RDMC: A Reliable RDMA Multicast Protocol

Jonathan Behrens Ken Birman Sagar Jha Edward Tremel
Cornell University

Abstract

It is common for computing systems to replicate content among groups of nodes. The shared need is for memory-to-memory data transfers from a sender to a set of receivers: *reliable multicast*. Yet although this pattern was recognized long ago, we still lack a simple, fast, solution to the problem. Our new RDMC protocol runs on RDMA unicast, and uses parallel block-dissemination to achieve high levels of concurrency. Data moves not just on network links from source to receivers but also inner links between the receivers, minimizing delay and increasing throughput. Although very basic, RDMC still has useful guarantees: data is kept in the sender's order and is not lost, duplicated or corrupted. RDMC is extremely fast relative to existing options, and in many cases even outperforms specialized solutions such as MPI over Infiniband, a popular HPC library.

1 Introduction

Our work targets datacenter and compute-cluster applications that spread data among participants. The pattern is common: MapReduce replicates data as input to parallel tasks. Parallelized machine-learning systems iteratively exchange large files of model parameters as they converge. Gaming and immersive virtual reality systems continuously exchange updates to the state, financial systems distribute market data to client computing platforms, intelligent vision systems capture large amounts of data from sensors (videos, lidar, etc) and then distribute it to compute nodes which in turn may need to distribute their outputs to the robotic systems involved in navigation. VOIP systems generate video streams that must be continuously replicated on video server nodes.

Today, such systems are built over a diversity of popular technologies that include distributed filesystems, key-value stores, event notification and software atomic multicast, but existing realizations of these technologies lack an efficient low-level data replication capability, despite the pervasiveness of the data replication pattern. Replication also facilitates fault-tolerance and data durability, and permits cloud-hosted systems to scale up and handle huge numbers of clients. The common denominator is a need to reliably move what may be huge amounts of data, typically memory mapped, from one source to a potentially large number of destinations. We believe that with a more powerful multicast solution, such systems would be simplified. They wouldn't all need to reimplement the same functionality, and would benefit because a standard module can be heavily optimized.

Our new RDMA Multicast (RDMC), is a user-space library built along the lines of other contemporary user-level multicast libraries, such as Orchestra [16], which includes a general-purpose multicast subsystem called Cornet, or the MPI HPC package. Both leverage RDMA: Cornet runs on TCP, but TCP itself increasingly ties into RDMA (for example with Intel's iWarp). The MVPAPICH version of MPI is tightly optimized for Infiniband. But we wanted to create a more general, free-standing module, suitable for any setting offering RDMA unicast.

RDMC is accessed through a simple library API that can be used directly, dropped into the kernel, or integrated with higher-level replication solutions. Used directly, RDMC is a multicast, but it can also be combined with existing memory-mapped file system layers to implement a replicated sharable memory segment with file-system naming, sharing and protection functionality. Today RDMC runs node-to-node, but with further work it should be possible to integrate directly with video capture devices, NetFPGA cards, or NVRAM file systems.

This paper focuses on the memory-to-memory case and shows that for a transfer from one source to a small number of replicas RDMC achieves performance close to line rate speeds. We have experimented with as many as 512 receivers and even at this scale, multicasts still run at a substantial fraction of the optical unicast line rate.

The paper is organized as follows. We begin in Sec-

tion 2 with an analysis of the options, leading to a precise statement of the goals for the RDMC library and clarity concerning tradeoffs that arise. Section 3 describes and explains the architecture of the RDMC protocol, and Section 4 presents our experimental work. A more thorough discussion of prior work appears in Section 5.

2 Problem Statement

Reliable multicast is an old problem, widely studied in the 1990's when server replication was a major need, but has been pushed off to the side in most modern data centers. We see several reasons for this. First, many reliable multicast solutions support sophisticated models, such as state machine replication (Paxos) or virtual synchrony. Not all systems need these models, and they can be costly. For example, RDMC is so much faster than our own Isis2 virtual synchrony library[1] that we don't include the side-by-side comparison here.

A second issue is that reliable multicast has generally been viewed as an end-to-end functionality, best implemented in the application layer. For example, although every modern operating system supports IP multicast, the protocol lacks flow control, loss detection, or retransmission. The majority of today's systems implement multicast as a series of unicast transfers. This is how Cornet works: Cornet is implemented over TCP, but because TCP is often mapped to RDMA, is capable of reaching very high data transfer rates. On the other hand, Cornet is intended for movement of memory-mapped files in the Spark system, a version of Hadoop, and it would not be trivial to adapt it for uses such as publish-subscribe messaging or streaming video and real-time television broadcasts. Our work seeks to be general, useful in any distributed system that stores and computes with large data sets and exhibits a replication pattern, moving data from a memory-mapped region in a source node to memory-mapped targets either on a one-time basis or repeatedly.

The need on which we focus is increasingly recognized as a bottleneck. For example, Chowdhury et al. found that data replication can be performance limiting in Hadoop with even a few hundred megabytes of replicated data [16]. They also argue that replicated data must become available with minimal latency and as close to simultaneously as possible: if some compute tasks start while others must wait, scheduling of the entire parallel system can become unbalanced.

These considerations lead us to argue that it is time to revisit reliable multicast, but not in the elaborate forms seen in higher level packages. Rather, our goal here is to offer as minimal a mechanism as possible.

[1] `isis2.codeplex.com`

2.1 Issues with the IPMC Abstraction

It is instructive to consider the failings of IP multicast (IPMC). IPMC is broadly supported yet rarely used in today's operating systems, primarily because IPMC has a number of design flaws. For RDMC to become widely used, it will be important that we not repeat the same mistakes.

IPMC was originally intended to provide a cheap and scalable way to replicate data, discover services, and to promote the style of anonymous communication we now associate with publish-subscribe. It runs in the IP stack, fragmenting outgoing data into 1500 byte IPv4 packets, and offers best-effort reliability [17]. On IPv4, a message size limit of 64KB is hard-wired into the protocol.

The core problem is that IPMC is unreliable: packets can be dropped on the sender (even after the send has reported success), in the network, or on any subset of the receivers. For example, we tested IPMC on the same high-speed cluster used to evaluate RDMC in Section 4, sending messages of size 4 to 64KB in groups of various sizes, and counting even a single loss to a single receiver as an unreliable outcome. With a standard 1G Ethernet, IPMC bandwidth could easily approach .96Gb/s, but loss rates varied from .75% for groups of 2-4 members with 1KB messages to as much as 1.8% for 16KB payloads with groups of more than 8 receivers. With a 20Gb/s NIC and 16KB packets, bandwidth rose to 4.3 Gb/s but reliability dropped to 77% for 16 receivers, and with 64 receivers, to to 68%. Later we will see that RDMC can reach 25Gb/s with 2 receivers and 10Gb/s with 64, with perfect reliability.

Why is IPMC so ineffective? As noted, IPMC runs in the UDP protocol stack in the kernel, copying data between user space and kernel memory. IPMC packets will be dropped if a fragment fails to arrive in time, just as for UDP, or if the receiver socket overflows. At modern network data rates, no amount of kernel buffering is adequate to soak up a burst of IPMC or UDP packets arriving at the full speed of the optical layer. Further, network routers and even the kernel itself drop IP packets as a way to signal overload. This is effective when using TCP, which senses packet drop and adjusts its window sizes accordingly, but IPMC has no end-to-end logic and no windowing or flow control of any kind.

IPMC can also cause serious problems in the network, overloading a datacenter's switches, routers and NICs even in the absence of heavy traffic, as a side-effect of the way such devices are implemented. The issue is that the IPMC address space allows 2^{24} class-D addresses (and more with IPv6). These devices make routing decisions at optical line rates, and cannot afford a table lookup. Instead, they use a Bloom filter: they maintain a set of bitmaps for each link, and use a scheme in which the

IPMC address is hashed to a set of bits. When a new member joins an IPMC group the corresponding bits are set for links routing to it, and when a packet is sent, each device can hash the IPMC target address, test the bits, and make a forwarding decision in just a few clock-cycles. But these Bloom filters have limited size. Thus if enough IPMC addresses are in use, with recipients scattered around the network, the Bloom filters become saturated. The network will then forward every IPMC packet on every link, and cause it to be received by every machine, leading to a *multicast storm* that overloads end-hosts and triggers widespread loss [15, 28]. Not surprisingly, IPMC is disabled in many datacenters (for example, Amazon AWS tunnels IPMC over a TCP-based reflection service).

In what follows, we argue that our RDMC design decisions offer a constructive response to every one of these IPMC limitations. IPMC mirrored the properties of a failure-prone generation of hardware and software, but as that hardware is phased out, it is time to consider deprecating IPMC in favor of a solution like RDMC.

2.2 Background on RDMA

Although we have assumed that most readers are familiar with the basics of RDMA, it may be helpful to offer some details about the assumptions made by RDMC, since the RDMA hardware space has recently become very active, particularly with the evolution of RDMA from its origins in the Infiniband environment to the newer positioning of RDMA over optical Ethernet (i.e. RoCE).

All RDMA technologies are reliable, in the sense that the optical layer should very rarely lose or corrupt data, and any such problem would be detected and trigger an exception. In practice, an RDMA transfer will be reliable unless the hardware fails, or the sender or receiver crashes. Taken jointly, one has a weak end-to-end guarantee: if there are no crashes or hardware faults, then an RDMA transfer will result in the data reaching the receiver intact. Conversely, if some component does fail, then until a reset occurs, that RDMA connection cannot be used. Thus some healthy endpoint will eventually receive an exception or notice a timeout. Notice that failure sensing is somewhat weak; this is because a NIC and its receive queue could remain active for a while even if the end-host system has crashed.

A modern RDMA network moves data directly at optical line rates, which is quickly exceeding the speed that a CPU core can touch memory (RDMA runs at 20Gb/s today but will soon ramp to 100, 200, 400 and then 1Tb/s; in our testbed, memcpy runs at just 30Gb/s). The most basic and universal capability is for an RDMA-capable NIC to simply transfer data directly to and from host memory using DMA. To avoid the need for interrupts, the NIC typically monitors a set of outgoing transfer queues, onto which the host places send requests, each giving an address (or a scatter-gather address list) and a transfer size, then sets up a DMA send and transfers the whole object. On receipt, the NIC matches the transfer to one of the receive queues and drops the data directly where it is desired. The NIC then enqueues a completion event which the end-host picks up either in a polling loop, or via an interrupt.

The potential scope of RDMA is expanding with Intel's promotion of iWarp, a technology that drops the TCP stack directly into hardware. The intent is that one or both endpoints can use iWarp; if both do so, the TCP connection is seamlessly transformed to use RDMA transfers. This, though, poses non-trivial technical challenges: TCP is byte-oriented whereas end-to-end RDMA is feasible only if the source and receiver use identical buffer sizes.

The pace of development in RDMA networks is also evident in the growing range of APIs, including the original Verbs library (providing both bulk DMA copying and direct remote byte-level data manipulation), several mappings of TCP to the RDMA layer, UDP and UDP multicast implemented over RDMA, and the open-source Data Plane Developer's Kit (DPDK). To encourage a match of sender and receiver buffer sizes, several packages run at the socket layer and more or less mimic UDP, but with much stronger reliability. For our work, we use just one RDMA feature: reliable zero-copy unicast transfers, which are available across a full spectrum of hardware.

RDMA solutions generally lose substantial performance unless a thread is dedicated to poll (via busy-wait) the send and receive queues. The problem is that although most devices offer interrupts on I/O completion events, those interrupts would be fielded by the kernel, and by the time a user-level thread could be awakened it is likely that tens of thousands of instructions will have executed, causing microseconds of delay. At the speeds of an RDMA transfer, this can be a substantial issue, particularly if the user-level thread will be involved in relaying data, as occurs in our protocol.

A further issue relates to group membership. Some devices need a preestablished "connection" between each sender and receiver pair before communication can be performed, and resetting such connections after a failure is tricky. Designing a protocol that can robustly tolerate failures of some members while others continue to operate is problematic. Our solution to this involves treating each group configuration as a distinct session, with associated connections only while the membership remains unchanged.

A final hardware issue relates to the way RDMA NICs track the state of send and receive queues, but especially the latter. For speed, an RDMA NIC will typically cache

active queues; when an incoming transfer is initiated, this permits the NIC to rapidly find the best matching queue, determine whether a receive memory region has been posted on it, and agree to accept the incoming transfer. But NIC memory is of limited size, hence it is important to bound the number of concurrent sessions, an issue that emerged as a problem in the FaRM RDMA Key-Value store [19].

2.3 Reliability

In light of our desire to offer a reliable multicast primitive for RDMA, the question arises of what reliability should mean for RDMC. Our overarching approach is to view RDMC as a reliable but asynchronous channel: normally, we can trust it to be reliable, but rarely, when something does fail in a way that it can sense, an asynchronous notification should occur either on one of the receivers or in the sender. In adopting this model, we basically mirror the reliability of RDMA unicast, but extended to the multicast case.

While these guarantees are strong, they are still much weaker than those of atomic multicast closely integrated with group membership, for example in the virtual synchrony model [13]. The issue is that RDMC failures result in a complex situation, partly because the NIC might not realize that a receiver node has actually crashed, and partly because the RDMC protocol relays data, hence the failure of a relaying process will disrupt receipt even if the sender and some receivers remain healthy. Our design opts for minimal overhead, leaving it to higher level applications to strengthen the RDMC guarantees if desired, but without creating a need for higher level buffering of data, end-to-end acknowledgements, flow control, retransmission logic, message stability detection and garbage collection.

With this in mind, we still had several possibilities. One option is to implement some form of asynchronous reliability notification. The sender and receivers would see two events. The first would simply be a notification of successful completion of the send or receive action; the second notification would report on success or failure of the transfer as a whole to the full set of participants. However, to carry this out, RDMC would need some form of internal acknowledgement mechanism.

The second, less elaborate, would omit the second notification, leaving the application to clean up in the (very infrequent) case where a relaying node fails and disrupts the transfer for downstream nodes. The normal mode would just be a success. Abnormal cases would be sensed by the end-point applications themselves: a receiver might timeout because its upstream data source failed; a sender or relaying node might timeout trying to send to a receiver that never posts a receive buffer (the timeout is under application control).

```
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

A third option would be to hard-wire RDMC to keep trying until each multicast has reached all the receivers "healthy enough" to accept the data, giving up only if there is no copy of the message to resend. This is what Isis2 does: if a multicast reaches any receiver, it will reach every non-failed receiver. But this would delay garbage collection of pending messages until some form of end-to-end acknowledgement occurs. Even if such an acknowledgement could be generated in 10ms, at the data rates we anticipate hundreds of megabytes or gigabytes of data might need to be retained.

We opted for the second option: if a set of nodes participates in a transfer and something goes wrong, the application will sense the issue. This approach is remarkably inexpensive, yet adequate for the majority of higher level purposes outlined above. Of course, end-to-end logic would be needed when using RDMC in support of a strong consistency model.

3 System Design

Our design of the RDMC system reflects the problem space and basic design decisions outlined above. RDMC provides a simple interface to applications that allows them to hand-off memory directly to the network hardware, bypassing any copying in the kernel, without needing to worry about network-layer details such as reassembly of data blocks.

Figure 1 shows the interface that applications use to interact with RDMC. Before any send or receive can take place, the application must determine group membership using an out of band mechanism and then call `create_group` locally on each node: our protocol requires that every participant know the current membership. Within a group, only one node (designated as the "root") is allowed to send data. However, the application is free to create multiple groups with identical membership but different senders. We do not provide a way to change a group's membership or root as this can easily

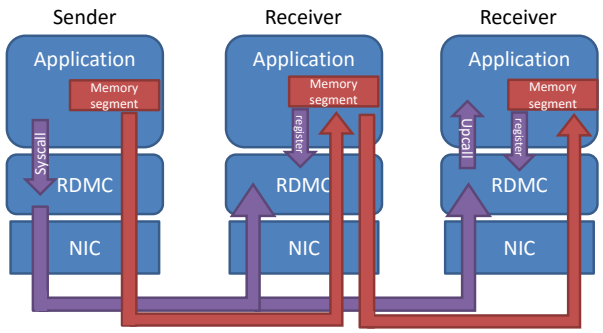


Figure 2: Overall system design of RDMC.

be accomplished by destroying and then recreating the group.

The `create_group` function does not require any communication and thus is a very inexpensive operation. All the sender needs to do is to record the group membership and initialize a few internal data structures. A notification callback is registered; RDMC uses it when reporting send or receive completion events and (infrequently) 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 multicast is locally complete. The only constraints are that before the first send, all recipients must have completed group creation and any previous multicasts to the group must have completed locally.

The transfer of messages is accomplished by a series of RDMA reliable unicasts between the user-level memory at the sender and the recipients. Internally, the RDMC library handles breaking the message into some number of fixed sized blocks and relaying of data by 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. The naive solution of transmitting the entire message from the sender to each recipient in turn does not scale well because at any point only one node is using any of its incoming or outgoing bandwidth. A significantly faster technique would be to construct a binomial tree in which once a node receives a message, it starts relaying it to other recipients that do not have it. However this method is also wasteful if implemented in such a way that a node cannot begin relaying until it has the entire message: half the nodes wouldn't receive the message until the last step and thus contribute no outgoing bandwidth at all.

To do better, we must divide the message into blocks that can be sent independently. Ganesan and Seshadri [20] analyze several different algorithms for cooperatively transmitting a multicast by using unicast transfers that do exactly this. Although they find that randomized scatter-gather-like algorithms perform very well in expectation, the overheads associated with negotiating which blocks should be sent to whom turns out to be prohibitive in the high-line-rate world of RDMA networks. However, under the assumption that each block takes the same amount of time to transfer between any two nodes (which is nearly true for RDMA), they were able to prove that a deterministic *binomial pipeline* algorithm is guaranteed to complete in the fewest possible number of steps.

This scheme operates by dividing the memory segment to be broadcast into blocks and then sending blocks along edges of a hypercube. At each step, every node with a block sends to another node that does not have that block. Since we are working with a hypercube, in its simplest form the algorithm only works on group sizes that are powers of 2. However, by assigning pairs of nodes to some vertices of the hypercube (in a manner we describe below), it can be used with any number of nodes.

The binomial pipeline proceeds in three stages, each of which are further divided into steps. During every step, all nodes with at least one block 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, the sender transfers a *different* block to another receiver, while the first receiver simultaneously sends its block on to a third receiver. This pattern continues until all nodes 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 nodes trade the highest numbered blocks they have.

Once the sender runs out of blocks, the algorithm enters the third stage. The sender repeatedly sends the last block, while the rest of the nodes continue to trade blocks in every step. Once the last block reaches every node, the multicast will be complete. If the number of receivers is a power of 2, every replica will be finalized simultaneously, which is beneficial in staged computations [16].

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.

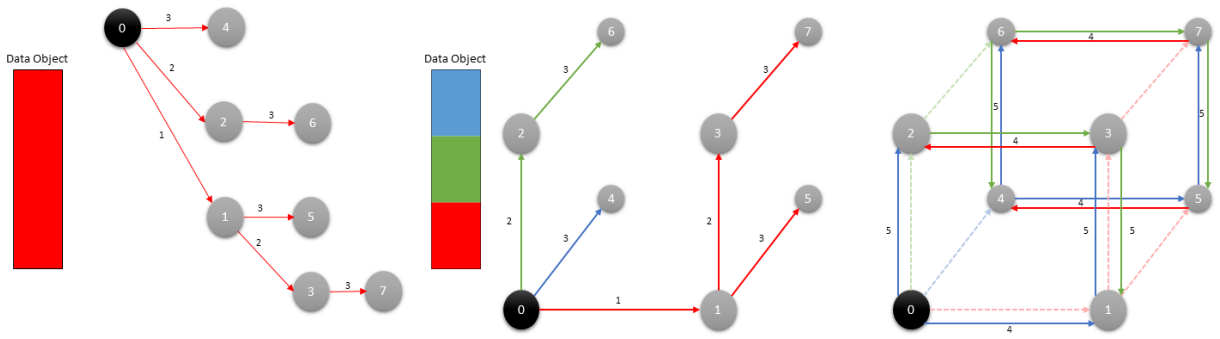


Figure 3: (Left) A standard binomial tree multicast, with the entire data object sent in each transfer. (Center) A binomial pipeline multicast, with the data object broken into three blocks, showing the first three rounds of the protocol. In this phase, the sender sends a different block in each round,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

When the group size is not a power of 2, we assign a pair of recipient nodes to some of the vertices of the hypercube so that the number of vertices is the nearest smaller power of 2. Note that when the group size is one less than a power of 2, all nodes except the sender are paired. In each step, vertices representing a single node proceed in the same way as if no nodes were paired. For vertices that represent a pair of nodes, the node having the highest numbered block sends in that step (if both have the same highest block, then any one node sends). The node not sending in that step receives the block sent to the vertex (if both are not sending, then any one receives). If the node(s) not sending also has a block that its partner does not have, it sends that to its partner. This ensures that the nodes sharing a vertex have at most one block difference with their partner at the end of every step. We run an extra step at the end to allow these nodes to exchange blocks that their partner does not have. Thus, we achieve the optimal algorithm for any group size, as shown by Ganesan and Seshadri. RDMC makes no effort to synchronize the transfer-completion upcalls simultaneously in this case, hence some occur one block-transfer-time before others. However, this is still close to simultaneous.

3.1 Evaluating Other Algorithms

RDMC can support a family of algorithms within a single framework. As part of our work, we implemented several other options, then compared them with the binomial pipeline for a variety of different message sizes. Later we will discuss these experiments, but the bottom line is that we found the binomial pipeline to indeed dominate other choices. It turns out that direct comparisons can be deceptive. For example, in one of our experiments, where we transferred 256 MB objects, chain replication was competitive with the binomial pipeline, but it turns out that for 64 MB multicasts (not included here for reasons of space), this would not have been the case: chain replication actually lags in groups with as few as 16 nodes and by 256 nodes it is down to 40% of our throughput. Chain replication is also the worst case for delivery skew, and is very sensitive to slow links. Nonetheless, if topology data were readily available, it might make sense to use chain replication between the racks of a fat-tree network, and the binomial pipeline within racks. RDMC already uses a hybrid solution for very small sends. As we will see later, the binomial pipeline imposes about 50us of overhead, which is negligible for large sends (where the optical line data transfer time might be 10's or even 100's of milliseconds), but excessive for very small ones. Accordingly, we implemented a special-purpose solution (discussed below) optimized for very small transfers. It reduces overhead by a factor of 50, but cannot be used for large objects.

3.2 Architecture Details

We implemented RDMC as a userspace library that runs on top of PSM, a QLogic API that offers optimized access to just the RDMA unicast operation and pushes less-commonly used functionality of the standard Verbs API (like native multicast) into software, though all the features we use can be emulated efficiently on any RDMA hardware supporting unicast and scatter-gather operations. PSM is nearly identical to the more broadly supported DPDK standard, hence our code would also run on any NIC supporting DPDK, or on any NIC support-

ing RDMA through the full verbs library. And although RDMC runs at the user level, the module could certainly be used within the kernel or hypervisor, 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.

The question now arises of how the sender should signal its intentions at the start of a large transfer, since any single RDMC group can transport messages of various sizes. Here, we took advantage of an RDMA feature that uses 64-bit tags to filter sends. Every send and receive buffer has an associated tag, while receive buffers also have a mask that control which bits of the tags are compared. A given receive buffer is considered for an incoming transfer from a send buffer only if the non-zero masked bits of both tags match. In our solution, we post a range of buffers for different kinds of control messages, using tags to ensure that the right one is chosen for each message type. Since control messages are small and infrequent, the memory overhead required to keep these buffers posted at all times is low. On a platform where tags are not available, this mechanism could be implemented by setting aside some receive queues and then using scatter-gather operations to pass this same information side by side with data.

Interaction between the library and the user process takes place by way of function calls into the library code and upcalls back to the user process. For example, RDMC upcalls in the receiver communicate the size of an incoming message, allowing the receiver to post a suitable size of message buffer. This allows for efficient and low latency communication between the two without requiring that the receiver repeatedly poll the RDMC library. Additional upcalls occur as each received multicast is completed.

3.3 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 repeatedly checks the RDMA receive queue for incoming messages, and the send queue for completion notifications. Although the hardware supports interrupts, we don't use them due to concerns about delay. Instead, our polling thread runs rapidly while a transfer is underway and for a short period after it ends (to ensure fast response if an application streams data), but then slows down (using a thread-level sleep) if nothing is going on.

3.4 Send Protocol

The send protocol relies on the binomial pipeline algorithm described previously, without any extra control messages. It is worth mentioning, however, that it is extensible and can easily support any other deterministic block dissemination algorithm. This is possible because receivers anticipate the blocks they are about to receive and only post a buffer for the next one they expect. And because RDMA transfers are constrained to only occur into pre-registered buffers, no message will be delivered out of order. By doing so we sharply reduce the amount of NIC resources used by any one multicast, addressing the concern mentioned in Section 3, where we noted that today's NICs exhibit degraded performance if the number of concurrently active receive buffers exceeds NIC caching capacity. RDMC is not at risk of overwhelming the NIC by having too many concurrently active receive queues: We only need a few per group, and do not anticipate having huge numbers of concurrently active groups.

Because the binomial pipeline is deterministic, each node knows which blocks it will receive and in which order in each active group. Furthermore, except in one case we'll address later, the first block a receiver gets does not actually depend on how large the message is. This means when a receiver joins a group, it is able to immediately register a receive buffer for the first block that it expects to receive.

Each block (including the first) includes a tag in which we encode the total number of blocks in the message. Thus, when the initial block of a message arrives it contains all the information necessary to conduct the remainder of the transfer, allowing us to send variable sized messages in the same group without any advanced coordination and virtually no overhead. One pitfall we must address is if the first block would be beyond the end of the message. In that situation we "forge" the index of the true first block to match what we expected and then correct it on the receiver.

It would be natural to design the system so that before sending data, some form of *prepare to receive* message would be transmitted, giving receivers time to post receive memory. However, this simple pattern is actually surprisingly slow because at RDMA data rates, it might consume more time than is needed to send the message itself. Yet we do want the application to allocate and manage memory: memory allocated by the OS on behalf of the application would need to be zeroed for reasons of security. Accordingly, we recommend that the receiver application post memory adequate for a first block as soon as the group is set up; when the block is received we do an upcall to notify the application of the incoming transfer size, but can already start forwarding the received block. Meanwhile, the receiver posts space ade-

quate for the whole incoming transfer. Although the first block of incoming message bytes won't be contiguous with the remainder of the memory segment, RDMC can use `mremap` to efficiently relocate it there, grabbing the first block that was posted as a new first block for the next incoming transfer, and so forth. Our scheme does waste space for small messages, but gains on performance.

Once a receiver has gotten its first block, it posts receive buffers for the remainder of the message, and then adheres to the chosen algorithm to complete the remainder of the send. After the receiver is done receiving all the blocks of a message and relaying them as necessary, RDMC issues an upcall back to the user process to signal completion.

Note that because our solution involves relaying by the participants, delay would have a highly amplified performance impact. In our current system, where CPU speeds are more or less matched to the optical network speeds and we can busy-wait for receive completions, this issue is minimized. But as optical network bandwidths rise by a factor of 500 over the next decade, our current approach will start to lag. Further, our test environment allows us to run on dedicated nodes. If RDMC were deployed in virtualized cloud settings, the much larger scheduling delays inevitable in any coresidency situation will distort performance. This leads us to believe that in the long run, the RDMC protocol or a similar mechanism will need to be implemented in hardware. For example, the RDMA standard could be extended to allow an enqueued receive request to have an associated *next action.* The hardware itself could then potentially receive an incoming message but immediately reenqueue it for output to a neighbor, notifying the application that the data is available only after all the relaying is completed.

4 Experiments

4.1 Experimental Setup

We conducted all experiments on 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 (20 Gb/s each direction). The Sierra cluster runs TOSS 2.2 which is a modified version of Red Hat Linux.

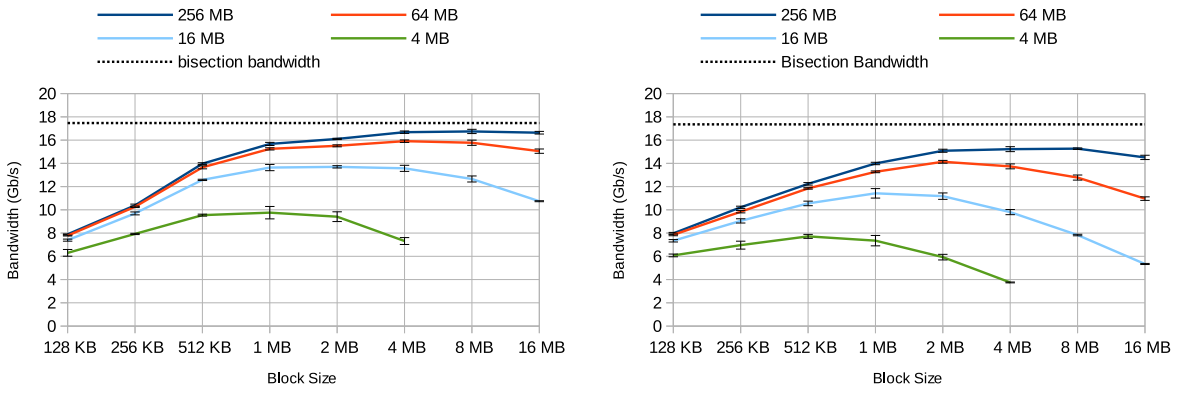
The cluster employs batch scheduling for jobs, and this creates an issue that should be noted: nodes within the cluster but not used by our experiment will be processing real workloads and generating unrelated network traffic. The reason this is a problem is that although we do have exclusive access to the nodes we are assigned (any cores not used by our tests will be idle), the cluster uses a fat-tree network and hence does not have full bisection bandwidth. As a result, our experiments compete with other network traffic, particularly at large scale, and we have no control over this, 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, particular at very large scale.

Our experiments depart in some ways from what might be seen in a general purpose, multi-tenant, cloud setting: Sierra is not a virtualized environment. On the other hand, application-layer access to RDMA is not yet common in commercial clouds. As this situation changes, one option is to link our framework directly into a hypervisor in an analogous fashion to the traditional network stack; another would be to run at a virtualized level and simply ride out the resulting scheduling delays. We hope to carry out further experiments in such environments to see what impact this would have.

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 random block of 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 5Gb/s throughput for a group of 512 receivers in Figure 5, we mean that all 512 receive identical replicas of the transmitted 5Gb object about one second after it was sent, and do so nearly simultaneously.

4.2 Microbenchmarks

The placement of nodes relative to the physical network topology impacts the achievable bisection bandwidth, which we measure and report here: before each test the lowest rank node measures the time it takes to concurrently send and receive a single large message to each of the other nodes in the test. The median is taken to be the bisection bandwidth for that experiment. However, as noted above, we have no control over node layout relative to the physical Sierra network, and for many runs ended up with more transfers that traverse the three cluster routers than is ideal. This creates link stress: these links can sometimes, but not always, have sharply lower performance (on Sierra, 5.8Gb/s bidirectional), whereas



(a) Measurements for groups of 4 nodes.

(b) Measurements for groups of 16 nodes.

Figure 4: Multicast bandwidth across a range of block sizes for message sizes between 4 and 256 megabytes.

links within a rack tend to have a much higher performance (almost 20Gb/s). But while we were able to measure this phenomenon, RDMC does nothing about it: dynamically discovering topology and bandwidth and then selecting the optimal block-transfer pattern ideal for a particular topology would be daunting.

4.3 Results

In Figures 4a and 4b 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 5 shows the bandwidths for various sizes of multicasts across a range of group sizes. With a group size of 2 (the degenerate case with 1 sender and 1 receiver) we actually get 134% of the bisection bandwidth for nodes in the same rack. This is possible because the sender is not using any of its incoming bandwidth.

Message size has a very large impact on overall bandwidth. As we saw earlier, when selecting a block size we must balance the number of blocks with the bandwidth possible when sending a block. For messages under tens of megabytes, these two factors directly conflict and we are unable to pick any size that will get extremely high overall bandwidth.

As group size gets larger, we unsurprisingly get degraded performance. Notice however, that even with 512 nodes the time spent distributing a large file is only several times what it would take to send it across any single link. With groups this large, we inevitably included more and more slow (5.8Gb/s) links in our overlay, and by the time groups reach size 128, those slow links apparently limit our transfer speeds.

Although RDMC is probably most interesting in the case of large transfers, we have also tuned the protocol to do well for smaller ones. Figure 6 shows the resulting performance. We have not shown the corresponding numbers for MPI over Infiniband, which achieves extremely low delays for tiny objects but appears to use a non-RDMA feature of Infiniband to achieve this. Specifically, Infiniband supports direct remote memory access (one processor can directly write to the memory of another, as on a NUMA shared-memory machine), and we suspect that MPI on Infiniband uses this feature, but only for small objects. This is feasible because the MPI model is one of gang-scheduled HPC applications where each replica knows the memory layout of all the others, unlike in our target setting, where source and destination could be different programs. We would recommend that RDMC users who expect to stream huge numbers of small objects consider buffering the sends, then transmitting smaller numbers of large messages; the network utilization for that pattern is far better. There are, of course, some applications that might not be able to do this (high-frequency trading systems for which minimal latency is critical come to mind), but we believe that the tradeoffs embodied in our solution are a good balance that matches well with what we anticipate will be the more common use cases.

Figure 8 compares the performance of RDMC bino-

mial 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 replication scheme uses the method described in [27], in which blocks are relayed along a chain.The only experiment not run using RDMC itself was the MPI_Bcast; this data 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 message size) MPI has better performance. Several factors apparently contribute to the MPI advantage in this case, most notably its integration with the LLNL cluster scheduler and node layout algorithm.

Finally, Figure 9 compares cases where we created several groups using the same processes, and then initiated concurrent multicasts from different roots. This was surprisingly hard to set up: the LLNL computers have almost synchronized clocks, but even so, the transfers performed here were so short that we do not know how much concurrency was actually achieved on the wire. The main takeaway is that while contention seems to slightly reduce performance, the impact is minor.

We omit head to head comparison with user-space multicast libraries such as the various Paxos libraries [1], the Isis2 group communication system [9], or the Orchestra Cornet library [16]. All of those run over the IP network stack, and the Paxos libraries additionally log messages to nonvolatile storage in support of the Paxos durability property, hence comparison would involve an apples to oranges scenario heavily biased in favor of RDMC, which offers orders of magnitude speedup relative to any system of this kind.

4.4 Discussion

When Ganesan and Seshadri examined tree and chain topologies for performing multicast in [20] they found them to be unfeasibly slow over TCP/IP. They attributed the suboptimal performance to the fact that these 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, this proves to be much less of an issue for RDMA than for TCP/IP over Ethernet. With 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 30Gb/s and is not likely to rise as quickly as optical network speeds will scale up. By contrast, thanks to hardware support for reliable sends we are able to consistently get upwards of 17 Gb/s bidirectional bandwidth (more than 85% of the line rate) per link by doing reliable point-to-point sends, and this should track the evolution of optical network speeds.

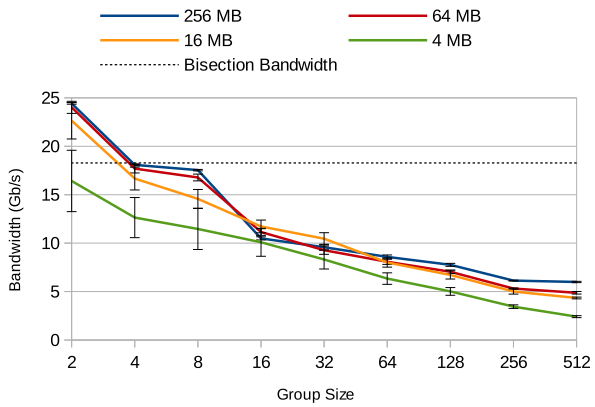


Figure 5: Bandwidth of a multicast for several message sizes.

While the simple chain-replication scheme can sometimes be quite fast, it has terrible delivery skew and is very sensitive to slow links. 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 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. This can happen even when that link is only slower by a few tenths of a Gb/s.

Our scheme is also cache-friendly, which is not the case for traditional IPMC-based mechanisms or end-user multicast libraries, in which data touching and copying are common events. While it is difficult to precisely measure such effects, in the aftermath of a transfer using those classical options, the L2 processor cache will be heavily contaminated with data sent or received during the multicast, causing small CPU stalls as application

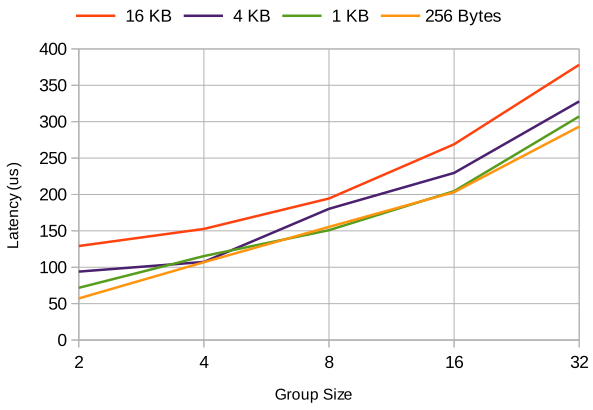


Figure 6: Estimated delay from when a small message is sent to when it is received when using our main multicast algorithm. For small objects, these delays are excessive, motivating us to support the specialized small-send code path evaluated in Figure 7. For large sends, overhead is not a concern because optical line-rate transfer time can reach 10's or even 100's of ms.

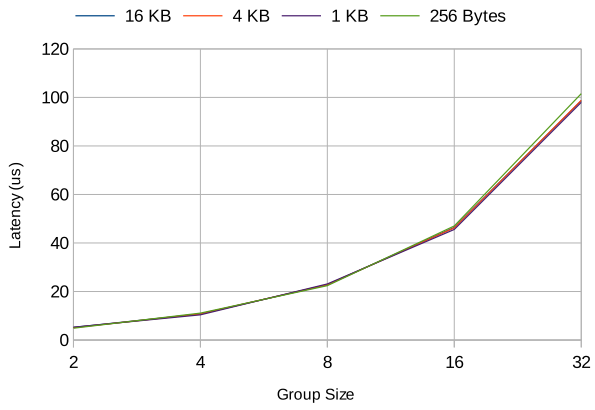


Figure 7: RDMC includes a specialized code path that reduces delays for small sends. As seen in this plot, delay drops 50-fold relative to that seen in Figure 6. However, this method cannot be used for large objects.

logic resumes execution. The zero-copy RDMA transfer shouldn't have any deleterious impact on cache performance at all.

Finally, notice that in Figure 5, achieved bandwidth is highest in the unicast case, where the hardware can actually outperform its 20Gb/s specification, then is fairly stable for replication groups of up to 8 nodes (a 1-level hypercube). Performance tails off for very large scenarios, but 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.

But there is a second and deeper factor at work, namely the scheduling delays associated with relaying that concerned Ganesan and Seshadri. Here we point to a queuing theory analysis reported by Basin et al. in [7], 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 more complex tree, with a different protocol, and using a more complex chunking mechanism, yet an analogous result probably still applies in our setting.

5 Related Work

Reliable multicast is an area rich in software libraries and systems. Pure communications solutions include Isis2, Spread, Totem, Horus, Transis and the Isis Toolkit [4, 5, 9, 12, 18, 26]. Durable state machine implementations include Paxos, Chubby, Rambo, Zookeeper and Corfu [1, 3, 6, 14, 21–23]. However, the RDMC primitive presented here focuses on a very low level data replication abstraction, whereas these all provide stronger of application-level consistency guarantees. For example, most pure communication systems offer managed group membership views, often synchronized with respect to multicast delivery ("view synchrony") and combined with various multicast ordering guarantees, supporting application-layer models such as virtual synchrony [10, 11, 13] or state machine replication [25]. The persistent versions also offer durable, totally ordered event logging and can guarantee coherence across disruptive server failure and recovery sequences. Our hope would be that all of these systems could benefit from RDMC, and that some might also be simplified by moving to our solution.

RDMC per se is better viewed as a next generation alternative to the IPMC abstraction [17]. Fundamentally, we see IPMC as a product of the time period within which it was offered; RDMC is a closer fit to a new generation of hardware, and can support a very similar API. On the other hand, IPMC supports an anonymous

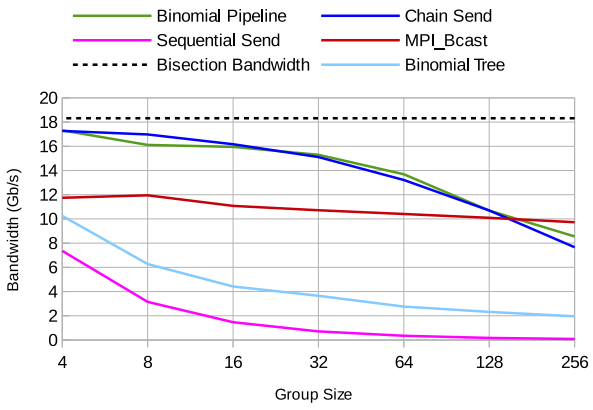


Figure 8: Bandwidth achieved by alternative algorithms across a range of group sizes for 256 MB multicasts. Chain Send and Binomial Pipeline have similar performance in this particular scenario. However, Binomial Pipeline is far more robust, as explained in Section 3.1.

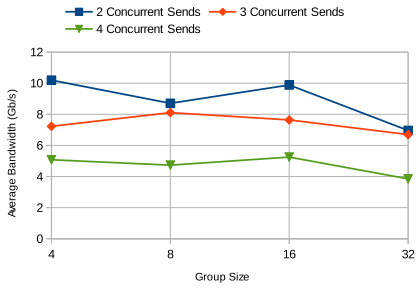


Figure 9: Average bandwidth for concurrent multicasts by distinct senders.

style of communication, in which a publisher of data uses IPMC to send to an unknown and anonymous group of subscribers, which also makes it useful as a kind of Internet *anycast* protocol, used to seek a member of a service, or rendezvous. RDMC does not support either style of use. For these reasons we do not think IPMC should be completely eliminated.

We are not the first to ask how RDMA should be exploited in the operating system. The early RDMA work grew out of the 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 iWarp, which offers RDMA through a TCP-like abstraction implemented in the hardware (RDMA over converged Ethernet, or RoCE), 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.

Our core argument here comes closer to the ones made in recent operating systems papers, such as FaRM [19], Arrakis [24] and IX [8]. 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 less ambitious than these other solutions, focusing on 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. As RDMC becomes integrated with OS features such as file systems and memory sharing, its ultimate impact could be very broad. Further, if reliable RDMA multicast hardware becomes available, RDMC could easily leverage it, and applications built on the abstraction would port to the new hardware without change. In contrast, today's mix of solutions has yielded a plethora of highly tuned, fragile systems.

6 Conclusion

Our paper introduces RDMC: a new reliable memory-to-memory multicast implemented over RDMA unicast. Performance is very high when compared with the most widely used general-purpose options, and even exceeds that of a speciality solution such as MPI over Infiniband when replicating large objects to as many as 128 servers (although MPI wins in even larger scenarios, and is also faster for high rates of very small sends). RDMC is quite general and could replace IPMC in many settings where today, IPMC is the only multicast option available. It takes the form of a library and can be linked into user space or 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.

Acknowledgements

We are grateful to Greg Bronevetsky and Martin Schultz at LLNL for generously providing access to their large InfiniBand computer cluster. Support for this work was provided, in part, by DARPA under its MRC program and by NSF under its Computing in the Clouds program.

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] 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.

[8] 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.

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

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

[11] BIRMAN, K. P. Replication and Fault-tolerance in the ISIS System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1985), SOSP '85, ACM, pp. 79–86.

[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] BIRMAN, K. P., AND JOSEPH, T. A. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst. 5*, 1 (Jan. 1987), 47–76.

[14] 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.

[15] CHOCKLER, G., MELAMED, R., TOCK, Y., AND VITENBERG, R. Constructing Scalable Overlays for Pub-sub with Many Topics. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 109–118.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

[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] SCHNEIDER, F. B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[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.