Toward Deploying Scalable and Effective RDMA as a Service at End-hosts through Reliable Connection

Anonymous
Paper ID: 1570477397

***Abstract*—Due to the high throughput, low latency and low CPU overhead, RDMA has recently attracted a lot of attention on building the in-memory computing applications. However, it is not trivial to quickly migrate current system to the new network environment. With the increase of connections the customer may see the degraded performance. The barrier is that the scalability and effectiveness of RDMA-based application is highly related to many low-level details of RDMA operations. To address this problem, we present a simple and scalable RDMA as Service (RaaS) to mitigate the impact of RDMA operational details. RaaS provides careful message buffer management to improve CPU/memory utilization, improve the scalability and maintain the performance of RDMA operations through a lock-free resource sharing approach. These optimized designs lead to simple and flexible programming model for customers. We have implemented a prototype of RaaS, and evaluated its performance on a cluster with a large number of connections. Our experiment results demonstrate that RaaS achieves high throughput for thousand of connections and maintains low CPU and memory overhead for many applications.**

I. Introduction

Remote Direct Memory Access (RDMA) technique provides the messaging service that directly access the memory on remote machines. Since data can be copied by the network interface cards (NICs), RDMA provides minimal operating system involvement and achieves low latency data transmission. The technique of RDMA has been widely used by the High Performance Computing (HPC) community through the support of InfiniBand (IB) network [1], which is not compatible with the Ethernet and requires the support of specific hardware.

Recently, with the compatible design of RDMA over Ethernet (RoCE) and the decreasing price of RDMA hardware, the modern data centers have been deploying RDMA for the distributed computing platforms to alleviate the communication bottleneck of the TCP/IP network [2], [3]. To make RDMA network scalable to support hundreds of thousands of nodes in the data center, the protocol of routable RoCE (RoCEv2) [4] was defined and deployed in the modern data center [3], [5]. By leveraging the advanced techniques of lossless network design [6], [7] and quantized congestion notification (QCN) [3], it has indicated the potential of RDMA by replacing TCP for intra-data center communication [5].

Rich transport modes and operations have been provided by the native RDMA programming interface, called *verbs*. Different from the TCP/IP stack, in RDMA communication, a consumer queues up a series of service requests to be executed by the hardware, i.e., RDMA NIC (RNIC). When the consumer submits a work request (WR), an instruction called Work Queue Element (WQE) is placed in the work queue, called a Queue Pair (QP): one for send operations and one for receive operations. RDMA NIC executes WQEs in order without involving the kernel. When finishing a request, a Completion Queue Element (CQE) is placed in the completion queue (CQ) [8]. Two kinds of transmission semantics can be applied for WQEs. One is SEND/RECV verb called channel semantic, and the other is READ/WRITE verb called memory semantic.

Compared with TCP/IP, RDMA exposes a wider interface to applications with many parameters. Attaining the benefits of RDMA is not always straightforward. The best-fit RDMA-aware system design for a specific application requires careful parameters turning, which highly relies on the low-level knowledge associated with the NIC architecture, memory management, transport semantics, etc. [9]–[13]. With regard to the various demands of applications and the shared infrastructure, applying the native RDMA may not be a wise move for the application-layer developers with little knowledge about RDMA [9], [10], [12]–[14]. In general, the developers need to overcome the following challenging issues when migrating applications to the RDMA-aware systems.

The first challenge is the scalability. Scalability is a practical concern for the distributed system running in data center environment, where a large number of connections are generated in one machine. Due to the limited cache space, RNIC can not store the metadata of all the active connections simultaneously. The current connection-oriented transport which provides exclusive access to send/receive queues. Thus, the information will be exchanged between RNIC cache and system memory across the PCIe bus when a large number of connections are active at the same time.

The scalability issue for RDMA-based system is not new which has been pointed out in many works [10], [13], [15]. Queue Pairs (QPs) sharing is a common solution to address this problem [16]. Because threads need to contend for the shared resource, it may reduce CPU efficiency which reduces the per-core throughput of reliable transport by up to 5.4x according to the test given in [13]. Instead of using reliable connection, for small messages, some works like key-value store [17] and RPC [13] exploited to apply the unreliable datagram transport, which allows to use only one datagram QP to handle the work requests from all remote nodes. However, the two-side operations supported by the unreliable transport require the involvement of CPUs at both sender and receiver

sides. To address this challenge, a more efficient lock-free design is provided in our system to achieve high scalability of reliable transport but maintain the low CPU overhead of reliable transport.

The second challenge is the low utilization of resource. When considering the shared infrastructure environment, the Recv operation needs to post receive work requests (WRs) beforehand to handle incoming messages. Holding a lot of work queue elements (WQEs) in each receive queue may result in the waste of memory space or wire usage. Besides, data sink consumer may not be aware of the starvation of RQ. The corresponding polling thread leads to the waste of CPU resource [10]. The shared receive queue (SRQ) model partially solves this problem which posts receive WRs to a queue that is shared by a set of connections. However, under the native RDMA consumer queuing model, each application maintains and manages its own RDMA queues. Our key observation is that the resource such as SRQs can be further shared among multiple applications that are running on the same physical machine. To this end, we provide careful memory management among transport connections of consumers that achieves an high utilization of memory usage, low CPUs overhead for the low latency, and high throughput RDMA network service.

The third challenge is the operational availability. Currently, the available RDMA network functionality is tightly coupled with the targeting system. The basic units of RDMA network functionality, e.g., RDMA enabled buffer management, data structure and polling thread, usually work as modules embedded in the system. The coupled design makes it hard to reuse the code for other applications. In addition, since the low-level details are important for RDMA system design [12], the application performance is highly determined by the choice of RDMA operations.

Unfortunately there is no one-size-fits-all approach. The optimal setting of a given system can be the poison to other ones. It is not easy to train the system designers to master all the low-level details of RDMA technologies and understand the guidelines [12], [18] under various conditions in a short time, which hinders the fast development of RDMA-based system in data centers. Thus, it is necessary to ensure that the application is not sensitive to the RDMA network operations by introducing a shim layer to mitigate the influence of low-level details. To address this challenge, in our design the flexible RDMA network functions are abstracted by Socket-like network interfaces. All the conditional sections in the program are described with parameters. Generally, the low-level details are not exposed and the performance is guaranteed by adaptive RDMA primitive selection in the shim layer. For any specific requirements on RDMA operation, their demands can be realized with customized setting through macro arguments.

In summary, the contribution of this paper is described as follows. First, we present a simple and scalable *RDMA as a Service (*`RaaS`*)* for fast deploying RDMA-capable services in the data center. The interface semantics are fairly straightforward and friendly to developers. Developers only need to determine the usage of connection-oriented transport or datagram transport for their services, since `RaaS` has mitigated the impact of low-level details. The good performance is achieved by improving the scalability and resource utilization of reliable connection-oriented RDMA transport. Second, we have implemented a `RaaS` prototype, and evaluated the performance in a cluster running a large number of connections. `RaaS` shows more efficient usage of memory and CPUs, and achieves high throughput for thousands of connections. We believe that this work can serve as a general low-layer transportation for storage [9], [17], RPC [13] and other applications [10], [19]. This work is also a good starting point to start thinking about how to share the RDMA network by many applications.

The rest of this paper is organized as follows. Section II explains the background of RDMA to be used in the paper. Section III and Section V describe the design and implementation of scalable `RaaS`, which include the selection of communication primitives, resource management, the lock-free design of `RaaS` and the programming model and optimization. In Section VI we evaluated the performance of `RaaS` using micro-benchmark and multiple memory-intensive applications. Section VII introduces the related works. Finially, Section VIII concludes this paper.

II. Background on RDMA

RDMA allows user-space applications to directly read or write remote memory without the involvement of CPU, kernel of either host. RDMA NICs provides asynchronous network programming interfaces, *RDMA verbs*, to application for submitting work requests to the adapter and returning the completion status. The queue used to hold these services requests is referred to as a Work Queue (WQ). Work queue including send and receive queues is always created in pairs, called a Queue Pair (QP).

For communication, the QP needs to be initialized and associated with one completion queue (CQ) on both sides of the connection. The application must register a memory region with the NIC for local and remote access. The registration process pins the memory pages to prevent the pages from being swapped out and writes the key as well as the virtual to physical address table in the network adapter. When the consumer submits a work request (WR), an instruction named Work Queue Element (WQE) is placed in the work queue. Without involving the kernel, RDMA NIC executes WQEs uses DMA to access the right memory. When finishing a request, a Completion Queue Element (CQE) is placed in the completion queue (CQ) associated with the QP. At the sender side, applications poll the CQ to get the CQE and check the completion status of the WQE. At the receiver side, applications poll the CQ to get the CQE and receive the data.

In the RDMA verbs, both channel semantics and memory semantics are provided to users. The channel semantics are two-sided operations, called *SEND/RECV*, which is similar to the traditional TCP/IP in I/O channel. The memory verbs are one-sided operations, embedded in the *READ, WRITE*

TABLE I
OPERATIONS AND MAXIMUM MESSAGE SIZE SUPPORTED BY EACH TRANSPORT TYPE

	SEND/RECV	WRITE	READ	Max Message
RC	✓	✓	✓	1GB
UC	✓	✓	✗	1GB
UD	✓	✗	✗	MTU

TABLE II
TESTBED SETTING

Component	Configuration
CPU	4x Xeon E5-2650v2 (24 cores, 2.1 Ghz)
DRAM	8x 8GB DDR-3
RNIC	Mellanox ConnectX-3, 40 Gbps, RoCE
OS	CentOS 7.1

as well as *WRITE-with-immediate-data* and *ATOMICs* operations, which allow the initiator of the transfer to specify the source or destination location without the involvement of the other endpoint CPUs.

Three transport types are provided in current RDMA implementation: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). Reliable transport guarantees lossless transfer and ensures in-order delivery of packets by using packet acknowledgment. The unreliable transport, providing no sequence guarantee, consumes less bandwidth due to no ACK/NACK packets. It is notable that connection-oriented transport, i.e., RC and UC, require one-to-one QP. In the datagram transport, i.e., UD, one QP can communicate with multiple QPs. Table I shows the operations available in each transport mode. Specifically, it is notable that RC and UC support message size up to 1GB. The message is divided into MTU-sized frames during transmission on Ethernet. UD supports the maximum size of a message up to MTU size.

III. RDMA AS A SERVICE (RAAS)

This section explains the design and implementation of `RaaS`. First, we describe the scalability and effectiveness issues when using different transport types in the shared environment. Then we describe the resource management and optimization approach of `RaaS`. The details of all components in `RaaS`, the programming model and the implementation are introduced which are evaluated in the next section.

A. Communication Primitives

We perform our primary experiments on two machines with 40Gbps RDMA NICs, which are connected to an Ethernet switch. We evaluate the performance of different RDMA transport options while avoid the impact of multiple paths and hops on the network. The testbed setting is shown in Table II.

One-side verbs (READ/WRITE) bypass the remote CPU to operate directly on remote memory. This advanced feature is a popular option for data transfer in the distributed system, because it releases the CPUs resource for the computation services. As shown in Figure 1, RC Write performs as well as UC Write. The throughput of RC READ is close to RC

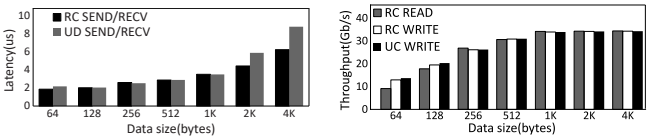

Fig. 1. Comparision of RDMA operations

write for large message. Therefore we use RC as the default RDMA transport instead of UC when users decide to use the connection-oriented transport. The selection of RC READ and WRITE can be adaptively adjusted based on the current CPU and memory consumption of servers (see Section V-A).

Unreliable Datagram (UD) has good scalability, since the connectionless service allows to use one QP to communicate with multiple remote QPs. UD operates with less state maintained at each end-host, which is suitable for latency-sensitive applications with small messages, such as key-value store [17] or RPC service [13]. One drawback of UD SEND/RECV operations is that they require the involvement of CPU at both sender and receiver sides. Moreover, since the maximum message size is constrained by MTU, a large UD message will be conveyed by multiple packets, which is not suitable for throughput-sensitive applications.

B. Resource Management Model

To address the scalability issue while maintain good performance, the main idea is to share RC QPs but avoid the usage of lock for accessing the shared resource. We first introduce three components related to our design: the management of registered buffer, the sharing of SRQ and the lock-free design of QPs sharing.

Different from the the TCP/IP network, the buffer is explicitly managed by the application for the RDMA network. To process the RDMA requests, a memory region need to be registered with the NIC before it can be made available for remote access. Memory region registration avoid the expensive data copy operations. However, the memory management leads to frequent makes application unable to reuse the buffers. Therefore, some existing works suggest to on-load the memory management to the operating system [15].

Another operation is to When preparing the sending buffer, applications first register the data buffer, send the data to network and release the buffer after completion of operations on this buffer. Or applications can pre-register a free buffer and copy the data to the pre-registered buffer when sending data to network and reuse the registered buffer after completion of operations on this buffer. The balance point here is the performance of memory copying and memory registration that both are time-consuming operations. In our benckmark and the previous work [20], memory registration is better than memory copying when the buffer size is over 128KB, but memory copying is better than memory registration when the buffer size is below 128KB. For modern data center applications, small data (below 50KB) dominates the workload [21]. Pre-

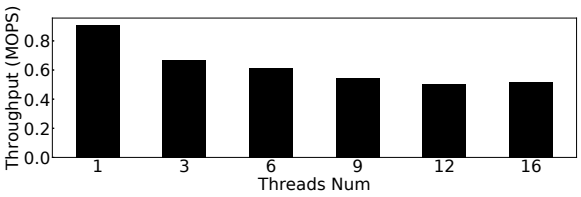

Fig. 2. Performance Degration Caused by Lock Contention

registering free buffers and copying data to the buffers when sending or receiving data is optimal.

According to this result, RaaS copys data to the pre-registered buffers when the data size is smaller than 128KB and registers data buffer otherwise. It is worth noting that small data (below 50KB) dominates the workload [21] in modern data center. It indicates the advantage of buffer management in RaaS.

RC transport provides all kinds of operations and reliable transmission but the scalability bottleneck exists due to that RNIC can not store the metadata of all the active reliable connections simultaneously. If the accessing information is not cached in the RNIC, RNIC needs to fetch the desired data in memory. This process slows down the packets processing of RDMA transport. Dynamically Connected Transport (DCT) has been proposed as a hardware solution to the scalability problem [1]. DCT reduces RNICs cache miss by dynamically creating and destroying one-to-one connections. However it requires three additional network messages when the target machine of a DCT queue pair changes: a disconnect packet to the current machine, and a two-way handshake with the next machine to establish a connection [22]. DCT increases the number of packets associated with each RDMA request [13].

Several studies have considered the resource sharing to mitigate the impact of limited RNIC cache [10], [13], [15], [23], [24]. FaRM [10] and LITE [15] applied lock mechanism on the shared resources (like QPs). FaSST [13] has demonstrated that with the increase of the number of threads, the scalability bottleneck lies in the mutex_lock. We also ran a micro-benchmark to show the degraded performance due to lock contention (more details in Section VI). As shown in Figure 2, when the number of threads sharing the same QP is 3, the throughput drops to 66% of the result without sharing. And when the number increases to 9, we observe only 56% throughput left. Therefore such synchronous QP-sharing is not scalable with the increase of connections. We need to design a new model to eliminate the bottleneck.

When receiving data from remote QPs, applications can post WRs to each RQ of QPs and poll each associated CQ to receive data separately. Or applications can use one SRQ to receive data from part or all of remote QPs held by counterpart applications. The SRQ is provided by verbs for one consideration that the receiver applications cannot predict the incoming rate on a given RQ. Posting sufficient WRs on each RQ to hold the possible incoming rate wastes

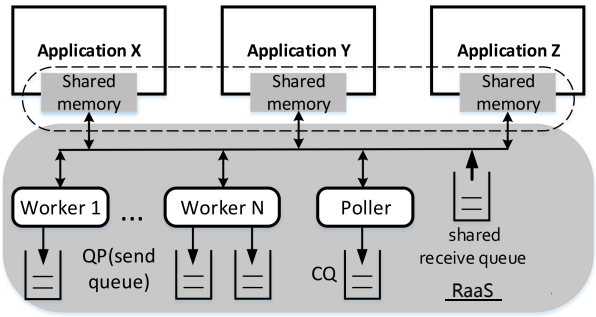

Fig. 3. Overview of RaaS

WRs. Lacking WRs on each RQ, the backoff mechanism will be triggered at sender side and unnecessary latencies will be incurred. Though the SRQ reduces resource consumption and mitigates the starvation of the RQ, the standard SRQ provides by verbs is applications isolated. While hundreds of applications exist in data center, it still consumes huge volume of resources. Under this circumstances, multiplexing the SRQ among multiple applications is a promising method.

IV. Architecture and System Design

We propose a model called RaaS to manage low-level shared resources like QP, CQ, SRQ. As illustrated in Figure 3, each application shares a memory with RaaS for traffic delivering. RaaS handles applications' traffic in three aspects: connection establishment, traffic processing and traffic receiving. For connection establishment, applications establish connections through RaaS. *Connections to the same remote node share the same resources (QPs and the dedicated thread named* Worker*).* After connections are set up, applications can send or receive data on the connections like using sockets. When sending data, corresponding WRs are pushed into each connection. In traffic processing, *Worker* checks the WRs in connections and posts WRs into the corresponding QPs. The completion statuses of WRs are checked by a dedicated thread named *Poller*. When receiving traffic, *Poller* polls the CQ associated with the SRQ and delivers the received data to right connections.

A. Connection Establishment and Resource Allocation

In connection establishment, RaaS needs to differentiate connections sharing the same QP and allocates the registered shared buffer used by each connection. In the shared environment, the connection is not identified by the QP. Instead, RaaS maintains an unique identifier (vQPN) for each connection. The identifier is placed in WRs' `imm_data` domain. After connection is setup, each peer of the connection holds each other's identifier, based on which RaaS is able to forward the arrival traffic to the right remote connection. For the optimal performance of data delivering and decoupling applications from low-level details, RaaS allocates, pre-registers and manages a region of shared memory for each connection. Applications can directly call the provided interface to copy small data to the shared memory and notify the RaaS for

the data delivering. When applications call the interface to transfer large data, RaaS automatically registers the data buffer in both side for transmission and deregisters the buffer after the transmission, hiding the details from applications.

B. Asynchronous Lock-Free QP-sharing in Traffic Processing

In traffic processing, RaaS need to quickly find the connection that is requesting for the data delivering and process the data on the shared memory. The problem is that how does the *Worker* quickly find the requesting connection, especially when the total number of connections is large. For achieving this, when generating a new connection, a pair of unix socket file descriptors and an associated egress queue are also allocated, stored in the connection. One of the unix socket file descriptor pair is returned to the application, the other unix socket file descriptor is registered into the file descriptor list of the associated *Worker*. The egress queue is used to store WRs referring to data on the shared memory. After pushing WRs into the egress queue, the application writes notification to the unix socket file descriptor. *Worker* captures the notification occurred on the unix socket file descriptors through epoll function provided by the Linux kernel, dequeues the WRs in the egress queue and posts WRs into the shared QP. In synchronous QP-sharing model, application threads need to acquire the lock guard for the shared QP when posting WRs. We apply the egress queues of the multiplexer layer of RaaS to decouple the application request and QPs forwarding. This design constructs an asynchronous QP-sharing model. The overhead caused by lock contending in synchronous QP-sharing model can be eliminated by the new model. In addition, the epoll used by one *Worker* can listen up to 65536 file descriptors. And its events report time complexity is O(1).

C. Multiplexing SRQ in Traffic Receiving

To further reduce the resource consumption, RaaS shares the SRQ among all applications on the same physical machine. As the application puts the unique identifier of connection into the WRs' imm_data domain, the unique identifier will be generated in the CQEs' imm_data domain when the SRQ at receiver has received the data. Thus, the receiver polls the CQEs from the associated CQ and identifies the arriving data based on the unique identifier in the filed of imm_data. Then the *Poller* pushes the received CQEs to the corresponding ingress queues of the connections. Finally, the *Poller* will notify the corresponding application that the data has been successfully received and ready to deliver to high level for further processing. The notification is also accomplished by the unix socket file descriptor mentioned above. It's worthy noting that the SRQ is filled with receiving WQEs referring to same-sized free buffers. The size of receiving buffer is equal to the sending buffer as requested by verbs. Extra traffic spliting and assembling is necessary when the data size is slightly large than the receiving buffer size. But it's simple and only requires some offset computation in both sender and receiver.

```
int connect(Target* t, int FLAGS);
int listen(Target* t, int FLAGS);
int accept(int fd, int FLAGS);
uint32_t send(int fd, void* buf, uint32_t len, int FLAGS);
uint32_t recv(int fd, void* buf, uint32_t len, int FLAGS);
uint32_t recv_zero_copy(int fd, void** buf_addr, uint32_t len, int FLAGS);
uint32_t sendto(int fd, void* buf, uint32_t len, Target* t, int FLAGS);
uint32_t recvfrom(int fd, void* buf, uint32_t len, Target* t, int FLAGS);
uint32_t recvfrom_zero_copy(int fd, void** buf_addr, uint32_t len, int FLAGS);
```

Fig. 4. RaaS's API

V. Implementation

A. Programming Model

RaaS supports simple interfaces to invoke RDMA functions by hiding the complicate details. Figure 4 shows the defined operations. They are socket-like operations which are friendly to users. The first three interfaces connection, listen, accept are used to set up connections. Here Addr is an unified encapsulation of host address, including IPv4, and IPv6. The default value of FLAGS means RC operations in our design. Users can use connect to initiate active connections and accept to passively wait for new incoming connections. The field of fd is used to specify on which logical connection the interface needs to be invoked. fd is the unix socket file descriptor mentioned in IV-B, which is a lite inter process communication (IPC).

Normal users can simply use send, recv operations for connection-oriented data transport, and sendto, recvfrom operations for datagram transport[1]. RaaS will adaptively select RDMA SEND/RECV for data block of small size and RDMA READ/WRITE operations for large data according to the explicitly indicated data size in the parameter of 'len'. Specifically, to reduce latency, RaaS chooses one-side verbs based on the current CPU consumption and work load, which is easily measured by the RaaS daemon process. It is notable that a parameter called FLAGS is introduced in the RaaS's interface. FLAGS is used to specific RDMA transport for one user with any special requirement on RDMA operations, e.g., $RC|WRITE$ or $Atomic$. This design provides the flexibility to help knowledgeable users to achieve customized setting.

We use send for outbound data transmission. Both recv and recv_zero_copy are used for receiving incoming data. send and recv work in the non-blocking model. The interface of recv_zero_copy is provided for application which works in the blocking mode. recv_zero_copy interface is popular in high performance system. Since in the blocking mode, it may take longer time to pick the data from the receive buffer of RDMA, recv_zero_copy will deliver the incoming data directly from receiving buffers to the pre-registered private memory of the application, instead of occupying the shared memory of RaaS.

We do not provide send_zero_copy due to the revelation of related work [20]. The reason is as follows. For sending

[1]Here we only explain the operations for connection-oriented transport due to the limited space. The operations for datagram transport are similar.

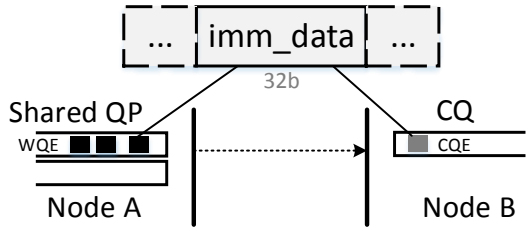

Fig. 5. vQPN in the imm_data field of WQE/CQE

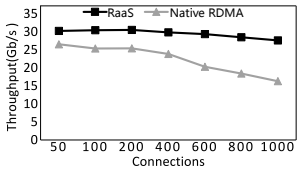

Fig. 6. Scalability

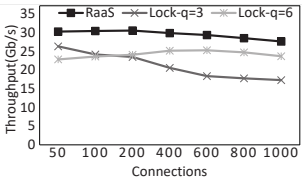

Fig. 7. Throughput vs. QPs sharing

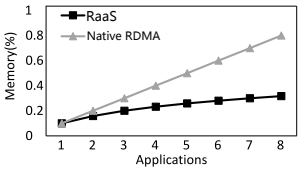

Fig. 8. Memory usage

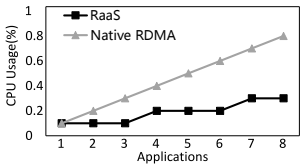

Fig. 9. CPU consumption

tasks, RNICs must get the physical address of outbound data to initiate a DMA operation. The memory region in which outbound data lying must be registered to RNICs beforehand. One way is to copy applications' outbound data to pre-registered memory region, called $memcpy$. Another approach is to register the memory region of applications' outbound data to RNICs every time, called $memreg$. However, in the case of relative small size of outbound data, $memcpy$ achieves better latency, but $memreg$ outperforms $memcpy$ when the size of outbound data increases. Therefore, `send_zero_copy` may be inefficient in sending the relatively big size messages and `RaaS` is going to handle this according to the previous work [20].

B. Synchronization for RDMA READ/WRITE

It is notable, however, that for one side verbs like RDMA READ/WRITE, the destination node is bypassed so that the RNIC at the destination is unable to generate CQEs to indicate the ending of transmission. FaRM [10] used a zeroed memory region as indicator of incoming RDMA Write, and applied a thread to continuously checking the status of the indicator. After processing the incoming data, the indicator will be reset. Since both sides of the communication should carefully synchronize several pointers, this design is complicated and hard to maintain.

To this end, `RaaS` generates an additional small control message to pass the synchronization information from the initiator to the target node. Since RDMA READ or WRITE are used to deliver large message (usually over 1MB), the overhead of such additional control message is negligible. Specifically, this process can be further simplified for RDMA Write operation by applying the `RDMA_Write_with_immediate_data` which allows transfer of another piece of status information simultaneously with the message and returns as a special field of the RECEIVE CQE status. Thus, applying the RDMA Write with immediate data operation can avoid issuing additional control messages.

C. Remote Atomic Operations

RDMA verbs also supports remote atomic operations like `CMP_AND_SWAP` and `FETCH_AND_ADD`, which is widely used for distributed transaction processing [14], [25]. However, in our practical and studies in [18] revealed that RDMA verbs remote atomic operations are not efficient. The poor performance of atomic operations occurs due to internal lock in RNICs. RNICs have multiple Processing Units (PUs) and the atomic operations always harm the parallelism, which produces a bad influence on the overall performance.

To solve this problem, we provide more efficient remote atomic operations by moving the atomic operations from RNICs to `RaaS`. `RaaS` takes advantage of C++11 library, which supports atomic_uint_least64_t type of data. Objects of atomic types are the C++ objects that are free from data races. Therefore, `RaaS` atomic operation only needs to create a map with unsigned 64 bits address as key and objects of atomic_uint_least64_t as value. By so doing, client can initiate SEND operations with 64 bits address in payload and the server will get the specific objects of atomic_uint_least64_t according to the address given in the payload. To this end, `RaaS` not only removes lock contention in RNICs but also ensure lock free in the whole data path of the remote atomic operations.

D. Reducing CPU Resources and Latency

Finally, we briefly introduce the method to save the CPU resource in our model. All threads in `RaaS` including *Worker*, *Poller* and the reloading SRQ thread are designed to work in blocking-interrupted-running mode. If there is no task (such as sending WRs to QPs, polling CQEs from CQ, reloading receive WRs to SRQ), threads will proactively block and wait for being waked up by interruption of new events. `RaaS` uses the polling for new tasks until there is no active task for a fixed interval. If so, those threads will block and wait for interruption events to avoid wasting CPU resources. Since in heavy load environment, *Worker* will keep processing WRs of connections and *Poller* will keep busy polling the CQ to get CQEs, low latency is guaranteed here.

VI. Evaluation

We evaluate the performance of `RaaS` on a cluster with four nodes. Each node ran CentOS 7.3 on four 2.1 GHz Intel Xeon processors with a total of 24 cores. Each machine has

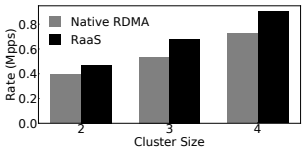

Fig. 10. Scalability in Cluster

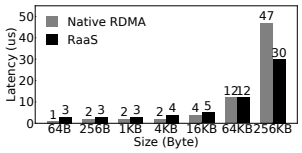

Fig. 11. Latency Comparision

64 GB of memory and a Mellanox ConnectX-3 Pro 40 Gb RoCE network interface card. All machines are connected to a Mellanox MSN 2100 40 Gb Ethernet Switch.

A. Micro Benchmark

To evaluate the scalability of RaaS design, up to 1000 connections were generated on one machine to randomly read 64KB data from other machines. We compare the RaaS Read operations with the native RDMA READ verbs where the QPs are not shared by those connections. As shown in Figure 6, the throughput of native RDMA started to drop when the size of connections, i.e., QPs, exceeded 400. RaaS shows stable performance since RaaS allows the reuse of QPs to communication with multiple QPs in remote nodes. It is notable that RaaS outperformed the native RDMA when the number of connections was even less than 400. This is because the reuse of QPs has higher opportunity of batching WRs than the case of one QP per thread. With the increase of connections, RaaS has a higher probability to realize QPs sharing. Next we adjusted the cluster size, established 192 connections to each node and measured the request rate. As shown in Figure 10, RaaS outperformed than native RDMA no matter what size of cluster and number of connections are. This was also due to the QP sharing and batching processing of WRs in RaaS.

We compare RaaS with the design of sharing QPs with locks [10], where each QP is shared by q threads. Figure 7 shows results when q is 3 and 6 by running random read. We can see that sharing QPs can reduce throughput when threads contend for locks. RaaS can mitigate the impact of lock contention. Specifically, our design is not sensitive to the number of q and cluster size since RaaS performance scales well with the connection number.

We compare the RaaS resource (memory and CPU) consumption with native RDMA. The resource consumption is normalized according to the resource required by one application to set up the connection. Figure 8 shows the memory consumption and Figure 9 shows the CPUs overhead respectively. We can see that the resource required by the native RDMA operations increases linearly with the increase of applications. However, the design of RaaS proposes more effective usage of resource which results in a slow growth of memory/CPU consumption.

Then we show the latency comparision between native RDMA and RaaS. For removing the batching effect of WRs, we let the native RDMA and RaaS initiate only one request each time, wait for the completion and compute the latency.

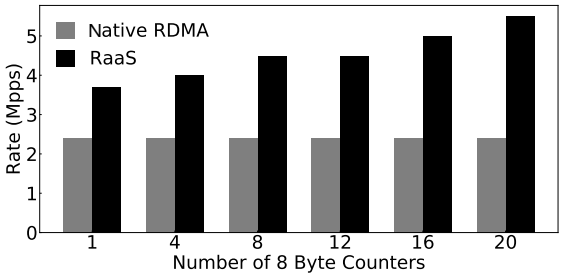

Fig. 12. Atomic operations

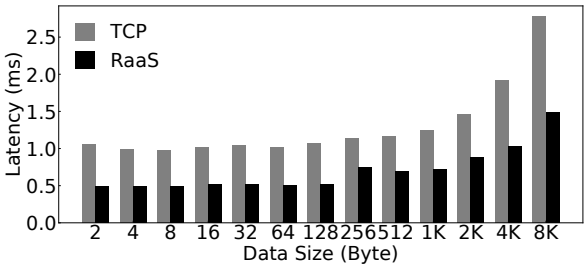

Fig. 13. Latency of gRPC on TCP and RaaS

Figure 11 shows the result. When the sending data size was smaller than 64KB, RaaS only introduced about 1-2 us. This overhead vanished when the sending data size is 64KB. It's notable that RaaS cost 37% less latency when the sending data size is 256KB. The reason was that pipeline happened between applications and RaaS when applications split data and continuously delivered to RaaS. Therefore RaaS can guarantee the low latency.

Figure 12 shows the result of remote atomic operations implemented by RDMA verbs and RaaS. We create 1 to 20 8-byte counters at server side and generate parallel atomic operations at client side. When using the native verbs atomic operations, it achieves 2.4 million operations per second. While RaaS can achieve up to 3.7-5.5 million operations per second with regard to different counters, which verified the effectiveness of our remote atomic operations.

B. Macro benchmark

We have implemented and deployed RaaS on an RDMA cluster to user applications and evaluate its effectiveness using multiple workloads running on gRPC, ps-lite, MXNet and memcached. Using RaaS, latency of these applications reduce by 50% to 70% over TCP/IP.

We used RaaS to accelerate two opensource network communication library, gRPC and ps-lite. We can either simply replace the original Socket APIs with RaaS's Socket-like APIs or rewrite the underly communication module with our APIs.

We replaced the Socket APIs in gRPC [26] with our RaaS APIs and sent the increasing size of data from one machine to another. Figure 13 shows the latency of TCP-based and RaaS-based gRPC. When the data size increases from 2B to 8K, RaaS-based gRPC is up to 1300us faster than TCP-based

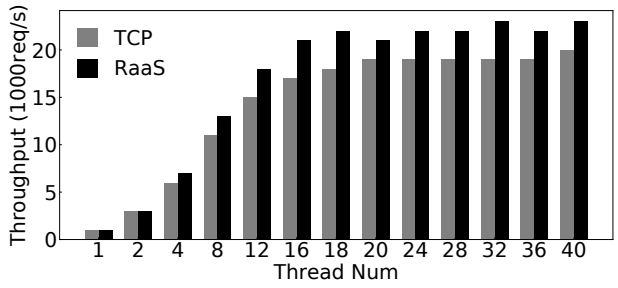

Fig. 14. Increased throughput of gRPC on RaaS

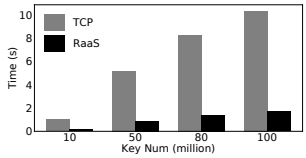

Fig. 15. 1 worker, 1 server

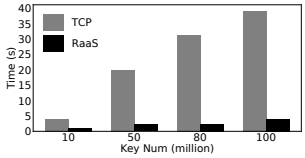

Fig. 16. 4 workers, 4 servers

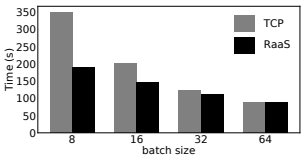

Fig. 17. MNIST training time in MXNet with different batch sizes

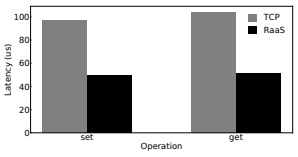

Fig. 18. Latency in memcached

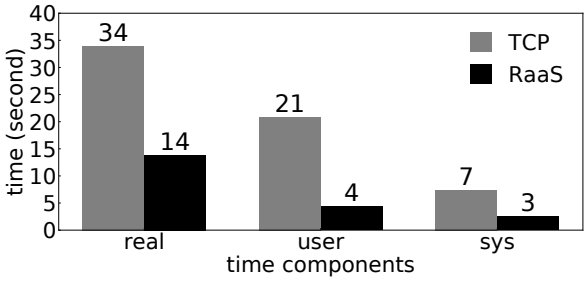

Fig. 19. Latency of migrating database

gRPC. Figure 14 shows the throughput of gRPC when the threads of clients increasing. Each thread sends 8 bytes data and the server echoes this 8 bytes to client thread. As the client threads increase, the throughput of gRPC with RaaS is stably about 3000req/s larger than TCP.

We also evaluated RaaS in MXNet. MXNet is a flexible and efficient library for deep learning. It supports distributed training based on ps-lite. We replaced the network transporting module in ps-lite with RaaS. We repeated requesting 1000 key-value pairs for 1000 rounds in different worker and server numbers. Figure 15 and Figure 16 shows that RaaS outperforms TCP/IP by $4\times$-$13\times$ as the total number of key-value pairs increasing. We can also see that with more workers and servers, the total latency of TCP-based ps-lite decreases because of contending and communication between threads. But RaaS maintains a low latency in all cases, which means that RaaS has better scalability than original TCP/IP.

From the former two cases, we can see that rewriting the network transporting module performs much better than simply replacing Socket APIs in original applications. Much upper logic for TCP/IP in origin applications is unnecessary in our RaaS's transport. As we can see, the latency of ps-lite reduces up to 70% while gRPC only reduces less than 50%. We recommend construct a new transport module based on RaaS, not only for better performance, but also less modification to original codes of applications.

Then we run the classic MNIST training with 60000 samples of 10 classifications in MXNet with modified ps-lite. Figure 17 shows that RaaS outperforms TCP in small batch sizes. TCP and RaaS come close when batch size increases. Because large size of batch reduces the communication between worker and server, which weakens the advantage of low latency in RaaS.

We used our RaaS in memcached. Memcached is a free, open source, high-performance and distributed memory object caching system. We replaced the Socket APIs with our RaaS APIs in memcached. We tested the latency of setting and getting a 8-byte object. Figure 18 shows that modified memcached only has 50% percent of the original memcached latency for both setting and getting operations.

We used RaaS to speedup a database migration procedure by using XtraBackup [27], which is a complete online backup solution for MySQL. The size of our migrating database is about 1.3GB. RaaS enables fast development of service by simply replacing the traditional TCP Socket interface. Therefore, we only show baseline (TCP) results. We used *time* command in Linux to show the total time taken to run the application, the amount of time spent in user mode and kernel mode. As shown in Figure 19, RaaS reduces the migration period by 60%. RaaS reduces much time in user mode because it omits the copy and preparation process of the meta data for kernel. Figure 20 shows that RaaS involves less CPU cycles by around 40% due to the introduction of one-side RDMA write operation. The long delay is mainly caused by snapshot generation of XtraBackup instead of network transmission. It is notable that a short period of high CPU cost at time slot $[10, 11]$ happens because RaaS needs to deregister the memory region allocated for RNIC when the database migration accomplished.

In conclusion, with our easy-to-use RaaS APIs, users only need to make a little modification of their origin programs to achieve the advantages of RDMA and do not take much time on the details of underlying RDMA.

VII. Related Work

RDMA has received a lot of attention in recent years to deal with the growing application demand of ultra-low latency, very high bandwidth. The current usage of RDMA is primarily limited to optimizing individual applications, such

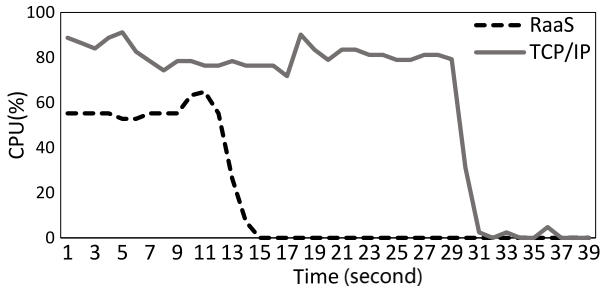

Fig. 20. CPU utilization of migrating database

as the in-memory key-value store, [17], [28], [29], large-scale graph-computing [2], and distributed computing system [10], [30], [31]. Sending the same data from machine A to machine B can be accomplished by any of READ, WRITE, or SEND/RECEIVE operations over three different transport types, each combination with its own advantages, drawbacks, and constraints [12].

Due to the limited cache space on the RNIC, frequent cache misses occur when a large number of active connections are accessed [10], [12], [13]. Several approaches have been proposed to improving the scalability of RDMA-based systems. FaRM [10] applied the large page (2GB) and QP sharing to overcome the constraint of limited cache space in RNIC. FaSST [13] applied UD (unreliable datagram) for resource sharing to improve the network I/O performance. LITE [15] allows applications to safely share resources in the Linux kernel. For reliable communication (RC), FaRM and LITE applied the synchronous mode of resource sharing where the lock can be the performance bottleneck.

It is worth noting that previous resource sharing methods are ineffective. At the sender side, one send queue is shared by multiple application threads [10], [12], [15], and these threads need to contend for the lock that protects the send queue. When the number of threads increases, the performance (such as throughput) declines. At the receiver side, application threads need to frequently post free Work Queue Elements to each Queue Pairs, which wastes the memory space. The Shared Receive Queue (SRQ) partially solves this problem. However, SRQs are application isolated which can be shared by multiple applications.

There are several RDMA-based user-level libraries including the standard OFED library [32], rdma-cm [33], Rsockets [34] and Accelio [35]. The standard OFED library provides the base function of RDMA. The rdma-cm handles part of details of RDMA. Rsockets implements a socket-like abstractions. Accelio supports full parallel and multi-thread operations for applications by allocating separate hardware resources (QPs and CQs) and event loops for each thread. These libraries can not provide both friendly abstraction and scalability at the same time.

`RaaS` can provide flexible transport types and operations for these applications while mitigating the scalability problem by sharing the resource among multiple applications. `RaaS` handles all resource and performance details of RDMA, and provides friendly interface for these applications that only need to leverage the advantage of RDMA.

VIII. Conclusion

In this paper we describe the design and implementation of a simple and scalable RDMA as Service (RaaS) to mitigate the impact of RDMA operational details. RaaS provides careful message buffer management to improve CPU/memory utilization and improve the scalability of RDMA operations. These optimized designs lead to simple and flexible programming model for common and knowledgeable users. We have implemented a prototype of RaaS, and evaluated its performance on a cluster with a large number of connections. Our experiment results demonstrate that `RaaS` achieves high throughput for thousand of connections and maintains low CPU and memory overhead through adaptive RDMA transport selection. Part of this work has been devoted to the Ceph community and the source code is available online [2].

References

[1] "Mellanox connect-ib product brief," http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Connect-IB.pdf, 2015.

[2] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou, "Gram: scaling graph computation to the trillions," in *ACM Symposium on Cloud Computing (SoCC)*. ACM, 2015, pp. 408–421.

[3] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *ACM SIGCOMM*. ACM, 2015, pp. 523–536.

[4] I. T. Association, *Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE)*, 2014.

[5] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *ACM SIGCOMM*, 2016.

[6] I. 802.11Qau, *Priority based flow control*, 2011.

[7] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *ACM SIGCOMM*, 2018.

[8] *InfiniBand Architecture Volume 1 and Volume 2, released specification*, 2015, https://cw.infinibandta.org/document/dl/7859.

[9] Y. Wang, X. Meng, L. Zhang, and J. Tan, "C-hint: An effective and reliable cache management for rdma-accelerated key-value stores," in *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[10] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *USENIX NSDI*. USENIX, 2014.

[11] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using rdma and htm," in *European Conference on Computer Systems (EuroSys)*. ACM, 2016.

[12] A. K. M. Kaminsky and D. G. Andersen, "Design guidelines for high performance rdma systems," in *USENIX ATC*, 2016, p. 437.

[13] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *USENIX OSDI*, 2016.

[14] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," vol. 9, no. 7. VLDB Endowment, 2016, pp. 528–539.

[15] S. Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *SOSP*, 2017, pp. 306–324.

[16] *Mellanox Connect-IB produce brief*, 2015, http://www.mellanox.com/related-docs/prod_adapter_cards/PB-connect-IB.pdf.

[17] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *SIGCOMM*. ACM, 2014.

[18] M. Documentation, *Performance Tuning Guide for Mellanox Network Adapters*, http://www.mellanox.com/related-docs/prod-software/.

[2]The code has been put online in 2016. We temporarily hide the online code for double-blind review.

[19] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with rdma for big data processing: Early experiences," in *HOTI*, 2014.

[20] P. W. Frey and G. Alonso, "Minimizing the hidden cost of rdma," in *ICDCS*, 2009.

[21] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1851182.1851192

[22] "Dynamically-connected transport service," https://www.google.com/patents/US20110116512, 2011.

[23] B. Chandrasekaran, P. Wyckoff, and D. K. Panda, "Miba: A micro-benchmark suite for evaluating infiniband architecture implementations," in *Computer Performance Evaluations, Modelling Techniques and Tools.*, 2003, pp. 29–46.

[24] S. Sur, A. Vishnu, H. W. Jin, and D. K. Panda, "Can memory-less network adapters benefit next-generation infiniband systems?" in *Symposium on High Performance Interconnects*, 2005, pp. 45–50.

[25] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *SOSP*, 2015.

[26] *gRPC*, https://grpc.io.

[27] "Percona xtrabackup," https://www.percona.com/software/mysql-database/percona-xtrabackup.

[28] C. Mitchell, Y. Geng, and J. Li, "Using one-sided rdma reads to build a fast, cpu-efficient key-value store." in *USENIX Annual Technical Conference*, 2013, pp. 103–114.

[29] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "Rfp: When rpc is faster than server-bypass with rdma," in *European Conference on Computer Systems*, 2017, pp. 1–15.

[30] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *USENIX Conference on Annual Technical Conference (ATC)*, 2015, pp. 291–305.

[31] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap." in *NSDI*, 2017, pp. 649–667.

[32] "Openfabrics alliance," https://www.openfabrics.org, 2004.

[33] intel, "Rdma communication manager," https://linux.die.net/man/7/rdma_cm, 2010.

[34] S. Hefty, "Rsockets," in *2012 OpenFabris International Workshop, Monterey, CA, USA*, 2012.

[35] *Accelio: OpenSource I/O, message, and RPC acceleration library*, http://www.accelio.org/.