Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing

Youmin Chen
Tsinghua University
chenym16@mails.tsinghua.edu.cn

Youyou Lu
Tsinghua University
luyouyou@tsinghua.edu.cn

Jiwu Shu*
Tsinghua University
shujw@tsinghua.edu.cn

Abstract

RDMA provides extremely low latency and high bandwidth to distributed systems. Unfortunately, it fails to scale and suffers from performance degradation when transferring data to an increasing number of targets on *Reliable Connection (RC)*. We observe that the above scalability issue has its root in the *resource contention* in the NIC cache, the CPU cache and the memory of each server. In this paper, we propose ScaleRPC, an efficient RPC primitive using one-sided RDMA verbs on reliable connection to provide scalable performance. To effectively alleviate the resource contention, ScaleRPC introduces 1) *connection grouping* to organize the network connections into groups, so as to balance the saturation and thrashing of the NIC cache; 2) *virtualized mapping* to enable a single message pool to be shared by different groups of connections, which reduces CPU cache misses and improve memory utilization. Such scalable connection management provides substantial performance benefits: By deploying ScaleRPC both in a distributed file system and a distributed transactional system, we observe that it achieves high scalability and respectively improves performance by up to 90% and 160% for metadata accessing and SmallBank transaction processing.

Keywords RDMA, Scalability, Resource Sharing

ACM Reference Format:
Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3302424.3303968

*Jiwu Shu is the corresponding author.

1 Introduction

In-memory processing, by placing data directly in DRAM, has been widely adopted to satisfy the increasing demands of extremely fast data storage and processing [3, 4, 19, 23]. Recently, Remote Direct Memory Access (RDMA), due to its low latency and high bandwidth, is becoming a promising technology to further narrow the gap between the network and storage. As such, it has been extensively used in distributed systems, including the in-memory key-value stores [16, 24, 33], transaction processing systems [9, 13, 18, 35], file systems [4, 15, 23], and deep learning systems [26].

However, we observe that the throughput (a.k.a, IOPS) of RDMA drops dramatically when the number of connections increases. By launching multiple clients to concurrently access a metadata server of an RDMA-enabled distributed file system, we are surprised to find that the throughput of `Stat` operations decreases sharply by almost 50% when the number of clients increases from 40 to 120. Besides, we also measure the raw throughput of the RDMA `write` verb, which declines from around 20 Mops/s to 2 Mops/s when the number of clients grows from 10 to more than 200. Many existing distributed systems adopt the client/server (C/S) model, thus forming the "one-to-many" message passing paradigm: For example, the communication between clients and key-value server to *get* or *put* key-value pairs, metadata accessing to the metadata server (MDS) in distributed file system, accessing to the sequencer or time server in distributed transactional system, and the data exchange between the parameter server and training nodes in distributed deep learning. As a result, the poor scalability of RDMA becomes a fatal blow for distributed systems

In this paper, we propose ScaleRPC, an RDMA-based RPC primitive on reliable connection (RC) mode to provide scalable message transferring performance. ScaleRPC achieves the scalability[1] goal by multiplexing the memory space, NIC cache and CPU cache efficiently in both time and space dimensions. Specifically, ScaleRPC first bounds the number of currently being served connections to a maximum to avoid the thrashing in the NIC cache. This is achieved by organizing the connections into groups and serving these groups in a time-sharing way. ScaleRPC then proposes virtualized mapping to allow one physical message pool to be shared

[1]In this paper, the term "scalability" refers to "the ability to transfer data to (from) one server from (to) an increasing number of clients without deterioration of the overall throughput".

among all the connections. Therefore, the messages from an increasing number of connections can be buffered with limited memory space. Besides, virtualized mapping also reduces the cache thrashing in the Last Level Cache (LLC) of the CPU, as it exports fixed memory addresses to the CPU cache for different connections.

To further improve the flexibility and efficiency of ScaleRPC, we are still required to handle the following challenges: 1) ScaleRPC should be flexible to meet the different requirements of different clients. In real-world applications, these clients may have different frequencies when posting requests and their behavior may change over time; 2) With virtualized mapping, switching between different groups should be light-weight and efficient, so as to avoid the server CPUs from been underutilized. To address these issues, we further optimize ScaleRPC by proposing *priority-based scheduling* to dynamically adjust the client groups, and *requests warmup* to reduce the group switching overhead.

Recent studies [1, 16–18] suggest using Unreliable Datagram (UD) or Dynamically Connected Transport (DCT) to design scalable software. However, both of them have to pay more effort in the software layer before being used in real-world applications. The detailed discussion is shown in Section 5.1. In this paper, we choose to improve the scalability of RDMA with efficient resource sharing at the system software layer. As far as we know, this is the first work to address the above issue from the system perspective rather than from the hardware. Besides, we target the RDMA-based RPC design because: Existing systems incorporate RDMA either by redesigning the system software [13, 23, 24, 35], which completely changes the I/O path via one-sided verbs, or substituting the network parts in existing software [16, 18] with RDMA-based RPC. Generally, it is a more feasible choice to only replace the RPC subsystem [25], since the RPC implementation is transparent to the applications and existing software can be easily ported to use ScaleRPC.

Our major contributions are summarized as follows:

1) We propose ScaleRPC to provide scalable RDMA on reliable connections. *Connection Grouping* and *Virtualized Mapping* are introduced to efficiently share the hardware resources between the concurrent clients.
2) We extensively evaluate our proposed ScaleRPC, and the experimental results show that ScaleRPC achieves competitive scalability and performance with FaSST RPC, state-of-the-art RDMA-based RPCs deployed on unreliable datagram (UD) mode.
3) To further verify the feasibility of ScaleRPC in distributed systems, we deploy ScaleRPC both in a distributed file system and a distributed transactional system. With RC verbs, ScaleRPC also enables us to optimize the transaction protocol by co-using the one-sided `read`/`write` and RPC primitives. Both the two systems with ScaleRPC achieve high scalability and improve performance by up to 90% and 160% respectively for metadata accessing and SmallBank transaction processing.

Table 1. RDMA verbs and MTU Sizes in Different Modes.

	send/recv	write/imm	read/atomic	MTU
RC	√	√	√	2 GB
UC	√	√	×	2 GB
UD	√	×	×	4 KB

The rest of this paper is organized as follows. Section 2 gives our observations that motivate this paper. We present the design and evaluation of ScaleRPC in Section 3. The implementation and evaluation of the two distributed systems are shown in Section 4. Discussions and related work are respectively given in Section 5 and Section 6, and the conclusion is made in Section 7.

2 Background and Motivation

2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) provides low-latency data transmission by directly accessing remote memory. It bypasses the operating system and supports zero-copy networking, and thus achieves both high bandwidth and low latency. RDMA can be configured in three modes: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). While UD supports both unicast (one-to-one) and multicast (one-to-many) without establishing connections, RC and UC need to establish connections first and support only one-to-one data transmission. Another difference between UD and RC/UC is the Maximum Transmission Unit (MTU). The MTU in UD is only 4 KB, while the MTU in RC/UC is as large as 2 GB. The difference between RC and UC is the reliability in the fabric. RC ensures data transmission is reliable and correct in the network layer, while UC doesn't have such guarantee.

RDMA accesses remote memory with two types of verbs, which are *message semantics* and *memory semantics*. Similar to socket programming, message semantics send and receive requests with RDMA `send` and `recv` verbs. RDMA `recv` is posted at the receiver before the sender posts `send` request, so as to specify the address of the incoming message. Since both the sender and the receiver are involved in the data transmission, they are also known as two-sided verbs. Memory semantics, such as RDMA `read` and `write` and their variants like `write_imm` and `atomic`, are capable of reading or writing data directly to (from) the remote memory without the involvement of remote CPUs. Thus, they are also known as one-sided verbs. These verbs are supported differently in different modes, which are shown in Table 1.

2.2 Poor Scalability of RDMA

As mentioned before, RDMA exhibits poor scalability when the number of connections increases. To better understand

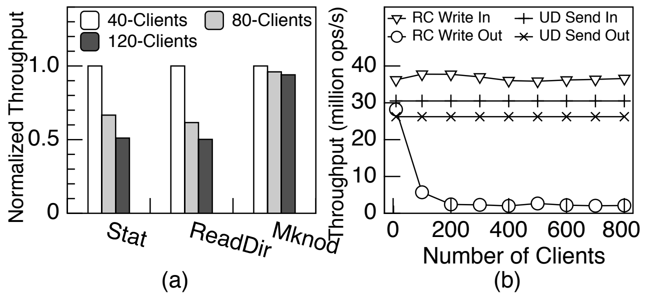


Figure 1. (a) Scalability Issue of RDMA in Distributed File Systems. (b) Raw Throughput of RDMA Verbs.

this issue, we first evaluate the metadata throughput of a distributed file system named Octopus [23]. Octopus is designed with light-weight software stack, so as to better exploit the hardware performance of RDMA and NVM. For simplicity, Octopus is configured with a single metadata server, and is accessed with an increasing number of clients. The server node and client nodes are connected via a 56 Gbps Mellanox SX-1012 switch (the detailed experimental setup is described in Section 3.6.1). As shown in Figure 1(a), when the number of clients increases from 40 to 120, the throughput of `Stat` and `ReadDir` operations decreases significantly (by almost 50%). We attribute this to the poor scalability of RDMA since Octopus itself never imposes any locking overhead for those read-oriented operations. We also notice that the throughput of `Mknod` declines slightly by about 5%. This is because Octopus has to do more work for update-oriented operations and the software layer is the main bottleneck.

We further measure the raw throughput of both inbound and outbound RDMA verbs (shown in Figure 1(b)). Among them, inbound verbs are the number of verbs that multiple remote machines (the clients) issue to one machine (the server); outbound verbs are those that one machine issues to multiple remote machines. Specifically, 10 threads are launched at one server to send (recv) 32-byte outbound (inbound) messages to (from) a number of clients. We observe that the throughput of outbound `write` drops from 20 Mops/s to 2 Mops/s as the number of clients grows from 10 to 800. The throughput of both inbound `write` and UD send is never affected.

2.3 Resource Contention in RDMA

Message Flow of RDMA. To better understand the root cause of the scalability issue of RDMA, we firstly cast a glance at the message flow of RDMA `write` between two servers: During the initialization phase, a Queue Pair (QP) is firstly created between two servers. It consists of a send queue and a receive queue to store the posted requests. Besides, one or multiple memory regions are allocated and registered with `ibv_reg_mr`, enabling them to be directly accessed by remote CPUs. As shown in Figure 2, to post a `write` request, the CPU of the sender initiates a verb request and sends it

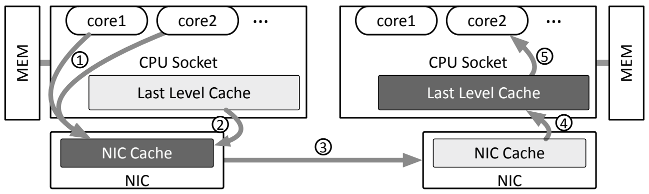


Figure 2. Message Flow With RDMA Network.

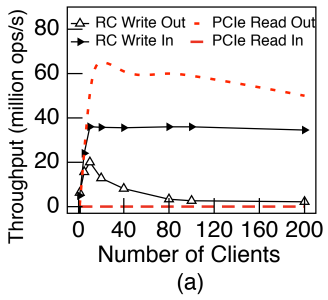
to the local NIC with MMIO (Memory Mapped I/O) (step 1). After the NIC is notified with the verb request, it collects the transferred data with DMA read (step 2), and sends it out (step 3). When the NIC at the receiver receives this message, it transfers the data to the memory with DMA write according to the memory address (step 4). The CPU checks the memory region repeatedly until discovering a new message (step 5).

While RDMA bypasses the operating system with the lightweight network stack, we find that *resource contentions* in the NIC cache, the CPU cache, and the memory significantly degrade system performance:

Cache Contention in the NIC Cache for Outbound Verbs. NICs mainly cache three types of information: ① the mapping table between the virtual and physical address of the registered memory, ② QP states, and ③ a work queue elements (WQE) cache [17]. FaRM [13] uses 2 GB huge pages while LITE [32] directly register physical memory so the mapping table size is greatly reduced (i.e., ①). However, when the number of connections grows, the total size of both QP states and WQE cache will be larger than the NIC cache size and cause cache thrashing (i.e., step 2 in Figure 2).

To quantitatively understand such issue, we collect the hardware counters of PCIe events when testing the raw performance of RC `write` (with the same configuration as in Figure 1(b)). The results are given in Figure 3 (a). We can find that before the outbound RC `write` reaches its peak performance, the corresponding PCIe read rate almost keeps in step with the `write` throughput. This is because, in each `write` request, the NIC has to read the payload via DMA read (step 2 in Figure 2). However, when the number of clients continues to grow, the throughput of PCIe read becomes far higher than that of the RC `write`. This is because, when the number of connections increases, the WQEs and QP states are more likely to be evicted out of the NIC cache by the newly posted one, and the NIC has to read them back from memory when processing them. Such extra efforts lead to the degradation of overall performance. On the contrary, the inbound `write` verbs exhibit almost constant performance and the PCIe read rate always keeps at a low standard, because the NIC only needs to store the messages to the local memory without modifying the cached states.

Cache Contention in the CPU Cache for Inbound Verbs. We can observe that the inbound `write` verbs don't cause the thrashing in the NIC cache. However, with DDIO



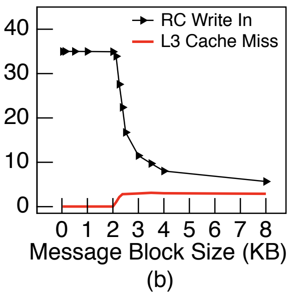


Figure 3. (a) Inbound/Outbound RC Write Throughput and the PCIe Read Rates. (b) Throughput of Inbound RC Write with Different Message Block Sizes.

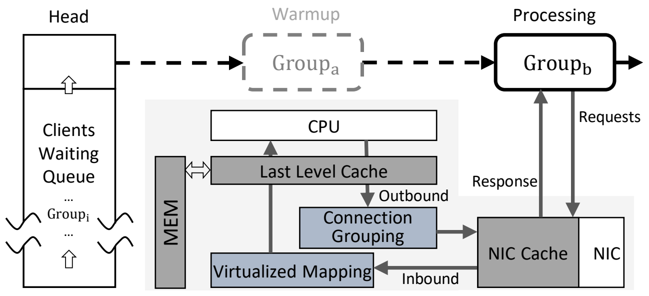


Figure 4. ScaleRPC Overview. (Clients are partitioned into groups and ScaleRPC only servers one group at a time slice)

(Data Direct I/O) technology [11], the NIC is allowed to directly write data to the last level cache in the CPU (step 4 in Figure 2) to improve the performance. The NIC accesses the LLC of the CPU with either *Write Update* or *Write Allocate*. *Write Update* allows in-place update if the accessed memory space already resides in the LLC. Otherwise, the NIC uses *Write Allocate* to write data to the newly allocated space in the LLC. However, *Write Allocate* mode is restricted to 10% of the last level cache in typical Intel CPU [11]. Therefore, if the remotely accessed memory is not efficiently cached in the LLC, the write allocate operations cause extra cacheline swapping operations and affect the overall performance. To reveal the above effect, we collect both the performance of inbound `write` and the L3 cache miss rate by varying the size of the accessed memory region at server side (as shown in Figure 3 (b)). Specifically, we partition the memory region of the server into message blocks, and clients are isolated to write to different blocks (most RDMA-based RPC adopts such design). In our configuration, the total number of clients is 400, each client has 20 message blocks to support batching, and the message size is 32-bytes. From the figure, we observe that as the message block size grows to larger than 2KB, the overall throughput shows a sharp drop from 35 Mops/s to less than 10 Mops/s, while the L3 cache miss rate increases accordingly. With 2KB message blocks, the size of the totally accessed memory is around 16 MB (2 KB×400×20), which is comparable to the LLC size. When there are more connections or with larger message block sizes, the efficiency of LLC cache is reduced, which further limits the scalability of inbound verbs.

We conclude that RDMA's scalability issue has its root in the *resource contention* of each server. Enhancements with more fine-grained management are required in the system software to improve the RDMA scalability.

3 ScaleRPC

3.1 Overview

Similar to the design of FaRM RPC [13], ScaleRPC chooses the RC mode and uses one-sided RDMA `write` verb for the communication between the server and client nodes, so as to meet the high reliability and performance requirements of the distributed systems.

As shown in Figure 4, ScaleRPC improves the RPC scalability from two aspects: 1) *Connection grouping* to reduce contention in the NIC cache for *outbound messages*. By grouping the number of connections being served, ScaleRPC serves a limited number of connections at one time, in order to balance the saturation and contention in the NIC cache. 2) *Virtualized mapping* to improve the CPU cache efficiency for *inbound messages*. It schedules the groups of connections to share one physical memory space. which is mapped into different logical pools for different connections. The limited usage of memory space also improves the memory utilization. To reduce the switching overhead between the groups of clients, ScaleRPC allows the next group of clients to warm up their requests in advance.

In ScaleRPC, the RPC request initiator is called the *RPCClient*, and the service that processes the RPC request is called the *RPCServer*. Once the *RPCServer* is started, it first allocates and registers huge pages (typically 2MB for each page) of memory from Linux kernel using `mmap`, which works as the message pool for remote clients. This message pool is formatted as contiguous message zones. Each zone is further cut into message blocks, whose size determines the largest message size it can support. Different *RPCClients* are mapped into different message zones, where clients can send RPC messages by directly posting RDMA `writes` to a specific message zone. In *RPCServer*, different message zones are owned by different working threads. Each thread polls new request from its own zones, invokes the RPC handler for requests processing, and sends the response message to the clients with RDMA `write` verbs. Previous work [13, 16] shows that RDMA updates memory in increasing address order, so we adopt a right-aligned layout for each message with three fields: `Data`, `MsgLen` and `Valid`. The `Valid` field at the end of each message is used for new message detection: Once the `Valid` is set as valid, the other two fields are confirmed to have been finished sending. Therefore, the *RPCServer* can decide the arrival of new requests by polling the value of

`Valid`. The `MsgLen` field represents the length of `Data` filed, which is the content of RPC message.

3.2 Connection Grouping

ScaleRPC introduces *connection grouping* to alleviate the resource contention in the NIC cache. Before the clients post RPC requests, they are organized into different groups by the *RPCServer*. The working threads at the *RPCServer* then serve them in a round-robin way. With such scheduling, the clients from the same group can send RPC requests simultaneously to the *RPCServer* during one time slice, and keep idle for the rest of time slices. Thus, the clients from different groups are strictly isolated with the time order. The grouping-based processing can avoid the cache thrashing in the NIC cache because the number of clients being served in each time slice is bounded to a maximum.

Priority-Based Scheduler. In real-world applications, different clients have different access frequencies to the *RPCServer*, and their behaviors may change over time. Therefore, a naive grouping strategy is not always optimal regarding the varying properties of different clients. Accordingly, we propose a priority-based scheduler to dynamically adjust the group size and time slice. It achieves this by monitoring the performance information of each client in real time. When connecting to the server, each *RPCClient* will be assigned a unique ID. For $client_i$ whose ID is i, *RPCServer* will record its throughput during the current time slice as T_i and the average request size as S_i. The priority of $client_i$ is defined as $P_i = T_i/S_i$. Hence, the clients with higher priority are more likely to send requests more frequently while carrying less payload. The priority-based scheduler manages those clients that have the same class of priority into the same group. The group with higher priority contains a fewer number of clients and a longer time slice. Such management is dedicated to squeezing the shared time wasted by those idle clients to serve those busy ones. Besides, the clients may dynamically log in or log out during execution, which makes the group size changes over time. Hence, the scheduler will lazily split or merge the groups when the current group size is out of the legal range. Based on our empirical results, the group is adjusted once its size is not within $[\frac{1}{2}, \frac{3}{2}]$ of the default group size.

Note that the group size setting has a high impact on the overall performance of ScaleRPC. With a small group size, the number of clients being served in each group is limited, and they cannot reach the peak performance of the NIC and server's CPU. On the other hand, larger group size aggravates contention in NIC caches, limiting the scalability. Hence, a proper choice of the group size balances the saturation and contention of the NIC cache. In our implementation, the group size is dynamically determined by two factors: 1) the cache size and the processing capacity of both NIC and CPU; 2) and the aforementioned scheduling policy.

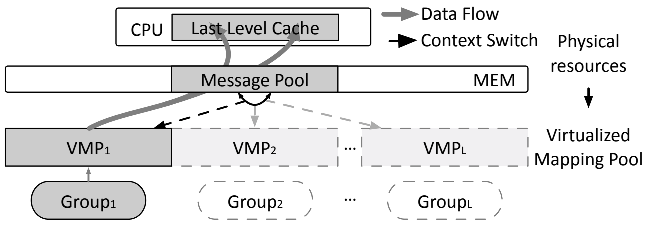


Figure 5. Virtualized Mapping at RPCServer.

3.3 Virtualized Mapping

The contention in the memory and CPU cache is another major bottleneck that affects the inbound throughput of UC/RC RDMA-based RPCs. With DDIO mechanism, the NIC directly writes the inbound messages to the last level cache of the CPU. However, the static mapping makes the size of the message pool be proportional to the number of the clients. As a consequence, the message pool fails to reside in the last level cache when serving too many clients. This leads to large amounts of *Write Allocate* operations and causes performance degradation.

In ScaleRPC, we reduce memory consumption and CPU cache miss ratio with *virtualized mapping*. As mentioned in Section 3.2, with grouping-based message processing, only the clients in the group being served are allowed to post RPC requests. Therefore, we only need to allocate one physical message pool which is exactly capable of serving one group of clients, in a virtualized mapping way. As shown in Figure 5, the physical message pool is virtualized to multiple logical message pools, each of which is used for RDMA communication with one group of clients. In this way, a single physical message pool serves all connections. Note that the message pool is stateless, which means that the messages in the memory pool become obsolete immediately after the request has been processed. Based on this property, the physical message pool is shared between different groups without memory resetting. Similarly, the data that cached in the last level cache of CPU also becomes invalid, and does not need to be evicted. Thus, clients in the next groups can directly overwrite the message pool without evicting or reloading the cache.

Context Switch. To achieve such virtualized mapping, each virtualized message pool is associated with its context metadata, including the *ClientID*s, *offset*s, and *performance counter*s for the clients of each group. The metadata is saved and reloaded by the priority-based scheduler at the end of each time slice (a.k.a., context switch point). Before switching to another group, each working thread is notified by the scheduler to process and clear the suspended requests in the message pool, and notify the corresponding clients with a `context_switch_event`, which is piggybacked directly in the response message. For the clients in the current group

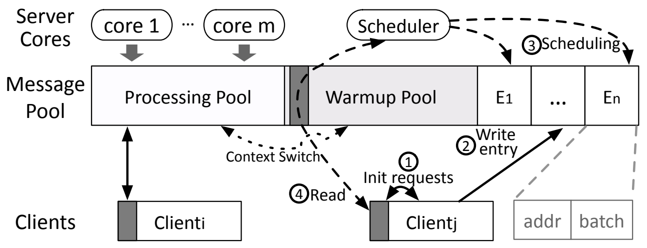


Figure 6. Warmup and Scheduling. Client$_i$ is in Process state, while Client$_j$ is under warmup.

that don't have any active requests in the message pool, *RPC-Server* also needs to notify them the `context_switch_event` with extra RDMA `writes`. The ratio of the introduced RDMA `writes` is lower than 0.01% when compared to the real payload, introducing insignificant effect on the overall performance. After processing all the remaining messages, the scheduler saves the context metadata, and recovers the next group by reloading its metadata.

Requests Warmup. Switching between two groups often makes the working threads at the *RPCServer* idle for a while to wait for the new requests from the next group, which impact the overall performance dramatically. We address this issue by allowing the clients from the next group to warm up the requests before being served. As shown in Figure 6, we introduce a *Warmup Pool* and an array of *Endpoint Entries* to reduce the context switch overhead.

3.4 Putting Everything Together

After establishing a connection with *RPCServer*, the client will first move to `WARMUP` state, during which the RPC requests are firstly initialized locally (step 1 in Figure 6). Then the client sends a tuple formatted as $<$ req_addr, batch_size $>$ of local requests to the corresponding endpoint entry using RDMA `write` (step 2, write to E_j according to the Client ID). The scheduler at *RPCServer* will choose the clients to form the warmup group according to the scheduling strategies described in Section 3.2 (step 3). It then fetches the requests actively from those clients' memory with RDMA `read` according to the endpoint entry information (step 4). When the context switch occurs, the warmup pool becomes the processing pool and the primary processing pool begins to warmup the next group of clients. The working threads begin to process the requests in the new processing pool (previous warmup pool). When the client in the `WARMUP` state receives the first response message from *RPCServer*, it moves to `PROCESS` state. At this time, the client is allowed to write new request directly to the processing pool with RDMA `write`. When the client receives a response message with `context_switch_event`, it moves to `IDLE` state, and starts again by repeating from step 1 again (Figure 7 shows the state transition of each client in detail).

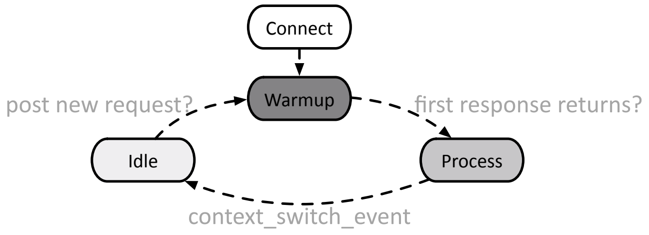


Figure 7. State Transition of the clients in ScaleRPC.

With the warmup pool, the server threads are enabled to process the requests *in a pipelined way*. The execution of context switch is completely hidden from the critical path. With virtualized mapping, ScaleRPC can support unlimited number of clients theoretically, as all the clients share a same physical message pool. On the contrary, static mapping approach (such as in HERD RPC) only supports a limited number of clients once the message pool has been formatted.

3.5 Deployment Considerations

ScaleRPC assumes each *RPCClient* execute independently, where there is no synchronization among them. This is usually the case for many distributed systems following client-server architecture. Furthermore, the *RPCServer* and *RPC-Client*s are assumed to cooperate together to make the aforementioned optimizations work properly. We achieve this by implementing a group of easy-to-use APIs to trigger the execution of remote procedures, which are *SyncCall()*, *Async-Call()* and *PollCompletion()*. Among them, the two asynchronous APIs (i.e., *AsyncCall* and *PollCompletion*) are provided to enable the clients to post multiple (*batch* of) remote calls at a time before collecting their response messages. Hence, for those distributed systems with "one-to-many" data transferring, ScaleRPC can be directly incorporated in by calling those APIs. However, when the clients need to access multiple *RPCServer*s simultaneously, such as in distributed transaction processing, we need to introduce *Global Synchronization* to make ScaleRPC applicable (in Section 4.2).

Connection grouping in ScaleRPC improves the overall throughput and shortens the average latency, but prolongs the maximum latency compared to RawWrite. However, we also observe that UD-based RPC (such as FaSST and HERD) exhibits even higher tail latency: The tail latency of them is more than 200 μs when serving 120 clients, which is much higher than ScaleRPC (in Section 3.6.2). In conclusion, ScaleRPC successfully achieves scalable performance and reliable data transmission, with the price of relatively higher tail latency. We believe that such sacrifice is worthwhile, because one-sided RC verbs with the scalable and reliable property are vital for the design of distributed systems (see Section 4). Another limitation is that the RPCs with long execution time may fail for the first time since they may be half-executed before a context switch occurs. To address this issue, their information is recorded. Any subsequent requests with the same call type will be executed by a separate thread in legacy mode. Anyway, this is not always the case in in-memory storage systems because the execution time of RPC tends to be short as the data is stored in DRAM.

3.6 Evaluation of ScaleRPC

In this section, we evaluate ScaleRPC's performance as well as the benefit breakdown from each mechanism design.

3.6.1 Experimental Setup

Evaluation Platform. Our cluster consists of 12 nodes, each of which is equipped with two 2.2GHz Intel Xeon E5-2650 v4 processors and 128 GB of memory. Each processor has 12 physical cores. All these servers are installed with CentOS 7.4 and are connected with a Mellanox SX-1012 switch using MCX353A ConnectX-3 FDR HCAs (56 Gbps over IB and 40 GbE). To evaluate the performance of ScaleRPC, we use coroutines provided by Boost C++ library to simulate hundreds of clients. In client nodes, each thread creates multiple coroutines, and each coroutine initiates one *RPCClient*. The working threads schedule the coroutines in a round robin manner. Each time one coroutine posts a batch of requests with the asynchronous APIs, it yields to the next coroutine. The response messages are polled before each coroutine sends the next batch of requests. In ScaleRPC evaluation, we choose one node to work as *RPCServer* and the rest of the servers are used to run *RPCClient*s.

Table 2. RPC Implementations for Comparison.

RPC	Description
RawWrite RPC	a baseline RPC implementation based on RC `write` verbs
HERD RPC [16]	a scalable RPC with a hybrid of UC `write` and UD send verbs.
FaSST RPC [18]	a scalable RPC based on UD `send` verbs.

Compared RPC Primitives. We compare ScaleRPC with three RDMA-based RPC primitives (as listed in Table 2). In the three RPC implementations, RawWrite is a variation of ScaleRPC with all the optimizations disabled (just the same as the FaRM RPC [13]). HERD RPC [16] uses UC `write` to send request messages to the server, while the server uses UD `send` to send the response message back. FaSST RPC [18] uses UD `send` to post both request and response messages. We configure the FaSST RPC to work asymmetrically, where multiple clients post requests to a single server. The clients of both RawWrite, HERD and FaSST are simulated using coroutine just like that of ScaleRPC. In our evaluation, the time slice and group size of ScaleRPC are set to 100 μs and 40 respectively by default. The message pool is formatted as continuous message blocks with size of 4 KB [2].

[2]We choose 4 KB as default message block size because this is the largest message size supported by UD-based RPC like FaSST [18] and HERD [16].

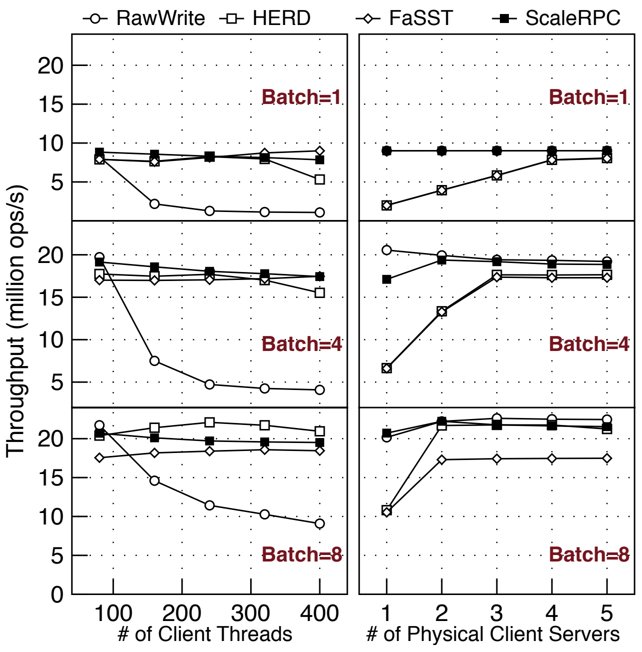


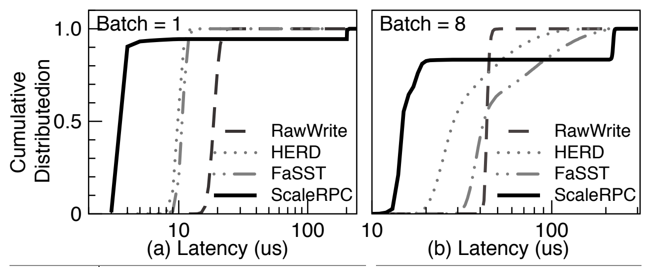
Figure 8. Throughput Evaluation for Different RPC Implementations.

3.6.2 Overall Performance

We evaluate ScaleRPC's scalability by collecting both throughput and latency results. The default message size is 32-byte.

Throughput. We collect the throughput results with two kinds of experiments. One is to vary the number of clients from 40 to 400, which are distributed evenly to 11 physical client servers (shown in the left half of Figure 8). The other is to vary the number of physical client servers from 1 to 5, with a total number of client threads of 40 (shown in the right half of Figure 8). From this figure, we make the following observations:

(1) ScaleRPC achieves comparable scalability to FaSST, and significantly outperforms RawWrite, as shown in the left half of Figure 8. RC-based RawWrite exhibits a significant performance drop due to the resource contention in the RC mode. HERD has much better scalability than RawWrite, but its performance still drops when the number of clients is large, especially when the batch size is small. This is mainly owing to the static mapping design of the message pool, which fails to reside in the CPU cache when the number of clients increases. FaSST uses UD `send` to post both request and response messages, so it doesn't need to create QPs for each client. Besides, the addresses of the incoming requests are determined by the FaSST server (by posting UD `recv`), indicating that FaSST doesn't need to create separate buffers for different clients. As a whole, FaSST shows stable throughput despite variations in the number of clients. In comparison, ScaleRPC using RC verbs achieves similar scalability to



	RawWrite	HERD	FaSST	ScaleRPC	RawWrite	HERD	FaSST	ScaleRPC
Median(us)	19	10	11	4	44	29	42	15
Average(us)	19.7	15.8	15.8	13.5	52.0	45.3	53.6	46.9
Max(us)	25	16	19	**217**	57	187	**367**	230
TP(Mops/s)	6.1	7.7	7.7	8.9	18.5	21.2	17.7	20.5

Figure 9. Latency Evaluation for Different RPC Implementations (120 clients).

UD-based FaSST, and keeps almost constant performance when the number of clients increases from 40 to 400.

(2) ScaleRPC and RawWrite are more effective in exploiting the RDMA hardware benefits than FaSST and HERD. As shown in the right half of Figure 8, when the batch size is 1, FaSST and HERD require 40 client threads to be distributed to at least 4 physical client servers to saturate its throughput. With larger batch sizes, FaSST and HERD still require multiple physical servers to saturate the throughput. In contrast, Scale RPC and RawWrite can be saturated with at most two physical servers. This is because, for each UD-based RPC, the client needs to post `recv` verbs beforehand and use *ibv_poll_cq* to poll the response message, rather than directly check the local message pool. With such working mode, CPU is more likely to be the bottleneck.[3] We conclude that RC-based RPCs verbs are effective in exploiting the hardware benefits than UD-based ones.

Latency. Figure 9 shows the cumulative latency distribution of all the evaluated RPCs. We set the number of clients to 120 with a varying batch size (1 and 8 respectively). The median, average, maximum latencies and corresponding throughput are also shown in the table. We record the latency of each batch as $T_2 - T_1$, where T_1 is the start time when posting each batch of requests; T_2 is the end time when all the response messages of this batch return. From the results, we make the following observations:

(1) ScaleRPC shows a bimodal distribution of latencies due to the grouping-based scheduling, while the other RPCs are more smooth in the latency distribution. ScaleRPC is capable of keeping most of its requests with extremely low latency, e.g., most of the requests have latencies of around 4 μs with batch size of 1, and 15 μs with batch size of 8. In contrast, RawWrite, HERD and FaSST respectively have median latencies of 19 μs, 10 μs and 11 *us* when the batch size is 1. When the batch size is 8, both HERD and FaSST have a wide latency spectrum, most of which falls in the range from 20 μs to 200 μs.

(2) ScaleRPC shows much higher maximum latency when the batch size is 1. This is mainly because we are using lighter workloads and the network is not fully saturated. Thus, connection grouping increases tail latency unnecessarily. But even so, ScaleRPC is still more efficient in delivering higher throughput than the compared systems.

(3) ScaleRPC has close or smaller maximum latencies with high concurrent workload (e.g., when the batch size is set to 8). In this case, the network is fully saturated. It's easy to understand that the maximum latency of ScaleRPC is closely related to the size of time slice and the number of groups. However, UD-based RPCs like HERD and FaSST exhibit a wide latency spectrum, and they show the same or even higher tail latency.

As a whole, ScaleRPC achieves comparable scalability with UD-based RPCs and is more effective in exploiting the hardware benefits. It also provides lower latencies to the majority of the requests, and keeps close or smaller maximum latencies. In the future, we expect to adopt more fine-grained scheduling regarding the sensitivity of different *RPCClient*s (e.g., different clients may have different requirements on either throughput or latency), thus combining the advantages of both low tail latency and high scalability.

3.6.3 Analysis of Internal Mechanisms

To understand the effects respectively from the internal mechanisms of ScaleRPC, we collect the hardware performance counters for analysis. These hardware counters are collected using the Processor Counter Monitor (PCM [2]) tool provided by Intel. Among them:

- `PCIeRdCur` indicates the number of operations of reading data blocks from memory to PCIe devices.
- `RFO` is the number of operations of writing partial data blocks to memory from PCIe devices.
- `ItoM` indicates the number of operations of writing full data blocks to memory from PCIe devices. The summary of `RFO` and `ItoM` gives the total number of write operations from PCIe devices to memory.
- `PCIeItoM` indicates the number of full data block write operations using the *Write Allocate* mode when writings data to memory from PCIe devices[4].

Effects of Connection Grouping. Figure 10 shows the throughput and the associated hardware counters respectively for RawWrite and ScaleRPC. From the figure, we can see that the `PCIeRdCur` counter of RawWrite increases dramatically to more than 20 *million ops/s* when the number of clients grows larger than 40. This tells that, when there are more than 40 clients, there are a large number of PCIe read operations. The reason lies in the NIC cache thrashing.

[3]private talk with Mellanox engineers

[4]Since `PCIeItoM` counter is valid only on Intel v1/v2 CPUs, we move *RPCServer* to a v2 CPU platform to replay the experiments.

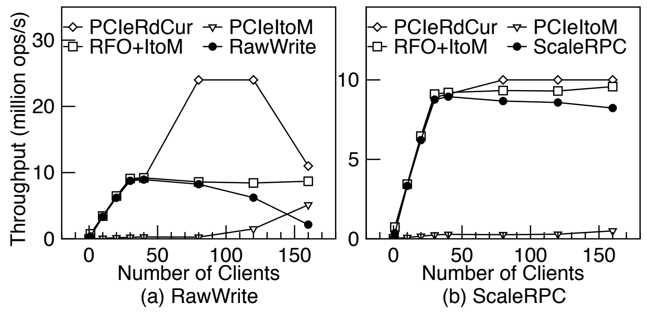


Figure 10. Analysis of the Internal Mechanisms using Hardware Counters.

When the NIC cache cannot keep all the QP states for the connected clients, they are evicted to the main memory. Also, the WQEs also need to be switched out and in from the NIC cache to the memory for different connections. Both of them result in extra PCIe read operations. When the number of clients is 150, the `PCIeRdCur` counter drops again, but the RawWrite performance is still poor. This is because the high rate of allocating writes (indicated by high `PCIeItoM` which will be discussed next) leads to slow performance. In comparison, the `PCIeRdCur` counter keeps pace with the throughput of ScaleRPC, and ensures its scalability.

Effects of Virtualized Mapping. As shown in Figure 10, the `PCIeItoM` counter shows a different pattern in RawWrite and ScaleRPC. In RawWrite, the `PCIeItoM` counter increases when the number of clients grows. This is because more message pools are allocated for the growing number of clients. The increased message pools result in more CPU cache misses. In other words, larger message pool sizes lead to poorer CPU cache efficiency. Therefore, with either larger block size in the message pool or more message pool blocks, the cache efficiency is reduced. In comparison, the virtualized mapping technique in ScaleRPC virtualizes different memory spaces using a single physical message pool. This single physical space reduces the chances of data eviction from the CPU cache when the virtualized memory spaces serve different clients. As shown in the figure, the `PCIeItoM` counter in ScaleRPC has a stable but low rate for a different number of clients.

3.6.4 Analysis of Uniform Workloads

As mentioned before, the group size and time slice size is partially determined by the cache size and the processing capacity of both NIC and CPU. Thus, we analyze these parameter settings with uniform workloads by sending 32-byte requests to the *RPCServer*.

Configuration of the Time Slice Size. In this part, we measuring the throughput of ScaleRPC with varied time slice size from 30 μs to 250 μs (80 clients, group size of 40 and batch size of 1, shown in Figure 11(a)). From the figure, we observe that the throughput improves from 7.6 Mops/s to

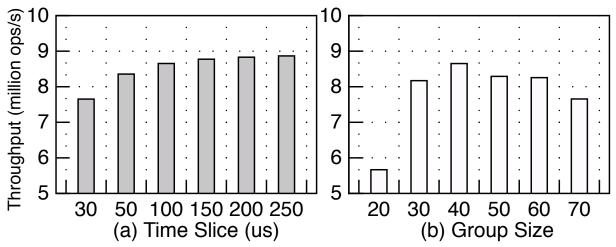


Figure 11. (a) Sensitivity to the time slice size. (b) Sensitivity to the group size.

8.9 Mops/s when the time slice grows from 30 us to 250 us. This is because with a smaller time slice, the frequent context switches incur much overhead and cause more network traffic of extra timeout notifications, thus limiting the performance. Using a larger time slice is helpful for improving the throughput, however, it incurs longer waiting time in the `IDEL` state, which amplifies the tail latency. In our evaluation, the 100 *us* time slice provides both high throughput and low latency, and is a feasible choice.

Configuration of the Group Size. In this evaluation, we vary the group size from 20 to 70 with a step of 10, and use two groups of clients to access the *RPCServer*. Figure 11(b) shows the throughput under different group size settings. From the figure we can see that the throughput of ScaleRPC is first increased and then decreased. For small group sizes such as 10, ScaleRPC only provides 5.7 million operations per second. The main reason is that the clients in a small group are unable to saturate the RDMA network. For large group sizes such as 70, ScaleRPC also encounters a slight performance drop. This is because the resource contention becomes more serious in the CPU cache and the NIC cache. As a whole, the group size of 40 provides high throughput in our hardware, which is optimal to balance the high throughput and low latency.

3.6.5 Analysis of Non-uniform Workloads

Apart from the hardware effects, the group size and time slice size are also dynamically affected by the behavior of different clients. To reveal the effects of the priority-based scheduler, we measure the throughput of ScaleRPC with imbalanced workloads. This is achieved by launching clients with different access frequency distribution. For comparison, we implement a *Static* mode based on ScaleRPC. It keeps both group size and time slice a fixed number.

Figure 12(b) shows the throughput of two modes when accessed by clients with different AFD. To simulate such imbalanced behavior, we inject different latencies for each client before they post the next request. The latencies we choose for different clients follow a Gaussian distribution (with σ set to 0.8 and 1 respectively in our evaluation). With dynamic scheduling, those clients with higher accessing frequency are organized into the same group and share a larger

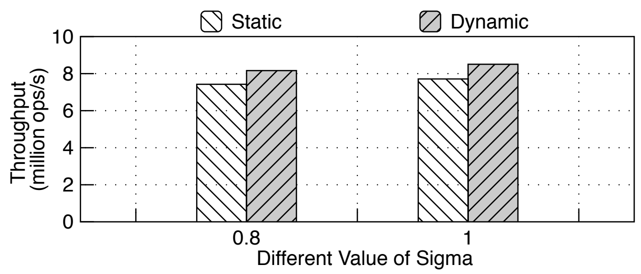


Figure 12. Throughput with different type of access frequency distribution.

time slice. Our experiment shows that *Dynamic* mode outperform *Static* mode by 9% and 10% with different σ.

4 Deployments of ScaleRPC

It's necessary to notice that the introduced techniques in Section 3 are not restricted in RPC design, but benefit all the systems deployed on RC/UC verbs. Based on such principle, we (1) transplant ScaleRPC into an existing distributed file system with little efforts, and (2) redesign a distributed transaction system named ScaleTX by rebalancing the loads between coordinators and participants by co-using the one-sided verbs and ScaleRPC.

4.1 ScaleRPC in Distributed File System

A typical distributed file system (DFS) consists of a single metadata server (MDS) and multiple data servers (DSs). The lack of highly scalable and parallel metadata processing with a single-node MDS is becoming an important performance bottleneck. In this section, we replace the RPC subsystem of Octopus [23] with ScaleRPC to reveal its effects in real-world workloads. Octopus is an efficient distributed file system deployed on emerging hardware like RDMA and NVM. It redesigns the software stack by abstracting a distributed shared persistent memory pool, so as to reduce redundant memory copying. It also introduces *self-identified RPC* for metadata access. Different from ScaleRPC, *Self-identified RPC* uses RDMA *write-imm* to post requests. In this way, the server threads can directly locate the new messages with the encapsulated *immediate* number, avoiding to scan the whole message pool.

We use *mdtest* benchmark to evaluate the metadata performance. Figure 13 shows the metadata performance when using self-identified RPC (abbreviated as selfRPC) and ScaleRPC respectively in Octopus with a varying number of clients. HERD and FaSST are not evaluated here because both of them use Unreliable Datagram mode, which has limited MTU of 4 KB, dissatisfying the requirement of variable-sized metadata access to the MDS. As shown in the figure, ScaleRPC outperforms selfRPC in all four evaluated operations. In detail, for those write-oriented metadata operations like `Mknod` and `Rmnod`, ScaleRPC slightly outperforms selfRPC by 5% - 6.5% with 80 and 120 clients. This is because both `Mknod` and

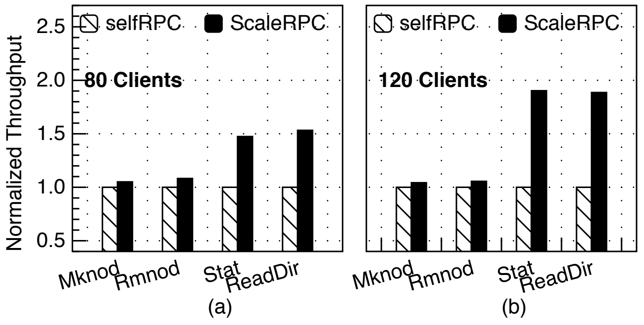


Figure 13. Performance of ScaleRPC in File Systems.

`Rmnod` operations require more complicated processing in the file system, which incurs higher software overhead in the file system itself than in the network. For the read-oriented metadata operations like `Stat` and `ReadDir`, ScaleRPC outperforms selfRPC respectively by 50% and 90% on average with 80 and 120 clients launched. Since those read-oriented operations introduce negligible software overhead in the file system, the scalability benefits of ScaleRPC dominate the overall system performance. In conclusion, ScaleRPC supports variable-sized reliable data transmission, which is necessary for file systems, and provides high performance and scalability simultaneously.

4.2 ScaleRPC in Transactional System

We further implement a distributed transactional system running on ScaleRPC named ScaleTX. It consists of two parts: the *coordinators* and the *participants*. The coordinators (also act as the clients) are responsible to initiate and coordinate the transaction, while the participants (also act as the storage servers) need to make responses when involved in the transaction processing. A key-value store is deployed on the storage servers and provides serializability with ScaleTX when operating on multiple key-value pairs. The key-value store is an in-memory hash table which has the same layout as that of MICA [20]. In our implementation, we deploy three servers (participants) to manage the key-value store, each of which store one shard. These three servers also act as *RPCServer* to process the transactional requests from the clients (i.e., coordinators).

Global Synchronization. As illustrated in Section 3.5, ScaleRPC targets at improving the scalability of "one-to-many" data transferring. Therefore, ScaleRPC cannot be directly used in "many-to-many" data transferring as required in ScaleTX: In the transaction processing, a client may access multiple servers simultaneously. In ScaleRPC, however, the *RPCServers* schedule the clients independently: When one client is in `PROCESS` state of one server, it may still in `WARMUP` state of another server, making the clients always stalling.

We use an NTP-like protocol to synchronize the *RPCServers*, so as to make them switch the groups of clients at the same pace. During the initialization phase, we choose one of the *RPCServers* to work as the time server, which is

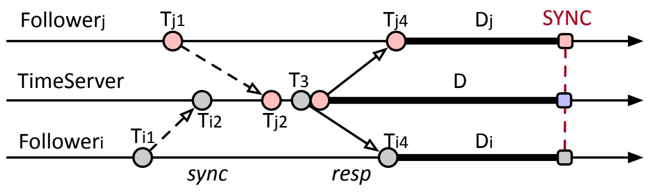


Figure 14. Synchronization Between *RPCServers*.

predefined with the configuration scripts. Other *RPCServers* (namely the followers) send *sync* requests to the time server periodically for global synchronization. As shown in Figure 14, the current time of T_{i1}, T_{i4} and T_{i2}, T_3 are recorded respectively by the followers and the time server when sending or receiving the *sync* or *resp* message. Besides, the time server will encapsulate the value of $(T_3 - T_{i2})$ (denoted as ΔT_i) in the *resp* message. After this, the time server and the followers will sleep for D and D_i respectively before the next context switch occurs, where D is a pre-defined value and $D_i = D - (T_{i4} - T_{i1} - \Delta T_i)/2$. With such design, all the *RPCServers* can schedule the clients with the same pace. In our implementation, the global synchronization event occurs once in every 100 ms, with an insignificant impact on the overall performance.

Similar to FaSST [18] and FaRM [13], we use optimistic concurrent control for serializability between transactions execution and two-phase commit for consistent commit. We further optimize the transactional protocol by co-using ScaleRPC and one-sided verbs, so as to reduce latency and improve throughput. As shown in Figure 15, the client acts as the coordinator and the KV servers act as the participants. The keys that read and updated by the transaction are denoted as *read set* (R) and *write set* (W) respectively. In this figure, the write set contains w_1 and the read set consists of r_1 and r_2.

1) **Execution:** The coordinator reads the key-value items from R and W by posting RPCs to the participants. Meanwhile, each item in W is locked by the KV servers, and the addresses of each KV item in R and W are sent back to the clients. Note that each KV item in the KV server has a co-located version number. To ensure the serializability, the versions and the version addresses of each element in R are sent back as well.
2) **Validate:** The coordinator checks the versions in R by posting RDMA `read`s according to the version addresses collected in the execution phase. If any version in R is modified by a concurrent transaction, the validation phase fails and this transaction is aborted.
3) **Log and Commit:** To commit a transaction if the validation succeeds, the coordinator first appends the log entries in each primary node of W by posting RPCs. If logging succeeds, the coordinator updates the primary key-value items in W by directly using RDMA `write`s. Meanwhile, the lock filed is released by zeroing the lock field of each KV item.

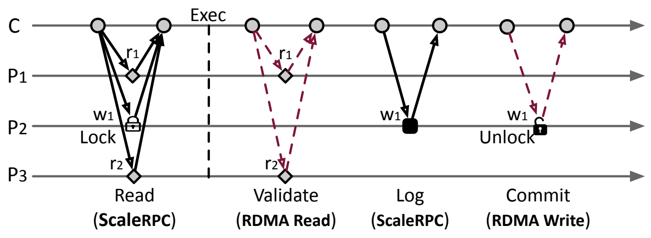


Figure 15. Protocol of ScaleTX. C indicates the coordinator and P1, P2 and P3 are the participants.

It is worth mentioning that ScaleTX uses one-sided `read` and `write` during the validation and commit phase. Such design has two advantages: (1) ScaleTX naturally inherits the good scalability of ScaleRPC. (2) ScaleTX uses one-sided verbs to offload a part of execution logic from the participants to the coordinators, improving the overall performance.

4.2.1 Evaluation of ScaleTX

We evaluate ScaleTX with two benchmarks: (1) an object store with different configurations of read set and write set, similar to the read-intensive OLTP benchmark in FaSST [18]; (2) a write-intensive benchmark named SmallBank [5]. We deploy RawWrite RPC, HERD RPC, FaSST RPC and ScaleRPC to run the distributed transaction protocol described in Section 4. For comparison, we disable the optimization of using one-sided verbs and replace them with RPCs. These transaction systems are respectively called RawWrite, HERD, FaSST and ScaleTX-O for short.

Object Store. In object store evaluation, we generate workloads with random keys and each transaction contains r items in read set and w items in write set, which is denoted as *(r,w)*. Figure 16 shows the performance of different implementations, and we have the following observations:

1. Generally, both HERD and FaSST have poor performance with 80 clients connected. This is mainly because UD connections are inefficient to achieve high throughput with the limited number of physical client nodes (as described in Section 3.6.2). The transaction with RawWrite shows the highest throughput in all evaluated workloads when connected by 80 clients, but it drops by 56 % on average when the number of clients increases to 160. This is mainly owing to the poor scalability of RawWrite RPC. Particularly, ScaleTX achieves the highest throughput among all the systems while keeping good scalability.

2. For read-only transactions as shown in Figure 16(a.1), we find that ScaleTX has the same throughput as ScaleTX-O. This is because posting one-sided verbs (i.e., RDMA `read`) doesn't reduce the network traffic. However, using one-sided verbs still reduces the execution latency by 15% with low-concurrent workloads (not shown in the figure).

3. For read-write transactions (in Figure 16(a.2)), ScaleTX can respectively outperform RawWrite, HERD, FaSST and

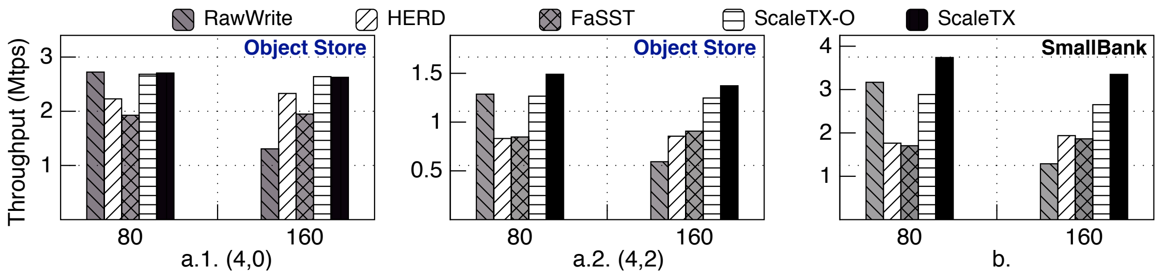


Figure 16. Performance of ScaleTX. (a.*) represent the performance of object store with varying size of write set and read set. (b) shows the performance of SmallBank.

ScaleTX-O by 131%, 60%, 51% and 10% with 160 clients connected. This improvement mainly owes to (1) the efficient and scalable design of ScaleRPC, and (2) the one-sided RDMA `write` used in multiple phases, which offloads the transaction committing overhead from servers to clients.

SmallBank. The SmallBank OLTP benchmark simulates simple bank account transactions. It is write-intensive with 85% of update transactions. In SmallBank evaluation, we load 1, 000, 000 bank accounts per server in advance, and let the 4% of the total accounts be accessed by 60% of transactions.

From Figure 16(b) we find that ScaleTX has the best performance and outperforms RawWrite, HERD, FaSST and ScaleTX-O by 18%, 112%, 120% and 30% respectively with 80 clients, and by 160%, 73%, 79% and 26% with 160 clients, which reflects both the high scalability of ScaleRPC and high efficiency of ScaleTX. We also notice that the performance gap between ScaleTX and ScaleTX-O increases when compared with that in the object store. This is because SmallBank is write-intensive, in other words, has a larger write set. During the transaction commit phase, all other four transaction systems have to pay more efforts to send RPCs for committing the transactions, while ScaleTX only needs to post write verbs without waiting for the feedback messages. This also proves that in large-scale, write-intensive transaction processing (which widely exists in banking, e-commerce, etc.), adopting one-sided verbs in RC RDMA to redesign light-weight transaction processing protocols is particularly important, and ScaleRPC provides such opportunity.

5 Discussion

5.1 Existing Approaches.

UD-based. Recent research has suggested the Unreliable Datagram (UD) mode to design the scalable software [16, 18]. Though the UD mode shows good scalability (as shown in Figure 1), its limitations are also obvious: (1) One-sided verbs are not supported by UD connections, which are the most important verbs to exploit the performance of RDMA network [13, 24, 34, 35]. (2) UD cannot transfer data larger than 4 KB. To support ordered transferring of large-sized messages, the data has to be cut into contiguous 4 KB slices, and the acknowledgment has to be made by the receiver before the next transmission. However, in our implemented prototype, such ordered transferring only provides the bandwidth of 0.8 GB/s with a single thread, which is merely 12.5% of that of the RC verbs. Transferring data in a pipelined way can improve the performance, but inevitably causing increased complexity in the software.

Dynamically Connected Transport (DCT). It is recently introduced in the new generation of Mellanox's HCAs to improve its scalability on reliable connections [1]. It achieves this by sharing the context between all the connections: the context is created each time the data transmission occurs by posting an inline message to the other side, and then destroyed immediately when switching to another connection. With the above approach, the size of cached connection status in the NIC is efficiently restricted. However, for small-sized network requests, DCT almost doubles the number of network packets compared to the real payload. DCT increases latency by 100 μs to 3 μs on RC mode [31].

Newer generation of HCAs. Mellanox's ConnectX-4 and ConnectX-5 HCAs are all equipped with larger cache space to improve scalability. However, eRPC [6] reveals that their throughput drops almost by half as the number of connections increases to 5000. DrTM-H [34] also has similar experimental results. Since these HCAs adopts the memory-less architecture, it's unlikely that improvements in NIC hardware will allow it to scale to unlimited number of connections.

5.2 Unique Properties of ScaleRPC

Remote procedure call (RPC), as a simple and versatile abstraction for inter-node interaction, has been widely adopted in existing distributed systems. Accordingly, we improve the scalability of RDMA by providing a scalable RDMA-based RPC primitive. Any existing software with "one-to-many" messaging paradigm can be easily ported to use ScaleRPC. Similar to existing approaches [17, 18], we achieve performance improvements with batching by providing a group of asynchronous APIs. Request batching doesn't impose extra restrictions on existing software since most of them [8, 12, 29, 37] always batch the requests before sending them. Others that don't leverage such opportunity still can use the synchronous APIs.

Besides, this paper implements ScaleRPC based on reliable connection (RC). We choose RC mode because 1) RC supports sending as large as 2 GB of data at a time, which meets the requirements of sending variable-length payloads in real-world applications. 2) One-sided RC verbs have higher performance compared with UD `send`, since one-sided `reads` or `writes` eliminate the MMIO overhead for the receivers to post `recv` verbs. 3) Improving the scalability of RC RDMA verbs enables us to co-use RC-based RPC primitives and one-sided verbs. As illustrated in Section 4.2, selectively using RPCs and one-sided verbs at different phases in the transactional protocol can provide higher performance, and this shares the same target as that of DrTM-H [34] and FaRM [14]. ScaleRPC steps further to provide scalable performance in the transaction processing.

6 Related Work

RDMA has long been used in high-performance computing (HPC) community to provide high aggregated I/O bandwidth, like in Lustre [10, 27], PVFS [36] and NFS [7]. RDMA optimizations have also been made for MPI implementations, like MPICH [21, 22] and OpenMPI [28]. Different from the HPC community, which focuses on achieving high I/O bandwidth, RDMA is getting widely used in recent in-memory storage systems, and the throughput and scalability become relatively more challengeable.

Improving the Scalability of RDMA. LITE [32] is a kernel abstraction for safe and scalable RDMA access in data centers. It directly registers physical memory to reduce the NIC cache thrashing. With bandwidth requirements, LITE only launches one thread to poll the completion of the posted verbs. We believe that the scalability of LITE will be further improved even for small-sized messages with optimizations like connection grouping and virtualized mapping equipped. Both FaSST [18] and HERD [16] propose to use unreliable datagram (UD) for inter-node communication. since UD supports "many-to-many" data transferring, the above two systems only need to create one QP for each working threads (instead of creating one for each connection), and thereby provide scalable performance. However, as described before, UD also faces many limitations.

General RDMA Optimizations. RDMA optimizations have been extensively explored from many perspectives. DrTM [9, 35] is a distributed transaction processing system which co-uses RDMA and Hardware Transactional Memory for high consistency and atomicity. FaRM [13] proposes a distributed computing platform, which provides the transactional interface for applications to access the shared memory. [17] provides guidelines on how to use RDMA efficiently from a low-level perspective. CQCN [38] proposes an end-to-end congestion control scheme for RoCE v2 to solve the fairness and performance issue.

RDMA in In-Memory Storage Systems. RDMA has been widely adopted in both distributed file systems and key-value stores to achieve high efficiency. Several key-value stores are carefully redesigned to work over RDMA-enabled network. Pilaf [24] improves the *get* performance by letting the clients directly post RDMA `read`. HERD enables both *get* and *put* operations to be processed at server side with the efficient design of RPC, which is a hybrid design of UD `write` and UD `send`. For distributed file systems, Crail [4] is a recently developed distributed file system built on DaRPC [30], which achieves good performance with both medadata and data communication. NVFS [15] provides a novel design of HDFS with byte-addressable NVM and RDMA network. Octopus [23] is a distributed shared persistent memory file system that combines the new features of RDMA and NVM by redesigning the software.

7 Conclusion

RDMA achieves low latency and high bandwidth by bypassing the operating system. In this paper, we notice that the *resource contentions* exist both in the NIC cache, CPU cache and memory of each server, which limits the scalability of RDMA network. To address this issue, we propose an efficient RPC primitive named ScaleRPC with internal mechanisms like *connection grouping* and *virtualized mapping*. We perform an extensive evaluation on ScaleRPC and build two distributed systems on it. Our experimental results show that ScaleRPC provides scalable performance while guaranteeing the reliable message transmission.

Acknowledgments

We sincerely thank our shepherd Dan R. K. Ports for helping us improve the paper. We also thank the anonymous reviewers for their feedback and suggestions. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011, 61772300), and Huawei Innovation Research Program (Grant No. HF2017070004).

References

[1] 2013. Mellanox Technologies. Connect-IB: Architecture for Scalable High Performance Computing. http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf.

[2] 2016. Processor Counter Monitor (PCM). "https://github.com/opcm/pcm".

[3] 2016. SAP HANA, In-memory computing and real time analytics. "http://go.sap.com/product/technology-platform/hana.html".

[4] 2017. Crail: A Fast Multi-tiered Distributed Direct Access File System. https://github.com/zrlio/crail.

[5] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The cost of serializability on platforms that use snapshot isolation. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 576–585.

[6] Kalia Anuj, Kaminsky Michael, and Andersen David. 2019. Datacenter RPCs can be General and Fast. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[7] Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach, and Omer Asad. 2003. NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI '03)*. ACM, 196–208.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[9] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 26.

[10] Sean Cochrane, K Kutzer, and L McIntosh. 2009. Solving the HPC I/O bottleneck: SunâĎć LustreâĎć storage system. *Sun BluePrintsâĎć Online, Sun Microsystems* (2009).

[11] Intel Corporation. 2012. Intel data direct I/O technology (Intel DDIO): A primer. "http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf".

[12] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[14] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[15] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. 2016. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM.

[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *SIGCOMM*.

[17] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*.

[18] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. 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)*. USENIX Association, 185–201.

[19] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*.

[20] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. *management* 15, 32 (2014), 36.

[21] Jiuxing Liu, Amith R Mamidala, and Dhabaleswar K Panda. 2004. Fast and scalable MPI-level broadcast using InfiniBand's hardware multicast support. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, 10.

[22] Jiuxing Liu, Jiesheng Wu, Sushmitha P Kini, Pete Wyckoff, and Dhabaleswar K Panda. 2003. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*. ACM, 295–304.

[23] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 773–785.

[24] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 103–114.

[25] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 7.

[26] Yufei Ren, Xingbo Wu, Li Zhang, Yandong Wang, Wei Zhang, Zijun Wang, Michel Hack, and Song Jiang. 2017. iRDMA: Efficient Use of RDMA in Distributed Deep Learning Systems. *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2017), 231–238.

[27] Galen Shipman, David Dillow, Sarp Oral, Feiyi Wang, Douglas Fuller, Jason Hill, and Zhe Zhang. 2010. Lessons learned in deploying the worldâĂŹs largest scale lustre file system. In *The 52nd Cray user group conference*.

[28] Galen M Shipman, Timothy S Woodall, Richard L Graham, Arthur B Maccabe, and Patrick G Bridges. 2006. Infiniband scalability in Open MPI. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.

[29] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), 1–10.

[30] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. 2014. DaRPC: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 1–13.

[31] Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh, Sourav Chakraborty, and Dhabaleswar K Panda. 2014. Designing MPI library with dynamic connected transport (DCT) of InfiniBand: early experiences. In *International Supercomputing Conference*. Springer, 278–295.

[32] Shin-Yeh Tsai and Yiying Zhang. 2017. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 306–324.

[33] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 22.

[34] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 233–251.

[35] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 87–104.

[36] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. 2003. PVFS over InfiniBand: Design and performance evaluation. In *Proceedings of the 2003 International Conference on Parallel Processing*. IEEE, 125–132.

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[38] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 523–536.