RDMC: A Reliable RDMA Multicast for Large Objects

Jonathan Behrens[1,2], Sagar Jha[1], Ken Birman[1], Edward Tremel[1]

[1] *Department of Computer Science, Cornell University*
[2] *MIT CSAIL*

Abstract

Multicast patterns are common in cloud computing and datacenter settings. Applications and infrastructure tools such as Spark frequently move large objects around, update files replicated to multiple nodes, or push new versions of programs to compute nodes. Some applications use replication directly, for example to increase fault-tolerance or achieve parallelism. Implementations of Paxos, block chains and other libraries often employ a hand-built reliable multicast as a primitive. Yet operating systems continue to be focused on point-to-point communication solutions such as TCP. Our system, RDMC (RDMA Multicast), offers reliable multicast functionality constructed from RDMA unicast. We discuss design choices, present a theoretical analysis of RDMC's robustness to delays and slow network links, and report on experiments that evaluate RDMC over Mellanox RDMA.

1 Introduction

Datacenter loads are heavily dominated by data copying delays, often from a source node to two or more destinations. By 2011, distributed file systems like Cosmos (Microsoft), GFS (Google), and HDFS (Hadoop) handled many petabytes of writes per day (hundreds of Gb/s) [6], and the throughput is surely far higher today. Many files are replicated to multiple storage servers [8]. The latency of this process determines overall write performance for end-user applications. At Facebook, Hadoop traces show that for jobs with reduce phases, the transfer of data between successive phases represents 33% of total run time [4]. Google's Borg has a median task startup latency of around 25 seconds (about 80% devoted to package installation) with upwards of 10,000 tasks starting per minute in some cells [22]. In some cases, copying VM images and input files takes substantially more time than computation [19].

Despite the importance of fast replication, effective general-purpose solutions are lacking. Today, cloud middleware systems typically push new data to nodes in ways that make one copy at a time. Content sharing is often handled through an intermediary caching or a key-value layer, which scales well but introduces extra delay and copying. In parallel platforms like Hadoop the scheduler often can anticipate that a collection of tasks will read the same file, yet unless the data happens to be cached locally, it will be moved point-to-point as each task opens and accesses that file. Cloud systems could substantially improve efficiency by recognizing such interactions as instances of a common pattern. Doing so makes it possible to recover network bandwidth and CPU time currently lost to extraneous transfers and unneeded copying. For time-critical uses, such a primitive would reduce staleness.

Our RDMA multicast protocol, RDMC, solves this problem, offering higher speed with sharply lower resource utilization. RDMC is inexpensive to instantiate, and offers a reliability semantic analogous to that of N side-by-side TCP links, one per receiver. The protocol is also robust to disruption and offers fair division of bandwidth, as we demonstrate using experiments that expose RDMC to scheduling delays, link congestion, and overlapping delivery patterns.

RDMC can also be extended to offer stronger semantics. In work reported elsewhere, we describe Derecho [9]: a new open-source software library layered over RDMC that supports atomic multicast as well as a classic durable Paxos. To gain these properties, Derecho introduces a small delay, during which receivers buffer messages and exchange status information. Delivery occurs when RDMC messages are known to have reached all destinations. No loss of bandwidth is experienced, and the added delay is surprisingly small.

The contributions of the present paper are as follows:

- We describe RDMC in detail, showing how it maps multicast transfers to an efficient pattern of RDMA unicast operations.
- We undertake an extensive evaluation of the system.
- We show that RDMC is robust to scheduling and network delays and discuss options for recovering in the rare event of a failed transfer.
- We argue that because RDMC generates a deterministic block transfer pattern, it offers a stepping stone towards offloading reliable multicast directly onto the NIC.

2 Background on RDMA

RDMA (remote direct memory access) is a zero-copy communication standard. It has been used for many years on Infiniband, but is now also working robustly on standard datacenter Ethernet [15, 25].

RDMA is a user-space networking solution, accessed via *queue pairs:* lock-free data structures shared between user code and the network controller (NIC), consisting of a send queue and a receive queue. RDMA supports several modes of operation. RDMC makes use of reliable *two-sided* RDMA operations, which behave similarly to TCP. With this mode, the sender and receiver bind their respective queue pairs together, creating a session fully implemented by the NIC endpoints. A send is issued by posting a memory region to the send queue, and a process indicates its readiness to receive by posting a memory region to the receive queue. The sender NIC will then transmit the data, awaiting a hardware-level ack. After a specified timeout, the NIC retries; after a specified number of retries, it breaks the connection and reports failure (as explained below, RDMC won't start to send unless the receiver is ready, hence a broken connection indicates a genuine network or endpoint failure). Once a send and the matching receive are posted, the data is copied directly from the sender's memory to the receiver's designated location, reliably and at the full rate the hardware can support. A completion queue reports outcomes. End-to-end software resending or acknowledgments are not needed: either the hardware delivers the correct data (in FIFO order) and reports success, or the connection breaks.

If processes P and Q wish to set up a two-sided RDMA connection, they must first exchange a form of key (RDMA lacks the equivalent of the TCP `listen` operation, and has no hardware-layer 3-way handshake). RDMC can support multiple overlapping sessions, and they can be created as needed, hence the need to exchange keys can arise without warning. To minimize delay, RDMC creates a full $N * N$ set of TCP connections during bootstrap, then uses them for RDMA connection setup and failure reporting, as explained below.

RDMA offers several additional modes: a *one-sided* read and write mode (Q authorizes P to directly access some memory region), data-inlining, unreliable point-to-point datagrams, and an unreliable multicast. These features are intended for small transfers, and because RDMC focuses on large transfers we did not find them useful, with one exception: as each receiver becomes ready to accept an incoming transfer, it does a a one-sided write to tell the sender, which starts sending only after all are prepared.

Evolution of RDMA NIC programmability. There is growing interest in programmable network devices. For RDMA NICs, this may introduce new request-ordering options.

Today's RDMA NICs guarantee two forms of ordering: (1) requests enqueued on a single send or receive queue will be performed in FIFO order (2) a receive completion occurs only after the incoming transfer is finished. Mellanox's CORE-Direct [14] feature proposes a third form of request ordering: it is possible to enqueue an RDMA send that will wait both until the prior request has completed, as well as for completion of some other RDMA send or receive, possibly even on a different queue pair. In cases where a node Q needs to relay data received from P to another node R, this avoids the software delay at Q to issue the relay operation after the receive is complete. We believe that CORE-Direct is just one of what will eventually be a wide range of new RDMA NIC programmability features.

RDMC was designed to anticipate this trend, although the hardware functionality isn't fully mature yet and hence serious evaluation of the potential will require additional work. RDMC can precompute data-flow graphs describing the full pattern of data movement at the outset of each multicast send. Members of a replication group could thus post data-flow graphs at the start of a transfer, linked by cross-node send/receive dependencies. The hardware would then carry out the whole transfer without further help. Offloading would eliminate the need for any software actions, but creates an interesting scheduling puzzle: if operations are performed as soon as they become possible, priority inversions could arise, whereby an urgent operation is delayed by one that actually has substantial scheduling slack. As these new hardware capabilities mature, we hope to explore such questions.

3 High level RDMC summary

We implemented RDMC using the two-sided RDMA operations described above. The basic requirement is to create a pattern of RDMA unicasts that would efficiently perform the desired multicast. In the discussion that follows, the term *message* refers to the entire end-user object being transmitted: it could be hundreds of megabytes or even gigabytes in size. Small messages are sent as a single block, while large messages are sent as a series of blocks: this permits *relaying* patterns in which receivers simultaneously function as senders. The benefit of relaying is that it permits full use of both the incoming and outgoing bandwidth of the receiver NICs. In contrast, protocols that send objects using a single large unicast transfer are limited: any given node can use its NIC in just one direction at a time.

This yields a framework that operates as follows:

1. For each RDMC transfer, the sender and receivers first create an overlay mesh of multi-way bindings: an *RDMC group*. This occurs out of band, using TCP as a bootstrapping protocol. RDMC is lightweight and can support large numbers of overlapping groups, but to minimize bootstrap delay, applications that will perform repeated transfers should reuse groups when feasible.
2. Each transfer occurs as a series of reliable unicast RDMA transfers, with no retransmission. RDMC computes sequences of sends and receives at the outset and queues them up to run as asynchronously as possible. As noted earlier, it should eventually be feasible to offload the entire sequence to a programmable NIC.
3. On the receive side, RDMC notifies the user application of an incoming message, and it must post a buffer of the correct size into which bytes are received.
4. Sends complete in the order they were initiated. Incoming messages are guaranteed to not be be corrupted, to arrive in sender order, and will not be duplicated.
5. RDMA apportions bandwidth fairly if there are several active transfers in one NIC. RDMC extends this property, offering fairness for overlapping groups.
6. If an RDMA connection fails, the non-crashed endpoint(s) learn of the event from their NICs. RDMC re-

```
// Create a new group with the designated members (first member is the root).
bool create_group(int group_number,
                  vector<int> members,
                  function<memory_region(int size)> incoming_message_callback,
                  function<void(char* data, int size)> message_completion_callback);

// Destroy the group, and deallocate associated resources.
void destroy_group(int group_number);

// Attempt to send a message to the group. Will fail if not the root.
bool send(int group_number, char* data, int size);
```

Figure 1: RDMC library interface

lays these notifications, so that all survivors eventually learn of the event. The application can then self-repair by closing the old RDMC session and initiating a new one.

4 System Design

4.1 External API

Figure 1 shows the RDMC interface, omitting configuration parameters like block size. The send and destroy_group functions are self-explanatory. The create_group function is called concurrently (with identical membership information) by all group members; we use the out-of-band TCP connections mentioned earlier to initiate this step. create_group takes two callback functions, which will be used to notify the application of events. The incoming_message_callback is triggered by receivers when a new transfer is started, and is also used to obtain a memory region to write the message into. Memory registration is expensive, hence we perform this step during startup, before any communication activity occurs.

The message completion callback triggers once a message send/receive is locally complete and the associated memory region can be reused. Notice that this might happen before other receivers have finished getting the message, or even after other receivers have failed.

Within a group, only one node (the "root") is allowed to send data. However, an application is free to create multiple groups with identical membership but different senders. Note that group membership is static once created: to change a group's membership or root the application should destroy the group and create a new one.

4.2 Architectural Details

RDMC runs as a user-space library. Figure 2 shows an overview of its architecture.

Initialization. When the application first launches, its members must initialize RDMC. At this point, RDMC creates the mesh of TCP connections mentioned earlier, registers memory, creates a single RDMA completion queue, and prepares other internal data structures. Later, during runtime, all RDMC sessions share a single completion queue and thread, reducing overheads. To avoid polling when no I/O is ocuring, the completion thread polls for 50 ms after each completion event, then switches to an interrupt-driven completion mode. It switches back to polling at the next event.

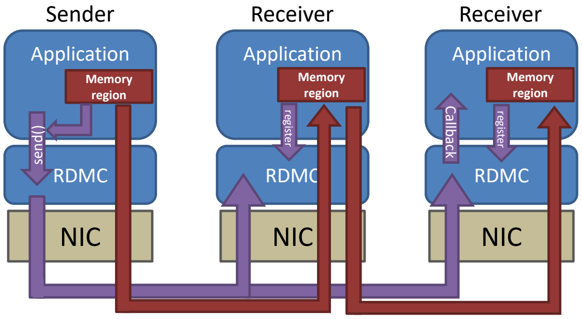


Figure 2: RDMC with a sender and 2 receivers.

Data Transfer. Although we will turn out to be primarily focused on the *binomial pipeline* algorithm, RDMC actually implements several data transfer algorithms, which makes possible direct side-by-side comparisons. To be used within RDMC, a sending algorithm must preserve the sending order, mapping message-sends to determistic sequences of block transfers.

When a sender initiates a transfer, our first step is to tell the receivers how big the incoming message will be, since any single RDMC group can transport messages of various sizes. Here, we take advantage of an RDMA feature that allows a data packet to carry an integer "immediate" value. Every block in a message will be sent with an immediate value indicating the total size of the message it is part of. Accordingly, when an RDMC group is set up, the receiver posts a receive for an initial block of known size. When this block arrives, the immediate value allows us to determine the full transfer size and (if necessary), to allocate space for the full message. If more blocks will be sent, the receiver can post additional asynchronous receives as needed, and in parallel, copy the first block to the start of the receive area. Then, at the end of the transfer, a new receive is posted for the first block of the next message.

The sender and each receiver treat the schedule as a series of asynchronous steps. In each step every participant either sits idle or does some combination of sending a block and receiving a block. The most efficient schedules are bidirec-

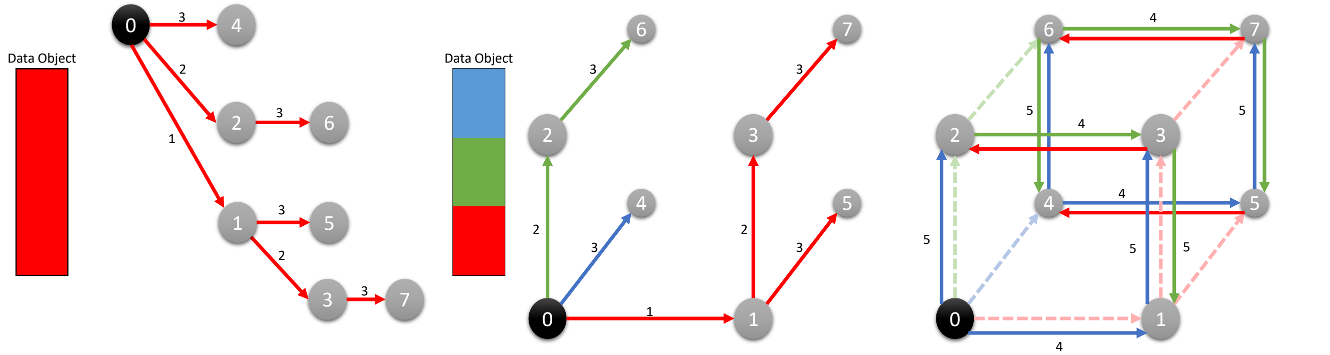


Figure 3: (Left) A standard binomial tree multicast, with the entire data object sent in each transfer. (Center) A binomial pipeline multicast, with the data object broken into three blocks, showing the first three steps of the protocol. In this phase, the sender sends a different block in each round, and receivers forward the blocks they have to their neighbors. (Right) The final two steps of the binomial pipeline multicast, with the earlier sends drawn as dotted lines. In this phase, the sender keeps sending the last block, while receivers exchange their highest-numbered block with their neighbors.

tional: they maximize the degree to which nodes will send one block while concurrently receiving some other block. Given the asynchronous step number, it is possible to determine precisely which blocks these will be. Accordingly, as each receiver posts memory for the next blocks, it can determine precisely which block will be arriving and select the correct offset into the receive memory region. Similarly, at each step the sender knows which block to send next, and to whom.

Our design generally avoids any form of out-of-band signaling or other protocol messages, with one exception: to prevent blocks from being sent prematurely, each node will wait to receive a `ready_for_block` message from its target so that it knows the target is ready. By ensuring that the sender never starts until the receiver is ready, we avoid costly backoff/retransmission delays, and eliminate the risk that a connection might break simply because some receiver had a scheduling delay and didn't post memory in time. We also sharply reduce the amount of NIC resources used by any one multicast: today's NICs exhibit degraded performance if the number of concurrently active receive buffers exceeds NIC caching capacity. RDMC posts only a few receives per group, and since we do not anticipate having huge numbers of concurrently active groups, this form of resource exhaustion is avoided.

4.3 Protocol

Given this high-level design, the most obvious and important question is what algorithm to use for constructing a multicast out of a series of point-to-point unicasts. RDMC implements multiple algorithms; we'll describe them in order of increasing effectiveness.

Sequential Send. The sequential pattern is common in today's datacenters and is a good choice for small messages. It implements the naïve solution of transmitting the entire message from the sender one by one to each recipient in turn. Since the bandwidth of a single RDMA transfer will be nearly line rate, this pattern is effectively the same as running N independent point-to-point transfers concurrently.

Notice that with a sequential send, when creating N replicas of a B-bit message, the sender's NIC will incur an IO load of $N * B$ bits. Replicas will receive B bits, but do no sending. With large messages, this makes poor use of NIC resources: a 100Gbps NIC can potentially send and receive 100Gbps concurrently. Thus sequential send creates a hot spot at the sender.

Chain Send. This algorithm implements a bucket-brigade, similar to the chain replication scheme described in [21]. After breaking a message into blocks, each inner receiver in the brigade relays blocks as it receives them. Relayers use their full bidirectional bandwidth, but the further they are down the chain, the longer they sit idle until they get their first block, so worst-case latency is high.

Binomial Tree. For large objects, better performance is possible if senders send entire messages, and receivers relay each message once they get it, as seen in Figure 3 (left). The labels on the arrows represent the asynchronous time step. Here, sender 0 starts by sending some message to receiver 1. Then in parallel, 0 sends to 2 while 1 sends to 3, and then in the final step 0 sends to 4, 1 sends to 5, 2 sends to 6 and 3 sends to 7. The resulting pattern of sends traces out a binomial tree, hence latency will be better than that for the sequential send, but notice that the inner transfers can't start until the higher level ones finish. For a small transfer, this would be unavoidable, but recall that RDMC aims at cases where transfers will often be very large. Ideally, we would wish to improve link utilization by breaking large transfers into a series of smaller blocks and pipelining the block transfers, while simultaneously minimizing latency by leveraging a binomial tree routing pattern.

Binomial Pipeline. By combining the Chain Send with the Binomial Tree, we can achieve both goals, an observation first made by Ganesan and Seshadri [7]. The algorithm works by creating a virtual hypercube overlay of dimension d, within which d distinct blocks will be concurrently relayed (Figure 3, middle, where the blocks are represented by the colors red, green and blue). Each node repeatedly performs one send operation and one receive operation until, on the last step, they all simultaneously receive their last block (if the number of nodes isn't a power of 2, the final receipt spreads over two asynchronous steps). The original work by Ganesan and Seshadri was theoretical, validated with simulations. Further, they assumed that the network is synchronous. We extended their approach to work in a fully asynchronous setting where a node is waiting for exactly one node to send a block. We also decoupled the send and receive steps so that a send step is only pending if the associated block hasn't been received. The resulting algorithm is exceptionally efficient because it reaches its fully-loaded transfer pattern quickly, ensuring that nodes spend as much time as possible simultaneously sending and receiving blocks.

Hybrid Algorithms Current datacenters hide network topology to prevent application behaviors that might defeat broader management goals. Suppose, however, that one were building an infrastructure service for datacenter-wide use, and that this form of information was available to it. Many datacenters have full bisection bandwidth on a rack-by-rack basis, but use some form of an oversubscribed top of rack (TOR) switch to connect different racks. When a binomial pipeline multicast runs in such a setting, a large fraction of the transfer operations traverse the TOR switch (this is because if we build the overlay using random pairs of nodes, many links would connect nodes that reside in different racks). In contrast, suppose that we were to use two separate instances of the binomial pipeline, one in the TOR layer, and a second one within the rack. By doing so we could seed each rack leader with a copy of the message in a way that creates a burst of higher load, but is highly efficient and achieves the lowest possible latency and skew. Then we repeat the dissemination within the rack, and again maximize bandwidth while minimizing delay and skew.

4.4 Analysis

We now offer a formal description of the binomial pipeline algorithm, starting with a precise description of the rule for selecting the block to send at a given step, and then proceeding to a more theoretical analysis of the predicted behavior of the algorithm during steady-state operation.

Let the number of nodes be n. Assume that n is a power of 2, $n = 2^l$ (for reasons of brevity we omit the general case, although the needed extensions are straightforward). Each node has an id in $\{0, 1, \ldots, n-1\}$, an $l-$bit number with node 0 as the sender. Let the number of blocks to send be k, ordered from 0 to $k-1$. The first block takes $\log n = l$ steps to reach every node. Since, the block sends are pipelined, the next block send completes in the next steps and so on. Thus, the number of steps to complete the send is $l + k - 1$. We number the steps from 0 to $l + k - 2$. Since all blocks are only at the sender in the beginning, it takes the first l steps for every node to receive at least 1 block. We refer to steps l to $l + k - 2$ as "steady" steps.

Let $\%$ denote integer modulus and $\oplus$ denote the bitwise XOR operation. Given the nodes, we can construct a hypercube of l dimensions where each node occupies a distinct vertex of the hypercube. The $l-$bit node-id of a node identifies the mapping from nodes to vertices as follows: A node i has edges to nodes $i \oplus 2^m$, for $m = 0, 1, \ldots, l-1$. The neighbor $i \oplus 2^m$ is along direction m from i.

Ganesan and Seshadri provide the following characterization of the algorithm:

- At each step j, each node exchanges a block with its neighbor along direction $j\%l$ of the hypercube (except if the node does not have a block to send or its neighbor is the sender).
- The sender sends block j in step j for steps $j, 0 \leq j \leq k-1$ and the last block $k-1$ for steps $j, k \leq j \leq l+k-1$. Other nodes send the highest numbered block they have received before step j.

From this specification, we devised a send scheme for a given node and step number, required for the asynchronous implementation of the algorithm. Let $\sigma(n, r)$ denote the number obtained by a right circular shift of the $l-$ bit number n by r positions. Let $tr\_ze(m)$ be the number of trailing zeros in the binary representation of m. Given step j, node i sends the block number, $b =$

$$\begin{cases} min(j, k-1), & \text{if } i = \sigma(n, j\%l) = 0 \\ \text{nothing}, & \text{if } \sigma(n, j\%l) = 1 \\ min(j - l + r, k-1), & \text{if } \sigma(n, j\%l) \neq 1 \text{ and } j - l + r >= 0 \\ \text{nothing}, & \text{otherwise,} \\ & \text{where } r = tr\_ze(\sigma(n, j\%l)) >= 0 \end{cases}$$

to the node $i \oplus 2^{j\%l}$, for each $0 \leq i \leq n-1, 0 \leq j \leq l+k-2$.

4.5 Robustness of RDMC's Binomial Pipeline

As will be seen in Section 5, the binomial pipeline remains stable even in an experimental setting subject to occasional delays in sending, has variable link latencies, and that includes congested network links. One can characterize multicast robustness in several dimensions:

- Tolerance of normal network issues of data loss, corruption and duplication.
- Tolerance of interference from other tenants sharing network resources.
- Delay tolerance : network delays, scheduling delays.

The first two properties arise from the hardware, which provides error correction and deduplication, and achieves fair bandwidth sharing in multi-tenant environments. Delay tolerance is a consequence of RDMC's block-by-block sending pattern and receiver-receiver relaying. In particular:

1. A delay ϵ in sending a block leads to a maximum delay of ϵ in the total time to send. If a block send takes about δ time, the total time without delay is $(l + k - 1)\delta$. Assuming $\epsilon = \mathcal{O}(\delta)$, the total time becomes $(l + k - 1)\delta + \epsilon$. If the number of blocks is large, $(l + k - 1)\delta >> \epsilon$, and thus the effective bandwidth does not decrease by much.
2. Since a node cycles through its l neighbors for exchanging blocks, a link between two neighbors is traversed on just $1/l$ of the steps. Thus a slow link has a limited impact on performance. For example, if one link has bandwidth T' and other links have bandwidth T, with $T > T'$, rough calculations show the effective bandwidth to be at least a

factor of $\frac{lT'}{T+(l-1)T'}$ of the bandwidth when each link is of bandwidth T. If $T' = T/2, n = 64$, this fraction is 85.6%. Contrast this to the chain replication scheme where each link is traversed by each block and the bandwidth is limited by the slowest link (T' in our example).

3. If a node i sends block b in round j, define $slack(i,j)$ to be j minus the step number in which i received b. The average slack for a given steady step j, $avg_slack(j)$is defined as $\frac{\sum_{i \text{ sends in } j} slack(i,j)}{\#\text{senders in } j}$. We found that $avg\_slack(j)$, for any *steady* step j is a constant equal to $2(1-\frac{l-1}{n-2}) = 2(1-\frac{\log n-1}{n-2})$. For moderate n, $\log n << n$, average slack is ≈ 2. A slack greater than 1 tells us that *the node received the block it must send on the current step at least 2 steps in the past.* This is of value because if the node is running slightly late, it may be able to catch up.

A more comprehensive investigation of robustness in the presence of delay represents an interesting direction for future research. Our experiments were performed both on a dedicated cluster and in a large shared supercomputer, and exposed RDMC to a variety of scheduling and link delays, but in an uncontrolled way. Performance does drop as a function of scale (presumably, in part because of such effects), but to a limited degree. The open question is the degree to which this residual loss of performance might be avoided.

4.6 Insights from using RDMC

We now have several years of experience with RDMC in various settings, and have used it within our own Derecho platform. Several insights emerge from these activities.

Recovery From Failure. As noted earlier, an RDMC group behaves much like a set of side-by-side TCP connections from the sender to each of the receivers. Although failures are sensed when individual RDMA connections report a problem, our policy of relaying failure information quickly converges to a state in which the disrupted RDMC group ceases new transmissions, and in which all surviving endpoints are aware of the failure. At this point, some receivers may have successfully received and delivered messages that other receivers have not yet finished receiving.

To appreciate the resulting recovery challenge, we can ask what the sender "knows" at the time that it first learns that its RDMC group has failed. Much as a TCP sender does not learn that data in the TCP window has been received and processed unless some form of end-to-end acknowledgement is introduced, an RDMC sender trusts RDMC to do its job. If a group is used for a series of transfers the sender will lack certainty about the status of recently-transmitted messages (RDMC does not provide an end-to-end status reporting mechanism). On the other hand, disruption will be sensed by all RDMC group members if something goes wrong. Moreover, failure will always be reported when closing (*destroying*) the RDMC group. Thus, if the group close operation is successful, the sender (and all receivers) can be confident that every RDMC message reached every destination.

For most purposes listed in the introduction, this guarantee is adequate. For example, if a multicast file transfer finishes and the close is successful, the file was successfully delivered to the full set of receivers, with no duplications, omissions or corruption. Conversely, if the transfer fails, every receiver learns this and the file transfer tool could simply retry the transfer within the surviving members. If the tool was transferring a long sequence of files and the cost of resending them were a concern, it could implement an end-to-end status check to figure out which ones don't need to be resent.

Systems seeking stronger guarantees can leverage RDMC too. For example, Derecho augments RDMC with a replicated status table implemented using one-sided RDMA writes [9]. On reception of an RDMC message, Derecho buffers it briefly. Delivery occurs only after every receiver has a copy of the message, which receivers discover by monitoring the status table. A similar form of distributed status tracking is used when a failure disrupts an RDMC group. Here, Derecho uses a leader-based cleanup mechanism (again based on a one-sided RDMA write protocol) to collect state from all surviving nodes, analyze the outcome, and then tell the participants which buffered messages to deliver and which to discard. Through a series of such extensions, Derecho is able to offer the full suite of Paxos guarantees, yet transfers all messages over RDMC.

Small messages. RDMC is optimized for bulk data movement. The work reported here only looked at the large message case. Derecho includes a small-message protocol that uses one-sided RDMA writes into a set of round-robin bounded buffers, one per receiver, and compares performance of that method with that of RDMC. In summary, the optimized small message protocol gains as much as a 5x speedup compared to RDMC provided that the group is small enough (up to about 16 members) and the messages are small enough (no more than 10KB). For larger groups or larger messages, and for long series of messages that can be batched, the binomial pipeline dominates.

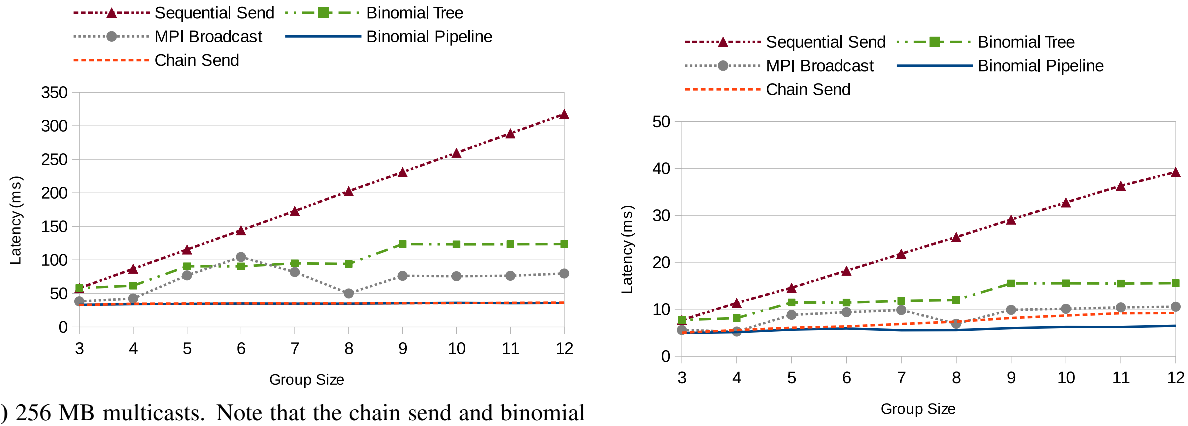
Memory management. RDMC affords flexible memory management. In the experiments reported here, we preregister memory regions that will be used with the RDMA NIC, but allocate memory for each new message when the first block arrives. Thus receivers perform a call to `malloc` on the critical path. In applications that can plan ahead, better performance can be achieved by performing memory allocation before the start of a long series of transfers.

5 Experiments

5.1 Setup

We conducted experiments on several clusters equipped with different amounts of memory and NIC hardware.

Fractus. Fractus is a cluster of 16 RDMA-enabled nodes running Ubuntu 16.04, each equipped with a 4x QDR Mellanox NIC and 94 GB of DDR3 memory. All nodes are connected to both a 100 Gb/s Mellanox IB switch and a 100 Gb/s Mellanox RoCE switch, and have one-hop paths to one-another.



(a) 256 MB multicasts. Note that the chain send and binomial pipeline achieve very similar latency.

(b) 8 MB multicasts.

Figure 4: Latency of MPI (MVAPICH) and several RDMC algorithms on Fractus. Group sizes include the sender, so a size of three means one sender and two receivers.

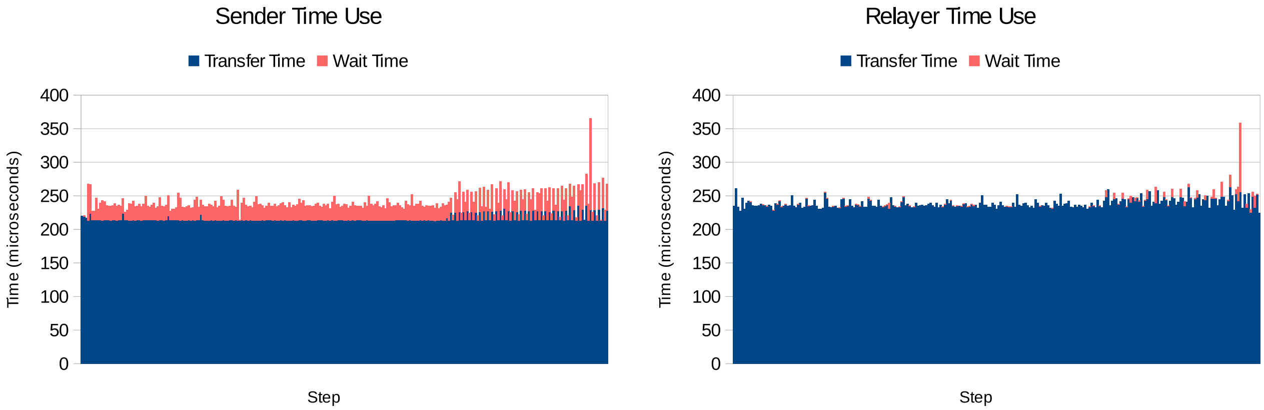


Figure 5: Breakdown of transfer time and wait time of two nodes taking part in the 256 MB transfer. The majority of time is spent in hardware (blue), but the sender (left) incurs a higher CPU burden (orange) than the receiver (right). Offloading RDMC fully into the hardware would eliminate this residual load and reduce the risk that a long user-level scheduling delay could impact overall transfer performance.

Sierra. The Sierra cluster at Lawrence Livermore National Laboratory consists of 1,944 nodes of which 1,856 are designated as batch compute nodes. Each is equipped with two 6-core Intel Xeon EP X5660 processors and 24GB memory. They are connected by an Infiniband fabric which is structured as a two-stage, federated, bidirectional, fat-tree. The NICs are 4x QDR QLogic adapters each operating at a 40 Gb/s line rate. The Sierra cluster runs TOSS 2.2, a modified version of Red Hat Linux.

Stampede-1. The U. Texas Stampede-1 cluster contains 6400 C8220 compute nodes with 56 Gb/s FDR Mellanox NICs. Like Sierra, it is batch scheduled with little control over node placement. We measured unicast speeds of up to 40 Gb/s.

Apt Cluster. The EmuLab Apt cluster contains a total of 192 nodes divided into two classes: 128 nodes have a single Xeon E5-2450 processor with 16 GB of RAM, while 64 nodes have two Xeon E5-2650v2 processors and 64 GB of RAM. All have one FDR Mellanox CX3 NIC which is capable of 56 Gb/s.

Interestingly, Apt has a significantly oversubscribed TOR network that degrades to about 16 Gb/s per link when heavily loaded. This enabled us to look at the behavior of RDMC under conditions where some network links are much slower than others. Although the situation is seemingly ideal for taking the next step and experimenting on hybrid protocols, this proved to be impractical: Apt is batch-scheduled like Sierra, with no control over node placement, and we were unable to dynamically discover network topology.

Our experiments include cases that closely replicate the RDMA deployments seen in today's cloud platforms. For example, Microsoft Azure offers RDMA over Infiniband as part of its Azure Compute HPC framework, and many vendors make use of RDMA in their own infrastructure tools, both on Infiniband and on RoCE. However, large-scale end-user testbeds exposing RoCE are not yet available: operators are apparently concerned that heavy use of RoCE could trigger data-center-wide instability. Our hope is that rollout of DCQCN will reassure operators, who would then see an obvious benefit to allowing their users to access RoCE.

In all of our experiments, the sender(s) generates a message containing random data, and we measure the time from when the send is submitted to the library to when all clients have gotten an upcall indicating that the multicast has completed. The largest messages sent have sizes that might arise in applications transmitting videos, or when pushing large images to compute nodes in a data analytics environment. Smaller message sizes are picked to match tasks such as replicating photos or XML-encoded messages. Bandwidth is computed as the number of messages sent, multiplied by the size of each message, divided by the total time spent (regardless of the number of receivers). RDMC does not pipeline messages, so the latency of a multicast is simply the message size divided by its bandwidth.

5.2 Results

Figure 4 compares the relative performance of the different algorithms considered. For comparison, it also shows the throughput of the heavily optimized `MPI_Bcast()` method from MVAPICH, a high-performance computing library that implements the MPI standard on Infiniband networks (we measured this using a separate benchmark suite). As anticipated, both sequential send and binomial tree do poorly as the number of nodes grows. Meanwhile chain send is competitive with binomial pipeline, except for small transfers to large numbers of nodes where binomial pulls ahead. MVAPICH falls in between, taking from 1.03× to 3× as long as binomial pipeline. Throughout the remainder of this paper we primarily focus on binomial pipeline because of its robust performance across a range of settings, however we note that chain send can often be useful due to its simplicity.

5.2.1 Microbenchmarks

In Table 1 we break down the time for a single 256 MB transfer with 1 MB blocks and a group size of 4 (meaning 1 sender and 3 receivers) conducted on Stampede. All values are in microseconds, and measurements were taken on the node *farthest* from the root. Accordingly, the Remote Setup and Remote Block Transfers reflect the sum of the times taken by the root to send and by the first receiver to relay. Roughly 99% of the total time is spent in the Remote Block Transfers or Block Transfers states (in which the network is being fully utilized) meaning that overheads from RDMC account for only around 1% of the time taken by the transfer.

Figure 5 examines the same send but shows the time usage for each step of the transfer for both the relayer (whose times are reported in the table) and for the root sender. Towards the end of the message transfer we see an anomalously long wait time on both instrumented nodes. As it turns out, this demonstrates how RDMC can be vulnerable to delays on individual nodes. In this instance, a roughly 100 μs delay on the relayer (likely caused by the OS picking an inopportune time to preempt our process) forced the sender to delay on the following step when it discovered that the target for its next block wasn't ready yet. The CORE-Direct functionality would mitigate this.

Remote Setup	11
Remote Block Transfers	461
Local Setup	4
Block Transfers	60944
Waiting	449
Copy Time	215
Total	62084

Table 1: Time (microseconds) for key steps in a transfer.

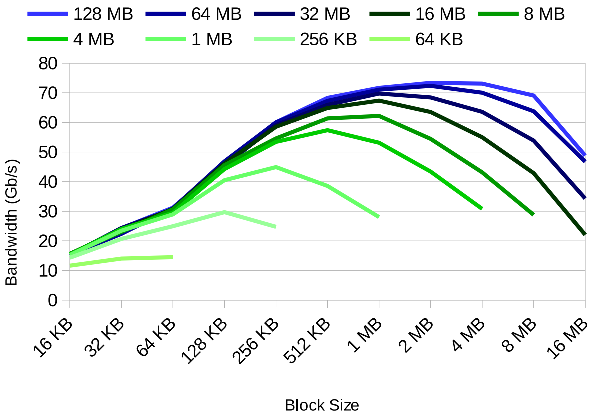


Figure 6: Multicast bandwidth (computed as the message size divided by the latency) on Fractus across a range of block sizes for messages between 16 KB and 128 MB, all for groups of size 4.

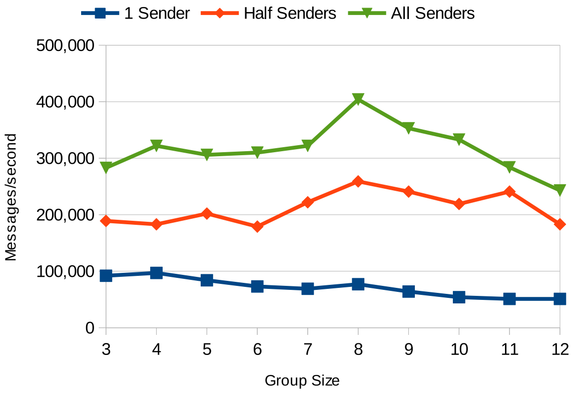


Figure 7: 1 byte messages/sec. (Fractus)

In Figure 6, we examine the impact of block size on bandwidth for a range of message sizes. Notice that increasing the block size initially improves performance, but then a peak is reached. This result is actually to be expected as there are two competing factors. Each block transfer involves a certain amount of latency, so increasing the block size actually increases the rate at which information moves across links (with diminishing returns as the block size grows larger). However, the overhead associated with the binomial pipeline algorithm is proportional to the amount of time spent transferring an individual block. There is also additional overhead incurred when there are not enough blocks in the message for all nodes to get to contribute meaningfully to the transfer.

Finally, Figure 7 measures the number of 1 byte messages

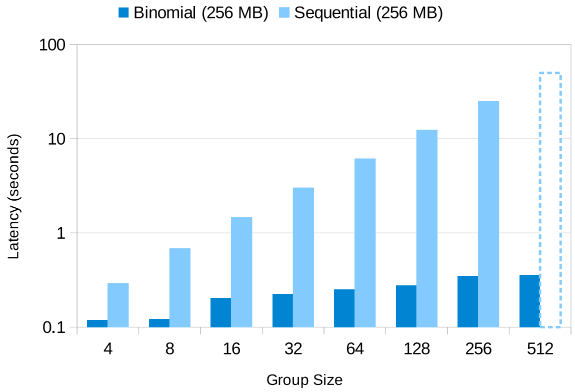


Figure 8: Total time for replicating a 256MB object to a large number of nodes on Sierra.

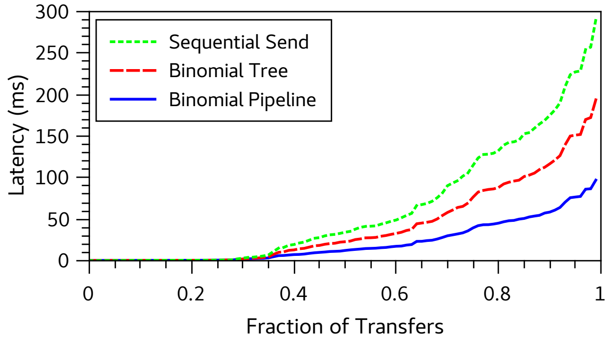


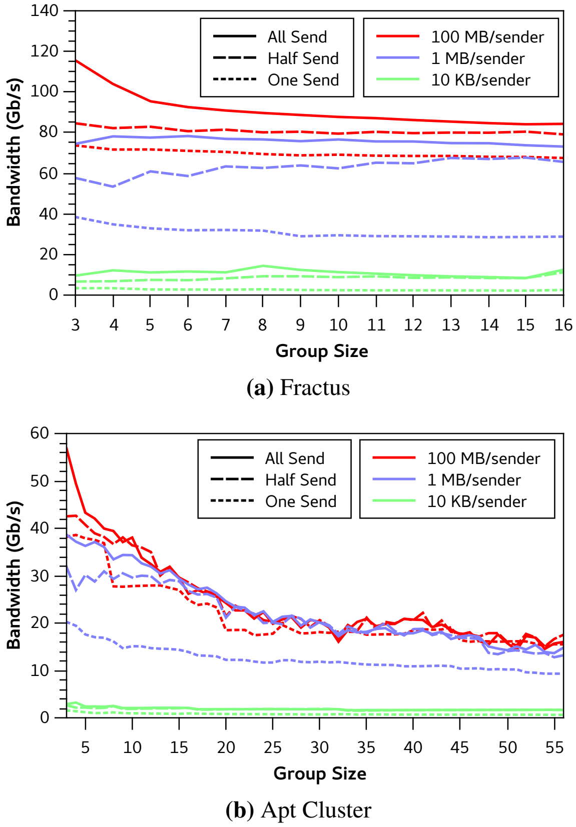
Figure 9: Distribution of latencies when simulating the Cosmos storage system replication layer.

delivered per second using the binomial pipeline, again on Fractus. Note, however, that the binomial pipeline (and indeed RDMC as a whole) is not really intended as a high-speed event notification solution: were we focused primarily on delivery of very small messages at the highest possible speed and with the lowest possible latency, there are other algorithms we could have explored that would outperform this configuration of RDMC under most conditions. Thus the 1-byte behavior of RDMC is of greater interest as a way to understand overheads than for its actual performance.

5.2.2 Scalability

Figure 8 compares scalability of the binomial pipeline on Sierra with that of sequential send (the trend was clear and Sierra was an expensive system to run on, so we extrapolated the 512-node sequential send data point). While sequential send scales linearly in the number of receivers, binomial pipeline scales sub-linearly, which makes an orders of magnitude difference when creating large numbers of copies of large objects. This graph leads to a surprising insight: with RDMC, *replication can be almost free:* whether making 127, 255 or 511 copies, the total time required is almost the same.

Although we did not separately graph end-of-transfer time, binomial pipeline transfers also complete nearly simultaneously: this minimizes temporal skew, which is important in parallel computing settings because many such systems run as a series of loosely synchronized steps that end with some form of shuffle or all-to-all data exchange. Skew can leave the whole system idle waiting for one node to finish. In contrast, the linear degradation of sequential send is also associated with high skew. This highlights the very poor performance of the technology used in most of today's cloud computing frameworks: not only is copy-by-copy replication slow, but it also disrupts computations that need to wait for the transfers to all finish, or that should run in loosely synchronized stages.



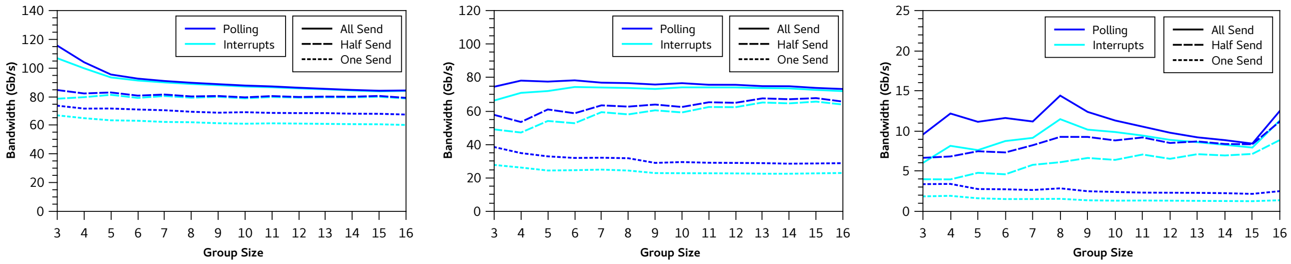
(a) Fractus

(b) Apt Cluster

Figure 10: Aggregate bandwidth of concurrent multicasts on Fractus and the Apt cluster for cases in which we varied the percentage of active senders in each node-group (in a group with k senders, we used k overlapped RDMC groups with identical membership). The Apt cluster has an oversubscribed TOR; our protocols gracefully adapt to match the available bandwidth.

Next, we set out to examine the behavior of RDMC in applications that issue large numbers of concurrent multicasts to overlapping groups. We obtained a trace sampled from the data replication layer of Microsoft's Cosmos system, a data warehouse used by the Bing platform. Cosmos currently runs on a TCP/IP network, making no use of RDMA or multicast. The trace has several million 3-node writes with random target nodes and object sizes varying from hundreds of bytes to hundreds of MB (the median is 12MB and the mean 29 MB). Many transfers have overlapping target groups.

To simulate use of multicast for the Cosmos workload, we designated one Fractus node to generate traffic, and 15 nodes to host the replicas. The system operated by generating objects



(a) 100MB Transfers **(b)** 1MB Transfers **(c)** 10 KB Transfers

Figure 11: Comparison of RDMC's normal hybrid scheme of polling and interrupts (solid), with pure interrupts (dashed). There is no noticeable difference between pure polling and the hybrid scheme. All ran on Fractus.

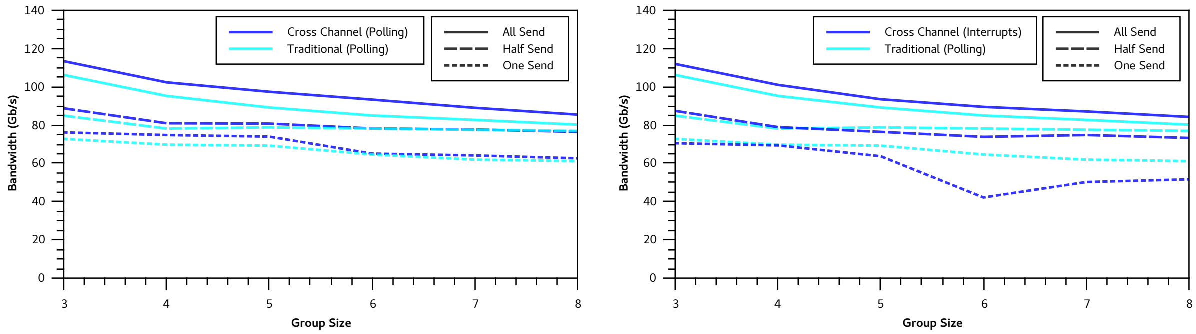


Figure 12: CORE-Direct experiment using a chain multicast protocol to send a 100 MB message. The left is a run using hybrid polling/interrupts; on the right is a run with purely interrupts. Both experiments were on Fractus.

filled with random content, of the same sizes as seen in the trace, then replicating them by randomly selecting one of the possible 3-node groupings as a target (the required 455 RDMC groups were created beforehand so that this would be off the critical path). Figure 9 shows the latency distribution for 3 different send algorithms. Notice that binomial pipeline is almost twice as fast as binomial tree and around three times as fast as sequential send. Average throughput when running with binomial pipeline is around 93 Gb/s of data replicated, which translates to about a petabyte per day. We achieve nearly the full bisection capacity of Fractus, with no sign of interference between concurrent overlapping transfer. The RDMC data pattern is highly efficient for this workload: no redundant data transfers occur on any network link.

A second experiment looked at group overlap in a more controlled manner with a fixed multicast message size. In Figure 10 we construct sets of groups of the size given by the X-axis label. The sets have identical members (for example, the 8-node case would always have the identical 8 members), but different senders. At each size we run 3 experiments, varying the number of senders. (1) In the experiment corresponding to the solid line, all members are senders (hence we have 8 perfectly overlapped groups, each with the same members, but a different sender). (2) With the dashed line, the number of overlapping groups is half the size: half the members are senders. (3) Finally, the dotted line shows performance for a single group spanning all members but with a single sender. All senders run at the maximum rate, sending messages of the size indicated. Then we compute bandwidth by measuring the time to transfer a given sized message to *all* of the overlapping groups, and dividing by the message size times the number of groups (i.e. the total bytes sent).

Again, we see that full resources of the test systems were efficiently used. On Fractus, with a full bisection capacity of 100Gbps, our peak rate (seen in patterns with concurrent senders) was quite close to the limits, at least for larger message sizes. On Apt, which has an oversubscribed TOR, the bisection bandwidth approaches 16Gbps for this pattern of communication, and our graphs do so as well, at least for the larger groups (which generated enough load to saturate the TOR switch).

5.2.3 Resource Considerations

RDMA forces applications to either poll for completions (which consumes a full core), or to detect completions via interrupts (which incurs high overheads and delay). RDMC uses a hybrid solution, but we wanted to understand whether this has any negative impacts on performance. Our first test isn't shown: we tested the system with pure polling, but found that this was not measurably faster than the hybrid.

Next, as shown in Figure 11 we compared RDMC in its standard hybrid mode with a version running using pure interrupts, so that no polling occurs. For the latter case, CPU loads (not graphed) are definitely lower: they drop from almost exactly 100% for all runs with polling enabled, to around

10% for 100 MB transfers and 50% for 1 MB transfers. With 10 KB transfers, there was only a minimal difference since so much time was spent processing blocks. Despite the considerable improvement in CPU usage, the bandwidth impact is quite minimal, particularly for large transfers. A pure-interrupt mode may be worthwhile for computationally intensive workloads that send large messages, provided that the slightly increased transfer delay isn't a concern.

On hardware that supports CORE-Direct we can offload an entire transfer sequence as a partially-ordered graph of asynchronous requests. Here, our preliminary experiments were only partially successful: a firmware bug (a NIC hardware issue) prevented us from testing our full range of protocols. Figure 12 shows results for chain send, where the request pattern is simple and the bug did not occur. The left graph uses a hybrid of polling and interrupts, while the right graph uses pure interrupts. As seen in the graphs, cross-channel generally provides a speedup of about 5%, although there is one scenario (a single sender transmitting in groups of size 5-8, in polling-only mode) in which our standard RDMC solution wins.

5.3 Future Work: RDMC on TCP

When Ganesan and Seshadri first explored multicast overlay topologies, they expressed concern that even a single lagging node might cause cascading delay, impacting every participant and limiting scalability [7]. This led them to focus their work on dedicated, synchronous, HPC settings, justifying an assumption that nodes would run in lock-step and not be exposed to scheduling delays or link congestion.

However, today's RDMA operates in multi-tenant environments. Even supercomputers host large numbers of jobs, and hence are at risk of link congestion. RDMA in standard Ethernet settings uses a TCP-like congestion control (DCQCN or TIMELY). Yet we do not see performance collapse at scale. Our slack analysis suggests a possible explanation: the binomial pipeline generates a block-transfer schedule in which there are opportunities for a delayed node to catch up. As we scale up, delays of various kinds do occur. Yet this slack apparently compensates, reducing the slowdown.

The observation has an interesting practical consequence: it suggests that RDMC might work surprisingly well over high speed datacenter TCP (with no RDMA), and perhaps even in a WAN network. In work still underway, we are porting RDMC to access RDMA through LibFabrics from the OpenFabrics Interface Alliance (OFI) [16]. LibFabrics is a mature solution used as the lowest layer of the message passing interface (MPI) library for HPC computing. The package uses a macro expansion approach and maps directly to RDMA as well as to other hardware accelerators, or even standard TCP. When the port is finished, we plan to closely study the behavior of RDMC in a variety of TCP-only settings.

6 Related Work

Replication is an area rich in software libraries and systems. We've mentioned reliable multicast, primarily to emphasize that RDMC is designed to replicate data, but is not intended to offer the associated strong group semantics and multicast atomicity. Paxos is the most famous state machine replication (consensus) technology. Examples of systems in this category include the classical Paxos protocol itself, our Derecho library, libPaxos, Zookeeper's ZAB layer, the head-of-log mechanism in Corfu, DARE, and APUs [1, 9, 10, 12, 13, 18, 24]. Derecho demonstrates that RDMC can be useful in Paxos solutions, but also that additional mechanisms are needed when doing so: RDMC has weaker semantics than Paxos.

We are not the first to ask how RDMA should be exploited in the operating system. The early RDMA concept itself dates to a classic paper by Von Eicken and Vogels [23], which introduced the zero-copy option and reprogrammed a network interface to demonstrate its benefits. VIA, the virtual interface architecture then emerged; its "Verbs" API extended the UNet idea to support hardware from Infiniband, Myrinet, QLogic and other vendors. The Verbs API used by RDMC is widely standard, but other options include the QLogic PSM subset of RDMA, Intel's Omni-Path Fabric solution, socket-level offerings such as the Chelsio WD-UDP [3] embedding, etc.

Despite the huge number of products, it seems reasonable to assert that the biggest success to date has been the MPI platform integration with Infiniband RDMA, which has become the mainstay of HPC communications. MPI itself actually provides a multicast primitive similar to the one described in this paper, but the programming model imposed by MPI has a number of limitations that make it unsuitable for the applications that RDMC targets: (1) send patterns are known in advance so receivers can anticipate the exact size and root of any multicast prior to it being initiated, (2) fault tolerance is handled by checkpointing, and (3) the set of processes in a job must remain fixed for the duration of that job. Even so, RDMC still outperforms the popular MVAPICH implementation of MPI by a significant margin.

Broadcast is also important between CPU cores, and the Smelt library [11] provides a novel approach to address this challenge. Their solution is not directly applicable to our setting because they deal with tiny messages that don't require the added complexity of being broken into blocks, but the idea of automatically inferring reasonable send patterns is intriguing.

Although our focus is on bulk data movement, the core argument here is perhaps closest to the ones made in recent operating systems papers, such as FaRM [5], Arrakis [17] and IX [2]. In these works, the operating system is increasingly viewed as a control plane, with the RDMA network treated as an out of band technology for the data plane that works best when minimally disrupted. Adopting this perspective, one can view RDMC as a generic data plane solution well suited to out-of-band deployments. A recent example of a database optimized to use RDMA is Crail [20].

7 Conclusions

Our paper introduces RDMC: a new reliable memory-to-memory replication tool implemented over RDMA unicast.

RDMC is available for download as a free, open-source library, and should be of direct use in O/S services that currently move objects either one by one, or over sets of side-by-side TCP links. The protocol can also be used as a component in higher level libraries with stronger semantics.

RDMC performance is very high when compared with the most widely used general-purpose options, and the protocol scales to large numbers of replicas. RDMC yields a benefit even if just 3 replicas are desired. In fact replication turns out to be remarkably inexpensive, relative to just creating one copy: one can have 4 or 8 replicas for nearly the same price as for 1, and it takes just a few times as long to make hundreds of replicas as it takes to make 1. Additionally, RDMC is robust to delays of various kinds: Normal network issues of data loss and duplication are handled by RDMA while RDMC's block-by-block sending pattern and receiver-receiver relaying compensate for occasional scheduling and network delays. The RDMC code base is available for download as part of the Derecho platform (https://GitHub.com/Derecho-Project).

Acknowledgements

We are grateful to the DSN reviewers, Michael Swift, and Heming Cui. LLNL generously provided access to its large computer clusters, as did the U. Texas Stampede XSEDE computing center. Additional support was provided by DARPA under its MRC program, NSF, and AFOSR. Mellanox provided high speed RDMA hardware.

References

[1] Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst. 31*, 4 (Dec. 2013), 10:1–10:24.

[2] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 49–65.

[3] Low latency UDP Offload solutions | Chelsio Communications. `http://www.chelsio.com/nic/udp-offload/`. Accessed: 24 Mar 2015.

[4] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing Data Transfers in Computer Clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 98–109.

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

[6] Ed Harris. It's all about big data, cloud storage, and a million gigabytes per day. `https://blogs.bing.com/jobs/2011/10/11/its-all-about-big-data-cloud-storage-and-a-million-gigabytes-per-day`, Oct. 2011.

[7] Ganesan, P., and Seshadri, M. On Cooperative Content Distribution and the Price of Barter. In *25th IEEE International Conference on Distributed Computing Systems, 2005. ICDCS 2005. Proceedings* (June 2005), pp. 81–90.

[8] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[9] Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., Zink, S., Birman, K. P., and van Renesse, R. Building smart memories and cloud services with derecho, 2017.

[10] Junqueira, F. P., and Reed, B. C. The Life and Times of a Zookeeper. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2009), SPAA '09, ACM, pp. 46–46.

[11] Kaestle, S., Achermann, R., Haecki, R., Hoffmann, M., Ramos, S., and Roscoe, T. Machine-aware atomic broadcast trees for multicores. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 33–48.

[12] Lamport, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[13] LibPaxos: Open-source Paxos. `http://libpaxos.sourceforge.net/`. Accessed: 24 Mar 2015.

[14] Mellanox Corporation. CORE-Direct: The Most Advanced Technology for MPI/SHMEM Collectives Offloads. `http://www.mellanox.com/related-docs/whitepapers/TB_CORE-Direct.pdf`, May 2010.

[15] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 537–550.

[16] OpenFabrics Interfaces (OFI). LibFabric: Open-Source Library for Exploiting Fabric Communication Services. `https://ofiwg.github.io/libfabric/`. Accessed: 11 Apr 2018.

[17] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 1–16.

[18] Poke, M., and Hoefler, T. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2015), HPDC '15, ACM, pp. 107–118.

[19] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout Ali Ghodsi, Michael J. Franklin, Benjamin Recht, Ion Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale.

[20] Stuedi, P., Trivedi, A., Pfefferle, J., Stoica, R., Metzler, B., Ioannou, N., and Koltsidas, I. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Bulletin of the Technical Committee on Data Engineering, Special Issue on Distributed Data Management with RDMA 40* (2017), 40–52.

[21] van Renesse, R., and Schneider, F. B. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 7–7.

[22] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[23] von Eicken, T., Basu, A., Buch, V., and Vogels, W. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 40–53.

[24] Wang, C., Jiang, J., Chen, X., Yi, N., and Cui, H. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of the Eighth ACM Symposium on Cloud Computing* (Santa Clara, CA, USA, Sept. 2017), SoCC '17, ACM.

[25] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 523–536.