast path to rather expect an entry in the connection's work queue. Now, fast path first adds a RDMA header for each work queue entry and then copies the header as well as the data from the specified memory region to the TX buffer for that connection. If TX buffer does not have enough space to hold the entire message and headers, we write partial data to it and keep state to track the data which is yet to be sent. On receiving an acknowledgement for each TCP message, fast path frees up space in the TX buffer. We modify fast path to process the pending work queue entries when more space becomes available in the TX buffer. Once the entire data is sent, the status of the work queue request is modified to reflect that the request data is sent to the remote node but a response is pending. After an RDMA response is received from the remote node, we update the status of the request entry to reflect that it has completed, and notify the application. To avoid an extra copy of work queue entry, we merge the work queue and completion queue. Hence, the application is just notified of the index of the latest work queue entry that has completed.

On receiving data from the remote node, fast path would normally write the data to RX buffer, send back an acknowledgment and notify the application. In our implementation, like RDMA specification, the target application is never notified and it can in fact be sleeping or be in the scheduler's wait queue. First, the initial 16 bytes of data are parsed as the RDMA header and then the following data is written to the application's memory in the offset mentioned in the header. After this copy has completed, an RDMA completion response is sent back to the remote node. To send these RDMA responses, we put them in a *response queue* and then multiplex between the response queue and the work queue for that connection.

Although we only implement RDMA write operation in our current prototype, we believe that extending it to support RDMA read operation should have a complementary work flow and thus would be fairly straight forward. In addition, we expect our implementation to be easily extensible to dynamic memory regions with access control. However, extending this to support unreliable datagram (UD) operations using TAS may require substantial effort.

Source Code

The changes that we made to TAS app library, fast-path, slow-path and our evaluation tests are available publicly at https://github.com/mani-shailesh/rdma-tas.

5 EVALUATION

We conduct all our evaluation experiments on two lab machines connected via a Netberg Aurora 720 100 Gbps switch. Each lab machine houses two Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz processors for a total of 72 CPU cores. Both machines have 187 GB of main memory, 32 KB of L1d and L1i cache each, 1 MB of L2 cache, and 25 MB of L3 cache. We also set up 4096 huge pages of size 2MB each for running TAS on each machine. Both these machines run Linux and have an RDMA enabled Mellanox ConnectX-5 100Gbps NIC that we use for our experiments. We fix the CPU frequency to 3.0Ghz and disable on-demand frequency scaling when performing the experiment.

Figure 4 shows the throughput and latency of our implementation compared to the hardware based RDMA for one connection as we vary the message size. For both these measurements, the maximum number of in-flight messages was set to 500. For smaller message sizes, our implementation provides higher single-connection throughput than the traditional hardware based implementation with similar latency. However, as the message size increases, the throughput of our implementation saturates, and latency increases while the hardware implementation provides higher throughput and maintains the low latency. We attribute this increase in latency and drop in throughput with the increase in message size to the overhead added by an extra copy of data from the memory region to the TCP transmission buffer. This overhead increases with the message size and thus leads to worse performance with higher message sizes.

For our next experiment, we analyze the performance with varying number of maximum allowed in-flight messages. We keep the message size fixed at 4096 bytes and measure the single-connection throughput and latency achieved by our implementation as well as by hardware based RDMA. As seen in figure 5, the throughput of our implementation initially benefits from increasing the number of messages in-flight because this allows us to pipeline the steps in our software stack such as writing requests to the work queue and copying data into transaction buffer with the actual transmission of messages over the wire. With more than 4 pipelined messages, we achieve the single-connection throughput of more than 20 Gbps as opposed to around 10 Gbps by the hardware RDMA implementation. Although the throughput of our implementation increases, we get this boost at the cost of higher latency. As more messages are in-flight, the average delay between the time of writing a request and the time that fast path actually reads and transmits it is increased. Thus, although we transfer more messages in aggregate, each individual request has to wait longer on average in the work queue and then in the transmission buffer. Once again,

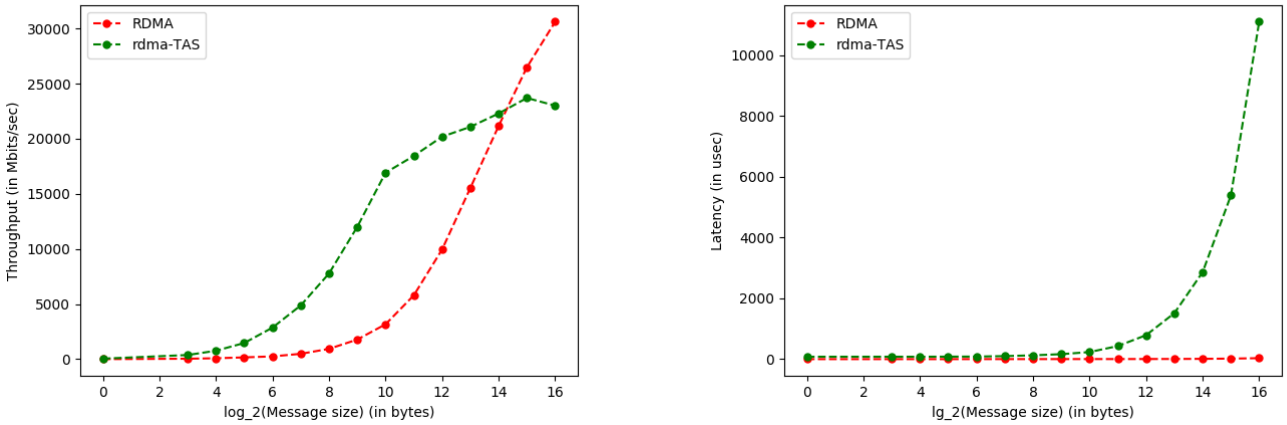

Figure 4: Throughput and Latency measured as a function of `RDMA_WRITE` size.

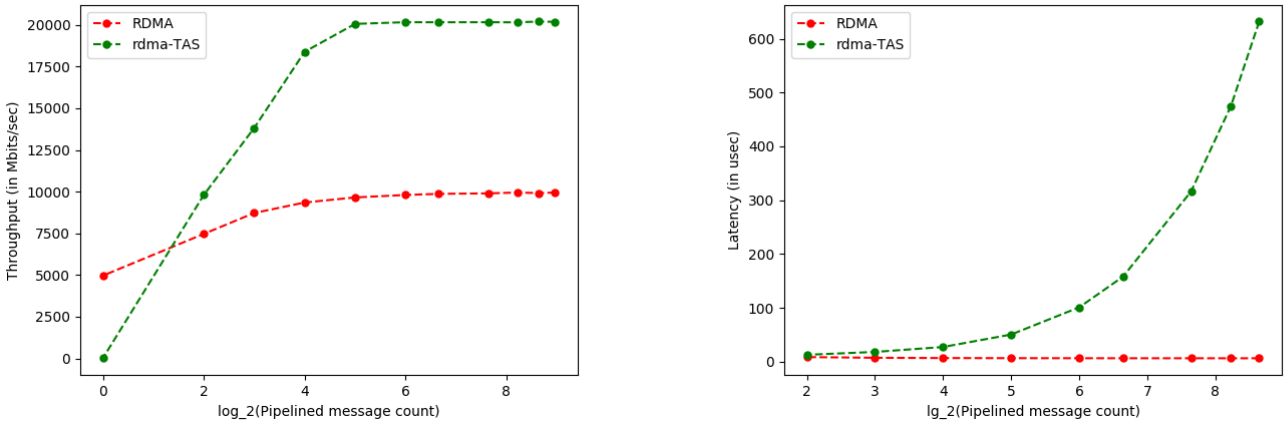

Figure 5: Throughput and Latency measured as a function of number of pipelined messages.

the hardware implementation of RDMA maintains the low latency.

Figure 6 shows the effect of increasing the number of connections while keeping the other parameters constant. We keep the message size fixed at 4096 bytes and the number of maximum allowed in-flight messages at 500. It must be noted that in all these experiments, TAS was run with just 1 fast-path core. Hence, as we run our RDMA implementation with multiple connections, we increase the amount of state that we have to maintain in the software and at the same time TAS fast path core has to multiplex between different connections. This adversely affects both the throughput and the latency of our implementation. On the other hand, the throughput of hardware implementation scales well with the increase in number of connections. It benefits from the parallel data transfer and hardware-enabled application memory access. The hardware implementation achieves nearly 70 Gbps of throughput with minimal increase in latency with 10 connections. Even with the hardware implementation, increasing the number of connections beyond 10 does not help with the throughput and adversly affects the latency.

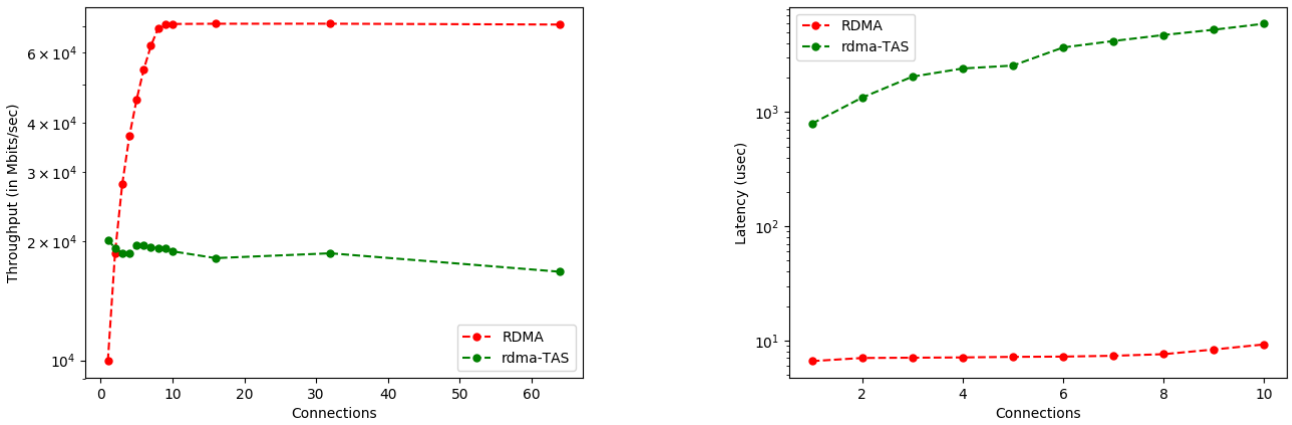

Figure 6: Throughput and Latency measured as a function of number of connections.

The single-connection throughput and latency achieved by our implementation indicate that providing RDMA interface over TCP using commodity NICs is not only feasible but also comparable to the hardware implementation. Using a fast user-space TCP stack, like TAS, we are able to eliminate OS from the common data path and avoid the associated overhead. We believe that the current challenge we face with running multiple fast-path cores significantly limits our ability to scale well with the number of connections. This, and more such challenges that we face are described in detail next.

Challenges

(1) **Mellanox RDMA NIC setup and configuration**: Getting RDMA to work was a tedious, confusing and often hopeless task. Poor online support and documentation made it extremely hard for us to get the RDMA up and running. Firstly, Mellanox OFED drivers are not compatible with Debian 10 OS which was running on our experimental infrastructure. After getting past this barrier, we were stuck on building DPDK Poll Mode Drivers (PMD) support for Mellanox NICs. We discovered that the drivers provided by the Mellanox package were obsolete and incompatible with DPDK. After days of fiddling with OS and switch configuration and countless hacks, we were able to stabilize our experimental infrastructure.

(2) **Erratic behaviour TAS with Mellanox NICs**: We encountered a host of problems with running TAS on our experiment setup.

- Auto-scaling and switching to interrupt mode features of TAS are not compatible with Mellanox DPDK Poll Mode Driver (PMD).
- We were unable to reproduce line rate with TAS using the echo benchmark test suite on our setup. We tuned several parameters including number of TAS fast-path cores, transmission message size, number of connections, number of in-flight messages, application cores, TCP buffer size, congestion control algorithm and other advanced NIC options such as Priority Flow Control, NIC ring buffer size, receive and transmit offloads. Our best settings yielded only 13Gbps throughput with TAS and in comparison Linux was able to achieve 37Gbps on the same benchmark.
- Unexplained drastic throughput drops after running steadily for several minutes at high throughput. Once the throughput drops, it slowly decays to 0.
- Running with more than 1 TAS fast-path core degrades the throughput of a single connection. Each flow is exclusively handled by a fast-path core. Adding more cores should help us with scaling to more connections. We observe the opposite behaviour as seen in Table 2.
- NIC statistics obtained from `ethtool` do not match with TAS level aggregate statistics. For instance, we observed that the aggregate TAS throughput (new transmission + retransmission) was about 145Kbps, but the throughput at the NIC physical layer as reported by `ethtool` was 37Gbps.

Connections	Throughput (Mbps)	
	1 FP core	2 FP cores
1	20184.53	1085.05
2	19234.88	1089.85
4	18661.75	11165.04
8	19262.18	12080.69
16	18138.48	13029.52
32	18654.46	13301.79
64	16749.93	14169.75

Table 2: Anomalous behavior with FP cores

- TCP Congestion Control makes the traffic fluctuate wildly. When congestion control is disabled (no rate limits on the flow), throughput is steady and drops are small.

If these problems are fixed, we expect our implementation to perform much better and scale to higher number of connections. Furthermore, having a stable setup will help us identify and eliminate bottlenecks in our implementation in a reliable way.

6 RELATED WORK

iWARP is an alternative to RoCEv2 that implements RDMA verbs API support over TCP/IP stack. However, iWARP is implemented in hardware and therefore suffers from all of the limitations of hardware based solutions mentioned in the motivation of our paper. We refer the reader to [6, 10] for a detailed comparison. *SoftiWARP* implements the iWARP stack as a patch to the linux kernel[9]. This solution incurs the cost of kernel crossing and the message copy with this solution and is not designed to be provide high performance.

[7] also implements RDMA verbs support in user-space using DPDK. But, this solution develops its own reliability protocol called Trivial Reliability Protocol (TRP) instead of using TCP. TRP may be incompatible with existing network hardware as switches and firewalls snoop on the packet header information and actively block any unknown protocols. Furthermore, TCP is a reliable and most widely deployed network protocol. Implementing RDMA verbs on top of TCP enables us to take advantage of correctness and stability guarantees of TCP. In addition, our implementation can be modified to support both TCP and RDMA interfaces on top of TAS and this is an added advantage. Finally, TAS is highly optimized for datacenter networks and RDMA deployments in datacenters can take benefit from the performance optimizations of TAS by using our solution.

7 ACKNOWLEDGEMENTS

We would like to thank Prof. Simon Peter and Tim Stamler for the lab infrastructure, providing necessary background on TAS and helping us with issues we faced in setting up TAS on Mellanox NICs.

8 CONCLUSION

Current implementations of the RDMA semantics use specialized hardware and pose many restrictions on the underlying network. We propose a software-based implementation of these semantics using TAS, a fast user-space networking stack. We avoid the overheads imposed by OS and provide the flexibility of software implementation. We show that the performance of our software-based prototype is comparable to that of specialized hardware. We mention the challenges that need to be solved and believe that a scalable software-based implementation of the highly effective RDMA interface is a potential avenue of future work.

REFERENCES

[1] 2019 (accessed December 9, 2019). *RoCE v2 Considerations.* https://community.mellanox.com/s/article/roce-v2-considerations

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[3] 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.* ACM, 202–215.

[4] Jim Pinkerton Renato Recio Jeff Hilland, Paul Culley. 2003. *RDMA Protocol Verbs Specification.* Technical Report. http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf

[5] 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

[6] John Kim, Tim Lusting, and Fred Zhang. 2018. *RoCE vs iWARP.* https://www.snia.org/sites/default/files/ESF/RoCE-vs.-iWARP-Final.pdf

[7] Patrick MacArthur. 2017. Userspace RDMA Verbs on Commodity Hardware Using DPDK. In *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI).* IEEE, 103–110.

[8] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

[9] Bernard Metzler, Fredy Neeser, and Philip Frey. 2009. Softiwarp. In *Open Fabrics Alliance Sonoma Workshop.*

[10] 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.

[11] P. Culley J. Hilland D. Garcia R. Recio, B. Metzler. 2007. *A Remote Direct Memory Access Protocol Specification.* RFC 5040. https://tools.ietf.org/html/rfc5040

[12] Renato Recio. 2006. *A Tutorial of the RDMA Model.* Technical Report. https://www.hpcwire.com/2006/09/15/a_tutorial_of_the_rdma_model-1/