Datenbank Spektrum (2020) 20:199–210
https://doi.org/10.1007/s13222-020-00355-7

SCHWERPUNKTBEITRAG

RDMA Communciation Patterns

A Systematic Evaluation

Tobias Ziegler[1] · Viktor Leis[2] · Carsten Binnig[1]

Received: 18 June 2020 / Accepted: 29 August 2020 / Published online: 29 September 2020

Abstract
Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency accesses to a remote node's main memory. Although there has been much work around RDMA, such as building libraries on top of RDMA or even applications leveraging RDMA, it remains a hard problem to identify the most suitable RDMA primitives and their combination for a given problem. While there have been some initial studies included in papers that aim to investigate selected performance characteristics of particular design choices, there has not been a systematic study to evaluate the communication patterns of scale-out systems. In this paper, we address this issue by systematically investigating how to efficiently use RDMA for building scale-out systems.

1 Introduction

1.1 Motivation

Scale-out data processing systems are the typical architecture used today by many systems to process large data volumes since they allow applications to increase performance and storage capacity by simple adding further processing nodes. However, a typical bottleneck in scale-out systems is the network which often slows down the computation if communication is in the critical path or, even worse, degrades overall performance when adding more nodes [22].

With the advent of high-speed networks such as InfiniBand and its networking protocol RDMA this changed. Network latencies dropped by orders of magnitude while throughput increased [5] making scale-out solutions more competitive. However, this improvement does not simply come by switching the network technology stack, but requires the usage of RDMA's low-level communication primitives, e.g., SEND or WRITE.

Although there has been much work around RDMA, such as building libraries on top of RDMA [1, 8] or even applications leveraging RDMA [2, 6, 7, 12, 17, 25], it remains a hard problem to identify the most suitable RDMA primitives and their combination for a given problem. While there have been some initial studies included in papers that aim to investigate selected performance characteristics of some design choices [5, 10, 11], there has not been a systematic study to evaluate the communication patterns of scale-out systems.

1.2 Contribution

We address this issue by systematically investigating how to efficiently use RDMA for building scale-out systems. For this study, we model communication between nodes using a request-response pattern (i.e., an RPC-style communication). This request-response communication pattern can be used for many data-centric communication scenarios such as a key-value stores or transactions in distributed databases.

For implementing a request-response communication pattern, the combination of RDMA verbs and other design considerations (e.g. the communication model or the communication topology) provide a huge design spectrum one

✉ Tobias Ziegler
tobias.ziegler@tu-darmstadt.de

Viktor Leis
viktor.leis@uni-jena.de

Carsten Binnig
carsten.binnig@tu-darmstadt.de

[1] TU Darmstadt, Darmstadt, Germany

[2] Friedrich-Schiller-Universität Jena, Jena, Germany

has to navigate. In our study, we combine all of the above options and evaluate different performance characteristics with varying message sizes of 64B to 16KB to simulate workloads of different applications (e.g., a different tuple width in a distributed database). As a result, we shed light on important metrics such as the bandwidth, latency but also important CPU statistics.

To evaluate this design space, we perform an extensive analysis and experimental evaluation of the following dimensions:

1. *RDMA Verbs:* RDMA verbs allow an application to specify whether one-sided communication primitives (RDMA READ/WRITE) or two-sided communication primitives (RDMA SEND/RECEIVE) should be used. While there have been already almost "religious" fights which communication primitive is better in existing papers, the usage of these primitives has only been evaluated in particular settings (e.g., only in a distributed database using an M-to-N communication topology) but not in the full design space. Moreover, there are many low-level optimizations such as inlining, selective signalling, DDIO and many other low-level configuration options (e.g., if RDMA is run in single-threaded communication or not) that have not been investigated.
2. *Communication Model*: A second dimension, we aim to analyze in the design space is how threads are connected. In this paper we analyze two typical connection models: One option is to use a so called communication *Dispatcher* where worker threads delegate the communication to one dedicated thread. Another option is that each worker thread directly executes all communication by using a *Private Connection* to each other worker thread resulting in a much higher number of overall connections (i.e., not just between dedicated communication threads).
3. *Communication Topology:* The last dimension is the communication topology between requesters and responders. In this paper, we analyze different options: *One-to-One*, *N-to-One*, *N-to-M*. For instance, while a key-value-store typically uses an N-to-One communication topology (i.e., *N* requesters, 1 responder), whereas a distributed database often uses an M-to-N communication topology. In addition with the communication model, the communication topology significantly influences the number of connections and thus the contention on internal data structures of high-speed network components (e.g., connection queues in InfiniBand).

We believe that our evaluation framework is an interesting analysis tool for other research groups or industry that plan to leverage RDMA for building distributed data processing systems. To foster this and allow follow-up research to use our findings, we made the code available on GitHub [23].

1.3 Outline

The rest of this paper is structured as follows: We first introduce the relevant background w.r.t RDMA in Sect. 2 and present related work in Sect. 3. In Sect. 4 we then describe our methodology before we start our study with Sect. 5 in which we evaluate single threaded performance in a One-to-One topology. We continue with the scale-out scenarios in Sect. 6 and 7, where we look at N-to-One and N-to-M topologies, respectively. In Sect. 8 we finally discuss the effect of further RDMA optimizations before concluding in Sect. 9.

2 RDMA Background

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency accesses to a remote node's main memory. RDMA achieves low-latency by using zero-copy transfer from the application space to bypass the OS kernel. A number of RDMA implementations are available – most notably InfiniBand and RDMA over Converged Ethernet (RoCE) [24].

RDMA implementations provide several communication primitives (so called verbs) that can be categorized into the following two classes: (1) one-sided and (2) two-sided verbs.

- *One-sided verbs:* One-sided RDMA verbs (READ/WRITE) provide remote memory access semantics, in which the host specifies the memory address of the remote node that should be accessed. When using one-sided verbs, the CPU of the remote node is not actively involved in the data transfer.
- *Two-sided verbs:* Two-sided verbs (SEND/RECEIVE) provide channel semantics. In order to transfer data between a host and a remote node, the remote node first needs to publish a RECEIVE request before the host can transfer the data with a SEND operation. In contrast to one-sided verbs, the host does not specify the target remote memory address. Instead, the remote host defines the target address in its RECEIVE operation. Consequently, by posting the RECEIVE, the remote CPU is actively involved in the data transfer.

To setup the connection between requesters and responders, RDMA uses so called send/receive queues: (1) While a *send queue* is used by the requester to issue operations such as READ, WRITE, SEND as well as ATOMICS (2) the *receive queue* is used by the responder to issue RECEIVE requests. With RDMA, a connection between a requester and a responder bundles these two queues and is therefore called queue pair (QP). More precisely, to initiate a connection between a requester and a responder, the application needs to create queue pairs on both ends and connect them. To actually then issue communication operations, a client creates a *work queue element* (WQE). The WQE specifies parameters such as the verb to use but also other parameters (e.g., the remote memory location to be used, if the element is sent signaled/unsignaled see Sect. 8). For sending the WQE, the requester then puts the WQE into a send queue and informs the local RDMA NIC (RNIC) via Programmed IO (PIO) to process the WQE. For a signaled WQE, the local RNIC pushes a completion event into a requester's completion queue (CQ) via a DMA WRITE once the WQE has been processed by the remote side.

3 Related Work

Research and industry have widely adopted high-speed networks [15, 21] to improve scale-out systems. In the following, we compare our evaluation with related work and provide a broader overview of work done in the database community.

3.1 RDMA Evaluation

In this paper we aim at providing a systematic evaluation of communication patterns. Therefore, an important body of work are existing low-level RDMA evaluations [5, 10, 11]. Kalia et al. [11] discuss which RDMA operations should be used and how to use them efficiently. The resulting guidelines provide a low-level analysis on the RDMA verbs and how they can be optimized. In this evaluation we build on this findings but focus on a more higher-level evaluation using a request-response pattern to simulate various use-cases in a data processing system.

3.2 High Performance RDMA Libraries and Systems

To simplify the use of RDMA, Alonso et al. [1] propose a high-level API for data flows for data intensive applications. Fent et al. [8], in contrast, present a library to accelerate ODBC-like database interfaces and propose a message-based communication framework.

Several recent systems adopted RDMA to speed-up their performance. There are many RDMA-enabled key/value stores [13, 14, 18–20] and distributed database systems based on RDMA [2, 6, 7, 12, 17, 25]. Depending on their design, these systems use different communication patterns. For instance, FaSST [12] discusses how remote procedure calls (RPCs) over two-sided RDMA operations can be implemented efficiently. FaRM [6, 7], in contrast, leverages the benefits of one-sided verbs to implement distributed transactions. Further, several researchers optimized specific algorithms of a database, including distributed joins [3, 4, 22], RDMA-based shuffle operations [16], and indexes [26].

While all these papers include some micro benchmarks to evaluate individual design options, there has not been a systematic study of the broader design space in one unified setup to evaluate the communication patterns of scale-out systems.

4 Methodology

In this section, we describe the evaluation methodology used to evaluate the design space of RDMA across the different dimensions discussed before. To isolate the fundamental properties of different communication primitives, we implemented a *request-response framework* that allows us to evaluate all design dimensions (i.e., verbs, communication patterns, communication topology). For a fair comparison and to avoid potential overhead that stems from configurability (which would potentially falsify the measurements), the framework uses C++ templates to generate code for one design option (selected by template parameters). Furthermore, to focus on the communication aspects, we avoid executing application logic on the responder side such as a key-value lookup or a remote procedure call (RPC).

In the following, we describe the building blocks of our framework, the workload and the experimental setup used for our evaluation.

4.1 Building Blocks of Framework

In a request-response communication pattern, the request or the response can either be transmitted by using an RDMA WRITE or an RDMA SEND operation. In the following, we first explain the basic building blocks of how a WRITE / SEND operations can be used to implement a request or a response and then how these basic building blocks can be combined to implement the request/response communication pattern.

Basic Building Blocks using RDMA WRITE: As mentioned in Sect. 2, WRITE is a one-sided verb, meaning it directly writes to remote memory and bypasses the remote CPU. Yet, the responder needs to know when new messages arrive, which is challenging with CPU-bypassing. A common protocol to detect incoming requests, is to write messages to a specific memory region, i.e., into *mailboxes*. Each requester knows the memory address to its dedicated mailbox as depicted in Fig. 1a. The responder constantly iterates the mailboxes to detect new messages. Special care must be taken to (1) avoid reading partially transmitted messages and (2) avoid that the optimizing compiler removes code. To avoid reading partially transmitted

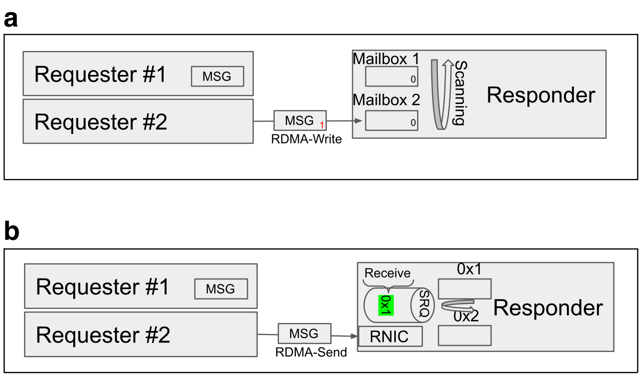


Fig. 1 Building Blocks – *Write with Mailboxes and Send with Shared Receive Queue*. **a** Write, *Mailboxes*, **b** Send, *Shared Receive Queue*

messages, the responder needs to verify if a message is completely written into RDMA memory. We can exploit the increasing address order of WRITE to check the last transmitted bit (flag). Although neither the RDMA nor the InfiniBand specification mentions the order, most hardware vendors implement WRITE with increasing address order [7, 10]. Scanning mailboxes is often implemented as a simple loop, which might be removed by the optimizing compiler. Because the optimizing compiler is not aware that a third party writes to the mailboxes, such a loop is considered unnecessary. In C++ the `volatile` construct ensures the optimizing compiler leaves the code for the loop unchanged.

The transmission process initiated by the requester begins with setting the last bit in the message as depicted in red in Fig. 1a. Afterwards, the message is written to the corresponding mailbox with WRITE. The responder will detect the incoming message by scanning the mailboxes. Finally, when processing is done the last bit in the mailbox is cleared to receive a new message. This approach is also applicable on the requester side, if the response is delivered by WRITE as well.

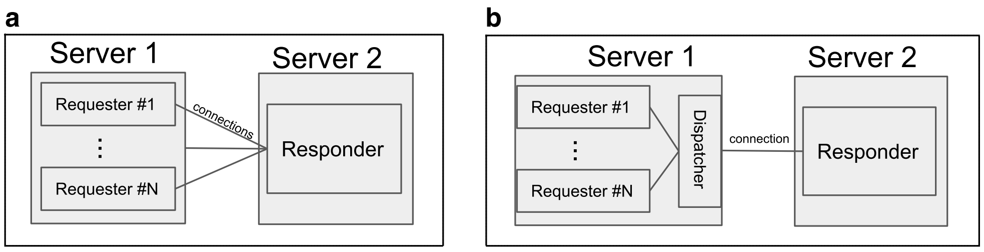
The mailbox design is very static and thus has some limitations. One needs to allocate the mailboxes and therefore decide in advance how many messages each requester can transmit. In our synchronous implementation, one requester is assigned to one mailbox. In the context of our evaluation, the WRITE verb always implies that the above mailbox architecture is set up on the remote side.

Basic Building Blocks using RDMA textscsend: In contrast to WRITE, SEND requires a RECEIVE posted to a Receive Queue beforehand. As mentioned in Sect. 2, the RECEIVE contains the information to which memory location an incoming SEND is copied to. Because it is not always known when to expect a SEND, a common pattern to prepare for multiple incoming SEND is to post multiple RECEIVES beforehand. When a SEND consumes one of the RECEIVES, the RNIC will notify the application that a message was received. To know which memory location, i.e., RECEIVE was used a common misconception is to exploit the order of the RECEIVES. For instance, the application registers two RECEIVES in the Receive Queue one with address *0x1* and afterwards one with *0x2*. Since the RECEIVES will be taken in order from the Receive Queue one might suspect that the first message is written to *0x1*, but hardware parallelism in the RNIC does not guarantee which memory location is used first. For instance, if two RECEIVES are consumed directly after each other the second memory location *0x2* could be written first. The receiver would now look into the first memory address, in which in the worst case nothing, partially transmitted messages or old messages are located. A better approach is to tag the RECEIVES with ids and when a SEND consumes a RECEIVE the RNIC returns the id, which can be mapped to a memory location. To detect incoming SEND messages, the Receive Queue has to be polled. This polling is expensive for the responder that typically has multiple incoming connections. These connections require that the responder constantly switches between the different receive queues. To mitigate this overhead, RDMA offers a Shared Receive Queue to which multiple incoming connections can be mapped. The Shared Receive Queue allows the responder to only poll a single queue to handle all requests as depicted Fig. 1b. We use the Shared Receive Queue always on the responder side, when requester transfers messages with SEND.

Combining the Basic Building Blocks: These two basic building blocks encapsulate one and two sided RDMA communication and can be freely combined to implement a request-response pattern:

Design 1 – Private Connection per Thread: The first design is fairly simple: each requester is directly connected to all responders (it has N QPs for N receivers, as shown in Fig. 2a). If the application has multiple requesters, the

Fig. 2 Designs – *N Requester and one Responder*. **a** Private Connection, *N Connections*, **b** Dispatcher, *Only 1 Connection*



amount of QPs increases linearly. This, in turn, can negatively impact performance because the RNIC needs to switch between connection states. In the worst case, a large number of connections can lead to swapping some state to the host memory over PCIe [11]. However, because each requester owns its state, no synchronization or delegation is required.

Design 2 – Communication Dispatcher: The second design, which is shown in Fig. 2b, aims to minimize direct connections by introducing a single communication dispatcher. The dispatcher solely handles communication for every requester on the same server. There is only one connection to every receiver, in contrast to the first design the number of QPs is decoupled from the number of requesters. This can lead to a higher RNIC utilization, but the communication dispatcher can also end up being the bottleneck. The dispatcher can only transfer messages from its own RDMA pinned memory. Hence, it is important that workers already prepare the messages in this memory region, otherwise an additional memory copy is needed. Furthermore, in high-contention scenarios an efficient method of delegating messages to the dispatcher is necessary. In our implementation, we use the same mailbox approach as for writes, but in a local setting. Requesters prepare the message in the RDMA memory and then set the last bit. The last bit again signals the dispatcher that the message can be transferred.

4.2 Workload of Framework

As mentioned before, we use different message sizes to simulate different application workloads. While the request in a request-response pattern typically transfers the request parameters (that are typically rather static in size), the response message can vary significantly. For example, in a key-value store the response depends on the width of the value. Whereas, in a distributed database, the response can vary between a few bytes and larger tuples grouped together in pages of multiple KB.

In our evaluation framework, we hence simulate the request-response pattern using the following setup:

- The requester always transmits 64 byte messages to the responder to simulate the request parameters. We use 64 byte messages since this allows us already to simulate typical request messages without giving up generality that we aim for: (1) We did not use smaller messages for the request, since this would not change the results since latency and bandwidth up to 64 byte messages is pretty stable [5]. (2) We did not use larger messages for the request since 64 byte messages allows us to already encode a large-enough number of parameters in the request.
- The responder replies with a response message, which we vary from 64 until we hit the bandwidth limit of our RNICs. Typically, as mentioned before, a response is very application specific and involves design decisions, such as memory allocation for each response or in advance. To avoid the effects of those decisions in our benchmark and concentrate on the effects of the communication, all response messages are pre-allocated in a fixed-size memory region of multiple GB from which a requester can retrieve items from.

4.3 Experimental Setup

For all experiments, we use a cluster with 6 nodes featuring two Mellanox ConnectX5 cards connected to a single InfiniBand EDR switch. Each node has two Intel(R) Xeon(R) Gold 5120 Skylake processors (each with 14 cores) and 256 GB RAM per NUMA node. Moreover, each node is equipped with two RDMA NICs (i.e., one per NUMA region). However, we will only use the second RNIC to create more load, i.e., more receivers, for our M-to-N scenario in Sect. 7. The reason is that M responders share all requests, therefore to get a similar load and ot make the numbers comparable to the previous experiments we utilize the second RNIC. The nodes run Ubuntu 18.04 Server Edition (kernel 4.15.0-47-generic) as their operating system and use the Mellanox $OFED$–5.0–1.0.0.0 as the network driver. The benchmark is implemented with C++ 17 and compiled using GCC 7.3.0.

5 Evaluation: One-to-One

In the first part of our evaluation, we initially focus on evaluating the One-to-One communication topology. Different from the following experiments (N-to-One, M-to-N), we use a single thread to isolate fundamental performance differences of alternative request-response designs. Hence, the main focus is on evaluating the following combinations of verbs for implementing the request-response pattern where the first verb stands for request and the second verb for the response:

- Write/Write (denoted as WriteWrite)
- Send/Send (denoted as SendSend)
- Write/Send (denoted as WriteSend)
- Send/Write (denoted as SendWrite)

Moreover, since we run in single-threaded mode, we only use the design with a private connection per thread. We excluded the dispatcher design from this experiment because it is not necessary to minimize communications (queue pairs) in a one-to-one single threaded scenario. To evaluate the

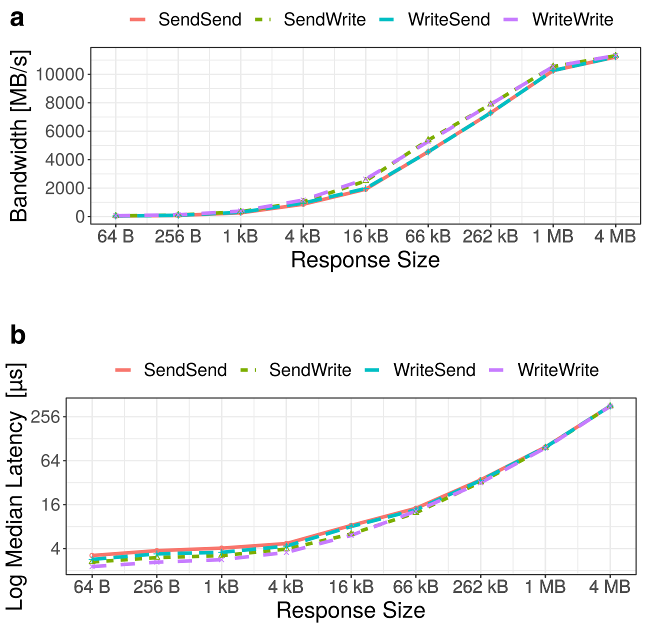


Fig. 3 Bandwidth and Median Latency – *For different response sizes in single threaded execution.* **a** Bandwidth, *One-to-One*, **b** Latency, *One-to-One*

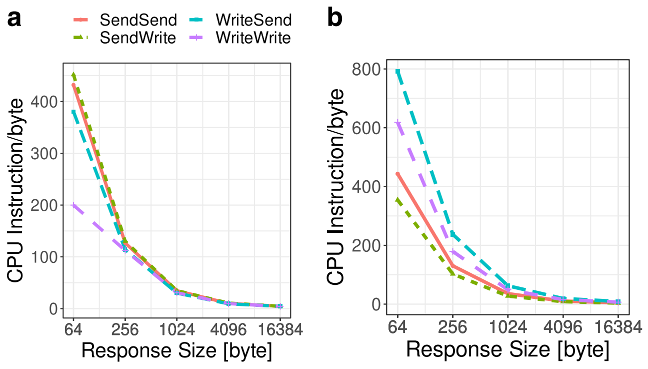


Fig. 4 Instructions per Byte for Requester and Responder – *For different response sizes in single threaded execution.* **a** Requester, *One-to-One*, **b** Responder, *One-to-One*

performance we requested 4 million responses and plotted the mean of 10 runs.

Fig. 3a compares the bandwidth in MB/s for different response sizes. For the 64 and 256 byte sized responses, only a minor difference in bandwidth is observable. However, the relative difference between SendSend (worst) and WriteWrite (best) is still around 40%. In general, all combinations using a WRITE to implement the response outperform the versions in which SEND is used. This performance difference is even more pronounced for larger messages. For example, when using 16 KB messages, the performance differs by 700 MB/s between WriteWrite and SendSend. Finally, when reaching the bandwidth limit with 4 MB response sizesthe gap closes again.

Next, we evaluate the corresponding median latency as shown in Fig. 3b. The Y-axis shows the median latency in microseconds and the x-axis the response sizes. For small response messages up to 256 byte all combinations achieve a latency below 4 microseconds. Nevertheless, SendWrite and WriteWrite are again more efficient with below 3 microseconds. The difference again becomes even more pronounced for 16 KB responses as there is almost a 2 microsecond difference between SendWrite/ WriteWrite and SendSend/ WriteSend. However, as in the bandwidth experiment the difference between all combinations disappears with increasing messages sizes. It is important to note that 4 KB is the maximum transmission unit (MTU) which increases the latency for packets larger than 4 KB.

In order to determine the cost for one byte transferred, we additionally analyzed the CPU instructions normalized per byte. Fig. 4a shows the instructions/byte on the requester and Fig. 4b for the responder. We observe that for 64 byte messages there is an instruction overhead of up to 800 instructions for each byte received and respectively 450 instructions on the requester. However, with increasing response sizes the instruction overhead is amortized. For brevity we only show up to 16 KB because instructions are not further amortized. In other words, to save CPU cycles per byte one can increase the message size and trade-off higher latency.

6 Evaluation: N-to-One

So far, we investigated the effect of increasing response sizes in a One-to-One setup using only single-threaded communication. In a distributed system, however, a requester often has multiple incoming connections. Therefore, in this experiment we next examine a N-to-One scenario. Due to space limitations we will only show bandwidth in the following experiments. For the responder in this experiment, we only use a single thread to show the effects of scaling the requesters in isolation. Using multiple threads for responders is evaluated next in the M-to-N evaluation. To show the effects on bandwidth of using multiple requester threads, we gradually increase the number of servers used for requesters from 1 to 5. On each requester server we run 8 threads, therefore the maximum scale out with 5 server results in 40 requester threads. Additionally, we evaluate both design patterns (1) private connection and (2) dispatcher. The dispatcher design significantly reduces the connections needed, namely in the maximum scale-out from 40 connections to 5 only.

Fig. 5 shows the bandwidth and the number of requesters used for the private connection per thread design. We again show the effect of different response sizes from 64 byte to 16 KB. Interestingly, the pattern which is observable in the 4 KB plot also applies for smaller sizes. For instance,

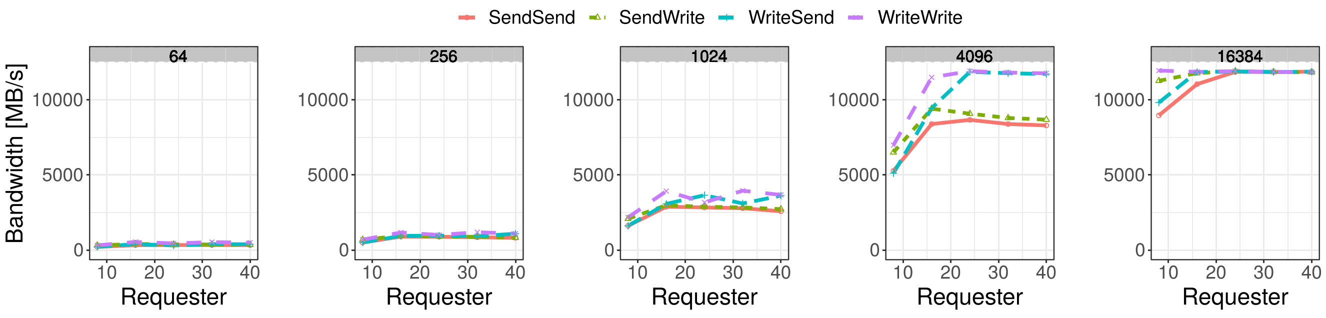


Fig. 5 Bandwidth – *Private Connection, For different response sizes in N-to-One scenario*

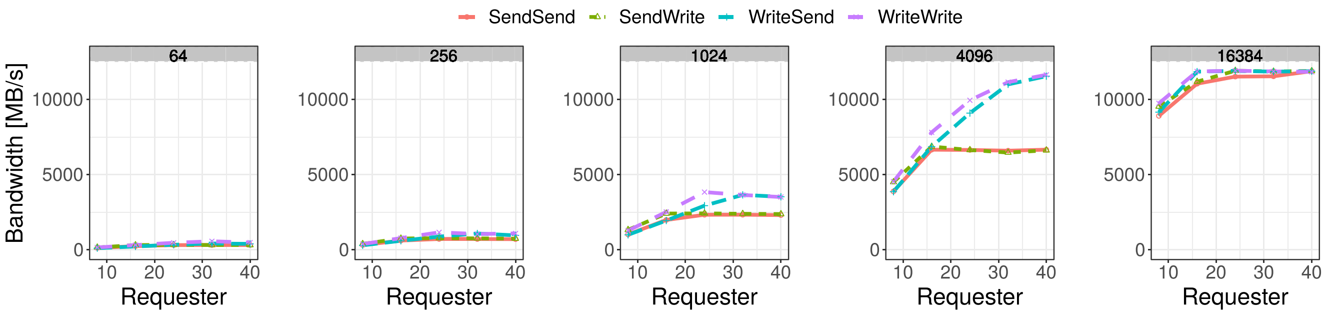


Fig. 6 Bandwidth – *Dispatcher, For different response sizes in N-to-One scenario*

64 and 256 byte responses SendSend achieves 500 MB/s WriteWrite 1.2 GB/s with 16 requesters. When scaling further up the performance remains stable for all verbs and is hardly affected. This performance gap also leads to the fact that WriteWrite and WriteSend reach the maximum possible bandwidth per NIC with 16 requesters and 4 KB responses. SendSend and SendWrite, in contrast, stagnate after 16 requesters and only reach between 9.5 and 8 GB/s. This performance difference diminishes when using 16 KB messages, because SendWrite and WriteWrite were already network bound with 4 KB SendWrite catches up with 16 requesters. SendSend, however, is only bandwidth bound with 40 requesters and clearly not as efficient as the other verbs.

The decline in bandwidth for SendSend and SendWrite with 4 KB responses is quite surprising when comparing the numbers with the one-to-one experiment. In the previous experiment especially SendWrite appeared to be very efficient. A possible explanation might be the RNIC's incapability to handle many incoming send connections as described in [11]. Consequently, we will test this hypothesis with the dispatcher design which reduces queue pairs to a minimum. From Fig. 6 we observe that the dispatcher has similar performance characteristics as the private connection per thread. We see the same pattern for smaller response sizes. However, for 4 KB we need 40 requester, instead of 16, to leverage the full bandwidth. Although, the dispatcher sends the messages asynchronously, the handover protocol (as mentioned in 4) from the requester thread to the dispatcher adds overhead. This overhead leads to a higher latency which in turn reduces bandwidth. Yet, we still have the decline in SendSend and SendWrite. Therefore, the number of connections did not cause the decline in our rack-scale experiment.

However, polling on the Shared Receive Queue could be more expensive than scanning over mailboxes in memory. Therefore we investigate the CPU cycles per 4 KB message measured on the responder shown in Fig. 7. When the responder only serves 8 requesters polling incurs a high CPU cycle overhead, because the responder spins some time until a message is received. This holds true for iterating the mailboxes as well for polling the Shared Receive Queue.

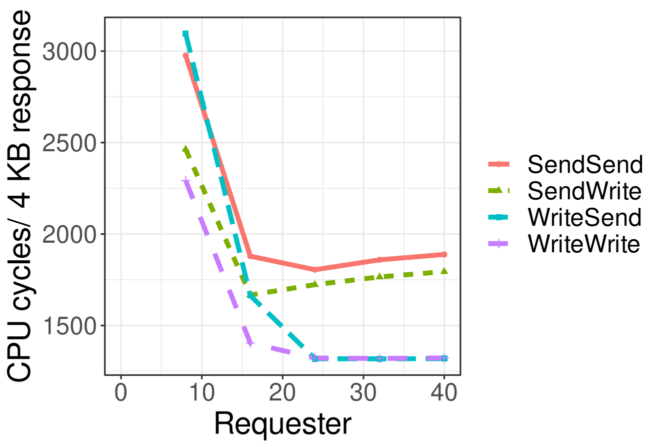


Fig. 7 CPU Cycles spent – *4 KB responses, N-to-One*

Table 1 Low-level metrics for SendSend (4 KB responses, 40 Requester threads, 1 Responder thread)

Method	Cycles %	Library
pthread_spin_lock	61%	libpthread-2.27.so
mlx5_poll_cq_1	17%	libmlx5.so.1.0.0

The more requesters (16) participate the less time is spent polling which decreases the cycles to around 1700 for all verbs. With even more requesters (> 20) the verbs diverge significantly, polling on the Shared Receive Queue always consumes around 1700 cycles and even increases again. In contrast, the cycles for scanning mailboxes is amortized with more requesters and continue to fall until the cycles per message stabilize at around 1100. When analyzing the Shared Receive Queue we observed that most cycles are spent in `phtread_spin_lock` and `mlx5_poll_cq` as shown in Table 1. Thus we conclude that polling on the Shared Receive Queue has a constant overhead which does not diminish with more requesters, i.e., messages.

7 Evaluation: N-to-M

Finally, in the last part of our evaluation, we analyze an N-to-M scenario by fixing the number of requesters but scaling the number of responders. However, different from the previous experiments we use both Mellanox cards and NUMA nodes per server to be able to run a requester / a responder per NUMA domain (i.e., in total we can simulate a cluster with 12 nodes).

From the previous experiment, we have seen that the network can be saturated with 16 requester threads per responder thread, therefore we scale responder threads in this experiment up to this ratio as well. More precisely, we use 9 nodes for requesters and run 10 threads per requester node (resulting in a total of 90 requester threads) while we use 3 servers for responders. The responder threads are scaled from 2, 4, and finally 6 per responder node resulting in 18 responder threads in total maximally.

For running the workload, each requester thread randomly chooses a responder thread for processing the request. Hence, with the setup that uses 6 responder threads in total, on average 15 requester threads need to be handled per responder thread while the other setups reflect a lower load per responder thread (as shown in Table 2). Moreover, we use the same setup as in the previous experiment and vary the message sizes while using both design patterns for the communication (private connections and dispatchers).

Fig. 8 shows the aggregated cluster bandwidth and total number of responder threads used. For small messages up to 1 KB we see that the aggregated bandwidth rises with additional responders. As seen in Fig. 4, small messages have a high instruction overhead causing a single core (responder) to be CPU bound. When adding new responders, i.e., CPU cores we thus can increase the throughput. Additionally, each requester needs to process fewer messages at the time. This effect can be observed in the 4 KB plot, in which SendWrite, surprisingly when looking at the N-to-One experiment, outperforms WriteSend with well over 40 GB/s. When comparing the aggregated bandwidth of WriteWrite with the one achieved in the N-to-One experiments its staggering that the bandwidth is far from the possible aggregated bandwidth of 66 GB/s. This effect can be explained with Table 2, when scaling the number of responders, the mailboxes are often empty and the threads thus simply poll without doing any actual work. Hence, the full aggregated bandwidth cannot be achieved.

To investigate this effect in more depth, we conducted the following microbenchmark: We evaluate a 90-to-1 setup to determine if the amount of mailboxes for the alternatives that use a Write operation (to implement the request) or connections mapped to the Shared Receive Queue for alternatives that use a Send operation to implement the request (i.e., SendWrite and SendSend) were limiting performance. Fig. 9a shows the result of this microbenchmark using 4Kb responses. These results show that the alternatives that use a Write for the request (i.e., WriteWrite and WriteSend) are able to leverage the full bandwidth despite the high mailbox count. The alternatives that use a Send for the request (i.e., SendWrite and SendSend), however, do not achieve a higher bandwidth, which is comparable to the findings from the N-to-One experiment.

Thus, focusing on Fig. 8 with 4 KB responses we can see that the SendSend and SendWrite alternative with 6 responders achieve the expected performance. The difference between the 90-to-1 microbenchmark and the M-to-N Experiment is the number of average requesters per responder. In the 90-to-1 setup every requester was constantly transmitting messages, but in the M-to-N scenario only a fraction of the requesters send to the same responder resulting in many empty mailboxes.

From Table 2 we can observe that when having 6 responders an average of around 15 requesters need to be served, which corresponds to around 83 percent of empty mailboxes. Therefore the next microbenchmark, Fig. 9b, shows the effect of empty mailboxes. We used the 90-to-1 setup but only a certain percentage of requesters transmit a mes-

Table 2 Average number of requesters threads per responder thread in different scenarios

Responder	Avg. Req./Resp.	Avg. % Empty Mailboxes
6	15	83.3
12	7.5	91.6
18	5	94.4

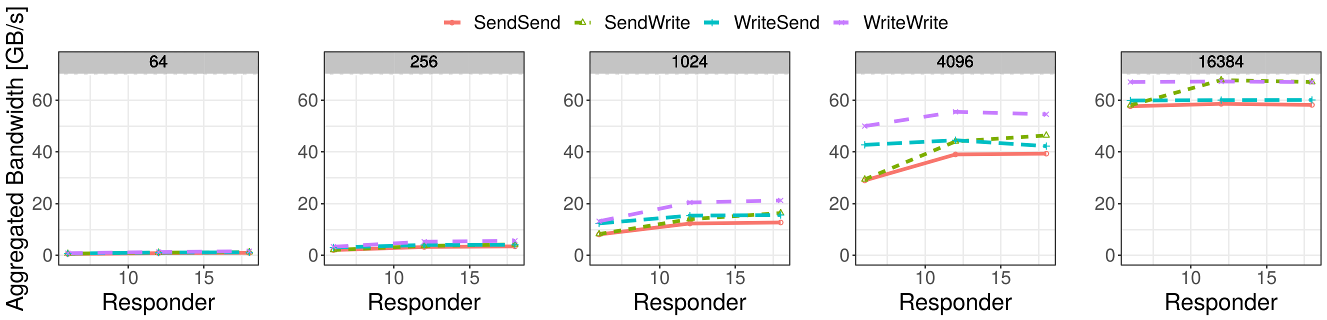


Fig. 8 Private Connection: Aggregated BW for increasing number of responders – *different response sizes, M-to-N*

sage. The rest waited as long as it would take to contact another responder, i.e, 8 microsecond latency. We observe that the bandwidth is quite stable until 70 percent. After 70 percent the performance drops significantly with 80 percent empty mailboxes around 8 GB/s can be achieved and with 90 only around 4 GB/s. The achieved bandwidth of 8 GB/s correspond to the bandwidth each responder achieves in the M-to-N scenario.

To summarize, the effect of empty mailboxes is quite high when having more than 80 percent empty. When focusing again on Fig. 8 we observe when increasing the number of receivers (see Table 2), i.e., reducing the average number of requester per responder, the bandwidth increases. For alternatives that use a Write operation to implement the request (i.e., WriteWrite and WriteSend) we observed that when having high empty mailboxes the instructions/message increases. Therefore, it takes longer to get a response on the wire. With more responders, this latency can be hidden, but only until enough messages are received, which explains why the bandwidth drops again after 12 responders.

In contrast to WriteWrite and WriteSend, SendWrite does not decline after 12 Responder threads instead it inclines. When looking at Table 2 we can see that only 5 requesters needs to be served on average. Therefore, the scenario is closer to the One-to-One scenario in which SendWrite outperformed WriteSend and SendSend. Furthermore, from Fig. 7 we observed that the cycles of other verbs get amortized with new requesters. In this scenario, we effectively reduce the number of requesters per responder. This diminishes the amortization advantage of other verbs and in turn SendWrite is more competitive.

Lastly, we analyzed the dispatcher design in the N-to-M scenario. Fig. 10 again shows the aggregated cluster bandwidth and total number of responder threads. For brevity we only show the measurements 4 KB and 16 KB as they are most interesting. For smaller messages up to 4 KB we can see that the private connection design from Fig. 8 outperforms the dispatcher design. Interestingly, with 16 KB messages WriteSend and SendSend in the dispatcher design achieve up to 5 GB/s more aggregated bandwidth compared to the private connection design. In contrast, SendWrite and WriteWrite are performing equally well in both designs.

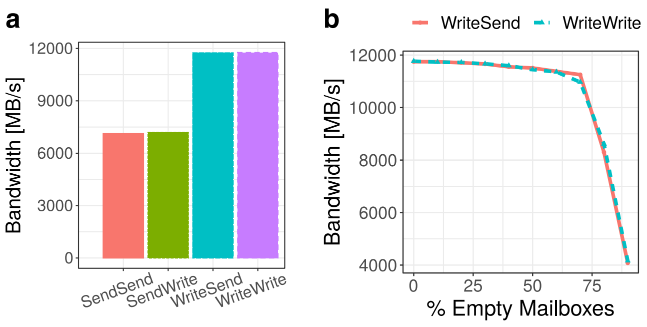


Fig. 9 Bandwidth and Effect of Empty Mailboxes – *4 KB responses*. **a** Bandwidth, *4 KB response, 90 to One*, **b** Effect of Empty Mailboxes, *90 to One*

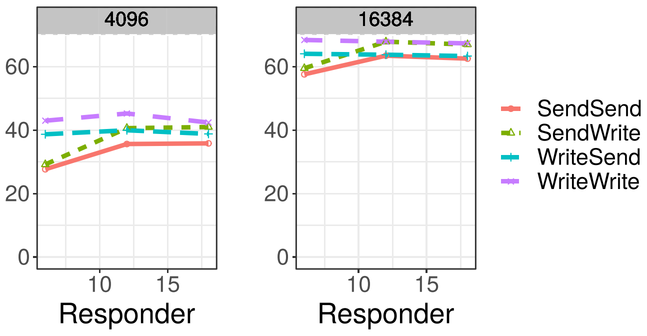


Fig. 10 Dispatcher: Aggregated BW for increasing number of responders – *different response sizes, M-to-N*

8 RDMA Optimizations

In the previous sections, we evaluated various communication patterns in a holistic manner. This section, in contrast, discusses several low-level RDMA optimizations we analyzed in our benchmark code. Additionally, we present some optimization techniques we have not yet implemented but are worth mentioning.

We have seen that RDMA can achieve low latency and high bandwidth. In fact, the latency is so low that small inefficiencies directly translate to decreased bandwidth and increased latency. For instance, this impact has been observed in Sect. 7 when the number of empty mailboxes were high and the polling became expensive. Therefore, we implemented techniques like *selective signaling* and *inlining* [5, 11] for small messages up to 220 byte.

Selective Signaling. As described in Sect. 2, a signaled WQE creates a completion event once processing finished. However, the completion event is generated by the RNIC which incurs overhead and reduces throughput as shown in [11]. To avoid completion event generation all together, the application can specify the WQE to be *unsignaled*. However, even though the completion event is not constructed, it still consumes completion queue resources. Because the completion queue is bounded, the application needs to periodically post a signaled WQE and process the generated completion to avoid depleting completion queue resources. Selective signaling is a technique to reduce completion events and to prevent depletion of resources [5], thereby improving performance. The application posts n-1 unsignaled WQE and then the n-th WQE signaled.

Inlining is another technique, applicable to messages up to 220 byte, to reduce work inside the RNIC. In general, once the RNIC processes a WQE it fetches the payload of the message over the PCIe bus. Inlining allows the application to directly attach the payload to the WQE, which eliminates the need to fetch data over the PCIe bus. Consequently, the message can be faster transmitted aqnd latency decreases, as shown by Kalia et al. [11].

Another important aspect for an efficient usage of RDMA is **memory allocation and preparation**. In particular, the NUMA architecture has to be considered carefully when allocating RDMA memory to ensure that the RNIC fetches NUMA-local data. Otherwise, data is fetched from remote NUMA regions via the QPI/UPI bus incurring a high overhead. To allow the RNIC to write and read memory, the region must be pinned and registered on the RNIC first. This registration should be avoided on the hot path as it is quite expensive. Furthermore, to translate virtual to physical memory addresses, the RNIC caches virtual to physical address translations. To reduce address translation cache misses often huge pages are used [11].

A feature we implicitly exploited is **Data Direct I/O (DDIO)**, which is provided by recent Intel CPUs (Sandy Bridge and later). With DDIO, the DMA executed by the RNIC to read (write) data from (to) remote memory, places the data directly in the CPU L3 cache. DDIO supports two modes of operation [9]: (1) *Write Update* if the memory address is resident in the cache an in-place update is performed, and (2) *Write Allocate* causing allocation in the L3-cache if data is not yet cached. To reduce cache thrashing, DDIO's Write Allocate is limited to 10% of the L3 cache [9]. We measured the effect of DDIO during our experiments, especially when using WRITE in combination with the mailboxes. In our benchmark, the responder polled mailboxes, which placed them in the CPU cache. Incoming messages are then directly written to the L3-cache with DDIO. This saves memory bandwidth and reduces latency by avoiding a full cache miss.

An optimization for the mailbox design that we have not implemented, but should be mentioned, is exploited by L5 [8]. On the responder site, L5 uses a dense mailbox buffer, in which each client occupies only a single byte. Additionally, a message buffer is used, similarly to our mailbox design described in Sect. 4, in which the payload is written. To transfer a message, the requester writes the payload to the message buffer and then issues a second write of a single byte to the dense mailbox buffer to signal completion. The responder benefits from decreased cache misses and instructions, because only the dense region is polled. For instance, with 64 clients only one cache line needs to be polled. Especially, our N to M experiment has shown that scanning empty mailboxes is expensive and the dense mailbox design would have mitigated that overhead. However, the drawback of this design is that the requester needs to write two messages instead of one, which in turn influences the latency and increases the operations per second in the RNIC.

Finally, for completeness, we used the following performance relevant settings in all experiments: MLX5_SINGLE_THREADED=1, MLX_QP_ALLOC_TYPE="HUGE", and MLX_CQ_ALLOC_TYPE="HUGE". The first setting disables locking when used in a single threaded application, and the two remaining variables allocate the queue pair and the completion queue on huge (2 MB) pages.

9 Discussion and Conclusions

In this paper, we have presented a systematic evaluation of RDMA communication patterns for various use-cases. Our main findings are: (1) In the One-to-One experiment, we have seen that WriteWrite and SendWrite perform best. There was a 40% difference in performance between WriteWrite and SendWrite compared to SendSend and WriteSend. We further observed that the CPU cycles per byte decline with increasing messages. This fact is quite interesting as the trade-off between latency and CPU overhead can be tailored to the application needs. (2) In the N-to-One experiment, WriteWrite maintained the performance observed in the first experiment. Surprisingly, however, WriteSend outperformed SendWrite. We have seen that the reason why SendSend and SendWrite did not scale is that the cycles spent do not get amortized with additional requesters as it is the case for WriteWrite and WriteSend. Therefore, the constant overhead leads to a stagnation in bandwidth. (3) The results from the M-to-N experiment confirmed the previous findings, but added another interesting insight: WriteWrite and WriteSend performance is determined by the number of empty mailboxes.

Acknowledgements This work was funded by the German Research Foundation (DFG) under grant BI2011/1.

Funding Open Access funding provided by Projekt DEAL.

References

1. Alonso G, Binnig C, Pandis I, Salem K, Skrzypczak J, Stutsman R, Thostrup L, Wang T, Wang Z, Ziegler T (2019) DPI: the data processing interface for modern networks. In: CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research Asilomar, 13.01.–16.01. (Online Proceedings. www.cidrdb.org (2019). http://cidrdb.org/cidr2019/papers/p11-alonso-cidr19.pdf)
2. Barthels C, Alonso G, Hoefler T (2017) Designing databases for future high-performance networks. IEEE Data Eng Bull 40(1):15–26 (http://sites.computer.org/debull/A17mar/p15.pdf)
3. Barthels C, Alonso G, Hoefler T, Schneider T, Müller I (2017) Distributed join algorithms on thousands of cores. Proc VLDB Endow 10(5):517–528. https://doi.org/10.14778/3055540.3055545 (http://www.vldb.org/pvldb/vol10/p517-barthels.pdf)
4. Barthels C, Loesing S, Alonso G, Kossmann D (2015) Rack-scale in-memory join processing using RDMA. In: Sellis TK, Davidson SB, Ives ZG (eds) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data Melbourne, 31.05.–4.06. Association for Computing Machinery, New York, pp 1463–1475 https://doi.org/10.1145/2723372.2750547
5. Binnig C, Crotty A, Galakatos A, Kraska T, Zamanian E (2016) The end of slow networks: It's time for a redesign. Proc VLDB Endow 9(7):528–539. https://doi.org/10.14778/2904483.2904485
6. Dragojevic A, Narayanan D, Castro M, Hodson O (2014) Farm: fast remote memory. In: Mahajan R, Stoica I (eds) Proceedings of the 11th USENIX symposium on networked systems design and implementation Seattle, April 2–4. USENIX Association, Berkeley, pp 401–414 (https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87)
7. Dragojevic A, Narayanan D, Nightingale EB, Renzelmann M, Shamis A, Badam A, Castro M (2015) No compromises: distributed transactions with consistency, availability, and performance. In: Miller EL, Hand S (eds) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015 Monterey, 4.10.–7.10. ACM, Monterey, pp 54–70 https://doi.org/10.1145/2815400.2815425
8. Fent P, Renen A van, Kipf A, Leis V, Neumann T, Kemper A (2020) Low-latency communication for fast DBMS using RDMA and shared memory. In: 36th IEEE International Conference on Data Engineering, ICDE 2020 Dallas, 20.04.–24.04. 2020 IEEE, Dallas, pp 1477–1488 https://doi.org/10.1109/ICDE48307.2020.00131
9. Intel Corporation Intel® data direct I/O technology: technology brief. https://www.intel.de/content/www/de/de/io/data-direct-i-o-technology-brief.html
10. Kalia A, Kaminsky M, Andersen DG (2014) Using RDMA efficiently for key-value services. In: Bustamante FE, Hu YC, Krishnamurthy A, Ratnasamy S (eds) ACM SIGCOMM 2014 Conference, SIGCOMM'14 Chicago, 17.08.–22.08. ACM, Chicago, pp 295–306 https://doi.org/10.1145/2619239.2626299
11. Kalia A, Kaminsky M, Andersen DG (2016) Design guidelines for high performance RDMA systems. login Usenix Mag. 41(3). https://www.usenix.org/publications/login/fall2016/kalia
12. Kalia A, Kaminsky M, Andersen DG (2016) Fasst: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In: Keeton K, Roscoe T (eds) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016 Savannah, 2.11.–4.11. USENIX Association, Berkeley, pp 185–201 (https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia)
13. Kulkarni C, Kesavan A, Ricci R, Stutsman R (2017) Beyond simple request processing with ramcloud. IEEE Data Eng Bull 40(1):62–69 (http://sites.computer.org/debull/A17mar/p62.pdf)
14. Li B, Ruan Z, Xiao W, Lu Y, Xiong Y, Putnam A, Chen E, Zhang L (2017) Kv-direct: high-performance in-memory key-value store with programmable NIC. In: Proceedings of the 26th Symposium on Operating Systems Principles Shanghai, 28.10.–31.10. ACM, Shanghai, pp 137–152 https://doi.org/10.1145/3132747.3132756
15. Li F, Das S, Syamala M, Narasayya VR (2016) Accelerating relational databases by leveraging remote memory and RDMA. In: Özcan F, Koutrika G, Madden S (eds) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016 San Francisco, 26.06.–01.07. ACM, New York, pp 355–370 https://doi.org/10.1145/2882903.2882949
16. Liu F, Yin L, Blanas S (2017) Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In: Alonso G, Bianchini R, Vukolic M (eds) Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017 Belgrade, 23–26.04. ACM, Belgrade, pp 48–63 https://doi.org/10.1145/3064176.3064202
17. Loesing S, Pilman M, Etter T, Kossmann D (2015) On the design and scalability of distributed shared-data databases. In: Sellis TK, Davidson SB, Ives ZG (eds) Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data Melbourne, 31.05.–4.06. Association for Computing Machinery, New York, pp 663–676 https://doi.org/10.1145/2723372.2751519
18. Lu X, Shankar D, Panda DK (2017) Scalable and distributed key-value store-based data management using rdma-memcached. IEEE Data Eng Bull 40(1):50–61 (http://sites.computer.org/debull/A17mar/p50.pdf)
19. Mitchell C, Geng Y, Li J (2013) Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In: Birrell A, Sirer EG (eds) 2013 USENIX Annual Technical Conference San Jose, 26–28.06. USENIX Association, Berkeley, pp 103–114 (https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell)
20. Ousterhout JK, Agrawal P, Erickson D, Kozyrakis C, Leverich J, Mazières D, Mitra S, Narayanan A, Ongaro D, Parulkar GM, Rosenblum M, Rumble SM, Stratmann E, Stutsman R (2011) The case for ramcloud. Commun ACM 54(7):121–130. https://doi.org/10.1145/1965724.1965751
21. (2012) Delivering application performance with oracle's Infiniband technology. https://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf

22. Rödiger W, Idicula S, Kemper A, Neumann T (2016) Flow-join: adaptive skew handling for distributed joins over high-speed networks. In: 32nd IEEE International Conference on Data Engineering, ICDE 2016 Helsinki, 16–20.05. IEEE Computer Society, Helsinki, pp 1194–1205 https://doi.org/10.1109/ICDE.2016.7498324
23. TU Darmstadt Data management lab: RDMA communication patterns code. https://github.com/DataManagementLab/RDMA_Communication_Patterns
24. Vienne J, Chen J, Wasi-ur-Rahman M, Islam NS, Subramoni H, Panda DK (2012) Performance analysis and evaluation of infiniband FDR and 40gige roce on HPC and cloud computing systems. In: IEEE 20th Annual Symposium on High-Performance Interconnects, HOTI 2012 Santa Clara, 22–24.08. IEEE Computer Society, Santa Clara, pp 48–55 https://doi.org/10.1109/HOTI.2012.19
25. Zamanian E, Binnig C, Kraska T, Harris T (2017) The end of a myth: distributed transaction can scale. Proc VLDB Endow 10(6):685–696. https://doi.org/10.14778/3055330.3055335
26. Ziegler T, Tumkur Vani S, Binnig C, Fonseca R, Kraska T (2019) Designing distributed tree-based index structures for fast rdma-capable networks. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD ’19. Association for Computing Machinery, New York, pp 741–758 https://doi.org/10.1145/3299869.3300081