An Evaluation of RDMA-based Message Passing Protocols*

Haoyu Huang, Shahram Ghandeharizadeh
Database Laboratory Technical Report 2019-02
Computer Science Department, USC
Los Angeles, California 90089-0781
{haoyuhua,shahram}@usc.edu

Abstract

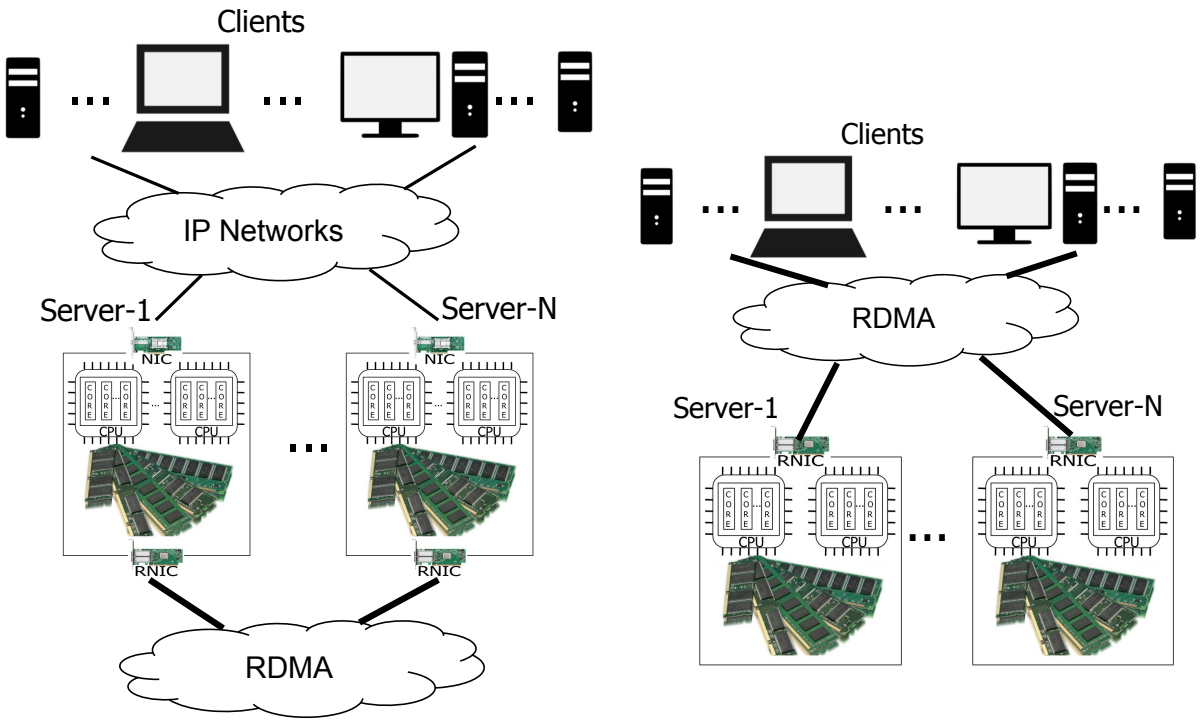
An enumeration of RDMA messaging verbs (READ, WRITE, SEND/RECEIVE) and queue pair types creates a diverse set of message passing protocols. This paper constructs three abstract communication paradigms to quantify the performance and scalability characteristics of five protocols. With each abstraction, different protocols provide different results and the identity of the protocol that is superior to the others changes. Factors such as the number of queue pairs per node, the size of messages, the number of pending requests per queue pair, and the abstract communication paradigm dictate the superiority of a protocol. These results are important for design and implementation of algorithms and techniques that use the emerging RDMA.

1 Introduction

Advances in hardware and processing have introduced high bandwidth Remote Direct Memory Access (RDMA). Similar to the Ethernet-based Internet Protocol (IP) networks, RDMA consists of a switch and a network interface card (RNIC) that plug into a node. It is novel because it supports a variety of protocols including READ and WRITE verbs that bypass CPU of a node to read and write its memory directly.

RDMA motivates novel architectures for data intensive applications. Fig. 1 shows two possible architectures. The architecture of Fig. 1(a) maintains the traditional IP network and extends it with RDMA. An application may separate client's network transmissions using the IP network from transmission in support of data processing using the RDMA [9]. Hence, the application may transmit terabytes of data using the RDMA network without slowing down the client requests issued using the IP network. Alternatively the application may use the IP network for control messages and the RDMA network for exchanging data across servers to implement the core functionality of the system, e.g., a join algorithm, a machine learning technique, or an analytic operation.

*Appeared in the *6th Workshop on Performance Engineering with Advances in Software and Hardware for Big Data Science*, co-located with IEEE BigData, Los Angeles, CA, December 9-12, 2019.



(a) (b)

Figure 1: Two possible architectures of a distributed key-value store.

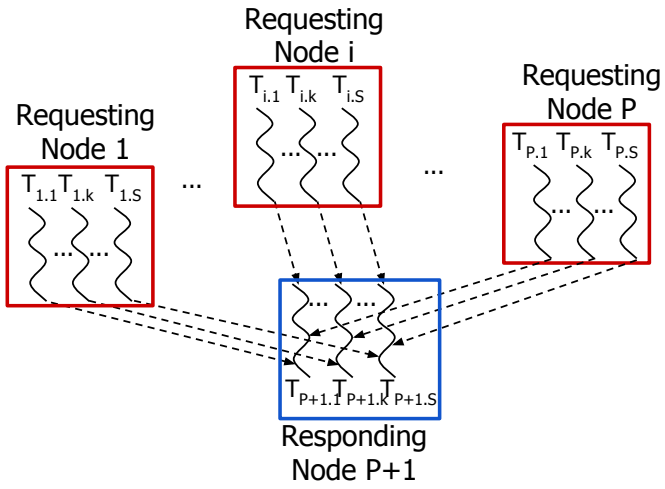
With the architecture of Fig. 1(b), the RDMA network replaces the IP network altogether, requiring clients and servers to communicate using RDMA [19]. Clients may use the READ verb to fetch data from a server, bypassing its CPU. Yet another possibility (not shown) may co-locate the client's software components on the same nodes that implement the server functionality [7]. With this architecture, a client accesses a server's data locally that is running on the same node. Moreover, the clients may share each other's memory using the RDMA network.

With the possible architectures of Fig. 1 (and others), an implementation of alternative data processing techniques using RDMA may result in different communication abstractions. We classify these into many-to-one (M:1), one-to-many (1:M), and many-to-many (M:M), see Fig. 2.

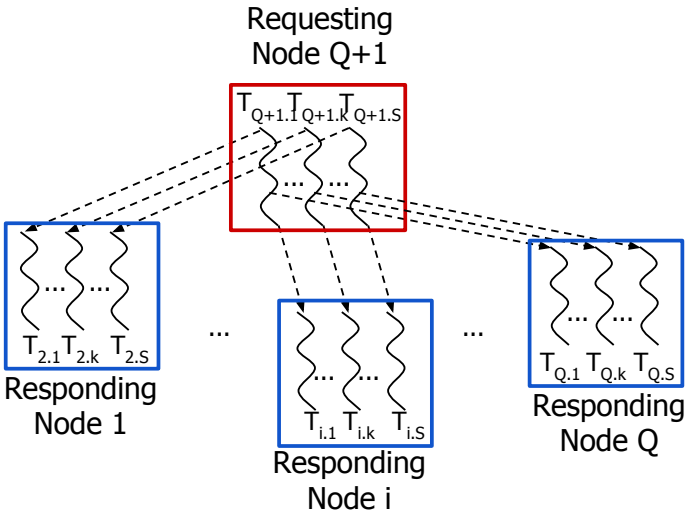
With M:1, many requesting nodes use their RNICs to fetch data from one responding node, see Fig. 2(a). An example usage is a detective load-balancing algorithm that requires idle nodes to pull data from a fully utilized (bottleneck) node to reduce its load [22, 17, 16, 15, 10, 11].

With 1:M, one requesting node fetches data from many responding nodes, see Fig. 2(b). An example usage is a distributed key-value cache that processes a multi-get request executing on the requesting node to fetch many referenced data items stored across several nodes [6, 24, 25, 1]. Yet another example is a primary-backup replication where the primary node replicates a data item across many backup nodes [32, 27].

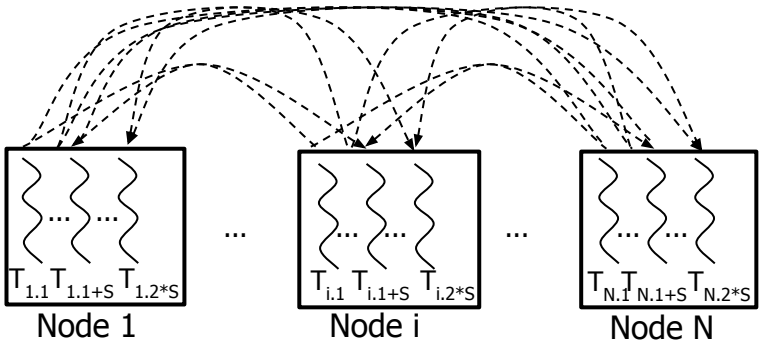
Finally, with M:M, a node may fetch data from any other nodes in a cluster using its



(a) M:1, P requesting nodes and 1 responding node.



(b) 1:M, 1 requesting node and Q responding nodes.



(c) M:M, N nodes and P=Q=N.

Figure 2: Three abstractions.

RNIC, see Fig. 2(c). An example usage is an implementation of a distributed data store where data is partitioned across many nodes and any node may process a request [7, 14, 5]. Another

Table 1: Three abstractions and their example use cases.

Abstraction	Example use cases
M:1	Load balancing algorithms where one server is the bottleneck [17, 16, 15].
1:M	Replication [32, 27]. Distributed key-value caches [6, 24, 25, 1].
M:M	Transaction processing system [7, 14, 5, 21, 23, 31, 30]. Analytical systems [3, 4]. Disaggregated operating system [26, 29].

example is an analytical system that uses RDMA to optimize distributed joins over data stored on many nodes [3, 4]. A disaggregated operating system breaks a monolithic server into disaggregated, network-attached hardware components [26, 29]. The communication between these components is also M:M.

The primary *contribution* of this study is to quantify the performance of the alternative RDMA protocols for these three abstractions with the reliable connected (RC) queue pair. We show a significant performance difference between the three abstractions and alternative protocols given an abstraction. These results enable a system architect to design and implement algorithms that maximize performance using RDMA. Moreover, they establish the theoretical upper bound on the performance that can be observed using an abstraction with a specific protocol. Table 1 identifies several recent studies that use RDMA and how they map to our provided abstractions.

Our evaluation uses the key-value characteristics of Facebook [2] specified as 36-byte key size and 365-byte value size as defaults. We consider one protocol superior to another if it provides a higher bandwidth and approximates the advertised RDMA bandwidth between any two nodes (56 Gbps). We vary the value size to investigate its impact on the superiority of different protocols and their observed bandwidths. We observe a significant difference between the alternative protocols under different settings. A node uses RDMA *queue pair* (QP) to communicate with other nodes. Typically, the highest bandwidth is observed using a few QPs.

1.1 Summary of results

Section 2 and 3 present alternative verbs and protocols. Their evaluation in Section 4 highlights the following lessons:

1. No single protocol is superior for all three abstractions. With M:1 and M:M, the protocol using the READ verb provides the highest bandwidth. With 1:M, those protocols that use the WRITE verb are superior to others. While the READ verb cannot be used to perform writes, the WRITE protocols can be used to perform reads. Hence, the results with the READ verb do not apply to write-heavy workloads.

2. The highest bandwidth observed with alternative abstractions is different. For example, the highest observed bandwidth with M:1 (42 Gbps) is 3x 1:M (14 Gbps). Fig. 3 shows

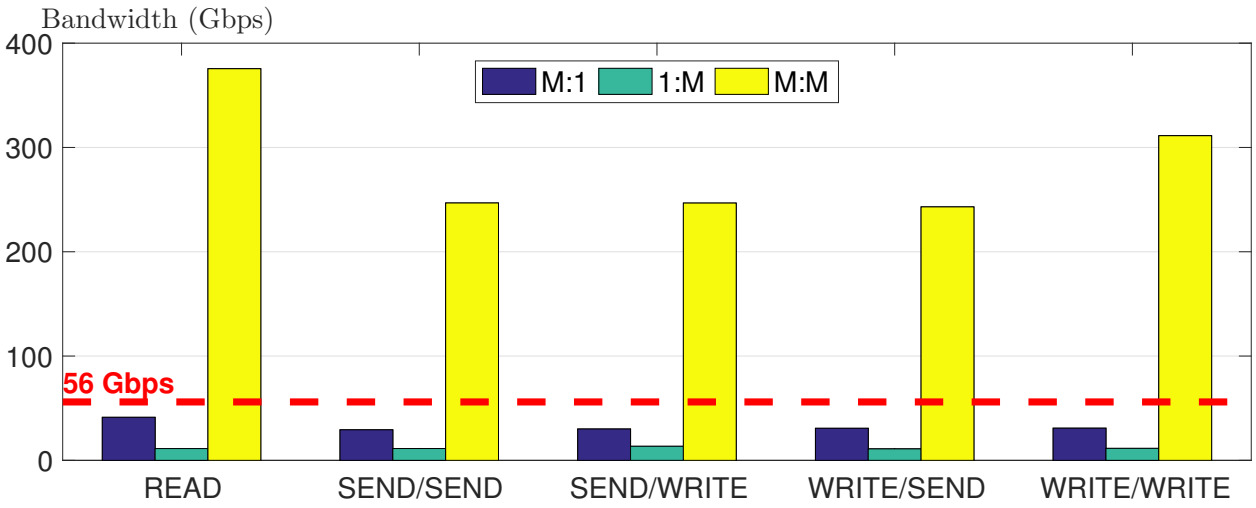


Figure 3: The highest observed bandwidth for each protocol with 28 nodes and 365 Bytes value size.

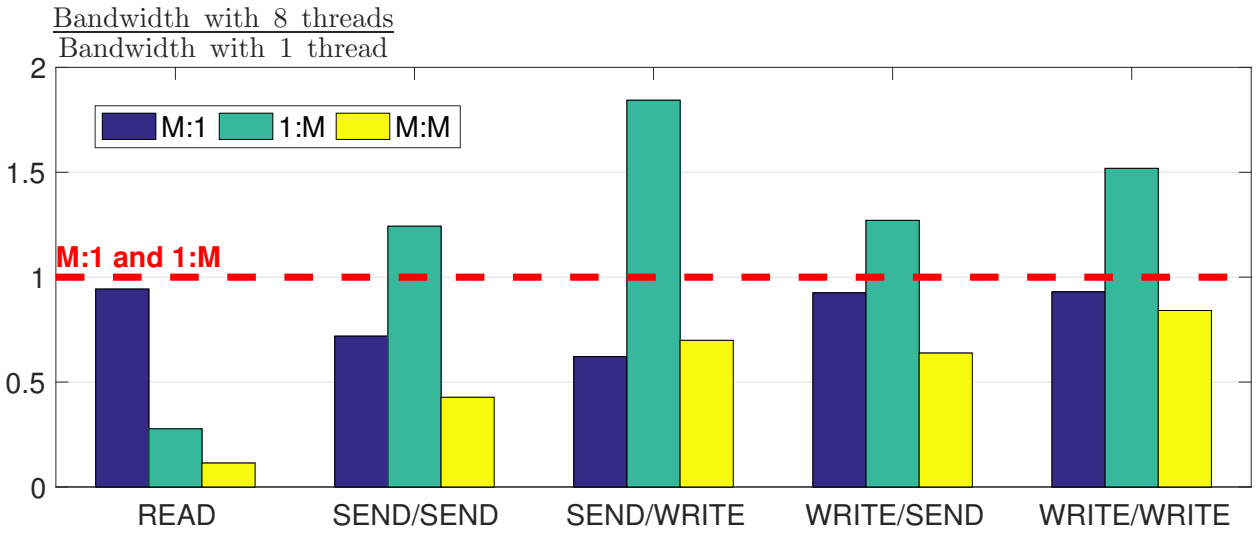


Figure 4: Vertical scalability: Ratio of observed bandwidth with 8 threads relative to that of 1 thread with 28 nodes and 365 Bytes value size.

the bandwidth observed with 28 nodes. The red line is the advertised bandwidth of one RNIC. Bandwidth observed with M:1 and 1:M is dictated by 1 RNIC. With M:M, the bandwidth is high because it uses different RNICs. The average bandwidth per node of M:M is lower than both M:1 and 1:M due to a higher number of QPs.

3. The alternative protocols do not scale vertically with M:1 and M:M. Fig. 4 shows the ratio of bandwidth with 8 threads relative to 1 thread. With the READ verb, using 8 threads reduces the bandwidth with all three abstractions. With 1:M, the scalability of all protocols except READ increases with additional threads. It is not eight times higher even though the overall observed bandwidth is approximately 10 Gbps.

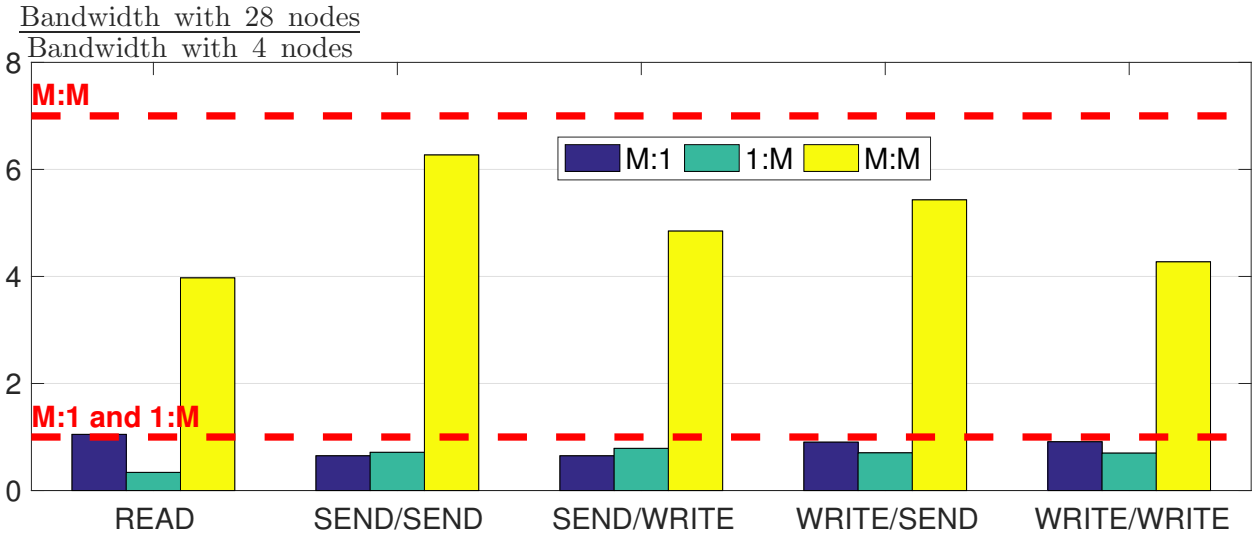


Figure 5: Horizontal scalability: Ratio of observed bandwidth with 28 nodes relative to that of 4 nodes and 365 Bytes value size.

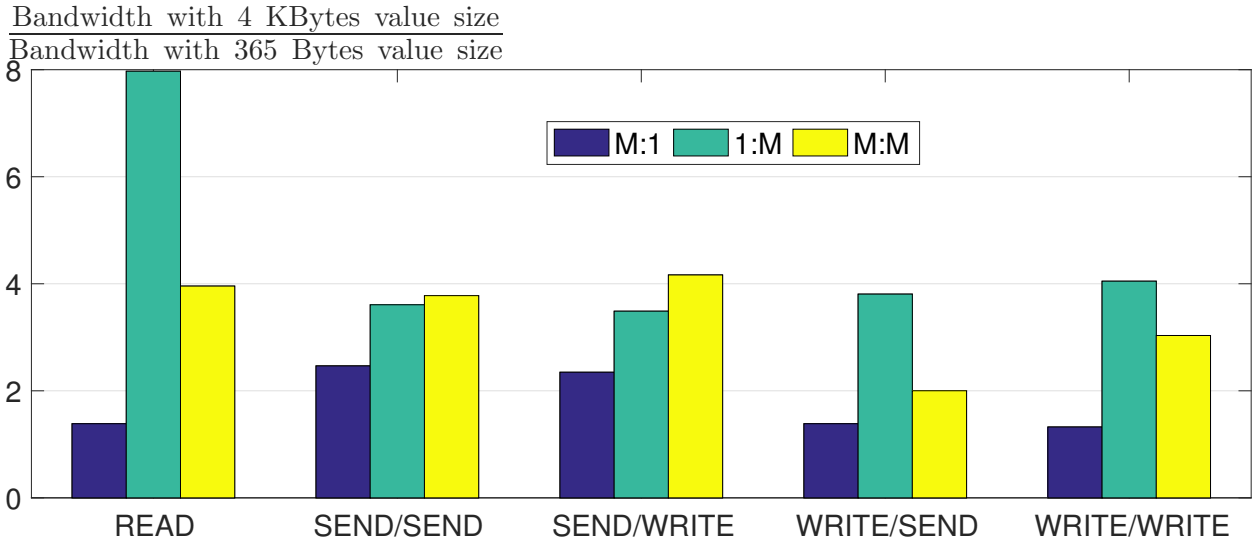


Figure 6: Value size: Ratio of observed bandwidth with 4 KBytes value size relative to 365 Bytes value size with 28 nodes.

4. The horizontal scalability of all protocols is sublinear. Fig. 5 shows the ratio of bandwidth observed with 28 nodes when compared with 4 nodes. The ideal ratio with M:1 and 1:M is one while it is seven with M:M, see red lines. Except for one scenario, READ with M:1, the different protocols with the alternative abstractions fall short of the ideal ratio. This is due to a high number of QPs.

5. The observed bandwidth increases as a function of value size for all protocols with all three abstractions. Fig. 6 shows the ratio of bandwidth observed with 4 KBytes value size relative to 365 Bytes. With M:1 and the protocol using the READ verb, the bandwidth increases from 35 Gbps to 50 Gbps. With 1:M, its bandwidth increases

from 2 Gbps to 16 Gbps. And, with M:M, its average bandwidth per node increases from 10 Gbps to 20 Gbps. The total bandwidth with M:1 and 1:M either drops or is unchanged as a function of the number of nodes (as one RNIC is used). With M:M, the total bandwidth is a function of the number of nodes and the average bandwidth observed per node. (With M:M, the bars for total bandwidth is identical to the one for the average bandwidth per node of Fig. 6.)

The rest of this paper is organized as follows. Section 2 and 3 present alternative RDMA verbs and protocols, respectively. Section 4 evaluates these alternatives. Section 5 describes example use cases. Section 6 surveys related work and Section 7 presents future work.

2 RDMA VERBS

RDMA verbs are a set of communication interfaces that enable nodes to communicate with each other. Two nodes use queue pairs (QP) to communicate with each other. A QP may be reliable connected (RC), unreliable connected (UC), or unreliable datagram (UD). A connected QP may only communicate with its connected remote QP.

A RC QP guarantees in-order delivery of messages, no message duplication, and no message loss. An UC QP may lose a message due to either software or hardware errors. An UD QP may communicate with other UD QPs on different nodes. Table 2 lists QP types and alternative verbs.

An application allocates a memory region and registers it in RNIC. A QP must exchange information such as the registered memory with a remote QP before communicating with each other. An application provides its registered memory region when using two-sided verbs. It must also provide the registered memory region of the remote QP when using one-sided verbs.

A QP consists of a send queue and a receive queue. Both send and receive queues have a corresponding completion queue. The send queue contains zero or more pending work requests (WR). A WR may either be a READ, a WRITE, or a SEND. The receive queue contains a list of WRs to accept requests. RNIC generates a work completion (WC) record in the completion queue when a WR completes. The completion queue contains zero or more WC records. An application must poll its completion queues to retrieve WC records continuously. This is because RNIC may not generate more WC records once the completion queue is full, resulting in applications stalls.

Sharing queue pairs or completion queues between different threads inhibit scalability. Both QP and completion queue have its own mutex [28]. A QP acquires its mutex before posting a WR. Similarly, a completion queue acquires its mutex before polling WC records.

We use the following terminology throughout the paper. A *requester* issues a request and a *responder* transmits a response. We term a node (thread) issuing a request as *requesting* node (thread). We call a node (thread) that transmits a response as *responding* node (thread).

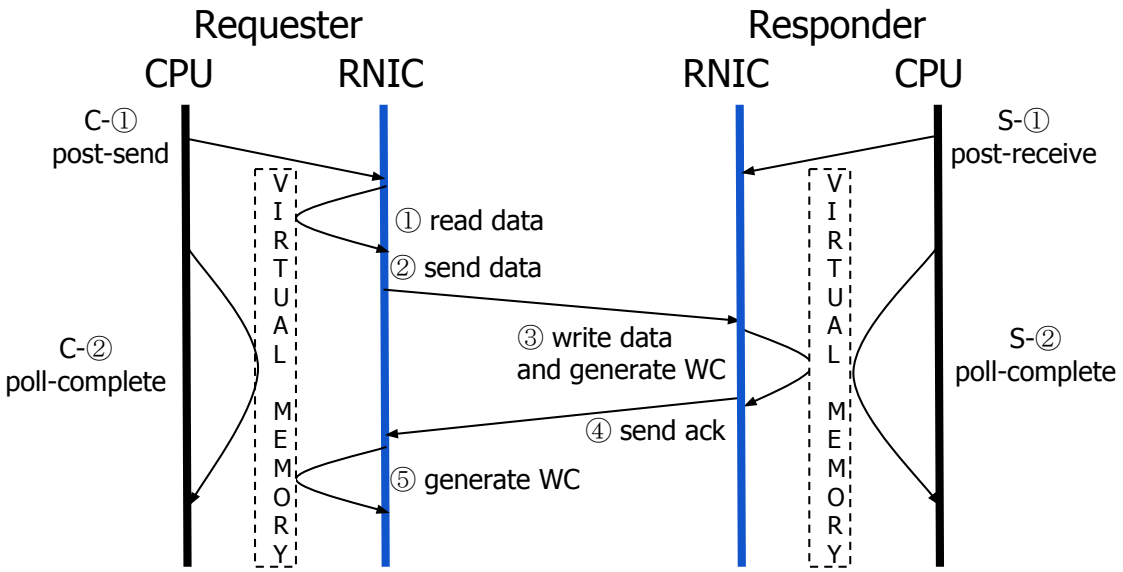


Figure 7: RC SEND/RECEIVE.

2.1 SEND/RECEIVE: RC, UC, and UD

A QP may use two-sided SEND/RECEIVE verbs to exchange messages, see Fig. 7. A responder must post a receive WR before a requester issues a request S-①. Next, the requester posts a SEND request to RNIC C-①. RNIC then issues a DMA to fetch the request ① and issues the request to the responder's RNIC ②. The responder's RNIC writes the incoming request to the posted receive buffer ③, acknowledges the request ④, and enqueues a RECEIVE WC record into the responder's completion queue. The requester's RNIC receives the acknowledgment and enqueues a SEND WC record into the requester's completion queue ⑤. Subsequently, the requester polls the SEND WC record from the completion queue.

With UC and UD QPs, the requester's RNIC generates a WC record immediately after it sends the data to the fabric ②. The responder's RNIC does not transmit an acknowledgment after it receives the request. Hence, step ④ is absent from Fig. 7.

2.2 WRITE: RC and UC

A WRITE issues a request to the responder's memory directly, bypassing its CPU. A requester first writes the request into a local buffer within its registered memory region. It also specifies a remote buffer within the responder's registered memory region to write this request to. Next, it posts a WRITE WR to RNIC C-①. RNIC fetches the content from the buffer ① and issues the request to the responder ②. The responder's RNIC receives the request and writes it into its specified remote buffer ③. Lastly, it transmits an acknowledgement ④ and the requester's RNIC generates a WC record to the QP's completion queue ⑤.

With UC QPs, the requester's RNIC generates a WC record immediately after it sends the data to the fabric ②. The responder's RNIC does not transmit an acknowledgment after

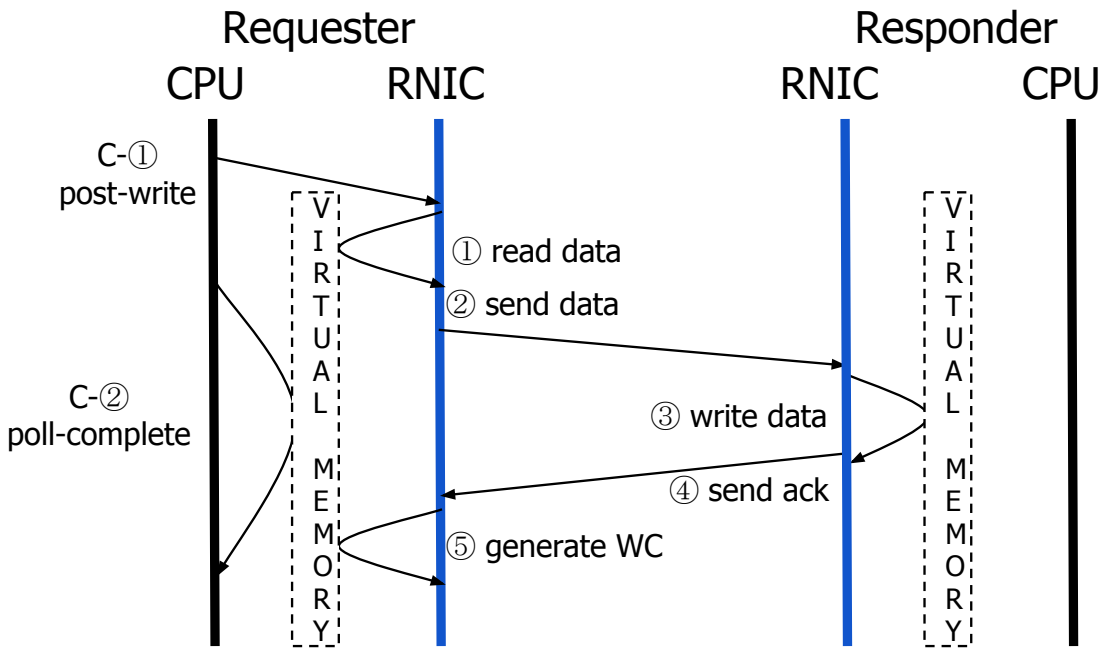


Figure 8: RC WRITE.

it receives the request, i.e., step ④ is absent from Fig. 8.

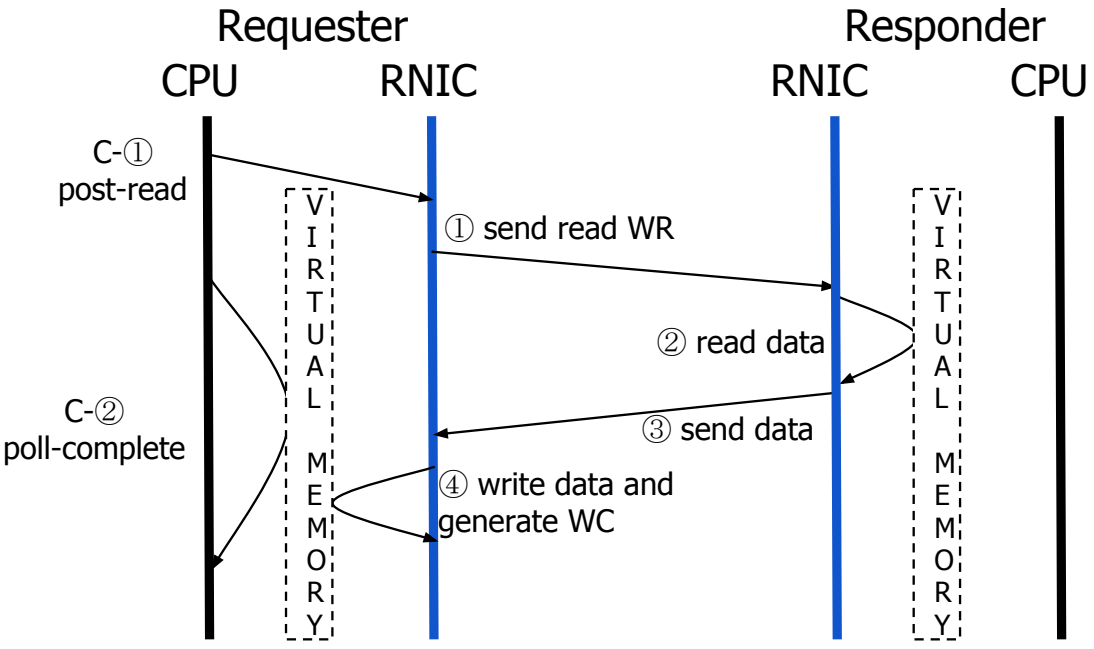


Figure 9: RC READ.

2.3 READ: RC

A READ fetches data from a responding node's memory directly, bypassing its CPU. A requester first provides a local buffer within its registered memory and a remote buffer within the registered memory of the responder. Next, it posts a READ WR to its RNIC C-①. RNIC issues the read request to the responder's RNIC ①. The responder's RNIC then fetches data from its specified remote buffer ② and transmits the data back to the requester ③. The requester's RNIC writes the returned data into the provided local buffer and generates a WC record ④.

Table 2: Queue pair types and their supported verbs.

	RC	UC	UD
SEND/RECEIVE	✓	✓	✓
WRITE	✓	✓	✗
READ	✓	✗	✗

3 Message Passing Protocols

The combination of one-sided and two-sided verbs results in a variety of message passing protocols. This paper evaluates five RC protocols shown in Table 3[1]. A requester may use either SEND or WRITE to issue a request. A responder may use either SEND or WRITE to transmit a response. With READ, we assume the requester knows the memory location of the referenced data. It uses RDMA READ to fetch the referenced data from the responder.

Table 3: Message passing protocols (RC).

Notation	Verb used at requester	Verb used at responder
SEND/SEND	SEND	SEND
SEND/WRITE	SEND	WRITE
WRITE/SEND	WRITE	SEND
WRITE/WRITE	WRITE	WRITE
READ	READ	N/A

When a requester uses WRITE to issue a request, a responder performs a local read to fetch the request from the same memory region. The local read may fetch a partial request if it is concurrent with a WRITE writing the request. We use markers to ensure memory safe reads. A request consists of a sequence of bytes. The first four bytes of a request are the byte length B of the request. The fifth byte is a marker and the following B bytes contain the request. The last byte is also a marker. A responder reads the length B of a request when the fifth byte equals to the marker. It then reads and processes the request when the last byte also equals to the marker. The marker indicates that the WRITE is complete and

[1]This paper defers the evaluation of UD and UC protocols to future work as they do not provide reliability guarantees and their performance is not comparable with RC protocols.

the responder reads a complete request. The responder zeros out $B + 6$ bytes after reading the request.

It is possible to construct protocols that are inefficient. For example, a naive protocol may require a requester to write its request to its local memory while the responder issues a READ to fetch the request continuously. Such a protocol imposes a high overhead on the available RNIC bandwidth and is not considered in this paper.

4 Evaluation

This section answers the following question: What is the bandwidth observed by the alternative message passing protocols for the three abstractions shown in Fig. 2?

We quantify an answer using the CloudLab APT cluster with 29 nodes of type c6220 [8]. Each node has 64 GBytes of memory and consists of 2 Xeon E5-2650v2 8-core CPUs. Hyperthreading results in a total of 32 virtual cores. All nodes are connected using Mellanox SX6036G Infiniband switches. Each server is configured with one Mellanox FDR CX3 Single port mezz card that provides a maximum bandwidth of 56 Gbps. This card connects to one CPU socket. Use of this CPU and its cores maximizes the overall observed bandwidth. In all reported experiments, we pinned the threads to the cores of this CPU. We term a thread k on Node$_i$ as $T_{i,k}$.

With M:1, P requesting nodes issue requests to one responding node. A requesting thread $T_{i,k}$ has one QP connecting to the responding thread $T_{P+1,k}$. A requesting thread has one QP and a requesting node has a total of S requesting threads (QPs). A requesting thread maintains R pending requests for each QP. A responding thread has P QPs and the responding node has a total of $P * S$ QPs.

With 1:M, one requesting node issues requests to Q responding nodes. A requesting thread $T_{Q+1,k}$ has one QP connecting to the responding thread $T_{i,k}$ for each of the responding nodes. A requesting thread has Q QPs. The requesting node has S requesting threads and a total of $Q * S$ QPs. A responding thread has one QP and a responding node has a total of S QPs. A requesting thread maintains R pending requests for each QP and thus a total of $Q * R$ pending requests.

With M:M, N nodes issue requests to each other. A node has S requesting threads and S responding threads. A requesting thread $T_{i,k}$ in Node$_i$ has one QP connecting to the responding thread $T_{j,k+S}$ for every other Node$_j$. A requesting/responding thread has $N - 1$ QPs. A node has a total of $(N - 1) * 2 * S$ QPs. A requesting thread maintains a fixed number of pending requests R for each QP.

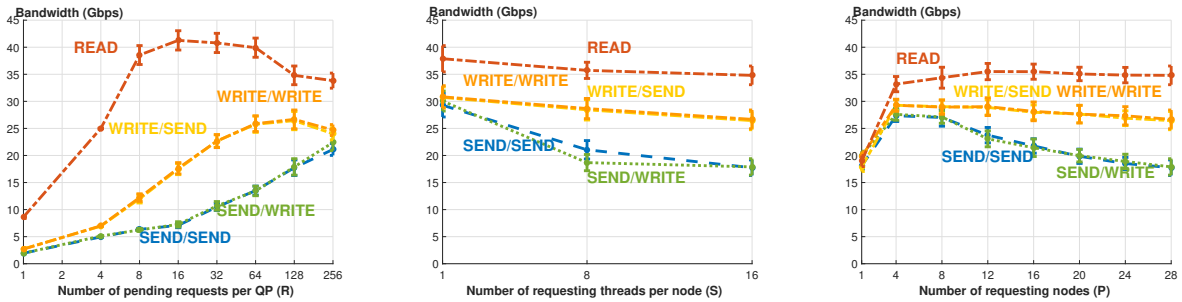
A QP has an array of buffers. Each element is $L + V$ bytes in size: the request is L bytes in size (i.e., identifies a key) and the response (i.e., a corresponding value) is V bytes in size. There are R pending requests per RC QP. Hence, the total size of the array is $(L + V) * R$. There are many ways to organize the bytes. Our implementation groups R requests of size L sequentially and R responses of size V sequentially. A request/response i indexes into this array trivially using the value of i.

In all experiments, we use Facebook [2] published mean key size of 36 Bytes as the request size, L=36 and value size of 365 Bytes as the value size, V=365. We consider other value sizes and report on the obtained results. RNIC uses inlining to issue requests to avoid the

use of DMA, providing higher bandwidth.

To provide the best performance, a thread polls the completion queue of its QP(s) continuously. A thread also checks each receive buffer for incoming requests (responses) continuously if the remote QPs use WRITE to issue requests (responses). We use a maximum of 16 threads pinned to cores of the CPU that connects to the RNIC. A thread utilizes its assigned core fully.

We sweep the parameter space and report the bandwidth by multiplying the number of messages per unit of time with the value size V. With M:1 and 1:M, we report the aggregated bandwidth of all requesting nodes. With M:M, we report the average bandwidth per node. We report the mean of three consecutive runs and error bars showing the maximum deviation from the mean.



(a) Number of pending requests per QP, P=28, S=16.

(b) Vertical scalability, P=28, R=128.

(c) Horizontal scalability, R=128, S=16.

Figure 10: M:1, V=365.

4.1 M:1

We use 28 requesting nodes to generate requests to one responding node, P=28 and Q=1. The observed bandwidth is limited by the maximum outbound bandwidth of the responding node's RNIC, 56 Gbps.

Summary of results: READ bypasses the responding node's CPU to provide a higher bandwidth than the other protocols. READ realizes RNIC's bandwidth at line rate with value size of 1 KBytes and higher. Using WRITE to issue a request outperforms SEND in most cases. SEND imposes a higher overhead on the responding node by requiring it to post a receive event for every SEND. With both, the responding node's use of either SEND or WRITE to generate a response provides a comparable performance.

4.1.1 Number of Pending Requests

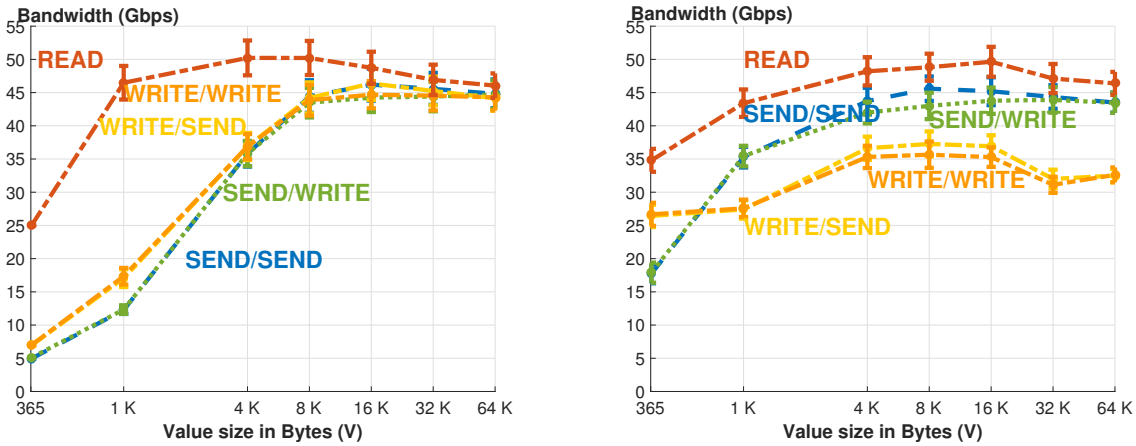
We report on the performance impact of the number of pending requests R per QP. We vary R from 1 to 256. The total number of pending requests is $R * S$ per requesting node. For example, when R=256, the total number of pending requests per requesting node is 4096 (256*16).

Fig. 10(a) shows the bandwidth observed with the alternative protocols as a function of R. The aggregate bandwidth observed by each protocol increases as we increase R. READ

achieves 43 Gbps with 16 pending requests per QP, R=16. Its bandwidth is two to six times higher than the other protocols when R=16. Using WRITE to issue requests is faster than SEND. The bandwidth difference is negligible between using SEND or WRITE to transmit responses. The bandwidth of WRITE/SEND and WRITE/WRITE plateaus at 28 Gbps when R=128. SEND/SEND and SEND/WRITE provide a lower bandwidth, 24 Gbps and 22 Gbps when R=256.

4.1.2 Vertical Scalability

Fig. 10(b) shows the bandwidth as a function of the number of threads S per requesting node from 1 to 16. When S equals 1, the responding node has 14 responding threads, each with 2 QPs to respond to two requesting threads. When S is higher than 1, the responding node has 16 responding threads, each with $\frac{S*P}{16}$ QPs to respond to the requesting threads.



(a) R=4 pending requests per QP.

(b) R=128 pending requests per QP.

Figure 11: M:1: value size, P=28, S=16.

READ provides the highest bandwidth across all S values. The bandwidth of READ reaches its peak 40 Gbps with only one thread per requesting node, S=1. Using WRITE to issue requests is second best and reaches its peak bandwidth of 33 Gbps when S=1. The bandwidth of all protocols decreases as we increase S. Using SEND to issue requests observes the most significant bandwidth drop. This is because SEND has a higher overhead as it requires the responding node to post one receive event for every SEND.

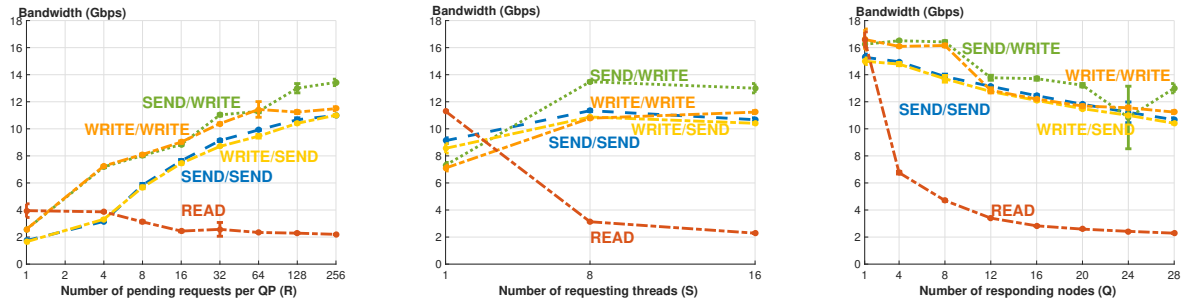
4.1.3 Horizonal Scalability

Fig. 10(c) shows the observed bandwidth as a function of the number of requesting nodes P is varied from 1 to 28. READ still outperforms all other protocols for all values of P. With one requesting node, the bandwidth of the alternative protocols is similar (20 Gbps). The bandwidth of READ stabilizes at 36 Gbps with 8 and more requesting nodes, $P \geq$8. The bandwidth with WRITE stabilizes at 30 Gbps when $P \geq$4. Obtained results show that the bandwidth of SEND to issue requests peaks at 28 Gbps with 4 requesting nodes and

drops gradually as the number of requesting nodes increases. The bandwidth drops to 19 Gbps with 28 requesting nodes.

4.1.4 Value Size

Fig. 11 shows the bandwidth of the alternative protocols as a function of value size from 365 Bytes to 64 KBytes. A larger value size results in a higher bandwidth for all protocols in most cases. READ provides the highest bandwidth. With 4 pending requests per QP, READ achieves the maximum bandwidth of 52 Gbps with 4 KBytes value sizes. The bandwidth of the other protocols peaks at 47 Gbps with 8 KBytes value sizes.



(a) Number of pending requests per QP, Q=28, S=16.

(b) Vertical scalability, Q=28, R=128.

(c) Horizontal scalability, R=128, S=16.

Figure 12: 1:M, V=365.

A higher number of pending requests R increases the bandwidth for small sized values. The bandwidth of a larger value size either decreases or remains the same with higher R values. For example, the bandwidth of READ increases by 14 Gbps with 365-byte value sizes but drops by 5 Gbps with 1 KBytes value sizes when comparing R=4 with R=128. The impact becomes negligible when the value size increases beyond 4 KBytes.

4.2 1:M

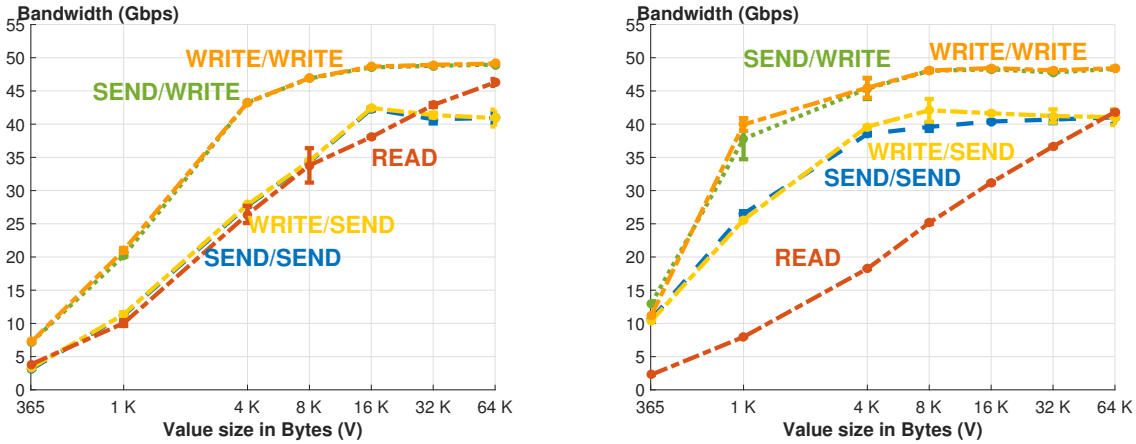
We use one requesting node to issue requests to 28 responding nodes, P=1 and Q=28. The maximum bandwidth is limited by the inbound traffic of the requesting node's RNIC, 56 Gbps.

Summary of results: READ performs the worst and its bandwidth decreases as the number of pending requests increases. Using WRITE to transmit responses outperforms SEND. It realizes RNIC's bandwidth at line rate with value size of 8 KBytes and higher. SEND imposes a higher overhead on the requesting node by requiring it to post a receive event for every SEND. With both, whether the requesting node uses SEND or WRITE to issue a request has an insignificant impact on the observed bandwidth.

4.2.1 Number of Pending Requests

Fig. 12(a) shows the observed bandwidth with an increased number of pending requests R per QP. When R=256, the requesting node generates 4096 pending requests to each

responding node. READ scales poorly and its peak bandwidth is only 4.2 Gbps when R=1. The bandwidth of READ stays low at 2 Gbps when R is 16 and higher. The bandwidth of the other protocols scales as R increases. Using WRITE to transmit responses is faster than SEND. Its peak bandwidth is 14 Gbps while SEND only achieves 11 Gbps.



(a) R=4 pending requests per QP.

(b) R=128 pending requests per QP.

Figure 13: 1:M: value size, Q=28, S=16.

4.2.2 Vertical Scalability

These experiments vary the number of requesting threads S from 1 to 16, see Fig. 12(b). A responding node has the same number of threads as the requesting node. Obtained results show that using WRITE to transmit responses performs the best. Its peak bandwidth is 13.8 Gbps. Using SEND to transmit responses is second best, achieving 11 Gbps. READ performs the worst. It reaches its peak bandwidth of 11 Gbps when S=1. Its bandwidth drops to 2 Gbps when S=16.

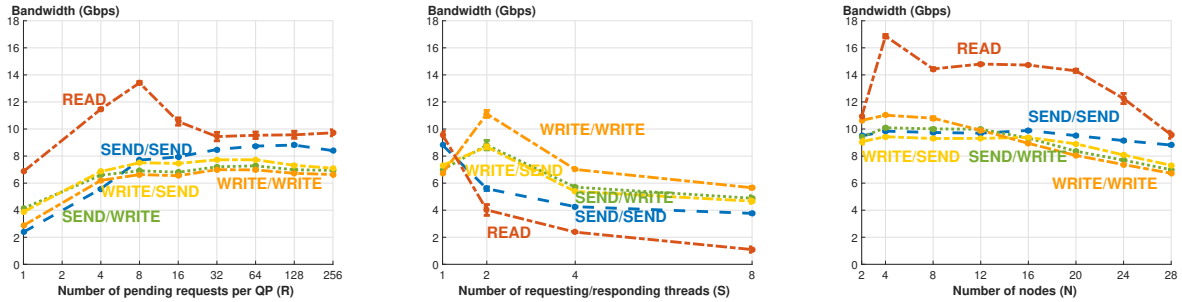
4.2.3 Horizontal Scalability

Fig. 12(c) shows the bandwidth as a function of the number of responding nodes Q from 1 to 28. A requesting thread generates $Q * 128$ pending requests. When Q=28, it generates 3584 pending requests. The bandwidth of READ drops from 17 Gbps to 2 Gbps as Q increases from 1 to 28. Using WRITE to transmit responses achieves the highest bandwidth of 16 Gbps. SEND is second best and achieves a maximum bandwidth of 15 Gbps. The difference of using SEND or WRITE to issue requests is still negligible across all Q values.

4.2.4 Value Size

Lastly, we evaluate the performance impact of various value sizes from 365 Bytes to 64 KBytes. Fig. 13 shows that READ scales as the value size increases. However, it is worse than the other protocols until value size is 64 KBytes and R=128. When the value size is 16 KBytes and higher, the bandwidth of the other protocols reaches their peak. While

using WRITE to transmit responses achieves 49 Gbps, using SEND to transmit responses only achieves 42 Gbps. Using WRITE to transmit responses is also superior to the other protocols when R=4.



(a) Number of pending requests per QP, N=28, S=1.

(b) Vertical scalability, N=28, R=128.

(c) Horizontal scalability, R=128, S=1.

Figure 14: M:M, V=365.

4.3 M:M

This experiment consists of 28 nodes, N=28. The maximum bandwidth is limited by the inbound or outbound traffic of each node's RNIC. With 28 nodes, the maximum theoretical bandwidth is 1568 Gbps (28*56 Gbps). We report the average bandwidth per node

Summary of results: All RC protocols provide their peak bandwidth with a low number of threads (typically 1) with READ outperforming the other protocols. Their performance decreases as we increase the number of threads (QPs) issuing requests. READ observes the most significant performance drop from 9 Gbps (1 requesting thread) to 1 Gbps (8 requesting threads). WRITE is superior to SEND for issuing requests and transmitting responses.

4.3.1 Number of Pending Requests

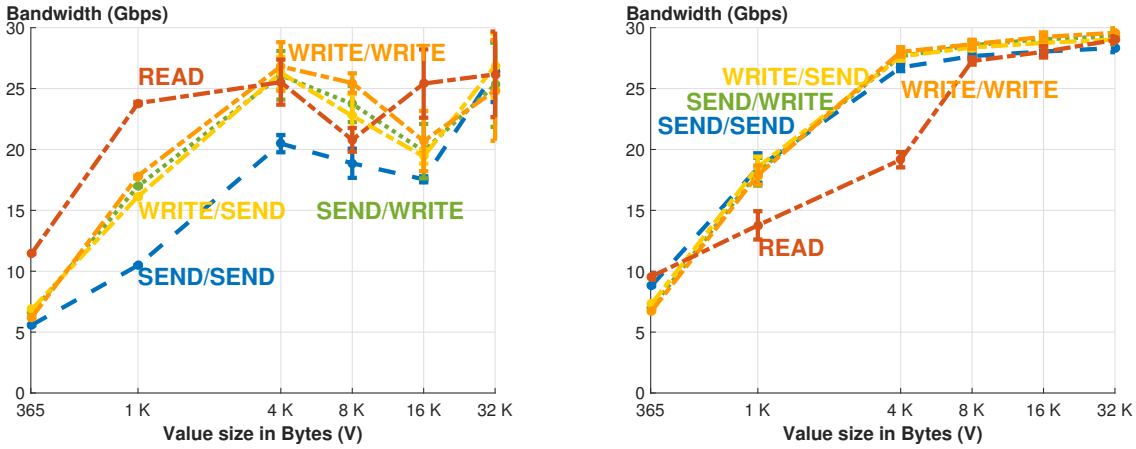
Fig. 14(a) shows READ achieves its peak bandwidth of 13.5 Gbps with 8 pending requests per QP, R=8.5 Its bandwidth is two times higher than the other protocols. As we increase R beyond 8, the bandwidth of READ decreases and stabilizes at 10 Gbps when R=32. The bandwidth of the other protocols increases as R increases. The bandwidth of SEND/SEND peaks at 9 Gbps when R=64. When R is 8 and higher, the bandwidth of SEND/WRITE, WRITE/SEND, and WRITE/WRITE stabilizes at around 7 Gbps. These bandwidths are significantly lower than the expected maximum, i.e., 56 Gbps.

4.3.2 Vertical Scalability

Fig. 14(b) shows the bandwidth as a function of the number of requesting/responding threads S. READ achieves its highest bandwidth, 10 Gbps, with one requesting thread. Its bandwidth drops to 1 Gbps as the number of requesting threads increases to 8. With 2 or more threads, the other protocols become superior to READ. When S=2, WRITE/WRITE achieves the highest bandwidth, 11 Gbps, among all protocols.

4.3.3 Horizontal Scalability

Fig. 14(c) shows the average bandwidth of all protocols decreases as the number of nodes N increases. A requesting thread maintains $(N-1)$ QPs and generates 128 pending requests to each QP. READ has the best bandwidth across all evaluated values of N. The other protocols provide a higher bandwidth and are comparable to one another. The observed bandwidth is a fraction of the anticipated 56 Gbps. This is due to the small value size of 365 Bytes.



(a) R=4 pending requests per QP.

(b) R=128 pending requests per QP.

Figure 15: M:M: value size, N=28, S=1.

4.3.4 Value Size

Fig. 15 shows the bandwidth of the alternative protocols as a function of the value size V. We use 4 and 128 pending requests per QP. The bandwidth of all protocols increases as the value size V increases. When R=4, READ achieves its highest bandwidth, 25 Gbps, with value size of 4 KBytes, V=4 KBytes. Using WRITE to issue requests and transmit responses achieves 17 Gbps when V=1 KBytes and 26 Gbps when V=4 KBytes. Using SEND to issue requests and transmit responses provides the lowest bandwidth. When R=128, READ has a higher bandwidth than the other protocols with value size of 365 Bytes. All other protocols observe a similar bandwidth across all V values. The bandwidth of READ becomes worse than the other protocols as the value size increases to 1 KBytes. With value size of 8 KBytes and higher, all protocols have a similar bandwidth.

5 Application Use Cases

A system architect may use our results to identify the protocol that is most suitable for an application use case. For example, a transaction processing system has a choice of protocols to execute a transaction that reads data from different nodes. Our results show that with a low number of threads per node, the system should use the READ protocol since it provides

the highest bandwidth. With a high number of threads per node, the system should use the WRITE protocol since it observes a higher bandwidth. This insight is particularly useful for adapting legacy software systems for use with RDMA.

A system architect may also use our results by mapping alternative designs to one of our abstractions to select the design that observes the highest RDMA bandwidth. For example, in the context of Fig. 1(a), a skewed access pattern may result in a few servers to fully utilize their NIC or CPU, dictating the overall performance of a thousand node cluster [13, 12]. One may implement load balancing techniques based on pull, push, or re-direction to resolve the bottleneck. With push, the bottleneck server *pushes* the most popular data to idle servers (1:M). With pull, idle servers *pull* the popular data from the bottleneck server (M:1). With re-direction, clients issue requests to any server. A server processes a client request by using its RDMA to fetch the referenced data from the server that stores the data (M:M). Our evaluation shows that *pull* observes an RDMA bandwidth that is four times higher than *push*. M:M observes poor horizontal and vertical scalability. We are developing techniques based on pull extended with use of server-side redirection and proxy. We refer the interested reader to [13] for details.

6 Related Work

Recent studies on databases [7, 5, 31, 21, 30, 23, 3, 4], key-value stores [27, 19], RPCs [20, 29, 18], and operating systems [26] use RDMA to enhance performance. Some use reliable queue pairs while others use unreliable queue pairs.

Kalia et al. [20] present design guidelines to achieve the highest performance using RDMA. It focuses on low-level optimizations for small messages and unreliable queue pairs. Its evaluation uses two machines and its sequencer implementation using these guidelines improves the performance by 50x. We apply some of these guidelines to achieve the maximum performance such as message inlining. This paper complements [20] by quantifying the performance and scalability characteristics of five protocols using a cluster of 29 nodes.

LITE [29] provides a kernel-level abstraction to ease the use of RDMA. LITE illustrates that the performance of RDMA does not scale due to the limited RNIC SRAM cache. LegoOS [26], built on top of LITE, uses RC WRITE to issue requests and transmit responses. Our extensive evaluation confirms that the alternative protocols perform the best with only a few threads issuing requests (a low number of QPs). In addition, our evaluation quantifies that using RC WRITE to issue requests and transmit responses provides a maximum of 1.5x higher bandwidth than SEND.

HERD [19] is a key-value store that uses UC WRITE to issue requests and UD SEND to transmit responses. It evaluates the inbound and outbound bandwidth of the alternative verbs extensively using one server machine and several client machines. Our evaluation complements this work by constructing three abstractions and focusing on five protocols using reliable queue pairs.

7 Future Work

Our performance results may be used to design novel algorithms for RDMA networks. One may conduct similar experiments using the unreliable connected (UC) and unreliable datagram (UD) queue pairs extended with a reliable protocol. The learned lessons guide the design and implementation of data reorganization techniques and data store architectures such as Nova [14] that employs components scattered across an RDMA network.

Acknowledgment

We gratefully acknowledge use of CloudLab network testbed [8] for all experimental results presented in this paper. We also thank the anonymous referees of the *6th Workshop on Performance Engineering with Advances in Software and Hardware for Big Data Science*, co-located with IEEE Big Data, for their valuable comments.

References

[1] M. Annamalai, K. Ravichandran, H. Srinivas, I. Zinkovsky, L. Pan, T. Savor, D. Nagle, and M. Stumm. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 445–460, Carlsbad, CA, 2018. USENIX Association.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[3] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1463–1475, New York, NY, USA, 2015. ACM.

[4] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed Join Algorithms on Thousands of Cores. *PVLDB*, 10(5):517–528, 2017.

[5] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proc. VLDB Endow.*, 11(11):1604–1617, July 2018.

[6] R. contributors. Redis, 2018.

[7] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[8] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[9] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 21:1–21:15, New York, NY, USA, 2018. ACM.

[10] S. Ghandeharizadeh, M. Almaymoni, and H. Huang. Rejig: A Scalable Online Algorithm for Cache Server Configuration Changes. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, TLDKS '19, 2019. To Appear.

[11] S. Ghandeharizadeh and H. Huang. Gemini: A Distributed Crash Recovery Protocol for Persistent Caches. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, pages 134–145, New York, NY, USA, 2018. ACM.

[12] S. Ghandeharizadeh and H. Huang. Hoagie: A Database and Workload Generator using Published Specifications. In *2nd IEEE International Workshop on Benchmarking, Performance Tuning and Optimization for Big Data Applications, co-located with IEEE BigData*, pages 3847–3852, Dec 2018.

[13] S. Ghandeharizadeh and H. Huang. Scaling Data Stores with Skewed Data Access: Solutions and Opportunities. In *8th Workshop on Scalable Cloud Data Management, co-located with IEEE BigData*, Dec 2019. To Appear.

[14] S. Ghandeharizadeh, H. Huang, and H. Nguyen. Nova: Diffused Database Processing using Clouds of Components [Vision Paper]. In *15th IEEE International Conference on Beyond Database Architectures and Structures (BDAS)*, Ustron, Poland, 2019.

[15] Y.-J. Hong and M. Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[16] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 8:1–8:7, New York, NY, USA, 2014. ACM.

[17] J. Hwang and T. Wood. Adaptive Performance-Aware Distributed Memory Caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX.

[18] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, 2019. USENIX Association.

[19] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, 2016. USENIX Association.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, Savannah, GA, 2016. USENIX Association.

[22] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

[23] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 37–50, New York, NY, USA, 2017. ACM.

[24] memcached contributors. memcached, 2018.

[25] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[26] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association.

[27] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 851–863, Boston, MA, 2018. USENIX Association.

[28] M. Technologies. libmlx4 driver. http://www.mellanox.com/downloads/ofed/MLNX_OFED-4.0-1.0.1.0/MLNX_OFED_LINUX-4.0-1. 0.1.0-ubuntu16.04-x86_64.tgz, 2019.

[29] S.-Y. Tsai and Y. Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 306–324, New York, NY, USA, 2017. ACM.

[30] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, Carlsbad, CA, 2018. USENIX Association.

[31] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, Feb. 2017.

[32] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.*, 12(11):1637–1650, July 2019.