The impact of RDMA on the design of networked systems

Konstantin Taranov  
ETH Zurich  
Switzerland  
ktaranov@inf.ethz.ch

Fabian Fischer  
ETH Zurich  
Switzerland  
fischerf@ethz.ch

Torsten Hoefler  
ETH Zurich  
Switzerland  
htor@inf.ethz.ch

ABSTRACT
Developers of networked systems often work with low-level RDMA libraries to tailor network modules to take full advantage of offload capabilities offered by RDMA-capable network controllers. Because of the huge design space of networked data access protocols and variability in capabilities of RDMA infrastructure, developers tend to reinvent and reimplement common data exchange protocols, wasting months of development yet missing various performance and system capabilities. In this work, we summarise and categorize RDMA data exchange protocols and elaborate on what features they can offer to networked systems and what implications they have on their memory and network management.

CCS CONCEPTS
• Information systems → Data access methods; • Networks → Network architectures.

KEYWORDS
RDMA, Networked Systems, System design

1 INTRODUCTION
Remote Direct Memory Access (RDMA) is a network mechanism that empowers applications to access memory of remote processes without the involvement of their CPUs. RDMA-capable network controllers (RNICs) perform memory accesses using dedicated DMA controllers, allowing specialized network transport protocols to write incoming packet payloads directly to the local memory and fetch outgoing payloads from the memory, bypassing the CPU.

RDMA network protocols use an asynchronous programming model, where applications submit non-blocking communication work requests to local RNICs, that use DMA to perform them without CPU involvement, and then asynchronously fetch corresponding completion events. The application can submit many communication requests in parallel without waiting for their completion, thereby achieving higher utilization of networking hardware. As a result, the CPU load is much lower with RDMA networks than with classical networks employing the blocking POSIX API.

In this work, we primarily focus on the IB verbs library [4], the most popular standard for RDMA programming, which defines RDMA requests and RDMA hardware capabilities that can be supported and implemented by RDMA infrastructure. RDMA requests. RDMA transport protocols can fundamentally offer SEND, WRITE, READ, and ATOMIC network requests to access remote memory and a local RECEIVE request to control destination memory of incoming SEND requests. SEND packets do not contain information about remote destination addresses, therefore, pre-posted RECEIVE requests are required to perform data reception. Unlike SEND, other requests are equipped with the destination address, allowing them to be executed without the control of the remote application (such access semantics is often called "one-sided access"). WRITE and READ requests allow writing content of local
buffers to remote buffers and reading the content of remote buffers to local buffers, respectively. ATOMIC requests are requests that allow performing compare-and-swap and fetch-and-add over the network.

The IB library also defines WRITE_WITH_IMM requests that extend silent WRITE requests to generate completion events at the receiver, but at the cost of involving the receiver’s CPU, which submits a RECEIVE request. The generated completion event contains the length of the WRITE request and an integer, called immediate data (IMM), specified by the sender.

**RDMA transports.** All discussed RDMA requests are supported by InfiniBand-based protocols: RoCE and InfiniBand [9]. These protocols traditionally ensure in-order packet delivery, though each message can consist of multiple packets. For the in-order delivery, static routing of packets and full message re-transmission in case of packet drops are employed. As a result, InfiniBand-based protocols may suffer from head-of-line blocking and network congestion [54].

The EFA [8] protocol alleviates these problems by adopting adaptive routing and allowing messages to be delivered out-of-order. With adaptive routing, packets may take up different paths in a network. To address the head-of-line blocking, each message of the EFA protocol consists of a single packet. However, the existing version of the protocol only supports SEND and READ requests.

Similarly, IRMA [44] limits messages to one packet to enable out-of-order packet processing and reduce congestion. Additionally, IRMA does not support SEND and WRITE requests as such packets with large payloads can overload the target RNIC or even cause side-effects such as unacknowledged memory writes (i.e., the target executes memory write but the initiator thinks that it failed). Fundamentally, the protocol only supports READ requests, however, the protocol offers emulation of WRITE requests that are implemented as requests to READ, therefore, requiring two full round trips.

RDMA has a long history in the high-performance computing field, where it was exposed via lower-level interfaces such as DMAPP or uGNI in Cray interconnects [41], libfabric [2], UCX [5], or Porthals 4 [10]. These interfaces aim at run-to-completion applications like MPI [21] and are only offered by specialized clusters (e.g., Cray supercomputers) while we aim to support client-server applications that can be deployed on commodity RNICs and in public clouds.

**RNIC capabilities.** RNICs implement various mechanisms that are used by applications to lower latency, increase throughput, and improve memory management. Scatter-gather list allows applications to specify source and receive buffers as a list of non-contiguous buffers. Shared receive allows applications to share RECEIVE requests between multiple connections. Device memory allows applications to allocate buffers in the memory of RNICs for networking. Inline requests allow senders to inline data to send work requests and receivers to inline incoming payloads to receive completions, removing one DMA transaction. On-demand paging empowers RNICs to dynamically fetch virtual to physical memory translation entries, removing the need to explicitly register the memory.

**Data ordering.** Three factors can affect the delivery order of transmitted bytes: message ordering, packet ordering of a single message, and DMA ordering. In-order message delivery (e.g., InfiniBand) guarantees the delivery of messages in the order they have been issued by the sender. Some protocols (e.g., EFA and IRMA) may reorder messages due to adaptive network routing. If a message consists of multiple packets, some protocols (e.g., InfiniBand with adaptive routing) can reorder packets of the message, resulting in out-of-order data placement at the receiver. EFA and IRMA do not support multi-packet messages at all. However, even if packets are delivered in order, it is not guaranteed that DMA operations are performed in the same order. Applications can intentionally relax DMA ordering to memory regions to improve the performance of RDMA requests to them. In our analysis, we say an RDMA protocol offers in-order byte delivery of a message if it ensures in-order delivery of its packets and in-order DMA operations.

### Table 1: Features of RDMA-capable protocols.

| Protocol          | Requests | Message delivery | Messages |
|-------------------|----------|------------------|----------|
| InfiniBand and RoCE [9] | Send, Write, Read, Atomic | In-order | Multi-packet |
| EFA [8]           | Send, Read  | Out-of-order     | Single-packet |
| IRMA [44]        | Read, Write   | Out-of-order     | Single-packet |

*Not all requests must be implemented by RNIC providers.

* Some RNICs offer out-of-order delivery [14, 34].

* A write is implemented as a request to read.

### 3 COMMUNICATION IN RDMA SYSTEMS

We focus on data access and data exchange protocols used by RDMA-enabled systems. We look at each data exchange as a uni-directional data channel that allows a sender to communicate data to a receiver. Essentially, all data exchange protocols need at least two uni-directional data channels to acknowledge the data reception for reliable data delivery. Exchange protocols often use network acknowledgments of reliable transports to constitute an explicit uni-directional data channel. Alternatively, systems explicitly send data via another uni-directional channel to acknowledge message reception. Some high-level protocols can piggyback acknowledgments with data sent via a multi-purpose uni-directional communication channel. For example, RPC replies can be used as an indicator of the RPC request reception. An alternative approach is to build a dedicated channel for only acknowledging the data reception.

Overall, applications can build various data exchange protocols by combining explicit and implicit uni-directional data channels, making them fundamental building blocks of data communicators.

Modern RDMA-enabled systems pose various requirements to network modules and underlying RDMA exchange protocols. To identify these requirements we analyzed numerous RDMA-enabled systems (see “Related Work” in Table 2) and reviewed fundamental features of their data exchange channels. We list these features below and elaborate on their importance in the system design.

**Performance.** Even though existing systems often argue about their performance in terms of latency and throughput, the performance of data exchange protocols is not portable across different RDMA networks. We propose to measure the performance of protocols in metrics that are independent of RDMA infrastructure and portable across RDMA networks. Instead of the latency, we count the number of half-round-trips (HRTs) required to send data via a data channel. Instead of the throughput, we count the number of issued RDMA network requests. Thus, the latency of the protocol is proportional to the number of HRTs and the throughput is inversely proportional to the number of issued RDMA requests. For example,
SEND and WRITE requests fundamentally perform alike as they require one request and one HRT to deliver data, even though WRITEs are slightly faster than SENDs in practice [24].

**Blocking receiver.** RDMA requests offer two ways of notifying the receiver about incoming data: the receiver can be notified with a completion event or it can poll/read a special value from memory indicating message reception. While memory polling is reported to provide a better performance, it wastes a lot of CPU cycles on actively loading data from the memory. The completion event, on the other hand, can be received in blocking mode via interrupts. As a result, the receiver can block at the completion events waiting for an interrupt, saving CPU cycles. For example, the NVMe-oF kernel module [28] blocks on receive completions to reduce the CPU load.

**True zero-copy.** Data copies waste a lot of CPU cycles and pollute CPU caches, encouraging systems to avoid them. Formally, if an initiator knows the source address of data and a protocol allows the sender to send this data without copies then we call the protocol *zero-copy on send*. If a communication initiator knows the destination address of data and a protocol allows the receiver to get data to that address without additional copies then the protocol is *zero-copy on receive*. For example, SEND-based protocols are not zero-copy on receive, as the sender has no control over the destination address at the receiver. Thus, if we want to build a sender which must deliver data to a specific NUMA region depending on a hash of a message, we cannot do it with a single SEND-based protocol, and the receiver needs to copy data from pre-posted receive buffers to its final destination. Note that the communication initiator can be the sender or the receiver depending on the protocol (e.g., READ initiators are data receivers, and their passive targets are data senders).

**Variable size messages.** Not all protocols can efficiently manage memory for the reception of variable size messages and can only receive messages of one predefined size. Such protocols force applications to pre-allocate sufficiently large receive buffers fitting all possible message sizes, thereby increasing the total memory usage of applications. For example, the main limitation of a SEND request is that it fully uses the pre-posted buffer at the receiver, even if incoming data was smaller than the receive buffer, causing significant memory fragmentation for variable messages.

**Memory need for N:1 communications.** Modern systems may include several communication nodes. Of course, any two communicating nodes of a cluster can establish a point-to-point datapath, but it leads to significant memory usage that grows with the number of remote nodes. For example, if one receiver needs to receive messages from N nodes using point-to-point datapaths it will require managing receive buffers for each sender independently. For the N:1 communication, we later present protocols which enable efficient memory sharing of receive buffers between independent senders. For the 1:N communication, which multicasts messages to N nodes, all studied protocols can efficiently share their data source.

**Passive networking.** It describes whether a networked system can manage a communication channel without explicitly working with the RNIC (i.e., without submitting verb requests and polling queues). It means that systems can send and receive data using only load and store CPU instructions, thereby removing contention from the RNIC’s queues. Formally, a *sender supports passive networking* if it does not submit any work requests to the RNIC (see "# Requests Send" in Table 2). A *receiver supports passive networking* if it does not submit any work requests to the RNIC and also does not support blocking receive (see "# Requests Recv" and "Blocking Recv" in Table 2), which entails polling completion queues.

4 **EXCHANGE PROTOCOLS**

We implemented [1] and analyzed twenty uni-directional data channel protocols that can be used as building blocks for any networked system. All proposed protocols are fundamentally different in memory management (i.e., how memory is organized on the sender and the receiver to communicate data) or in network management (i.e., how the sender and the receiver interact with the RNIC to enable communication). We group channels into six categories/families depending on memory organization and the used underlying RDMA requests. For each channel, we indicate supported networking features and requirements to the underlying RDMA transport in Table 2.

Before diving into data channels, we outline hardware and software tricks that can be used to enable some protocols. The former tricks require special RNIC capabilities, whereas the latter can be implemented on any modern operating system.

**Hardware tricks.** RNIC capabilities listed in Section 2 are extensively used to improve performance of systems [15, 24, 25, 39, 48]. We argue the inlining and on-demand paging capabilities do not fundamentally affect the memory and data layers of networked systems as they only reduce overhead on DMA engines or simplify memory registration. In contrast, the other capabilities, even though they have been introduced for improving performance, may offer new memory management and networking features to networked systems.

Scatter-gather lists can be used to achieve true zero-copy in the case when a channel needs to append protocol-specific prefixes or suffixes to immutable data buffers. These extra data can include data length or special signaling bytes that are expected by the receiver. For example, scatter-gather lists have been used to implement a zero-copy serialization library [39]. Shared receive queues can be used to share receive buffers across multiple RDMA connections, significantly decreasing memory usage for the N:1 communication. Device memory offers additional memory regions located closer to the network compared to the DRAM, thereby accelerating applications that often send recently received data [37].

**Software tricks.** Existing RDMA protocols use virtual addresses in their RDMA requests to access memory. The use of virtual addresses allows systems to avoid fragmentation of physical memory at anything coarser than page granularity. RDMA protocols can take advantage of virtual addresses as operating systems do. Importantly, physical memory segments can be mapped to several virtual addresses allowing to access the same physical memory via different memory addresses. MICA [29] and FaRM [16] use this trick to build a virtually circular RDMA-accessible buffer by mapping the virtual memory addresses right after the end of the buffer to the physical pages of the original buffer, thereby making the end of the buffer appear circular and contiguous in virtual space. As a result, local and remote accesses to such circular buffer can be performed without range checking and RDMA writes near the end of the buffer do not cause splitting an RDMA request into two requests.
### 4.1 Send/Recv protocols

The Send/Recv protocol family, where a sender uses SEND requests to send data and a receiver pre-posts fixed size RECEIVE requests to get data, offers three different channels: Normal, Shared, and Bufferless. These channels do not support variable messages as one SEND request consumes one RECEIVE request regardless of message sizes. As messages are received pre-posted buffers controlled the receiver, the sender has no control over the exact destination of the message, making the protocols not zero-copy on receive.

**Normal.** The sender and the receiver use a private channel to communicate data, limiting the receiver to receive messages only from one sender, entailing large overhead for the N:1 communication pattern. Senders with reliable transport often utilize network acknowledgments as an implicit data channel to confirm data reception. However, if the receiver does not promptly post RECEIVE requests, the reliable transport will cause delivery failure, which causes costly connection disconnects for InfiniBand and a request timeout for EFA and 1RMA. Therefore, systems often explicitly inform peers about the number of posted RECEIVE requests to prevent network failures.

**Shared receive.** Shared receive allows the receiver to share receive buffers between multiple senders. If not enough buffers are pre-posted, the reliable transport of InfiniBand is subject to connection failures during network bursts. Overall, shared receive has the same features as the normal version but helps to reduce memory usage for the N:1 communication.

**Bufferless.** Bufferless data channel allows sending an integer using payload-less messages with IMM data. The receiver must still post RECEIVE requests to receive IMM data but they may have no buffers attached. This channel is often used to enable low-latency notifications without the need to register and manage memory.

### 4.2 Write slot protocols

Write slot protocols rely on writing messages to pre-allocate fixed size memory regions, called mailboxes or slots, using WRITE requests. The main property of this family is that the sender and receiver support zero-copy. As WRITEs are silent, we outline all approaches to inform the receiver about incoming messages.

**Inlined bell.** This method reserves a special memory location within each mailbox slot that indicates that a message is fully written. The sender writes a message into the mailbox so that it changes the state of the bell field. After processing, the receiver clears the bell to reuse the slot. In some cases, systems use remote memory as temporal remote storage that is not required to be notified about incoming data. For example, InfiniSwap [19] silently swaps out pages to remote memory to fetch them later when they are required. We assign such use-cases to the inlined bell category since they both logically manage memory as plain fixed size mailboxes.

**Detached bell.** This method stores the bell outside the mailbox slot. In this case, the sender writes data into the mailbox and then sends an additional request into the bell to signal the message reception. Unlike the inlined bell approach, this channel requires sending two RDMA messages that must come in-order as the signal should arrive after the message is fully received.

**IMM-based.** This method uses one WRITE_WITH_IMM request to signal the message arrival via a completion event at the receiver. The sender can encode into the IMM data the used mailbox.

**Reserve slot.** The main disadvantage of the write protocols is that they have high memory usage, especially for the N:1 communication. The limitation comes from the fact that slots cannot be shared and the receiver allocates slots for each client individually. To address this issue, we can introduce a data path to lock and unlock slots for a
specific sender. The sender can use any two-way protocol to reserve slots before using them and then later unlock them, thereby enabling buffer sharing between senders at the cost of extra round trips. The slot reservation can be implemented via bufferless sends to reduce memory usage. Note, depending on the implementation, one sender can take up all slots preventing other clients from sending messages.

4.3 Ring buffer protocols

Ring buffers rely on writing messages to a remote pre-allocated buffer with WRITE requests. The pre-allocated buffer forms a circular ring, allowing the sender to write messages one by one into the same region. To enable efficient implementation of circular buffers, systems use the software trick that maps the virtual address after the ring buffer to the beginning of the buffer, allowing to access ring buffer if it was truly circular in memory (see Section 4). We distinguish the ring buffer algorithms by the way of informing the receiver about the incoming messages.

Detached bell. In this case, the detached bell stores the current head offset of the ring buffer. The sender writes a message into the ring buffer and then writes the new head into the bell. The RDMA messages need to come in-order as the bell should arrive after the message is written. To send messages of variable length, the sender needs to append the message length to each message.

Inlined bell without zeroing. This method appends the message length before the message as a bell. In addition, it appends the value "one" to the end of each message to inform the receiver that the message is fully written (see Figure 1). The sender writes the length, data, and the completion with one WRITE request. The receiver polls the length field of the upcoming message of the circular buffer and once it is not zero it polls the inlined completion after the message until it is not zero. Once the message is processed, the receiver needs to clear the memory region occupied by the message to unset all possible bells and completions of future messages.

Inlined bell without zeroing. The main disadvantage of the previous algorithm is the need to clear memory after message processing. We found that it is possible to implement a ring buffer without zeroing and still use a single WRITE request. For that we flip the order of messages in the ring buffer: instead of placing a new message after the last message, we place the new message before it in memory. Figure 2 shows how writing a message works. Each message starts with the value "zero" and is followed by the message length. When a new message is placed, the value "zero" unsets the bell for an upcoming message and the length value sets the bell of the current message, which is at the end of the current message and at the beginning of the previous message. As a result, the sender zeros the bell of the upcoming message, and the receiver does not need to clear all memory. The protocol requires in-order byte and message delivery, as written bytes must be strongly ordered even across messages.

4.4 Shared ring buffer protocols

Ring buffer protocols suffer from high memory usage when a receiver needs to receive messages from many endpoints. To reduce the memory usage we propose shared ring buffer protocols that allow sharing one ring buffer between many writers. Each sender before writing a message needs to reserve a slot in the shared circular buffer using ATOMIC operations. The sender atomically fetches the current head offset of the ring and adds the length of the message it wants to write. Then it writes the message to the fetched offset. Note that each writer needs to ensure that the receiver processed the previously written messages to that offset of the ring buffer. Thus, each writer sometimes needs to read the tail of the remote ring, which indicates the processed offset.

Inlined bell with zeroing. After fetching the head, each sender writes messages as in the corresponding ring buffer protocol. Though senders can write messages in any order after fetching the head, the receiver can process them only in the offset order, since it only knows where the next message starts. Therefore, a slow writer can block message processing.

Impossible of other protocols. The reverse buffer cannot be employed to receive messages without zeroing, as it is impossible to order WRITEs between different connections. Thus, one writer can accidentally unset the bell of the unprocessed message. We cannot have a detached bell protocol either, as the detached head at the receiver only indicates the intention of writers to write. Even with a second detached bell storing written bytes, clients could increment that bell in any order, falsely acknowledging incomplete messages from slow writers. If the writers use a two-directional channel to reserve slots instead of ATOMIC requests, then the protocol becomes the "write slot with reserve slots" method.

4.5 Read slot protocols

Read slot protocols rely on using READs to fetch data from predefined locations with zero-copy. This family of algorithms is not a good fit for message passing and therefore is usually used in combination with other channels that are used to inform receivers about readable memory regions.

Inlined bell. This method inlines a bell within the mailbox slot that would indicate that the message is ready to be read. The receiver fetches the region and checks the bell. We also assign all algorithms that read a predefined region using one READ request into this
Data of the message \( N-1 \)
Bell of the message \( N \)
Data of the message \( N \)
Bell of the message \( N+1 \)

with inlined bell in three steps (see Figure 3): first, it locally copies Table 2 shows that protocols offer different features and often support only certain RDMA networks. Developers can use our study to find channels according to their system requirements and capabilities of targeted networks. They can choose protocols that address primary needs such as low memory usages, scalability, or low latency.

Besides facilitating correct design of new systems, our study can be used to identify systems whose communication channels contradict the core requirements of primary workloads. For example, RDMA-enabled swapping systems aim to swap out 4 KiB pages to remote daemons with low-latency, low CPU cost, and efficient memory management at the receiver, which can be satisfied with the "Shared Send/Recv" protocol. However, all existing RDMA-enabled swapping systems [7, 19] employ "write slot with inlined bell", thereby wasting a lot of memory at the remote swap daemon.

### 6 EVALUATION

To demonstrate the correctness of the studied uni-directional channels we have implemented them and measured their performance under various workloads and deployment settings. With our evaluation we aim to answer the following questions:

- Does true zero-copy requirement allow networked systems to improve their performance?
- Do our performance metrics (HRT and # Requests) reflect the empirical performance of the protocols?
- Does our "ring inlined bell w/o zeroing" outperform other ring implementations?
- Can the device memory capability of RNICs improve the performance of shared ring protocols?
- What performance can be achieved by shared ring protocols for the N:1 communication and by read ring protocols for the 1:N communication?

**Experimental setup.** All experiments were performed on two machines interconnected by 100 Gb/sec Nvidia ConnectX5 NICs. Machines communicate with each other via a switch using RoCEv2 protocol. Each machine is equipped with AMD EPYC 7742 @ 2.25GHz and 256 GiB of DRAM.

**Implementation details.** All protocols and systems are implemented in C++ and depend on the following libraries: libibverbs, an implementation of the IB verbs, and librdmacm, an implementation of the RDMA connection manager. All channels are implemented as single-threaded message interfaces that expose the following API:

```c
// Sender API
// Send a region to the receiver. The send will be zero-copy if the protocol is zero-copy on send.
uint64_t SendRegion(Region *r);
// To test the completion of the send request
bool TestSendRequest(uint64_t handler);
// Receiver API without zero-copy on receive.
// Receive the next message.
Region* ReceiveRegion();
// Acknowledge the region processing to reuse it.
bool FreeReceiveRegion(Region *r);
// Receiver API with zero-copy on receive.
// Receive the next message.
RegionOption* CanReceiveRegion();
// Initiate zero-copy message reception.
uint64_t ReceiveRegion(RegionOption *o, Region *r);
// To test the completion of the receive.
bool TestReceiveRequest(uint64_t handler);
```

As mentioned earlier in Section 3, a network module should include at least two uni-directional channels in opposite directions to build a proper datapath with reliable data delivery: a receiver needs to acknowledge message reception and message processing. For example, senders of all ring-based protocols need acknowledgements.
6.1 Performance microbenchmarks

Zero-copy requirement. To motivate the importance of zero-copy communication for networked systems we measure the performance of the two-directional protocol based on the send/recv normal channels with and without the need to copy received messages. In the experiment, a client sends a message to a system via the channel, and the system after processing the request sends a response back. The response size is the same as the request size. In the "with copy" setting, the system needs to copy the content of the received message to log-structured in-memory storage with the capacity of 8 GiB.

Figure 4 shows that the copy requirement caused the client to experience higher request latencies with higher deviation (the 90% confidence interval is depicted). The difference becomes more pronounced for large messages. In addition, the average throughput was by 0.2M req/sec higher for the zero-copy system, showing the advantage of the zero-copy.

Performance metrics. To show that the latency of the data channels can be expressed as the number of half-round trips, we measure the round-trip latency of various channels with different performance characteristics. For the point-to-point channels, the latency includes the server’s reply message that has the same size as the original message. For shared and read rings, the receivers do not send the whole message back and only acknowledge it by sharing the tail of the ring.

Figure 5 shows that shared and read rings have the highest median latency as they require multiple HRTs to send data. All benchmarked channels requiring one HRT have approximately the same latency. However, the "ring inlined bell with zeroing" has a bit higher latency for large messages as the receiver needs to zero processed messages before sending a reply. The data also reveals that the "ring detached bell" channel has higher latency than other rings as it requires two work requests per message.

Interestingly, passive read rings have a dramatic drop in performance for large messages in Figure 5, which reports the median latency. Figure 6 helps to understand this artifact by providing empirical probability distribution of the latencies. The data reveals that the read ring channel follows a bimodal distribution. The first peak depicts latencies when the receiver fetched the new bell with one READ request. The second one is when the fetching the new bell took two READ requests. As the time required to write a message into the ring and update the bell depends on the message size, we can see the receiver was mainly in-sync with the sender for the small message. For the large message, however, the receiver was less fortunate and needed the second request.
6.2 Point-to-point channels

In Section 4.3, we have presented a new approach for delivering data to a ring buffer that does not require zeroing memory when the inlined bell is used. To evaluate the effectiveness of the proposed method we measure the throughput of all ring-based channels and the send/recv normal channel.

Figure 7 reports the average throughput of the discussed channels measured by a client of a system. The client could have multiple outstanding requests and the system was empowered to batch responses to ensure that the measured throughput is bottlenecked by the client to the server channel. The lowest throughput for small messages has been observed for methods requiring the receiver to post RECEIVE requests, which limited the speed of the protocol. For large messages, however, they achieve maximum performance.

Our inlined bell algorithm without zeroing outperforms the counterpart with zeroing, as it does not require the CPU to clear processed messages. Therefore, we conclude that our algorithm should replace the existing inlined bell method as they offer the same features to the networked systems. However, if the RDMA network does not offer in-order message delivery, then our algorithm cannot be used.

Due to the pipelining effect between outstanding requests, the detached bell method performed as the inlined bell method without zeroing for some sizes. Nonetheless, the second WRITE request always reduces the link utilization.

6.3 Shared and read ring channels

Device memory. The device memory extends the memory of networked systems with a small memory region that is closer to the network than the main memory. As the device memory is located at the RNIC, RDMA requests to it can be processed without PCIe involvement. To measure the role of the device memory on channels, we measured the performance of the shared ring inlined bell method with various deployment settings depending on the memory location of the head and the tail of the shared buffer. As a reminder, each sender issues an ATOMIC request (fetch and add) against the head to reserve a slot in the buffer, and occasionally a READ request against the tail to monitor the progress of the receiver.

Figure 8 shows performance observed by a client of a system that used the shared ring with the inlined bell algorithm. As fetching the head is on the critical path, moving the head to the device reduced the latency, whereas the tail location had no effect. The difference between "no device memory" and "head on device" is equal to the round trip latency of the PCIe bus. The throughput also is increased for the cases when the head is in the device memory. However, the increase was not dramatic as we allowed the sender to have at most 16 outstanding messages.

Figure 9a shows the cumulative throughput of the shared ring with the inlined bell under the load from multiple clients. The throughput is measured by the receiver. The plot shows that the performance of the channel has been bottlenecked by the performance of ATOMIC requests against the main memory. When the head has been moved to the device, the receiver experienced a more than 4x improvement in throughput, showing that the device memory can be efficiently used to speed up synchronization between clients with ATOMICS.

Broadcast. Read rings allow the sender to passively replicate data to multiple nodes without the need to work with RDMA requests. We compare the performance of the proposed algorithms under a workload of four clients that receive messages from the server by reading them from the read ring.
processing. Similar to PRISM, such proposals only reduce the network latency and not PCIe latency. Nonetheless, such smartNICs can partially offload complex protocols, proposing completely new data and network management opportunities as well as design challenges.

8 CONCLUSIONS

The choice of RDMA-based communication channels has a great impact on the design of networked systems and the requirements for the utilized RDMA networks. We have analyzed and categorized data communication algorithms, helping system developers choose them according to their needs. We have introduced new communication algorithms and discussed the effect of future RDMA products on the studied algorithms, ensuring the comprehensiveness of the study.

REFERENCES

[1] Implementation of studied protocols. https://www.dropbox.com/s/ps3lka87jmlf2it/ RDMA-and-systems.tar.gz. [Accessed 22-Aug-2022].
[2] Libfabric Programmer’s Manual. https://ofiwg.github.io/libfabric/. [Accessed 22-Aug-2022].
[3] Multi-Packet RQ. https://docs.nvidia.com/networking/display/rdmacore05Multi- Packet+RQ. [Accessed 22-Aug-2022].
[4] RDMA core userspace libraries and daemons. https://github.com/linux-rdma/ rdma-core. [Accessed 22-Aug-2022].
[5] UCX Documentation. https://openucx.org/documentation/. [Accessed 22-Aug- 2022].
[6] Tag Matching Verbs API and Implementation Example. https://community. mellanox.com/b/article/tag-matching-verbs-api-and-implementation-example, 2018. [Accessed 22-Aug-2022].
[7] Emmanuel Amaro, Christopher Branner-Augmon, Zhifan Luo, Amy Osterhout, Marcos K. Aguileria, Aura Jojita Panda, Sylvia Ratanasamy, and Scott Shenker. Can Fast Memory Improve Job Throughput? In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys ’20, New York, NY, USA, 2020. Association for Computing Machinery.
[8] Amazon Web Services, Inc. or its affiliates. Elastic Fabric Adapter. https://aws.amazon.com/bp/eفا/. 2020. [Accessed 22-Aug-2022].
[9] InfiniBand Trade Association et al. The InfiniBand Architecture Specification 1.4, 2020. https://www.infinibandta.org/iba-specification. 2018. [Accessed 22-Aug-2022].
[10] Brian W. Barrett, Ron Brightwell, Ryan E. Grant, Scott Hemmert, Kevin Pedretti, Kyle Wheeler, Keith Underwood, Rolf Riesen, Torsten Hoefler, Arthur B. Mac Nab, and Trammell Hudson. The Ports 4.2 Network Programming Interface. Technical report, November 2018.
[11] Matthew Burke, Somnath Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21, page 228–242, New York, NY, USA, 2021. Association for Computing Machinery.
[12] Qingtao Cai, Weinian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. Proc. VLDB Endow., 11(11):1604–1617, July 2018.
[13] Youmin Chen, Youyou Lu, and Jiuw Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys ’19, New York, NY, USA, 2019. Association for Computing Machinery.
[14] NVIDIA Corporation. NVIDIA InfiniBand Adaptive Routing Technology. NVIDIA Corporation, White Paper WP-10326-001_v01, 2021.
[15] Aleksandar Dragujevic, Dushyant Narayanan, and Miguel Castro. RDMA Reads: To Use or Not to Use? IEEE Data Eng. Bull., 40:3–14, 2017.
[16] Aleksandar Dragujevic, Dushyant Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI’14, pages 401–414. USENIX Association, 2014.
[17] P. Fent, A. v. Renen, A. Kirp, V. Leis, T. Neumann, and A. Kemper. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In 2020 IEEE 36th International Conference on Data Engineering (ICDE), pages 1477–1488, 2020.
[18] Vasils Gavrielatos, Antonios Kataraakis, Arpi Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagaran. Scale-out CcNUMA: Exploiting Slew with Strongly Consistent Caching. In Proceedings of the Thirteenth EuroSys Conference, EuroSys ’18, New York, NY, USA, 2018. Association for Computing Machinery.
