Pushing Bytes: Cloud-Scale Data Replication with RDMC

Jonathan Behrens* Sagar Jha† Edward Tremel† Ken Birman†
*Cornell University and MIT †Cornell University

Abstract

Data center infrastructures frequently replicate objects to create backups or to copy executables and input files to compute nodes. This task occurs under time pressure: data is at risk of loss until replicated for fault-tolerance, and in the case of parallel processing systems like Spark [33], useful computation can't start until the nodes all have a copy of the executable images. Cloud elasticity creates a similar need to rapidly copy executables and their inputs. To address these needs, we introduce RDMC: a fast reliable data replication protocol that implements multicast as a pattern of RDMA unicast operations, which maximizes concurrency while minimizing unnecessary transfers. RDMC can be used in any setting that has RDMA or a software RDMA emulation. Our focus is on use of replication as an element of the data center infrastructure. We evaluate overheads for the hardware-supported case using microbenchmarks and heavy-load experiments, and also describe preliminary experiments using a technique that offloads the entire data transfer pattern into the NICs, further reducing latency while freeing server resources for other tasks.

1 Introduction

Data center 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) [15], and the throughput is far higher today. All of these writes must be split into chunks, and replicated to several storage servers [17]. 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 [12]. Similarly, copying executables accounts for a significant portion of the startup time of large compute tasks, and speeding up this process would directly reduce overall task completion times. 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 [31]. In some cases, image copying takes far more time than computation [28].

Despite the importance of fast replication, effective general-purpose solutions do not exist. Today, cloud middleware systems typically push new data in ways that make one copy at a time, for example when copying VM images prior to launching a parallel task on a large number of nodes. 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, even if the scheduler can anticipate that a collection of tasks will read the same file, data movement still occurs only on demand.

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, lower delays can improve end-user experience and will be of growing importance as the cloud increasingly performs real-time control of large data-dependent physical systems, such as smart power grids and roadways.

Our new reliable RDMA multicast protocol, RDMC, solves this basic replication problem, emphasizing raw speed. This does not preclude layering stronger semantics over the solution: in work still underway, we are exploring the integration of RDMC into a library that supports reliable multicast with strong group membership semantics (as in [8]). When the library is configured for data persistence, we obtain a version of Paxos [21] that moves all data through RDMC. Other opportunities include use of RDMC for replication in systems that manage ordered append-only logs such as Corfu [4], key-value stores [14, 20, 25], or transactions [5]. However, the present paper limits itself to stand-alone scenarios.

RDMC makes the following contributions:

- It provides insights on how to efficiently use RDMA to conduct multicast transfers.
- It demonstrates multicast protocols that achieve stable and exceptionally high performance, outperforming even the heavily optimized broadcast primitive included in the MVAPICH library for HPC.
- It evaluates options for eliminating polling and copying, which free compute resources for other tasks.

- It explores the potential for offloading extended asynchronous transfer sequences into the NIC: “compiling” protocols directly into a form that can be executed by hardware.

2 Background on RDMA

RDMA (remote direct-memory access) is a zero-copy communication standard supported on a wide range of hardware (as well as by the Soft RoCE software implementation). RDMA is a user-space networking solution, accessed by creating what are called *queue pairs:* lock-free data structures shared between user code and the network controller (NIC), each consisting of a send queue and a receive queue. 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. If the receiver’s queue is empty when a send is attempted, the sender either retries later or issues a failure. A queue pair also has an associated completion queue which is used by the NIC to report the successful completion of transfers, as well as any detected faults.

RDMA supports several modes of operation. RDMC uses *two-sided* RDMA operations, which behave similarly to TCP: the sender and receiver bind their respective queue pairs together, creating a session fully implemented by the NIC endpoints. In this mode, 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. This can be remarkably fast. For example, in our lab, two-sided RDMA rates frequently approach the full 100Gb/s of the optical layer, a rate very difficult for IP protocols to match, and indeed more than 3x what single threaded `memcpy` can achieve for memory-to-memory copying internal to the nodes of our compute cluster (the memory module on our servers has 256Gbps total bandwidth but that bandwidth is divided among 16 cores). The gap may grow: RDMA speedup to 1Tb/s networking is expected within a decade, whereas per-core memory speeds are not expected to grow nearly as fast.

With today’s devices, two-sided RDMA is as reliable as a memory bus: data will never be corrupted during transfer, and the send-order is preserved. Thus there is no need for end-to-end acknowledgments or retransmissions, or even for checking the received data. If a hardware fault or an endpoint crash occurs, the hardware reports the failure and breaks the two-sided session.

RDMA emerged as an Infiniband technology, but is no longer limited to Inifiniband fabrics: RDMA over Converged Ethernet (RoCE) has become widely available on fast Ethernet NICs and routers. RoCE is not yet fully mature: early versions have suffered from limited scalability due to instability of the priority pause-frame feature (PPF) used for link-layer congestion management leading to data losses. Fortunately, recent advances have yielded two promising alternatives to PPF: DCQXN [34] and TIMELY [26]. Both are stable even at very large scale, and their performance is similar to that of Infiniband. A pure software implementation (Soft RoCE) enables RDMA software to be used on platforms that lack hardware support.

RDMA operations are asynchronous, hence the question arises of how to detect completions. One option is to dedicate a polling thread to monitor the completion queue. This minimizes latency but pins a core. A second option uses interrupts that fire when a completion occurs, but this can be slow. A third approach is to combine the two, using the interrupt mechanism most of the time, but switching to polling while a high rate of transfers is underway.

Although not used in RDMC, RDMA also supports reliable *one-sided* reads and writes, where one endpoint grants the other permission to perform RDMA reads or writes into a pre-prepared memory region. The initiator of a read or write will see a completion event, but the target isn’t notified at all. Finally, RDMA offers *unreliable datagrams*, including a feature that resembles IP multicast. However these leave it to the user to deal with message loss or reordering, and would require occasional retransmissions. Worse, they limit transfer size to the network MTU – at most a few kilobytes – so large messages would be split into thousands of chunks that could arrive out of order, requiring the receiver to reassemble again at the end.

RDMA resource management considerations introduce additional issues. Once an RDMA NIC starts doing a remote DMA operation, it must dedicate resources to the operation until the transfer finishes, and for very large operations this could potentially take a long time (the wire-level MTU does not limit the size of a DMA transfer, so even with a small MTU very large DMA transfers can occur). Operators of large multi-tenant cloud platforms have expressed concern that such data patterns could cause starvation if the NIC lacks a sufficient capacity for concurrent transfers. Accordling, very large transfers are typically broken into smaller ones (blocks) of a few hundred KB each, guaranteeing that the NIC will treat requests by other users fairly.

An additional concern is that RDMA NICs have limited memory capacity. Each connected queue pair consumes resources when actively sending or receiving. Some applications (notably, key-value systems [14]) struggle with these limits because over time, any node in an RDMA key-value system interacts with every other node. NIC memory is also consumed to cache DMA mappings, and non-contiguous IO operations can exhaust this limit.

In these respects, data replication is a relatively easier task, because replication patterns correspond to trees. A node will typically interact with just a few neighbors, and once a replication activity ends, the connections can be closed. Even with extensive concurrent use by overlapping multicast groups, the number of active connections never approaches the NIC limits, which are in the low thousands. The memory mapping issue is easily addressed by mapping a large amount of contiguous node memory, then allocating memory for messages within this pool. In future multi-tenant systems, one could go even further, by setting aside a shared RDMA memory region, then exposing it through a secure mapping as needed. One could also use large pages, as recommended by the developers of the FaRM key-value system.

Experimental Features RDMA guarantees two forms of request 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 [24] feature lets the application specify a third form of request ordering: RDMA send operations that first wait for completion of an RDMA receive, which could occur on a different queue pair. For example, suppose that node N issues a send on a queue pair bound to node M. M can post the receive request to accept the data from N together with a send that relays the same data to node O, specifying that the send should occur only after the receive completes. By doing so, the M avoids the need to wait for the receive complete before posting the relaying operation.

CORE-Direct is not widely used, but it intrigued us because in some situations it is actually possible to precompute data-flow graphs describing the full pattern of data movement for each multicast send. For example, members of a replication group could potentially post dataflow graphs at the start of a transfer, linked by cross-node write/receive dependencies. The hardware would then carry out the whole transfer without further help. In contrast, were the same task performed using the standard RDMA API, scheduling delays (i.e. between receives and subsequent sends) could easily accumulate to have a significant downstream impact.

3 High level RDMC summary

We implemented RDMC using the two-sided RDMA operations described above, with a compile-time switch to determine whether or not CORE-Direct features should be used (eventually, we hope to automate selection of the best setup at runtime). 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 (similar to BitTorrent except that for RDMC, the pattern of transfers is deterministic). 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. The sender and receivers first create a multi-way binding: an *RDMC group*. This occurs out of band, using TCP as a bootstrapping protocol. To minimize delay on the critical path, applications that expect to do repeated transfers can set up groups ahead of time and then reuse them.

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. When CORE-Direct is available, we enqueue the entire sequence at once.

3. On the receive side, RDMC will notify the user application of an incoming message, at which point it must post a buffer of the correct size to receive it into.

4. Sends complete in the order they were initiated. Incoming messages are guaranteed to not be be corrupted, arrive out of order, or get duplicated.

5. Multicast groups are isolated from each other such that any group is unaffected by the failure of nodes not in that group.

6. Any failures sensed by RDMA are reported to the application via a single event, after which no further operations occur in that group. (Of course, the nonfaulty members can always create a new group and resume communications in it.)

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, but group creation requires some elaboration.

Our design is such that the `create_group` function should be called concurrently (and with identical membership information) by all group members, initiated by

```
// 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

some out-of-band method. It takes two callback functions which are used to notify the application of events. The `incoming_message_callback` is triggered by receivers when a new transfer is started, and is used to obtain a memory region to write the message into. Because memory registration is an expensive operation, it should ideally take place in advance.

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.

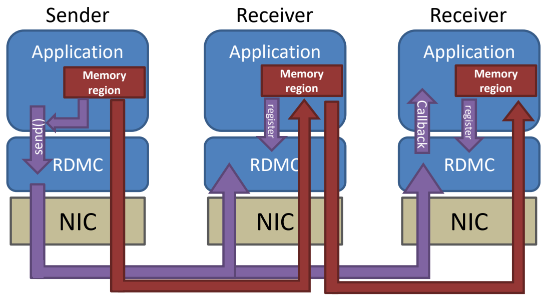


Figure 2: RDMC with a sender and 2 receivers.

Initialization Before an application can participate in RDMC transfers, it must go through a setup process to create an RDMC group. During this stage, RDMC exchanges connection information with all other nodes that may participate (using an existing channel, such as TCP/IP), creates queue pairs, and prepares internal data structures. Finally, it starts a thread that monitors RDMA completion queues for notifications about incoming and outgoing messages. If several RDMC transfers are underway concurrently, each determines its own block-transmission sequence, but separate transfers can share the same completion thread, reducing overheads. This thread seeks a balance between responsiveness and low overhead: it polls for 50 ms after each completion event, but switches to waiting for completion-triggered interrupts if there has been no activity.

Data Transfer Although we will turn out to be primarily focused on the *binomial pipeline* algorithm (described later), RDMC actually implements several data transfer algorithms, which makes possible direct side-by-side comparisons. To be used within RDMC, a sending algorithm must be deterministic, and if a sender sends multiple messages, must deliver them in sequential order. 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 more blocks will be sent, the receiver can post additional asynchronous receives as needed.

The sender and each receiver can now 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

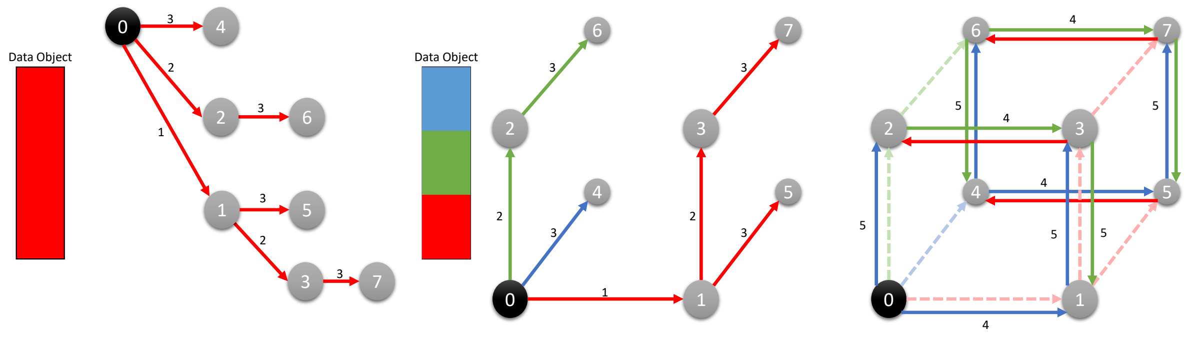


Figure 3: (Left) A standard binomial tree multicast, with the entire data object sent in each transfer. (Center) A binomial pipeline multicast, with the data object broken into three blocks, showing the first three rounds of the protocol. In this phase, the sender sends a different block in each round, and receivers forward the blocks they have to their neighbors. (Right) The final two rounds 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.

schedules are those that make sure all the nodes spend as much time concurrently sending and receiving as possible. 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 only starts when 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. It implements the naive 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. Modern RDMA NICs have a full bidirectional capability: a 100Gbps NIC can potentially send and receive 100Gbps concurrently. Thus sequential creates a hot spot at the sender, and leaves a great deal of potentially unused *inner bandwidth* between the replicas.

Binomial Tree Better performance is possible if receivers relay the message once they get it. This pattern, a binomial tree can be seen in Figure 3 (left); the labels on the arrows represent the asynchronous time step. Here, sender 0 starts by sending the entire 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. Thus many nodes are idle for the majority of the time, wasting the bandwidth of their incoming and outgoing links.

Chain Send This algorithm implements a bucket-brigade, similar to the chain replication scheme described in [30]. 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 Pipeline One might expect that the lowest possible latency would be that of a binomial tree used to transmit blocks instead of entire messages, but in fact one can do even better. This observation was first made by Ganesan and Seshadri [16], who proposed a binomial pipeline algorithm that combines bucket brigade and binomial trees. The algorithm works by creating a 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). This algorithm is exceptionally efficient because it ensures that all nodes spend as much time as possible simultaneously sending and receiving blocks. The details are described in Appendix A, along with the changes we made to turn Ganesan and Seshadri's synchronous solution into an asynchronous protocol, and to tune it to better match our setting.

The binomial pipeline makes balanced use of bandwidth: except for the initial and final steps, every node other than the sender maintains a steady load, sending and receiving one block on each logical step. Latency skew is 1: all replicas receive their final blocks simultaneously (if the number of replicas is a power of two), or on consecutive logical timesteps (if not).

4.3.1 Hybrid Algorithms

Lacking detailed data-center topology information, the binomial pipeline normally offers the best mix of latency and performance. Nonetheless, there may be situations in which other options are preferable.

Many data centers 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). While this works well until we reach a genuinely large scale, it isn't the best possible data relaying pattern.

In contrast, suppose that we were to use chain send in the top of rack layer, designating one node per rack as the leader for its rack. This would require some care: in our experiments chain send was highly sensitive to network topology and data pattern. However, a properly configured TOR chain would minimize load on the top of rack switching network: any given block would traverse each TOR switch exactly once. Then we could use the binomial pipeline within each rack.

Equally interesting would be 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 Leveraging CORE-Direct

In Section 2, we discussed the experimental Mellanox CORE-Direct feature. Because the transfer schedule is deterministic, RDMC can precompute the full set of operations required for each multicast, enqueuing an entire graph of partially ordered operations at each participating node. We implemented this idea and evaluate it below.

5 Experiments

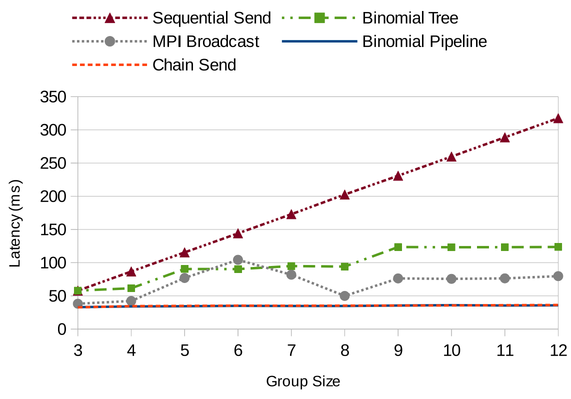
Our goal is to show that not only does RDMC have high performance and low overhead, but also that it scales well and can handle real world workloads.

5.1 Experimental Setup

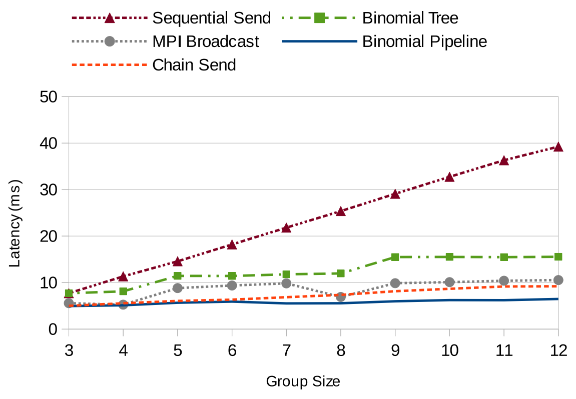
Fractus We conducted experiments on several clusters, beginning with our own internal Fractus cluster. Fractus contains 16 RDMA-enabled nodes running Ubuntu 16.04, each equipped with a 4x QDR Mellanox NIC and 94 GB of DDR3 memory. All nodes are connected to both a 100Gbps Mellanox IB switch and a 100Gbps Mellanox RoCE switch, and have one-hop paths to one-another.

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 The U. Texas Stampede 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.

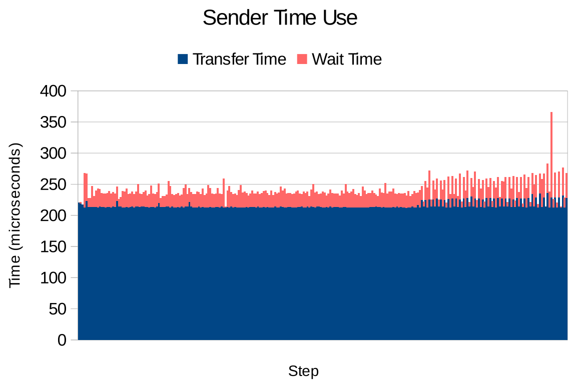


(a) 256 MB multicasts



(b) 8 MB multicasts

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



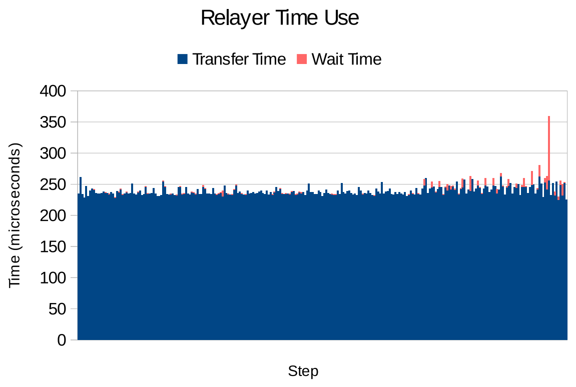


Figure 5: Breakdown of transfer time and wait time of two nodes taking part in the 256 MB transfer from the experiment used to create Table 1. Notice that the relaying node spends hardly any time waiting, while the sender transmits each block slightly faster (since it isn't receiving at the same time) and then must wait for the other nodes to catch up.

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. 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 seperate 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 $1.03\times$ to $3\times$ 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

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.

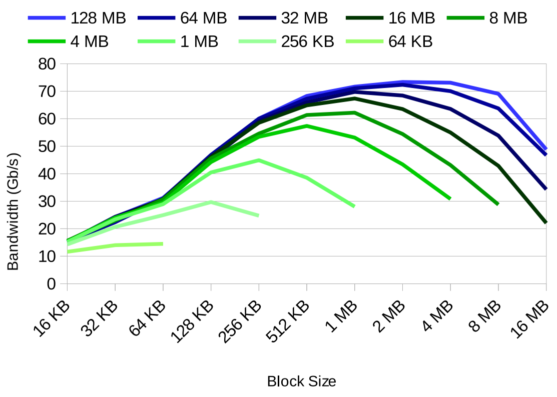


Figure 6: Multicast bandwidth (computed as the message size divided by the latency) on Fractus across a range of block sizes for messages of size between 16 KB and 128 MB, all for groups of size 4.

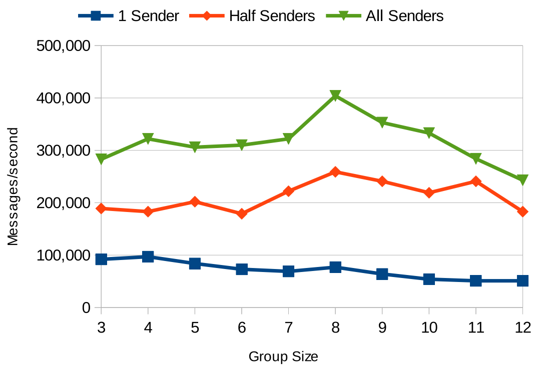


Figure 7: 1 byte messages/sec. (Fractus)

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.

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 propor-

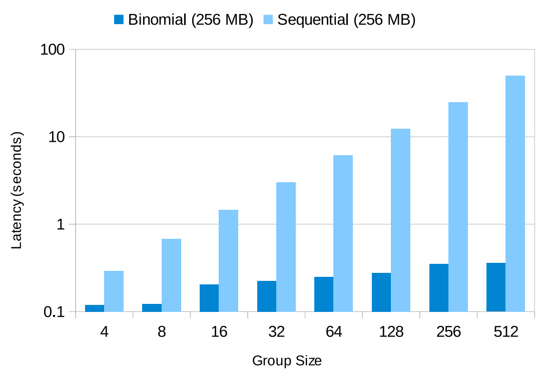


Figure 8: Total time for replicating a 256MB object to a large number of nodes on Sierra.

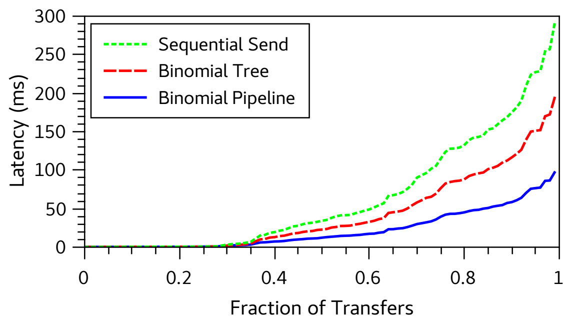
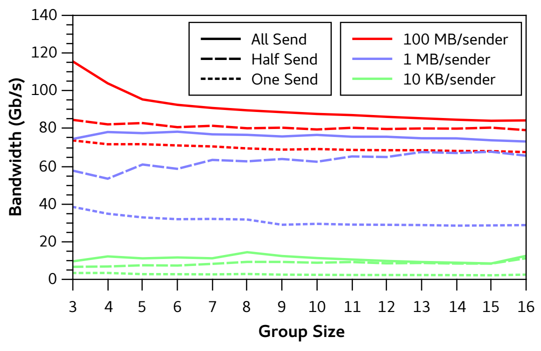
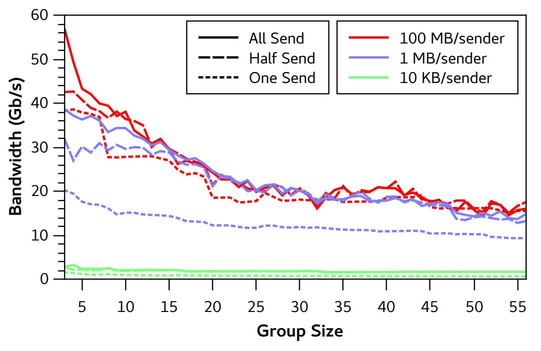


Figure 9: Distribution of latencies when simulating the Cosmos storage system replication layer.



(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).

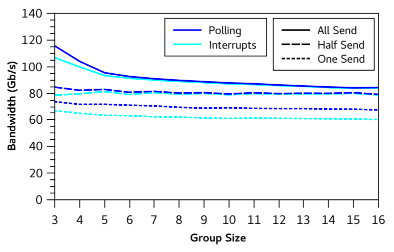
tional to the amount of time spent transferring an individual block. There is also additional overhead incurred when there are not enough blocks in the message for all nodes to get to contribute meaningfully to the transfer.

Finally, Figure 7 measures the number of 1 byte messages 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.

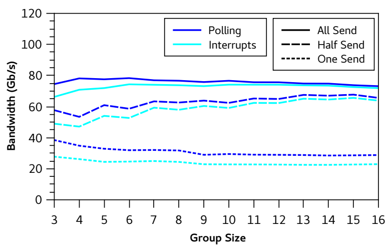
5.2.2 Scalability

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.

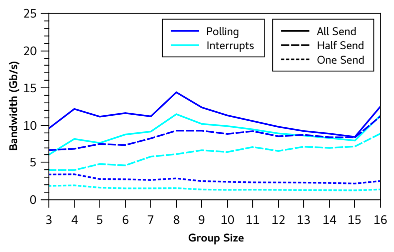
Although we did not separately graph end-of-transfer time, binomial pipeline transfers also complete nearly simultaneously: this minimizes temporal skew, which is important in parallel computing settings because many such systems run as a series of loosely synchronized steps that end with some form of 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



(a) 100MB Transfers

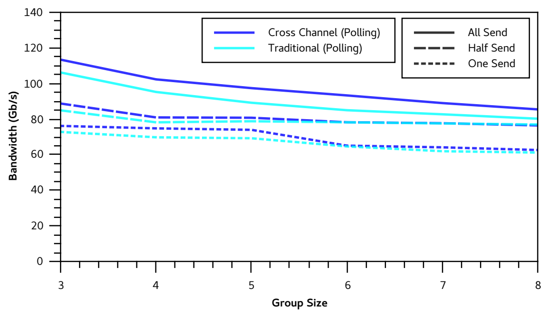


(b) 1MB Transfers



(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.



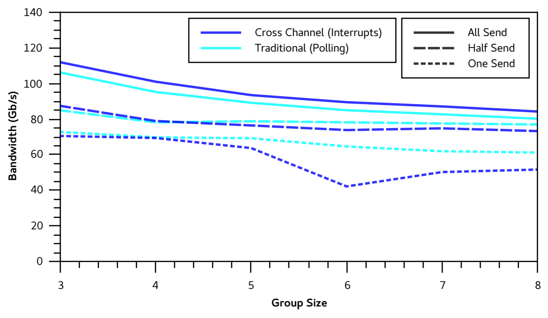


Figure 12: CORE-Direct experiment using a chain multicast protocol to send a 100 MB message. The left is a run using hybrid polling/interrupts; on the right is a run with purely interrupts. Both experiments were on Fractus.

that should run in loosely synchronized stages.

Next, we set out to examine the behavior of RDMC in applications that issue large numbers of concurrent multicasts to overlapping groups. For a first experiment we obtained a trace sampled from the data replication layer of Microsoft's Cosmos system, a production data warehouse used by the Bing platform. Cosmos as currently deployed runs on a TCP/IP network and makes no use of RDMA or multicast. The trace has several million 3-node writes, with varying object sizes and random target nodes.

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 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. The key take-away is that we are running at nearly the full bisection capacity of Fractus, and with absolutely no duplicative data transiting any network link: the RDMC data pattern is highly efficient for this sort of production workload.

A second experiment looked at group overlap in a more regular manner, using a single 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 three experiments, varying the number of senders to that set of group members. (1) In the experiment corresponding to the solid line, all members are senders (hence we would have, for example, 8 perfectly overlapped groups, each with the same members, but each having 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 with just one sender. To carry out the experiment, 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 rates (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 asymptotes to 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 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 driver bug (not in our code) 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.

5.3 Discussion

When Ganesan and Seshadri considered tree and chain topologies for performing multicast in [16] they thought them to be unfeasibly slow over TCP/IP. This is an interesting question for us, because RDMA can be understood as a hardware implementation of a TCP-like protocol. In their discussion, Ganesan and Seshadri predicted suboptimal performance, attributing this to a concern that highly structured topologies can allow a single lagging node to slow down the entire send for everyone. The binomial pipeline algorithm (which they recognized as theoretically optimal) is more susceptible to this phenomenon because each node is responsible for the transfer to all of its neighbors in the hypercube.

As we have seen, in our asynchronous implementation of their scheme, slowdown proves to be much less of an issue for RDMA than for TCP/IP over lossy Ethernet. With true hardware-supported RDMA we are able to achieve low latency, reliable transfers directly into user-space memory on the receiver, with no copying, which is important because `memcpy` peaks at 30 Gb/s per core and is not likely to scale up as quickly as optical network speeds will. By contrast, thanks to hardware support for reliable sends we are able to consistently get nearly line rates across a range of systems using reliable point-to-point sends, and this should track the evolution of optical network speeds. Thus the opportunity for application-induced scheduling delays is much reduced, and the size of such delays is also much smaller than in their analysis.

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. Good examples of systems in this space include Isis2/Vsync, Spread, Totem, Horus, Transis and the Isis Toolkit [2,3,7,9,13,29].

Paxos is the most famous of the persistent replication solutions, and again, RDMC is not intended as a competitor. But examples of systems in this category include Paxos, Chubby, Rambo, Zookeeper and Corfu [1,4,10,18,21–23].

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 [32], 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, Myranet, 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 [11] 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 [19] 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 [14], Arrakis [27] and IX [6]. 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.

7 Conclusion

Our paper introduces RDMC: a new reliable memory-to-memory replication tool implemented over RDMA unicast. Performance is very high when compared with the most widely used general-purpose options, and the protocol scales to large numbers of replicas. When replicating data to create backups, RDMC already 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. We believe this to be a striking finding, and of potentially broad applicability. Furthermore, because RDMC delivery is nearly simultaneous even within large groups of receivers, applications that need to initiate parallel computation will experience minimal skew in their task start times. RDMC has the potential to dramatically accelerate and simplify a wide range of important applications, while improving utilization of data center computing infrastructures.

Appendix A: Binomial Pipeline

The binomial pipeline was first described by [16]. The algorithm assigns each node to a single vertex on a hypercube. When the group size is a power of two, each node is assigned to its own vertex. Otherwise, some pairs of nodes must share a vertex. A vertex behaves externally as a single node: at any point it is sending and receiving at most one block from another vertex. However, as we will discuss later, nodes occupying the same vertex exchange blocks among themselves to ensure that they all receive the full message.

The binomial pipeline proceeds in three stages, each of which are further divided into steps. During every step, all vertices with at least one block have one of their members send across parallel edges of the hypercube. At the start of the first stage the sender transfers one block of the segment to a receiver. In the next step of the first stage, the sender transfers a *different* block to receiver in another vertex, while the first receiver simultaneously sends its block on to a third vertex. This pattern continues until all vertices have a single block.

Now that all nodes have a block, the second stage can be much more efficient. Previously we were wasting most of the network capacity because at each step every node was either a sender or a receiver but not both. In this stage, the sender continues to send successive blocks while all other vertices trade their highest numbered blocks. Once the sender runs out of blocks, the algorithm enters the final stage. The sender repeatedly sends the last block, while the rest of the vertices continue to trade blocks.

The progression of the binomial pipeline for a group of 8 nodes is illustrated in Figure 3, and contrasted with a more traditional binomial tree broadcast. It is worth noting that if the binomial pipeline is run with only a single block, it will produce a binomial tree.

All that remains is to discuss the interactions within vertices containing two nodes. Whenever a vertex is responsible for sending a block, exactly one of the nodes within it will have that block. During that step, the other node will send a block that only it has to its partner and receive the incoming block (if any) for the vertex. When all vertices have all blocks, the nodes within them exchange the final block they are missing, completing the send.

Our implementation of the binomial pipeline in RDMC is the first adaptation of this technique to an RDMA environment (the work described in [16] was evaluated purely with a simulation). This entailed several extensions: (1) Our implementation doesn't need to know global state or to compute the whole schedule. Instead it just computes the parts relevant to each individual node. Further, whereas the original version has a stage at which nodes gossip about which nodes have what blocks, we were able to eliminate that step entirely. (2) RDMC adjusts the algorithm to allow some nodes to run slightly ahead of others. The resulting small degree of asynchronicity eliminated stalls that otherwise were seen in the originally, fully synchronized protocol. (3) To minimize RDMA connection setup overhead, we adjusted the schedule to ensure that the first block each node receives always comes from the same relayer.

References

[1] Abraham, I., Chockler, G. V., Keidar, I., and Malkhi, D. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2004), PODC '04, ACM, pp. 226–235.

[2] Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M., and Budhia, R. K. The Totem Multiple-ring Ordering and Topology Maintenance Protocol. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 93–132.

[3] Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., and Stanton, J. The Spread Toolkit: Architecture and Performance. *Johns Hopkins University, Center for Networking and Distributed Systems (CNDS) Technical report CNDS-2004-1* (Oct. 2004).

[4] 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.

[5] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 325–340.

[6] 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.

[7] Birman, K. Isis2 Cloud Computing Library. `https://isis2.codeplex.com/`, 2010.

[8] Birman, K. *Guide to Reliable Distributed Systems*. No. XXII in Texts in Computer Science. Springer-Verlag, London, 2012.

[9] Birman, K. P., and Joseph, T. A. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1987), SOSP '87, ACM, pp. 123–138.

[10] Burrows, M. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

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

[12] 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.

[13] Dolev, D., and Malki, D. The Transis Approach to High Availability Cluster Communication. *Commun. ACM 39*, 4 (Apr. 1996), 64–70.

[14] 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.

[15] 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-mil` Oct. 2011.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

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

[22] LAMPORT, L., MALKHI, D., AND ZHOU, L. Reconfiguring a State Machine. *SIGACT News 41*, 1 (Mar. 2010), 63–73.

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

[24] 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.

[25] MITCHELL, C., GENG, Y., AND LI, J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 103–114.

[26] 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.

[27] 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.

[28] SHIVARAM VENKATARAMAN, AUROJIT PANDA, KAY OUSTERHOUT ALI GHODSI, MICHAEL J. FRANKLIN, BENJAMIN RECHT, ION STOICA. Drizzle: Fast and Adaptable Stream Processing at Scale.

[29] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Commun. ACM 39*, 4 (Apr. 1996), 76–83.

[30] 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.

[31] 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).

[32] 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.

[33] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 10–10.

[34] 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.