# Pushing Bytes: Cloud Scale Big-Data Replication with RDMC

Jonathan Behrens Ken Birman Sagar Jha Edward Tremel
*Department of Computer Science, Cornell University*

## Abstract

Cloud computing frameworks replicate large objects for diverse reasons, often under time-pressure. RDMC (Reliable DMA Multicast) is a reliable data replication protocol that runs at exceptionally high speeds and low latencies, implementing multicast as a pattern of RDMA unicast operations using a novel approach that maximizes concurrency. Users with knowledge of datacenter topology can configure RDMC to use a pattern of data flow matched to the network. In networks with full bisection bandwidth our fastest protocol creates a collection of binomial trees and embeds them on a hypercube overlay, setting speed records while also achieving exceptionally low delivery-time skew.

## 1 Introduction

Large-scale computing systems often must replicate content among groups of nodes. The need is widespread in cloud computing infrastructures, which copy huge amounts of data when reprovisioning nodes and must update VM and container images each time software is patched or upgraded. Replication patterns are also seen when a task is spread over a set of nodes in parallel machine-learning applications, and when a file or database is accessed by first-tier nodes charged with rapidly responding to client requests. With the growing embrace of IoT, immersive VR and multi-player gaming, replication must satisfy rigid time constraints.

Yet we lack effective general-purpose solutions. Today, cloud infrastructures typically push new images using specialized file copying tools. Content sharing is often handled through some kind of intermediary caching or key-value layer. Even if a scheduler like Hadoop can anticipate that a collection of tasks will read the same file, and tries to launch jobs where the data is cached, if those files are not present in the cache the applications just pull data on a one-by-one basis.

Cloud systems could derive substantial efficiencies by recognizing such interactions as instances of a common pattern, and our hope is that with a standard solution such as RDMC in hand, developers will chose to do so. However, the goal of high speed replication can lead in two distinct directions. For the present paper, our focus is on *reliably pushing bytes:* given an initiator with a source object, (1) making large numbers of replicas as rapidly as possible while leveraging RDMA, (2) optimizing for large objects and time-critical data transfer situations, and (3) ensuring that if failure occurs, the application will be notified and can retry the transfer.

The other direction includes offering stronger properties, perhaps packages as a reliable multicast with strong group membership semantics [11] or an implementation of Paxos [21]. Such systems go beyond RDMC by imposing an ordering on concurrent updates that conflict. They may also persist data into append-only logs [6] or introduce data semantics, for example by focusing on support for key-value stores [17, 20, 23], or a transactions [7]. We believe that RDMC could be useful in systems with such goals (and indeed are exploring that option in our own continuing work), but stronger semantics require synchronization protocols, which can conflict with the goal of achieving the highest possible data rates.

RDMC is an open-source project, written in C++ and accessed through a simple library API. The basic functionality is that of a zero-copy multicast, moving data from a memory region in the source to memory regions designated by the receivers. If an application maps files into memory, RDMC can transfer persistent data from the sender, and similarly one can persist received data by mapping a file and calling *fsync*. As we look to the future, RDMC should eventually be able to integrate directly with video capture devices, and with new kinds of persistent storage technologies like 3D-XPoint.

We've tested RDMC on a variety of platforms. RDMA hardware is increasingly common in data center settings, and is widely supported by high speed switches and NICs. The feature was first offered on Infiniband networks, but is now available on fast Ethernet (RoCE), and a software emulation (SoftRoCE) is available for portability to settings where RDMA hardware is lacking. User-level access to RDMA is possible in non-virtualized Linux and Windows systems, and Microsoft's Azure platform now supports RDMA in its containerized (Mesos) mode.

We start in Section 2 with a precise statement of goals. In Section 3 we describe the RDMC data dissemination options currently supported. By supporting multiple pro-

tocols, RDMC facilitates side-by-side comparisons; in our experimental cluster, a new protocol we call the *binomial pipeline* dominates.

## 2 Goals

RDMA (remote direct-memory access) is a zero-copy communication standard supported on a wide range of hardware (as well as by the SoftRoCE software). RDMA is a user-space solution, accessed by creating what are called queue-pairs: lock-free data structures shared between user logic and the network controller (NIC) consisting of send queue and a receive queue. A send is issued by posting a memory region to the send queue, and receive destinations are indicated similarly. If the receiver hasn't posted a receive buffer by the time a send is attemped, the sender either retries later or issues a failure. A queue pair also includes a 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 makes use of *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, a zero-copy transfer occurs from the sender 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-side RDMA rates can approach the full 100Gb/s of the optical layer, far faster than any IP protocol can reach, and indeed more than 3x what single threaded memcpy can achieve for memory-to-memory copying internal to the nodes of our compute cluster. (Total memory bandwidth is considerably higher, but is divided between 16 cores.) Moreover, speedup to 1Tb/s networking is widely expected within a decade.

Although not used in RDMC, RDMA also supports *one-sided* RDMA reads and writes where one endpoint grants the other permission to perform one-sided reads or writes into a preprepared memory region. The initiator of a read or write will see a completion event, but the target isn't notified at all. Finally, RDMA also supports some unreliable modes of operation, including a mode that resembles IP multicast. However this multicast mechanism leaves it to the user to deal with message loss or reordering, and would require occasional retransmissions. Worse, because transfer size is limited to the network MTU which is at most a few KB, large messages would be split into thousands of chunks and arrive out of order, requiring the receiver to undertake the CPU intensive process of reassembly.

In the two-sided mode used by RDMC, if nothing fails, data will be moved reliably from user-mode source memory to user-mode receiver memory, and the send-order is preserved. If a hardware fault or an endpoint crash occurs, the hardware reports the failure and breaks the two-sided session (RDMA is not tolerant of Byzantine behavior). Such a situation should be extremely rare: RDMA reliability is similar to that of a memory bus.

```
void create_group(group_number, root,
     members, notifications_callback)

void destroy_group(group_number)

void send_message(group_number,
     data, size)

void post_receive_buffer(group_number,
     data, size)
```

Figure 1: RDMC library interface

We set out to implement an RDMA multicast that extends the basic RDMA semantics to multiple receivers:

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 avoid incurring delay on the critical path, applications that will do repeated transfers should set groups up ahead of time.

2. The sender can enqueue a sequence of asynchronous send requests and can continue to do so for as long as the group is needed.

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. Any messages that arrive will not be corrupt, out of order, or duplicated and if no failures occur, all messages will arrive.

5. Any failures sensed by RDMA are reported to the application, but no automated recovery actions occur.

## 3 System Design

Figure 1 shows the RDMC interface, omitting configuration parameters like block size. To set up a group, the application starts by deciding group membership using an out of band mechanism. For example, a file copying program might launch on a set of nodes with the sender and receiver IP addresses supplied as a parameter. Next, all instances start up, pre-allocate and pin memory, and call

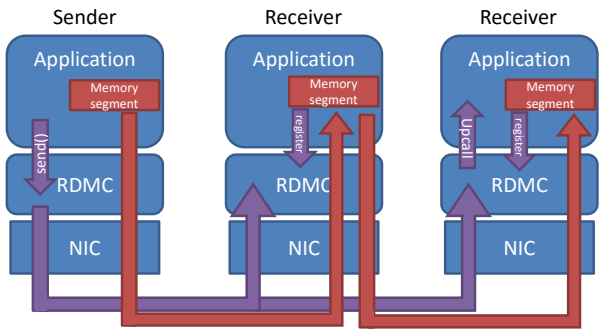



Figure 2: Overall system design of RDMC.

`create_group`, passing in the identical membership information. Within a group, only one node (designated as the "root") is allowed to send data. However, an application is free to create multiple groups with identical membership but different senders; we've tested this case and found that when groups are concurrently active with overlapping membership, bandwidth splits between them. We do not provide a way to change a group's membership or root as this can easily be accomplished by creating a new group and then destroying the old one.

The `create_group` function is inexpensive: Each member must simply exchange connection information with its neighbors and initialize a few data structures. RDMC issues callbacks to an application notification handler to report completion events and failures.

Sending a multicast is straightforward: a user-level process simply calls `send_message` with the group number and the memory segment it wishes to transfer, and gets an upcall when the sending action is locally complete. On the receivers, a notification upcall requests memory for each incoming message; a second upcall signals receipt.

Notice that neither the sender nor the receiver is notified when *all* receivers have successfully finalized reception. We recognize that a definitive outcome notification would often be useful, and our ongoing work includes ways of implementing a notification with strong semantics. However, the solution transforms RDMC into a full-fledged atomic broadcast, imposing semantics on the application that go beyond the least common denominator, while bringing non-trivial overheads.

By opting for the weakest semantics, we enable such applications to use RDMC without incurring any unnecessary overheads.

In summary, RDMC messages will not be corrupted or duplicated, and will arrive in order. RDMC will also not drop messages simply because of high load or insufficient buffer space. If no nodes fail, then all messages are guaranteed to arrive. Any failures that do occur are reported on a best effort basis.

The transfer of messages is achieved by a series of RDMA reliable unicasts between the user-level memory at the sender and the recipients. The RDMC library breaks each message into fixed sized blocks and relays the data, block by block, to intermediate recipients. When the multicast is locally complete, an upcall from the RDMC library on each member notifies them that data has arrived in their application's memory.

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. As noted in the introduction, RDMC actually supports multiple algorithms. We'll describe them in order of increasing effectiveness.

The very simplest solution is the "sequential send:" it implements the naive solution of transmitting the entire message from the sender one by one to each recipient in turn. This approach does not scale well because at any point only one node is using any of its incoming or outgoing bandwidth.

"Chain send" implements a bucket-brigade. After breaking a message into blocks, each inner receiver in the brigade relays blocks as it receives them. Relayers use their full bandwidth, but sit idle until they get their first block so worst-case latency is high.

A "binomial tree send" can be seen in Figure 3 (left). 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 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 unused bandwidth of their incoming and outgoing links.

Significantly lower latency is possible if we use a binomial tree to transmit blocks instead of entire messages. This observation was first made by Ganesan and Seshadri [18], who proposed an algorithm that combines bucket brigade and binomial trees to provide the benefits of each. They do so 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). As detailed Appendix A, we modified Ganesan and Seshadri's synchronous solution into an asynchronous protocol, and included several other small changes to better match it to our setting.

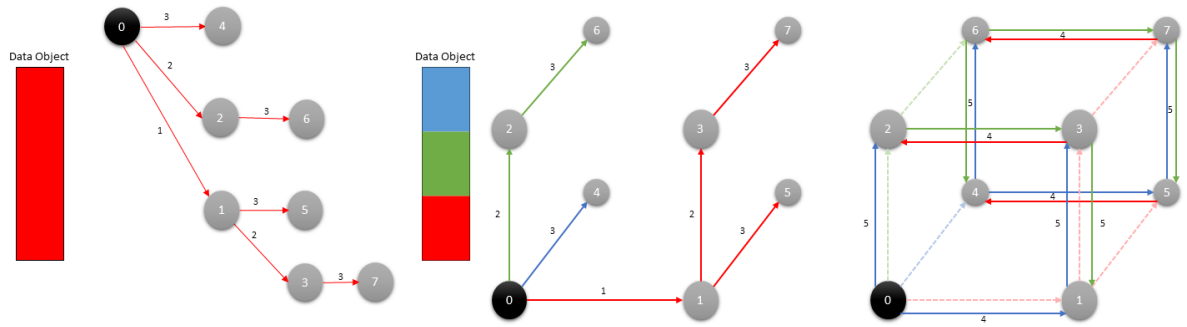



Figure 3: (Left) A standard binomial tree multicast, with the entire data object sent in each transfer. (Center) A binomial pipeline multicast, with the data object broken into three blocks, showing the first three rounds of the protocol. In this phase, the sender sends a different block in each round, 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.

## 3.1 Hybrid Algorithms

Although RDMC supports multiple algorithms, the binomial pipeline normally offers the best mix of latency and performance. Nonetheless, there may be situations in which other options are preferable.

For example, many of today's data centers have full bisection bandwidth on a rack-by-rack basis, but use some form of oversubscribed top of rack (TOR) network. If we were to use a binomial pipeline multicast without attention to the topology, half the data would traverse the TOR network (this is because if we build the overlay using random pairs of nodes, the average link would connect nodes that reside in different racks). The resulting data transfer pattern would impose a heavy load at that level.

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. When transferring 256 MB objects, chain send performed well, but when the same data was transferred using 64 MB multicasts, chain send lagged in groups with as few as 16 nodes and by 256 nodes, was achieving just 40% of the throughput of the binomial pipeline. Chain send is also the worst case for delivery skew, and is very sensitive to slow links. 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.

Even more 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 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.

## 3.2 Architectural Details

We implemented RDMC as a userspace library that runs on top of the IB Verbs library. Although we have tested only in user mode, we believe that RDMC could also run within the kernel or hypervisor, be dropped into the controller of an NVM storage unit, or embedded directly into the network hardware, all of which would drastically reduce scheduling latency and decrease overhead.

**Initialization** Before an application can participate in RDMC transfers, it must go through a setup process. During this stage, RDMC exchanges connection information with all other nodes that may participate and prepares any internal datastructures, and also posts receive buffers for all possible control messages. Finally, we start a polling thread that monitors RDMA completion queues for notifications about incoming and outgoing messages.

If several RDMC transfers are underway concurrently, each has its own block-transmission sequence to follow, but separate transfers can share the same completion polling thread, which reduces overheads. Even so, an issue arises of CPU load: while at least one transfer is active it is acceptable to poll continuously, pinning one core, but clearly when a system is idle, this overhead would be objectionable. Accordingly, after 50 ms of inactivity we transition to using interrupts to reduce CPU load at the expense of more latency in reacting to subsequent messages. For CPU bound workloads, it is possible to run RDMC exclusively with interrupts. Preliminary results for this

configuration show less than 1% CPU utilization and only about 20% decrease in bandwidth for large messages.

**Data Transfer Schedules** An RDMC sending algorithm must be deterministic, and if a sender sends multiple messages, must deliver them in sequential order. As summarized earlier, when a sender initiates a large transfer, our first step is to tell the receivers how big each 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 message to carry in 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 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 schedules are those that make sure all the nodes spend as much time concurrently sending and receiving.) 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 signalling 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 they are ready. By doing so 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 uses just a few receive queues per group, and since we do not anticipate having huge numbers of concurrently active groups, this form of resource exhaustion is avoided.

# 4 Experiments

## 4.1 Experimental Setup

We conducted experiments on several clusters, beginning with the Sierra cluster at Lawrence Livermore National Laboratory. The cluster 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. The clock speed is 2.8GHz; while system memory bandwidth is 256Gb/s, memcpy achieves just 30Gb/s. They are all 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 which is a modified version of Red Hat Linux.

The cluster employs batch scheduling for jobs, and this creates an issue that should be noted: nodes within the cluster but not used by our experiment will be processing real workloads and generating unrelated network traffic. The reason this is a problem is that although we do have exclusive access to the nodes we are assigned (any cores not used will be idle) and the cluster uses a fat-tree network, observed bandwidths are far below full bisection bandwidth. We speculate that this is caused by link congestion resulting from suboptimal routing. As a result, our experiments compete with other network traffic, particularly at large scale. We have no control over this phenomenon, although we can estimate the degree to which it is occurring. MPI, which is popular at LLNL, has an advantage in this sense: the LLNL scheduler is optimized for MPI jobs and selects node layouts that work especially well for it, particularly at very large scale.

We also conducted tests on two other clusters. The U. Texas Stampede cluster contains 6400 C8220 compute nodes each with 56 Gb/s FDR Mellanox NICs. Like Sierra, it is batch scheduled with little control over node placement. We measured unicast speeds of to 40 Gb/s, about double what was observed on the other two systems.

The Fractus cluster located at Cornell University contains 8 RDMA enabled nodes very similar to the ones on Sierra, each equipped with a 4x QDR Mellanox NIC and 94 GB of DDR3 memory, and running Ubuntu 12.04, and equipped with a 40Gbps Mellanox IB switch. All nodes have one-hop paths to one-another, hence latency and bandwidth numbers are consistent between runs. Unfortunately, during our experiments the cluster was misconfigured, limiting bandwidth to 20 Gbps each way.

In work currently underway at the time of submission, we upgraded Fractus with dual-capable 100Gbps Mellanox NICs that support both IB and RoCE, increased the number of RDMA-capable nodes to 19, and installed two 100Gbps Mellanox switches, one for IB and the other for RoCE (here the one way bandwidth is almost exactly 100Gbps but the full two-way limit seems to be around 165Gbps). Thus, we are now in a position to explore faster hardware, to microbenchmark with larger groups, and to compare the IB and RoCE cases. Here we include preliminary results for experiments run on 12 of the upgraded nodes using the IB switch. We've run some of the same experiments on RoCE and obtained similar findings.

Our experiments thus include cases that closely replicate the RDMA deployments seen in today's cloud platforms: for example, Microsoft Azure offers both RDMA over IB and RDMA over RoCE. They also include scenarios seen on today's large HPC clusters.

Not included are experiments with any form of virtualization. Although virtualized platforms are popular in cloud settings, it is not obvious how they could expose RDMA queue pairs to applications: multi-level page tables are in potential conflict with the zero-copy RDMA model, and there are evident issues of security. NIC hardware advances could perhaps address these concerns, but until that happens, we doubt that RDMA can be offered in fully virtualized clouds. Container models represent an appealing compromise, and in fact the Mesos OS, supporting Docker containers, underlies the Microsoft Azure cloud RDMA option. We believe that this model could become a de-facto standard for applications that need RDMA support, enabling the use of RDMC by cloud infrastructure developers, container application developers, and of course also by HPC solution developers.

With the exception of the concurrent sends experiment, we always select the lowest numbered node in our job to be the sender. The sender generates a message containing random data, and we measure the time from when the send is submitted to the library and when all clients get an upcall indicating that the multicast has completed. Bandwidth is computed as the message size divided by the total time spent, regardless of the number of receivers. Thus, when we report a 6 Gb/s throughput for a group of 512 members in Figure 7, we mean that all 511 receivers get identical replicas of the transmitted 256 MB object about a third of a second after the send was initiated.

## 4.2 Results

In Figure 4 we break down the time for a single 256 MB transfer with 1 MB blocks and a group size of 4 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 depicts 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 $\mu$s 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.

In Figures 6a and 6b 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. All block transfers involve 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.

Figure 7, Figure 8 and Figure 9 show the bandwidths for various sizes of multicasts across a range of group sizes running on LLNL (Sierra), Stampede, and our 100Gbps Fractus IB configuration. In these experiments we fixed the block size at 1MB.

Although Sierra had by far the largest number of machines available to us, the experiments on that platform posed a challenge for us since we often ended up with our nodes spread widely across the cluster. Further, the Sierra TOR switches exhibit surprising and very large load-dependent performance variations (a problem we did not see on Stampede). For example, we measured latencies as high as 20 microseconds and inconsistent bandwidths, which in the worst case were as low as 3 Gb/s. Despite these issues, our asynchronous implementation of the binomial pipeline proves to be surprisingly robust to delay and scheduling phenomena. For example, with 512 nodes we observed speeds of 6 Gb/s for 256 MB messages. It takes just 4x as long to make 511 replicas of a large object as to make 1.

Message size has an important impact on overall bandwidth. As we saw earlier, when selecting a block size we must balance the number of blocks with the band-

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

Figure 4: Times in microseconds for various steps of the transfer.

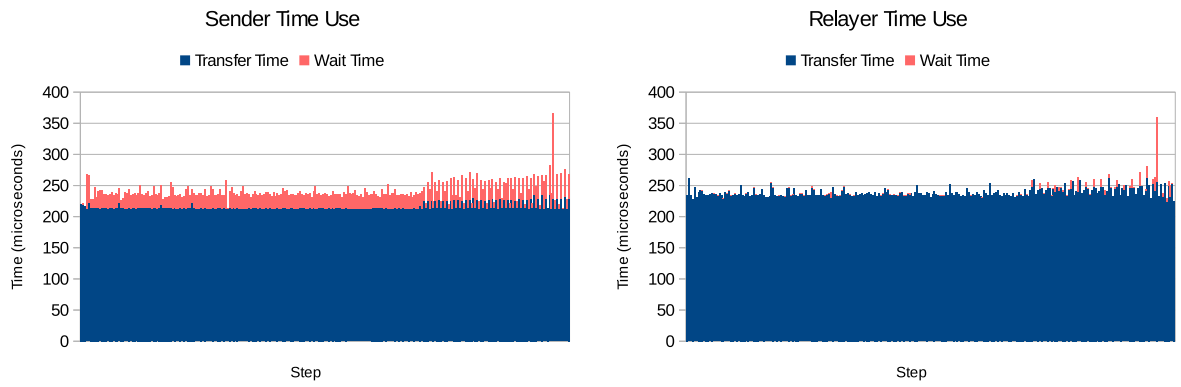
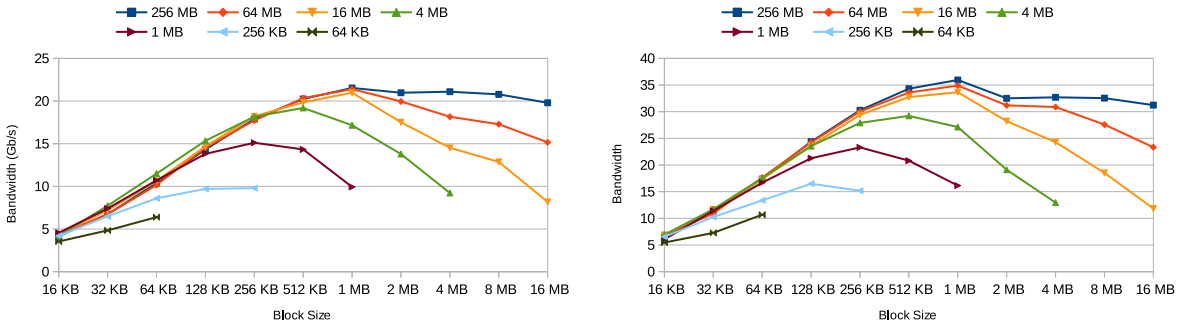



Figure 5: Breakdown of transfer time and wait time of two nodes taking part in the 256 MB transfer from the same experiment as Figure 4. Notice that the relaying node spends hardly any time waiting, while the sender transmits each block slightly faster (since it isn't receiving at the same time) and then must wait for the other nodes to catch up.




(a) Measurements from Fractus.

(b) Measurements from Stampede.

Figure 6: Multicast bandwidth across a range of block sizes for message sizes between 4 and 256 megabytes with a group size of 4. The ideal block size depends on both the message size and the unique characteristics of the network.

width possible when sending a block. For messages under roughly 10MB, these two factors directly conflict and we are unable to pick any size that will get extremely high overall bandwidth.

Finally, notice that in all three of these cases, achieved bandwidth is highest in the unicast case and remains high but fairly stable for replication groups of up to 8 nodes. At present we have only tested very large groups on Sierra, and although performance tails off at the largest sizes, RDMC even then achieves a high percentage of the possible unicast speed. In our current experiments, the limiting factor is almost surely topology, and could be avoided by minimizing the load placed on shared, higher-latency links. We believe this is best done in the job scheduler that selects nodes on which the application should run, as seems to occur when the LLNL scheduler launches MPI jobs.

Figure 10 compares the performance of RDMC binomial pipeline multicasts to that of several others, including MPI on Infiniband (shown as MPI_Bcast), as well as several others we implemented within the RDMC framework. Sequential send is the naive algorithm introduced earlier where none of the receivers help with relaying, while binomial tree is the slightly better one where receiviers begin relaying once they have the whole message. The chain send scheme uses the method described in [27], in which blocks are relayed along a chain.

We should note that of these cases, one was actually not run as a protocol within the RDMC framework: the MPI_Bcast performance was measured using the OSU Micro Benchmarks software package using MVAPICH 1.2, a version optimized for QLogic hardware.

Our algorithm outperforms the others for large transfers in small to medium sized groups, achieving higher bandwidth and lower latency. However, once the group size becomes large (128 replicas or more depending on the

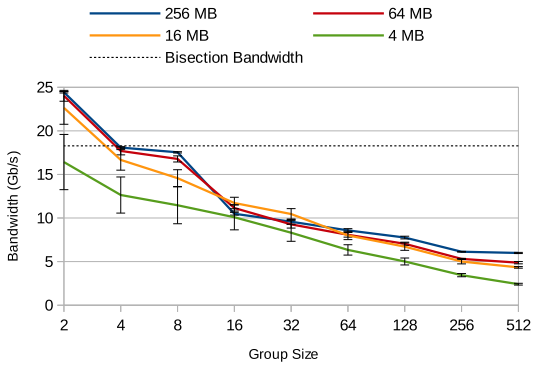



Figure 7: Bandwidth of a multicast for several message sizes on Sierra.

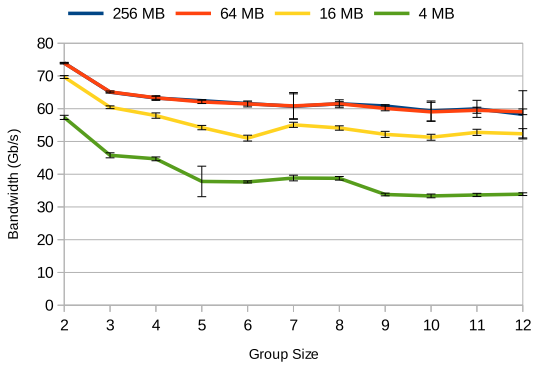



Figure 9: Bandwidth of a multicast for several message sizes on 100 Gps Fractus.

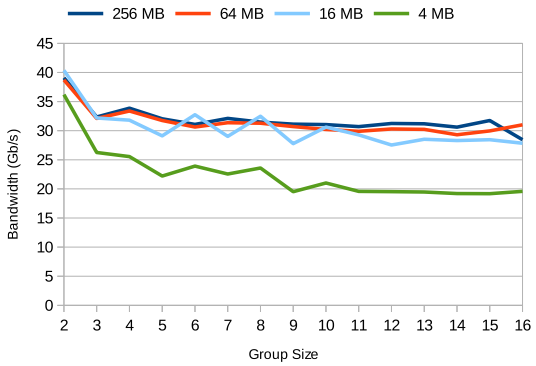



Figure 8: Bandwidth of a multicast for several message sizes on Stampede.

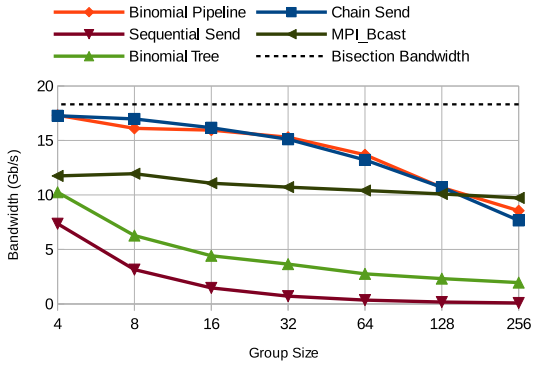



Figure 10: Bandwidth of various algorithms across a range of group sizes for 256 MB multicasts (Sierra).

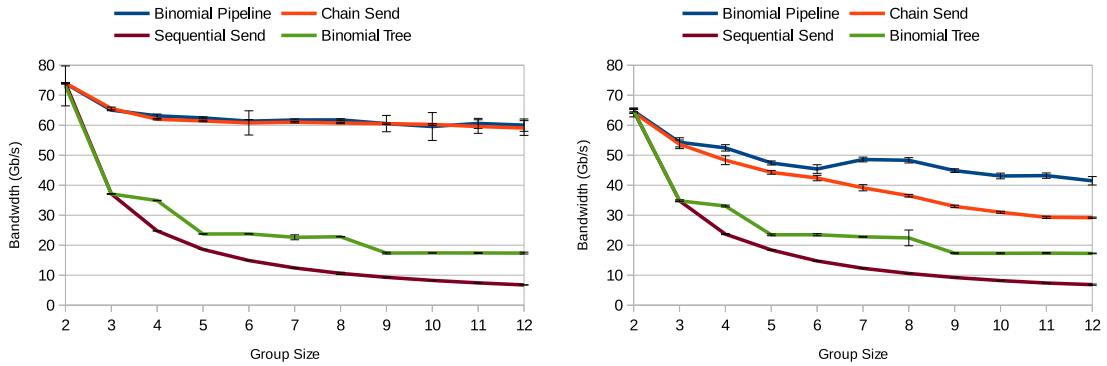
message size) MPI has better performance. An obvious question arises of why MPI_Bcast is so slow for smaller groups, yet experiences almost no performance degradation at larger scales. Unfortunately, we are not able to answer this because the algorithm used for MPI_Bcast is not well documented. In particular, although Sanders describes a 2-Tree approach in [25], our understanding is that the 2-Tree algorithm would not give such flat scaling across the full range of sizes seen here.

Next, we undertook a similar performance comparison on Fractus at 100Gbps, but now omitting MPI. As seen in Figure 11, chain send and binomial pipeline do extremely well here, while the sequential send and tree send, which transfer full objects before relaying can occur, degrade very quickly. When examining these results it is important to keep in mind that while the simple chain-replication scheme can achieve high throughput, it has terrible delivery skew and is very sensitive to slow links (Figure 12). With the binomial pipeline algorithm, we are able to transform these fast point-to-point send speeds into very efficient multicasts for large objects. The algorithm is able to take advantage of both the incoming and outgoing bandwidth from all the nodes in the group, not just some of them. As a result, when the block size is small compared to the message size, the theoretical (and observed!) time for the transfer is only slightly more than the time it would take to send the entire message between two nodes. In fact, for small groups, overheads can be so low that the total time taken to replicate an object can be less than it would take to perform a unicast transfer over the slowest link. We have seen this happen even when that link is only slower by a few tenths of a Gb/s.

Figure 12 compares scalability of the binomial pipeline on Sierra with that of sequential send (beyond the 4-node case, the sequential send data is an extrapolation).

Figure 13 is an experiment carried out on 40Gbps Fractus in which we created multiple groups using the same processes, and then initiated concurrent multicasts with each node acting as the root of one group. For comparison we include data for single sender transfers as well.

Finally, Figure 14 looks at rate variability when all members send in groups of various sizes on 100Gbps




(a) 256 MB multicasts

(b) 8 MB multicasts

Figure 11: Bandwidth of various algorithms across a range of group sizes on 100 Gbps Fractus.

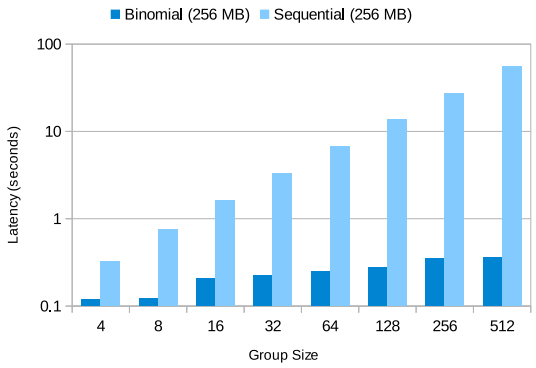



Figure 12: Comparison of the latency for sending messages using binomial pipeline and sequential send.

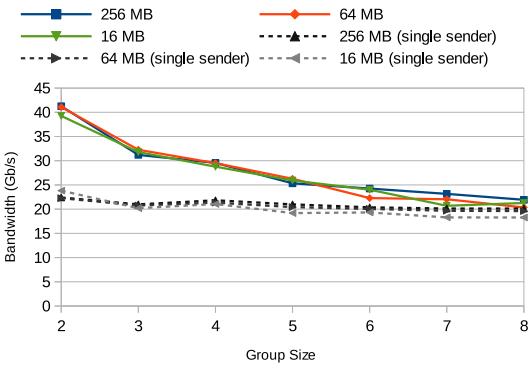



Figure 13: Average bandwidth for concurrent multicasts by distinct senders (Fractus).

Fractus, plotting mean and standard deviation, while Figure 15 measures the number of 1 byte messages per second that RDMC can send as a function of group size, again on 100Gbps Fractus. RDMC has not been optimized for this use case, although we've observed that performance depends heavily on the RDMA configuration parameters in use while being unchanged for all message sizes below the network MTU.

In the introduction, we noted that in ongoing work, we are integrating RDMC into versions of atomic broadcast and Paxos, but that concerns about the cost of synchronization argue for keeping RDMC itself as simple and free of delays as possible. In support of that point, it would be tempting to explore a head to head comparison with user-space multicast libraries such as the various Paxos libraries [1], the Isis2 (recently renamed Vsync) group communication system [10], or the Orchestra Cornet library [14]. However, we concluded that such comparisons would simply not be fair. All of these systems run over the IP network stack, and the Paxos libraries additionally log messages to nonvolatile storage in support of the Paxos durability property, hence any such experiment would be heavily biased in favor of RDMC, which offers orders of magnitude speedup relative to any system of this kind.

## 4.3 Discussion

When Ganesan and Seshadri considered tree and chain topologies for performing multicast in [18] 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.

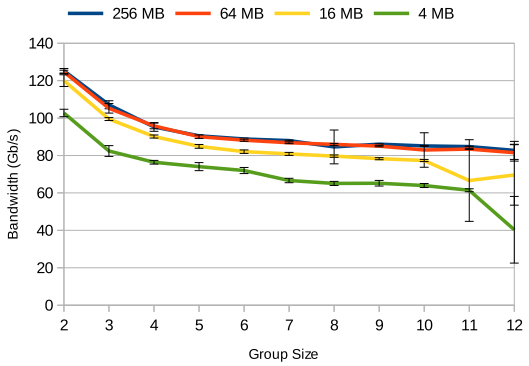



Figure 14: Average bandwidth for concurrent multicasts by distinct senders (Fractus 100 Gbps).

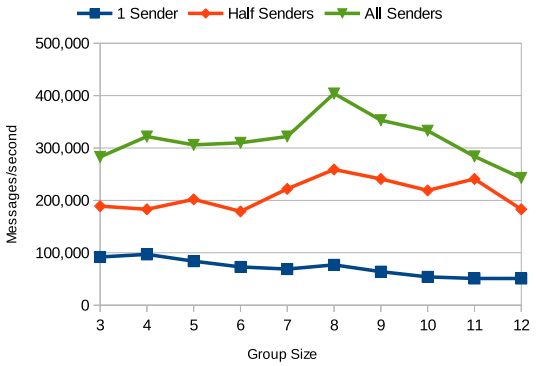



Figure 15: 1 byte messages/sec. (100Gbps, Fractus)

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 Ethernet.[1] With true hardware-supported RDMA we are able to achieve low latency, zero-copy, 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.

But there is a second and deeper factor at work that may ultimately dominate at very large scale. Here we point to a queuing theory analysis reported by Basin et al. in [8], where the cumulative effect of small delays for multicast overlays constructed from TCP links was explored. The analysis is somewhat TCP-specific and is carried out in a simpler binary-tree topology, but it predicts that above some threshold size, any overlay structure at risk of link-level forwarding delays would be expected to exhibit performance that degrades in the group size. In effect, as the number of nodes below a given sender increases, the probability rises that a relaying delay will occur somewhere in the forwarding tree and back up to cause a delay at the sender. We run on a binomial tree, but an analogous result probably applies.

[1] We have not experimented with SoftRoCE, but because SoftRoCE maps RDMA to TCP, when running on non-RDMA platforms RDMC may have to be adjusted accordingly.

# 5 Related Work

Reliable multicast is an area rich in software libraries and systems. We've mentioned reliable multicast (primarily to emphasize that RDMC 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 [4, 5, 10, 12, 16, 26].

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, 3, 6, 13, 19, 21, 22].

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 [29], 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. Verbs, though, is awkward, and this has spawned a number of other options: the QLogic PSM subset of RDMA, Intel's Omni-Path Fabric solution, socket-level offerings such as the Chelsio WD-UDP [2] 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 High Performance Computing (HPC) communications. MPI only uses a subset of RDMA functionality, hence a modern RDMA implementation will often have a stripped-down core (PSM or a similar library), on which Verbs and the full RDMA stack is implemented in software. UDP and UDP multicast on RDMA are also supported on such platforms.

Indeed, RDMC is best viewed as a bulk data copying

solution: an OS-layer primitive capable of playing a role that IP multicast (IPMC) was once expected to play [15], but in which it was never successful. We are not aware of any bulk-data replication solution that successfully used IPMC at scale, although many research efforts attempted to do so. Instead, researchers soon learned that IPMC was both hard to use because of its unreliability, and was capable of destabilizing data center switches and NICs by provoking broadcast storms [28]. Indeed, IPMC is notorious for enthusiastically discarding data — which can occur on the sender even immediately after a "successful" multicast Send operation — reordering data, and delivering duplicates. Many of today's cloud platforms either prohibit the use of IPMC or emulate it by tunneling over TCP (as in Amazon AWS). Yet it would be hard to say that IPMC was unsuccessful. Rather, it was very much a product of the time period within which it was offered, a period when reliability and flow-control were perceived as a problem to be addressed purely by the endpoints, and when it was believed that the network would be the main source of any packet loss (in the cases mentioned above, IPMC itself overloads the switches and routers and causes the loss).

Today IPMC and UDP have been mostly displaced by TCP, with built-in reliability and flow-control. RMDA, which embeds the properties of TCP into the hardware (indeed, many RDMA networks map TCP to RDMA), is a very natural fit to this new environment, and when seen in this light, we believe that RDMC is an appealing extension of RDMA to the multicast case. RDMC fully leverages the zero-copy reliability of the hardware, supports an intuitive API, and performs very well for its intended use cases. While the RDMC failure model is weak (more or less the equivalent behavior of having N one-to-one TCP connections side by side, which are unlikely to fail, but that could break during a transfer if the hardware fails or a node crashes), we've argued that this behavior is still a good match to many intended use cases, and in our own future work, have been able to extend RDMC into a full-fledged reliable multicast with strong semantics and to use it in a Paxos protocol. We feel comfortable making the case that while RDMC's model is weak, it is still "strong enough."

Although our focus is on bulk data movement, the core argument here is perhaps closest to ones made in recent operating systems papers, such as FaRM [17], Arrakis [24] and IX [9]. 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. RDMC is of course far less ambitious than these operating systems, offering just a bare-bones reliable multicast abstraction, and on achieving the absolute lowest overheads we can. However, because RDMC is a software library and highly portable, it could easily be used in a wide range of settings, and would integrate easily into the systems just listed. Further, by leveraging RDMC in file systems and memory sharing, its ultimate impact could be very broad.

# 6 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. At smaller scale one can literally have 4 or 8 replicas for nearly the same price as for 1; with really large numbers of replicas, it costs just a few times as long to make hundreds of replicas as it takes to make 1. We believe this to be a really striking finding, and of very broad potential applicability. Further, 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. Because our solution takes the form of a library it can run in user space but could also be dropped into the kernel. We believe that it could dramatically accelerate and yet also simplify a wide range of important applications, and also improve utilization of datacenter computing infrastructures. We are making the RDMC code base available for free, open-source download at http://rdmc.codeplex.com.

# Appendix A: Binomial Pipeline

The binomial pipeline is a scheme for distributing a collection of blocks originating at a single host, to some number of remote hosts and was first described by [18]. 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 vertices are assigned two nodes. 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 sequentially send 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 in every step.

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.

Now all that is left is to discuss the interactions within vertices containing two nodes. Whenever the 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. And once all vertices have all the blocks, the nodes within them trade the final block they are missing, thereby 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 [18] was evaluated purely with a simulation). This entailed several small 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 asynchronous 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.

## Acknowledgements

We are grateful to Greg Bronevetsky and Martin Schulz at LLNL for generously providing access to their large computer clusters, and to the U. Texas Stampede XSEDE computing center for providing access to that system. Support for this work was provided, in part, by DARPA under its MRC program and by NSF under its Computing in the Clouds program. An experimental grant from Microsoft is being used to evaluate RDMC on Azure (we hope to include those results in the future). Mellanox provided access to their high speed RDMA hardware, and AFOSR supports the cluster on which the majority of our experiments were performed.

## References

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

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

[3] 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.

[4] 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.

[5] 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).

[6] 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.

[7] 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.

[8] BASIN, D., BIRMAN, K., KEIDAR, I., AND VIGFUSSON, Y. Sources of Instability in Data Center Multicast. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware* (New York, NY, USA, 2010), LADIS '10, ACM, pp. 32–37.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] DEERING, S. E., AND CHERITON, D. R. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Trans. Comput. Syst. 8*, 2 (May 1990), 85–110.

[16] DOLEV, D., AND MALKI, D. The Transis Approach to High Availability Cluster Communication. *Commun. ACM 39*, 4 (Apr. 1996), 64–70.

[17] 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.

[18] 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.

[19] 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.

[20] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (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] 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.

[24] 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.

[25] SANDERS, P., SPECK, J., AND TRFF, J. L. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing 35*, 12 (2009), 581 – 594. Selected papers from the 14th European PVM/MPI Users Group Meeting.

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

[27] 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.

[28] VIGFUSSON, Y., ABU-LIBDEH, H., BALAKRISHNAN, M., BIRMAN, K., BURGESS, R., CHOCKLER, G., LI, H., AND TOCK, Y. Dr. Multicast: Rx for Data Center Communication Scalability. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 349–362.

[29] 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.