Spindle: Techniques for Optimizing Atomic Multicast on RDMA

Sagar Jha
Cornell University
Ithaca, NY, USA
srj57@cornell.edu

Lorenzo Rosa
University of Bologna
Bologna, Italy
lorenzo.rosa@unibo.it

Ken Birman
Cornell University
Ithaca, NY, USA
ken@cs.cornell.edu

***Abstract*—Modern networking technologies such as Remote Direct Memory Access (RDMA) promise huge speedups in I/O bound platforms, but software layering overheads must first be overcome. Our paper studies this issue in a system that replicates small data objects using atomic multicast: a case in which internal synchronization is unavoidable, and any delay will be particularly impactful. Spindle, the methodology we propose, entails a series of optimizations including memory polling integrated with novel sender and receiver batching techniques, null-message send logic, and improved multi-thread synchronization. We applied Spindle to Derecho, an open-source library for atomic multicast, and obtained significant performance improvements both for the library itself and for an OMG-compliant avionics DDS layered on it. Derecho's multicast bandwidth utilization for 10KB messages rose from 1GB/s to 9.7GB/s on a 12.5GB/s network, and it became more robust to delays even as latency dropped by nearly two orders of magnitude. While our focus is on the Derecho library and the OMG DDS, the same techniques should be relevant to databases, file systems, and IoT infrastructures.**

***Index Terms*—RDMA, atomic multicast, OMG DDS;**

I. Introduction

Software infrastructures used in settings such as avionics have strict dependability requirements that are addressed through data and service replication. Application developers adhere to standards like the OMG Data Distribution Service (DDS) [1], which represents data as distributed objects (*topics*). These objects are replicated across the subscribers in accordance with Quality of Service (QoS) policies, which offer various fault-tolerance and delay guarantees. Yet today's DDS systems stop short of the level of QoS needed for atomic consistency [2] because the protocols were felt to be too costly. Our work started with an effort to revisit this assumption in light of RDMA.

Originally created as a hardware-offload technique for communication in HPC systems, RDMA is increasingly popular in cloud computing and compute clusters. The technology enables direct data transfers between the virtual memory of processes on remote machines, in which the only role of the CPU is to request the operations. On cutting edge hardware, with large messages, throughput can be as high as $200\,\text{Gbps}$ and one-way latency as low as $0.75\,\mu\text{s}$.

Prior research applied RDMA to strongly consistent fault-tolerance models such as State Machine Replication (SMR) [3], yielding RDMA libraries for atomic multicast [4]–[6]. One might expect that simply by layering middleware over such a library we would gain commensurate speedups. Unfortunately, however, this is not the case: we tried doing so for the OMG DDS API, but encountered a series of overheads associated with cross-layer handoffs, due to coordination, locking and copying. The central dilemma is a tension between overhead and delay: to minimize delay we would want to send each new message promptly, but this incurs overhead on every message. Those overheads can be amortized by batching but this delays messages until the batch is full.

To tackle this class of inefficiencies, we created Spindle: a set of optimizations for layering data replication services such as communication middleware on a high speed RDMA library, namely Derecho [4]. The approach is general, and should be of value for the entire class of strongly consistent replication systems [7], as well as for other forms of middleware and for other high-speed networking technologies, e.g., DPDK [8]. We obtain very low latencies and big speedups, enabling much stronger DDS QoS guarantees (failure atomicity, total ordering, message logging with durability).

Our work on Spindle starts with a close study of cross-layer handoff delays in situations where the application initiating the send runs on its own threads, distinct from the threads used in the underlying library. Such scenarios are common because RDMA libraries typically use a dedicated polling thread that is able to react instantly when data becomes ready. The intent is that polling and spin-lock overheads would be low, but we found that they are actually very significant. We identified three major issues, all caused by the composition of application threads and underlying RDMA library. First, when a library must discover new messages from the application or from the NIC, it turns out that even the briefest delays can have a hugely amplified impact on performance. Second, message-sending and receiving costs grow disproportionately if the application has multiple incoming or outgoing message streams. And finally, protocol control messages such as low-level receipt or stability acknowledgments can be surprisingly expensive: the latency to send minimal-sized acknowledgments with RDMA is about the same as the latency to send multiple KBs of application data. Thus while a layering of a platform such as the OMG DDS on Derecho seems quite simple (it requires just a few dozen lines of code), the solution only performs well with very large messages.

These observations lead to a series of insights. The first concerns batching. Derecho supported batching only on receipt. Spindle extends this into a unified opportunistic batching mechanism, generating batches of varying sizes and applied at all the stages: interaction with application threads, sending, wire-level reception, stability (safety) sensing, and delivery of upcalls and acknowledgments. Unified batching not only slashes overheads but also makes the system far more tolerant of scheduling delays.

A second insight relates to send-rate variability. Many systems, including Derecho, are implicitly tuned for steady data streaming: they work best when there is always a next message to send as soon as the platform is ready. However, at RDMA speeds, this assumption is unrealistic. We introduce a new null-send mechanism that automatically sends empty messages from a lagging sender with minimal overhead.

Finally, we introduce zero-copy message construction, enabling concurrent in-place preparation of new messages, while the underlying library automatically performs sends as messages become ready. This lock-free concurrency in turn requires a new style of opportunistic batch sends.

Our work is best understood as a next step in a progression of insights concerned with leveraging modern high-speed communication devices. Prior work explored separating control and data planes [9] and optimizing the match between RDMA and data movement [10], [11]. Derecho introduced a novel monotonic representation of control data that facilitates opportunistic batching. Spindle shifts the focus to the interaction between the application and the RDMA library.

II. Background

A. Derecho atomic multicast protocol

Derecho implements a virtually synchronous membership model within which it offers atomic multicast and persistent replication using variants of Paxos [2], [4]. Application processes are considered to be *external* or *internal*. External members are basically clients of a service: their requests must be relayed through an internal member. Internal members comprise the service itself, and are said to belong to its *top-level group*. The top-level group, in turn, is *sharded* into smaller replication sets, which we refer to here as *subgroups*. Multicasts in one shard won't interfere with multicasts in other shards, enabling linear scalability.

Within each subgroup, membership evolves through a sequence of *views*. A view change or reconfiguration occurs on failures, node joins and leaves, which are assumed to be relatively infrequent events. Atomic multicast is used when updating data replicated within a subgroup: All members receive these in the same order, and if a failure occurs, pending multicasts are either delivered everywhere, or aborted (not delivered anywhere) and then resent in the next view. OMG DDS maps easily to this model: topics are hashed to pick the subgroup where the topic will be managed, publish is implemented as a multicast, and subscribe watching for matching multicasts and then does an upcall. For an external client, all of these are relayed.

Our work did not change the Derecho protocols, but unified batching and null-sending are best explained with reference to it major components: a control layer and an associated data structure called the shared state table (SST), a data layer called the small-message multicast (SMC), and a polling framework using predicates that orchestrates the two. We briefly review each.

B. SST

In Derecho, the control layer is represented by a data structure called *Shared State Table* (SST). SST models each node's local state using a pre-agreed set of state variables. Each process in the system has a local copy of the table, which it accesses just like any other local table. A process is considered to *own* one row: it updates the state data in this row to notify other processes that its state has changed. The remaining rows are read-only copies of the control data of each of its peers. The actual transfers of data from process to process are performed using one-sided RDMA writes: having updated its own state variables, process A enqueues RDMA write requests on its NIC, and this causes a series of one-to-one remote data transfers to occur, updating the A row on processes B, C, D, etc. There is no locking. Thus, although updates from any single sender preserve that sender's intended order, updates from different processes can show up in different orders. It turns out that the classic Paxos protocols can be reexpressed to operate over this encoding, and that doing so yields an exceptionally efficient atomic multicast solution [4].

In Table Ia we see an example SST, used by Derecho for atomic multicasts in a single subgroup. In this example, we have five application nodes with ids $\{0, 1, 2, 3, 4\}$ organized in three subgroups with memberships $\{0, 1, 2\}, \{0, 1, 3\}$ and $\{0, 2, 4\}$. There are two state variables, $received\_num$ and $delivered\_num$ for each subgroup, abbreviated as literals r and d in the table. Messages from each node in a subgroup are received by all members in FIFO order. Thus every message in the subgroup can be assigned a unique sequence number, seq_num which is its index in the delivery order. The value of $received\_num$ for a subgroup member is the highest seq_num s such that it has received all messages with $seq\_num \leq s$ in the delivery order. Similarly, the value of $delivered\_num$ for a subgroup member is the sequence number of the latest message it has delivered. Both counters are monotonic, starting from -1.

Indeed, Derecho's SST is designed for *monotonic* data: counters that steadily increase, booleans that shift from false to true, and lists of integers that are updated only via appends or prefix-truncation. For basic types, such as counters, each entry will fit in a cache line. This maps nicely to RDMA, which is cache-line atomic and sequentially consistent. As a result, every subgroup member is certain to see an increasing sequence of values for every table entry. For example, when node 0 sees that $received\_num[1]$ for node 1 increases from 21 to 25, it can conclude that node 1 received the next four messages. For updates to a list that spans multiple cache lines, SST updates the list data, pushes the update with a

	r[0]	r[1]	r[2]	d[0]	d[1]	d[2]
node 0	8	25	-1	6	21	-1
node 1	9	21	—	6	20	—
node 2	6	—	-1	6	—	-1
node 3	—	23	—	—	21	—
node 4	—	—	-1	—	—	-1

(a) State for atomic multicast

s[0][0]	s[0][1]	s[0][2]	s[1][0]	s[1][1]	s[2][0]
{...}, 1	{...}, 0	{...}, 0	{...}, 7	{...}, 6	{...}, -1
{...}, 0	{...}, 0	{...}, 0	{...}, 7	{...}, 6	—
{...}, 0	{...}, 0	{...}, 0	—	—	{...}, -1
—	—	—	—	—	—
—	—	—	—	—	{...}, -1

(b) State for SMC data. {...} is a substitute for message content

Table I: Sample SST state at node 0 for 5 application nodes and 3 subgroups.

first RDMA operation, then updates a *guard:* a monotonic counter used to signal that the data is ready, and pushes it with a second RDMA operation. The RDMA memory-fencing guarantee ensures that any member that sees the counter update value will also see the updated version of the guarded data.

C. SMC

SMC (small-message multicast) is a multicast protocol implemented using a portion of the SST as a ring-buffer. Each subgroup has a fixed, configurable number w (for *window size*) of columns in the SST where each column entry for a particular node is a slot for sending messages in that subgroup. A slot is composed of a message area of a configurable, but fixed size (thus the maximum message size is fixed) and a counter. To send a message from a subgroup member, the application obtains a slot in its row from SMC, generates the message in it and calls send. SMC then updates the slot counter and issues RDMA writes to push the message and the counter to the subgroup members. On the receiver side, each subgroup member monitors the counter of one slot for each sender in which it expects to receive a message. Since the slots are utilized in ring buffer order for consecutive messages, an increase in the value of the counter indicates to the receiver the presence of a new message.

Messages remain buffered until they have been delivered to the application by every recipient. Thus a sending node needs to track deliveries to know when it can reuse a slot (failing to do so could cause an undelivered message to be overwritten). The intent is that value of w be large enough so that before running out of slots, some slots will have been cleared, enabling continuous sending.

An example of SST columns corresponding to the SMC state are shown in Table Ib, where slots are abbreviated using the literal s. The first three slots are for subgroup 0, the next two are for subgroup 1 and the last one is for subgroup 2. Thus in node 0's copy of the SST, the counter value of slot[0][0] being 1 for node 0's row indicates that node 0 in subgroup 0 has received 2 different messages in slot 0 from itself, while the counter value of slot[2][0] being -1 for node 4's row indicates that node 0 in subgroup 2 has not received any message from node 4. Only nodes 0 and 1 are senders in subgroup 1, thus the slots in node 3's row are not used. If node 0 were to send a new message in subgroup 1, it will use slot[1][1] of node 0 which will result in the increment of the slot's counter value to 7. The window sizes of 3, 2, and 1 respectively are just to illustrate the concept: a w value in the range 50 to 1000 would be typical for small messages.

Both SST and SMC guarantee that the memory layout of the application during a view remains unchanged. Thus the required memory can be allocated at each node at the beginning of the view, registered with the NIC and the addresses exchanged with all nodes for RDMA operations.

D. Monotonic predicates over the SST

RDMA is so much faster than traditional messaging that per-event interrupts would be prohibitively slow. Accordingly, Derecho's core uses a single polling thread that watches for work to do, then performs the needed action instantly. Because there is a single thread, no locking is needed, and because only a few conditions are of interest, any event of importance - a new message ready to send, a new message from some other peer, etc. - will be sensed within a few clock cycles.

To express this in a very general manner, we think of the Derecho polling thread as an evaluator of a series of *predicates*, i.e., conditional statements about data in the SST or other control variables, such as a flag indicating that an application thread has finished preparing a new message to send. When a predicate is found to be true, the thread will execute one or more corresponding code (*predicate body*). To avoid busy waiting when there is no work to do, the polling thread will quiesce if it loops for a while ($1\,\mathrm{ms}$ in current settings) and nothing happens; in this state, the next event will ring a form of doorbell to wake it up. In the active state, the performance of the predicate thread is central to the performance of Derecho.

This is where the concept of monotonicity is useful. In Derecho prior to our work on Spindle, when a receiver thread detected that new messages had arrived, it would opportunistically discover a batch of size one or more messages, and deliver them in order but as a single predicated action. This reduces overheads and also rides out some forms of message delay. However, the technique was used only on the receiver, and only for this one purpose. Spindle, as we will see, takes it quite a bit further.

There are three predicates of importance for Spindle:

Send predicate: Detects that the application has prepared new messages that are ready to send.

Receive predicate: Monitors the SMC slot counter for every sender in the subgroup. When the counter increments, a new message is present, and the receive trigger runs. The trigger can then increment the $received\_num$ counter if the incoming message is complete (large messages arrive in chunks), then push the updated value to other subgroup members.

Delivery predicate: Checks to see if the next message in the delivery order, say with id s, has become deliverable by

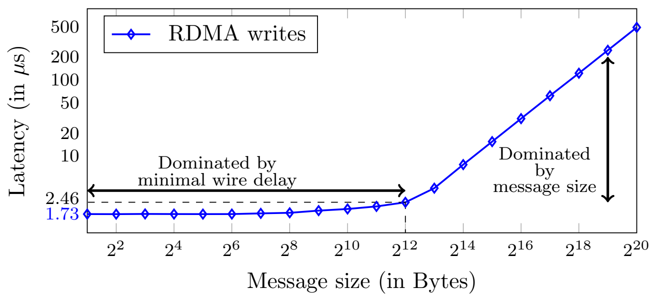


Figure 1: Latency vs data size on 100 Gbps Mellanox RDMA. Latency is nearly constant for up to 4KB message size.

checking if every member of the subgroup has received that message ($received\_num[i] >= s,\ \forall\ i$). The trigger delivers the message, updates the receiver's SST row, and then pushes the update.

All three predicates need to run at high speeds without thread scheduling delays. This explains the decision to employ a single predicate thread even though many subgroups share the SST: multiple predicate threads would contend for access to the SST memory as well as internal data structures shared across all subgroups, resulting in locking and possible cache-coherency delays. With a single thread, we lose the opportunity of multi-threaded parallelism, but also eliminate these overheads. Experiments made it clear that with our batching techniques, a single thread can efficiently handle tens of subgroups.

To illustrate these predicates in action, consider a new send by the application thread. The application first acquires a free SMC slot (meaning, the $delivered\_num$ entry of all subgroup members exceeds that of the slot). It constructs a message in the slot and updates the associated counter. The send predicate detects that the message is ready and initiates RDMA writes to other subgroup members. A lock is needed because the underlying data structures are shared with the predicate thread, and also because multiple application threads may be sending simultaneously. On the receive side, we see a similar stack, but now the receive predicate senses the incoming messages and the delivery predicate senses that they have become stable and can be delivered.

III. Spindle optimizations

Spindle was created as a response to a series of issues we identified by microbenchmarking a baseline version of the OMG DDS running on the Derecho atomic multicast. Although Derecho performance is outstanding with large messages, DDS publications are more frequently small. We found that performance for message sizes and sending patterns typical of DDS middleware (messages of up to a few KBs, a few dozen topics with subgroups that are heavily overlapping) was low. We set out to identify the issues that resulted in such poor numbers. In this section, we detail some of the issues and describe the Spindle techniques that respond to them.

A. Opportunistic batching

Performance of the baseline implementation is especially low for small messages because the latency of sending control data (acks for receiving a message, delivering a message) is comparable to the latency of sending the application messages themselves. Figure 1 plots RDMA write latency for different message sizes. Latency for small messages does not increase appreciably with the data size, increasing only marginally from 1.73 μs for 1-byte data to 2.46 μs for 4KB data.

The predicates described in Section II-D generate an ack (sent through fields in the SST) for every new message receive and delivery. This turns out to be expensive not only because of the comparatively high latency of control messages as described earlier, but also because posting an RDMA request to the NIC takes ∼1 μs. In the baseline system, the predicate threat spends more than 30% of its time posting RDMA writes.

A natural and effective way to address this is to batch events at different stages of the delivery pipeline: send, receive and delivery. Underlying this observation is the use of monotonicity for message sequence numbers: for instance, if 10 messages in a sequence are received before acknowledgment happens, the corresponding $received\_num$ entry can be simply advanced by 10 and a single RDMA write operation issued to push the acknowledgment through the SST. Batching acknowledgments will drastically reduce the number of RDMA writes issued which in turn reduces time spent by the predicate thread posting them.

The usual benefits of batching apply here as well. Batching can improve predicate thread efficiency (by improving locality of predicate evaluation) and can also allow a slow node to catch up with the rest by processing larger batches. In cases with multiple application subgroups, the original protocol makes no distinction between the predicates for different subgroups; that is, the predicate thread evaluates predicates of all the subgroups fairly. When some of the subgroups are not sending messages actively, this reduces the efficiency of the predicate thread, lowering performance. Batching, if done correctly, can help mitigate this issue by adapting batch sizes to the workload. Sending multiple application messages carries an additional benefit, that of sending a larger amount of application data in a single RDMA write, which results in better latency scaling as seen in Figure 1 and better wire-level efficiencies (RDMA data is sent in frames with a capacity tied to the network speed, and a very small message can leave much of a frame unused).

Batching is not new, but traditional fixed-size batches are poorly matched to RDMA, especially in latency-sensitive settings like network services. If a system ever pauses sending to accumulate the next batch, the associated delay in sending proves to be remarkably disruptive. In one experiment, we explored waiting to send a fixed batch of messages on top of receive and delivery batching. Performance collapsed and latency soared even for very small batch sizes.

Accordingly, we expanded the batching architecture into a unified technique that covers all stages of atomic multicast.

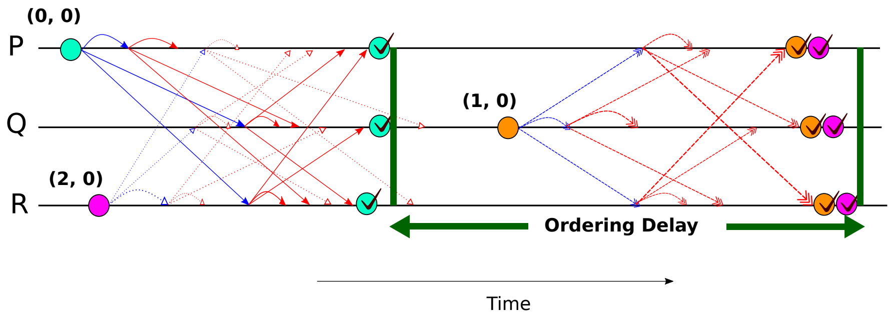


Figure 2: Delays at a sender can impact performance of the system by slowing down other senders.

The application thread generates messages as usual but the send predicate checks to see if one *or more* messages are ready. If so it aggregates them on the fly, and sends a batch. The receiver predicate looks through the sequence of slots for each sender, receiving all new messages that it can find. The delivery predicate delivers all messages that have become deliverable, in the right order. Opportunistic batching is *self-balancing*: a batch can be smaller or larger depending on the number of events a predicate discovers as it loops. This makes execution more robust to delays by allowing lagging nodes to catch up and does not involve waiting of any kind.

To implement these ideas, we modified the predicates described in Section II-D as follows:

Send predicate: The new version of predicate issues RDMA writes that send all the queued data generated in contiguous ring buffer slots to the other members. If the queued sends have wrapped around the ring buffer, it issues two RDMA writes per remote member accordingly. Since the messages go into discrete slots, each of a fixed size, the predicate pushes the leftover space in the slots too (if messages do not take up the entire slot area). We do not anticipate any downsides to doing so, since the latency for small messages does not rise appreciably and batching allows us to send multiple messages in a single RDMA write.

Receive predicate: For every sender, this predicate goes through the corresponding slots to find all messages that have arrived, stopping at the first empty slot. The trigger updates the count of received messages appropriately and pushes the updated value to the other subgroup members.

Delivery predicate: This predicate takes the minimum of the received count for the subgroup members to find all undelivered messages that have been received by all members in the subgroup. Those messages now become deliverable. The trigger delivers all those messages, updates the receiver's SST row, and then pushes the update.

The solution enables a highly efficient in-place message construction. In our new approach, a message constructor first obtains a pointer into a free SMC slot. This can be done without any locking, and the application thread can then asynchronously construct the outgoing object. When finished it marks the message as ready to send, and the send predicate does the rest. The solution is lock-free, zero-copy, and it will preserve ordering for single-threaded applications. In contrast, the one-to-one methodology of Kalia et al. [11] is also opportunistically batched, but their in-place object construction scheme is limited to one client request at a time, and the associated data must fit within an RDMA *immediate data* field (8 bytes). Our approach avoids both limitations.

B. Null-sends

Our second optimization is an enhancement to the round-robin message ordering used in Derecho's atomic multicast. We focus on failure-free runs (the epochs described in Section II-A), under the assumption that failures are relatively infrequent and that the delays for reconfiguration (a few milliseconds) aren't likely to be a major concern. Recall that each epoch is associated with a membership view listing some fixed, agreed upon list of members and known to every member of the system. Derecho transforms these properties into an atomic multicast delivery order. Specifically, the system operates in rounds, and during each round one message from every sender is delivered. This eliminates competition for multicast slots, which results in a back-and-forth messaging pattern in classic Paxos protocols.

The problem we address is that application sending rates can be variable - it is impossible to guarantee that every sender will continuously be ready to send the next message on demand. For example, senders may need to interact with IoT devices when constructing messages, may be delayed by CPU stalls, or there could be some sort of locking delay.

In one experiment we even saw a situation in which a library used a C++ spin-lock rather than a mutex lock. The intent was that spin-locks would be faster than a mutex, but in fact the experiment revealed a case where this was exceptionally slow: the C++ 17 implementation of spin-locks turns out to be unfair on NUMA hardware and can favor one thread while disadvantaging other threads. In the particular case, Linux decided which core each thread would run on, and the application's sender thread turned out to be running very slowly compared to the Derecho thread that checked for new messages!

We illustrate this issue in Figure 2. We have three nodes, P, Q, R, that each send a message denoted by green, orange and pink circles, respectively. The blue arrows denote the send of a message, while the red arrows denote the send of an acknowledgement. Sender Q sends its message much later than P and R. Since the delivery order is P, Q, R, we see that while P's message is delivered as soon as each node learns that it has been received by all the nodes, R's message has to wait until Q's delayed message is delivered. With Derecho's ring buffer implementation multiple messages can be sent at the same time, but delays in sending by a single member can leave multiple messages stuck waiting at the receivers, if that delayed sender is next in the round-robin order. The ring buffers of active senders will soon fill up with undelivered messages, preventing them from sending more messages.

The obvious way to deal with this issue is to detect when a sender has fallen behind and then send dummy 0-sized messages (called *nulls*) from that sender to expedite the delivery of application messages from other senders. At the time of delivery, null messages can simply be discarded and the resulting sequence of delivered messages will still be same across the members. The problem is that we detect the potential delay in receiver logic, and yet a null-send is logically a sender-side action. Moreover, designing an efficient null-send scheme that decides when and how many nulls to send at RDMA speeds has not been explored in the literature: Prior null-send protocols ran on older TCP networks, where the processor was so fast relative to the network that small sending delays did not risk appreciable performance loss.

Sending a null too soon at RDMA speeds and latencies interferes with normal message delivery, because the sender may have been about to send a legitimate application message. Over time, this will result in sending too many nulls which, owing to RDMA's relatively high 1-Byte latency (Figure 1), will add up to a significant cost. On the other hand, sending a needed null even a few microseconds too late is undesirable because these tiny delays still represent significant lost bandwidth. We desire four properties:

1) **Sender-invariance**: Performance with only a subset of senders sending continuously does not drop appreciably.
2) **Low-overhead**: Performance does not degrade significantly when all senders are sending actively compared to the same case without any null-send scheme in-place.
3) **Correctness**: Under all circumstances of senders sending at different rates (and possibly some senders not sending at all), the delivery pipeline never stalls.
4) **Quiescence**: When all senders are inactive, the system attains a quiescent network state where no nulls are sent.

In Spindle, when a sender node receives a message, it immediately sends a single null message if this null will precede the received message in the delivery order. This determination can efficiently be made by checking the per-sender index of messages sent and global sequence numbers.

Correctness and Quiescence can be proved as follows. Denote a message in a subgroup as $M(i,k)$ where i is the sender rank (in the senders list) and k is the sender index, equal to the number of messages it has sent in the subgroup. Round robin delivery imposes a total ordering $<$ on the messages: $M(i_1,k_1) < M(i_2,k_2) \iff k_1 < k_2 || (k_1 = k_2 \wedge i_1 < i_2)$. Assume that node i receives a message from a sender with rank j in round $k, M(j,k)$. Without loss of generality, assume $i < j$. The null-send scheme will send a null iff current round number of sender i, l (equal to the number of messages sent by i), is such that $l < k$.

Suppose a null is sent. It is an easy induction to deduce that $l = k - 1$ (consider what happened when i received $M(j, k - 1)$). That is to say, that the null-send keeps every sender within one round of each other in terms of the number of messages it has received vs. sent. Thus after $M(j,k)$ has been received by all nodes, their own round number is greater than or equal to k. This statement is imprecise for nodes with rank $> j$ but without loss of generality, we can take j to be the highest ranked sender. This implies that sends of all messages that precede $M(j,k)$ have been initiated, meaning that $M(j,k)$ will be delivered barring failures. Hence no deadlock arises.

We now show that the system reaches a quiescent state when no application messages are being sent. The complicating factor is that $M(j,k)$ may itself be null. However, if $M(j,k)$ is null, it was sent by sender j in response to another message $> M(j,k)$ received by it. This chain cannot go on forever and thus will finally terminate in a non-null, application message. Thus, if no sender is actively sending messages, no nulls will be sent. This explains why we do not need to check if the received message is a null. In fact, sending a null in response to another null may expedite the delivery of subsequent application messages (this would be useful if messages arrive in different orders at different nodes). It is straightforward to combine null-sends with batching: After the receiver predicate finishes an iteration, it sends the determined number of nulls as a single integer.

The intention is that Spindle's null-send scheme should dynamically adjust to real-time delays, lagging nodes, and other disruptions, maintaining high levels of performance when a sender is unintentionally delayed. If an application deliberately will not send messages from some source for an extended period of time, it should declare the number of rounds of inactivity to the subgroup members which can then appropriately modify the message delivery sequence. If a node is never going to send again, it can be marked as a non-sender when it joins the subgroup or later, during a reconfiguration. Thus, the (small) overheads of the null-send scheme are seen only in the event of unanticipated delay. Moreover, the null-send optimization is quite general, and has features that could be used even in other round-robin protocols that already use null-sends, such as Ring Paxos [12], [13]. We will return to this point in Section V.

C. *Efficient thread synchronization*

Efficient thread synchronization is crucial to high performance. For systems that rely on polling (see Section II-D), we noted one potential inefficiency: when application threads interact with the polling thread, a lock is required that protects against concurrency conflicts. Yet, this can delay other critical operations that use the shared state: when concurrent accesses are frequent, the entire system slows down. For instance, in Derecho we found that many predicates interleave access to SST data with RDMA write operations. RDMA writes are costly to post: they can consume 20-50% of the time spent in the entire predicated test and code block.

Accordingly, we restructured all Derecho predicates so as to place the RDMA write calls only at the end. This can be done safely when a predicate's logic does not depend on the shared state of the SST at a remote node, but only what is present in the local SST. Then we can safely release the lock before we proceed to issuing RDMA writes. Any parallel access of the SST by other threads is safe because of the following two properties of the SST: (1) simultaneous reads and writes to the SST are safe since the variables fit within cache-lines and (2) any updates to the variables being pushed that might occur between when the predicate releases the lock and when the push actually occurs are monotonic. Thus, the eventual push will simply batch the original information with additional data.

D. Delays caused by the receiver

Derecho offers safe and consistent delivery of messages. The application "acts on" (or consumes) a message only when it is delivered. The protocol delivers messages in the critical path - the predicate thread discovers that a message is deliverable, calls into the application with that message, and updates and pushes the corresponding $delivered\_num$ after the upcall returns. Waiting until all receivers have consumed the message makes it safe for Derecho to reuse the associated slot for sending a fresh message.

As a result, delays in the delivery upcall have a dramatic performance impact, because the predicate thread cannot continue its run until the upcall returns. To quantify this effect, we built an application in which message delivery upcalls take $1\,\mu s$, $100\,\mu s$ or $1\,ms$ and found that performance decreases by about 9%, 90%, and 99% on average, respectively. For larger delays of $100\,\mu s$ and $1\,ms$, performance degenerates to one message delivered per delay time. This finding confirms that the protocol is highly sensitive to the time taken by the application in processing the message.

To mitigate the impact of delays in delivery, we offer two viable options: (1) Applications can support a batched delivery upcall, which consumes all messages that are deliverable. If processing a batch of delivered messages takes less time overall, we obtain performance speedups. (2) Applications can simply move the data into a separate memory area via memcpy and return from the upcall. For small messages, costs of memcpy are not terribly high. We evaluate the overhead of memcpy during delivery in Section IV-D.

IV. Evaluation

In this section, we evaluate the impact of the Spindle optimizations discussed in the previous section. The evaluation focuses on throughput, defined as the amount of application data delivered per unit time (GB/s, averaged over all nodes). Each test is run 5 times - we plot the average values and show error bars corresponding to one standard deviation. We test on our local cluster consisting of 16 machines connected with a 12.5GB/s ($100\,Gbps$) RDMA Infiniband switch. Each machine has 16 physical cores and 100GB of RAM.

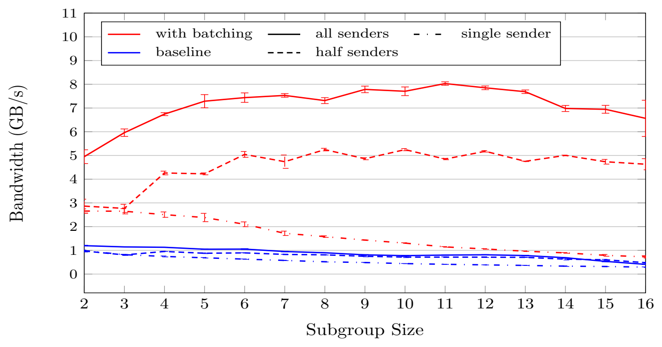


Figure 3: Performance for single subgroup with opportunistic batching. Performance improves by up to 16X.

We evaluate a multitude of scenarios with one or multiple subgroups with one or all of them sending messages actively, senders sending continuously or with delays, delays in various parts of the protocol. For each optimization, we show the cases most directly impacted by the change. Subsequent optimizations are evaluated on top of the previous optimizations, showing incremental improvements. Finally, we look at the overall impact of Spindle on the application that motivated our effort: an avionics DDS.

A. Opportunistic batching

1) Single subgroup continuous sending: Many systems have just one replication group, for example to replicate a component or data or to support event notifications. In this case, all senders continuously stream messages in a tight loop. We vary the subgroup size from 2 to 16 using message sizes 1B, 128B, 1KB and 10KB, in three patterns - all senders (every member is a sender), half senders (only half of the members are senders) and just one sender. Small message sizes can go as far as few hundred KBs, but by limiting it to 10KB, we can leverage the power of aggregation while keeping within the limit. Consistent with the SMR approach, all members are receivers in all cases and deliver all sent messages in the same order. Each sender sends a total of 1 million messages. The experiment finishes when all messages have been delivered.

Figure 3 plots performance for this test for 10KB messages and compares it against the baseline performance. As is clear from the graph, opportunistic batching alone outperforms the baseline by about 9X for all senders, 6X for half senders and 3X for one sender on average. The peak bandwidth attained is 8.03GB/s for 11 members, giving a maximum network utilization of 64.2%. Performance also scales much better with increasing number of senders, for instance, it is 16X of the baseline performance with 16 senders. Performance with just one sender declines with the subgroup size as the algorithm pays the price of increased coordination overheads.

Consistent with Figure 1, performance is proportional to the data size for both the baseline and the optimized version. Hence, the number of messages delivered per second remains about the same for different small message sizes. Figure 4 confirms this observation for the optimized version. As such, all subsequent experiments only show data for the 10KB case.

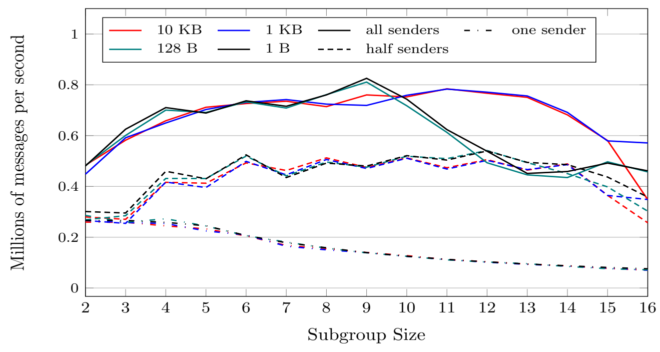


Figure 4: Rate of delivery for single subgroup with opportunistic batching. Derecho has a second communication larger, RDMC, for very large subgroups or messages. Although RDMC was not evaluated in our work, shifting to it might be advisable for subgroups with more than 12 members.

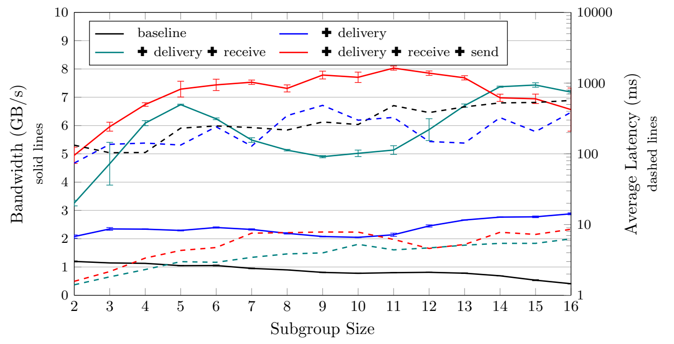


Figure 5: Performance gains with batching applied to successively more stages of the pipeline for all senders. Throughput (left Y-axis) is shown by solid lines, and latency (right Y-axis) using dashed lines. Both metrics show significant improvements relative to our baseline system.

It is interesting to learn the impact of batching at different stages of the protocol. Figure 5 shows the incremental effect of adding delivery, receive and send batching successively. It is particularly noteworthy that our optimizations improve *both* throughput and latency across the full range of subgroup sizes. In contrast, as noted earlier, traditional forms of sender-side batching sharply increase latencies, and may significantly reduce bandwidth by leaving the RDMA network idle while waiting to accumulate the next batch.

We computed some metrics for the 16 senders case to gain an insight into the improvements. Comparing the baseline against the optimized version, we find that the number of RDMA write requests goes down from 18.2M to 1.1M, time spent by the polling thread in posting RDMA writes goes down from 64.84s to 4.29s and the sender thread spends time waiting to find a free buffer only for 52.7% of the much reduced experiment time (as opposed to 97.6% of the total runtime of the baseline).

2) Suitable ring buffer size: Batch sizes for our batching optimization are influenced by the subgroup window size. After all, the number of messages that can be sent or received in one batch is limited by the number of slots. An unreasonably

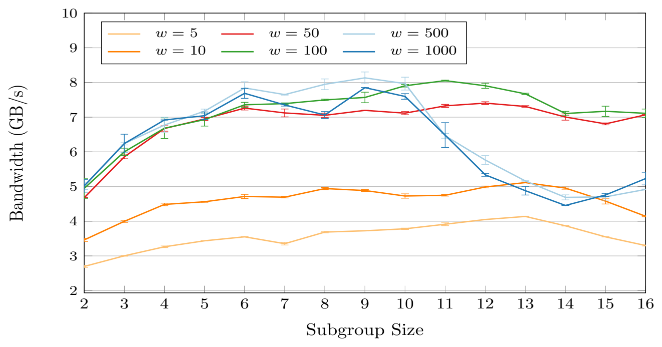


Figure 6: Performance with different window sizes when all nodes are sending messages continuously

small window size does not allow for optimal batching, while an excessively large window size forces the predicate thread to cover too large a memory area. In this experiment, we measure the performance of the single subgroup all senders case, varying ring buffer size.

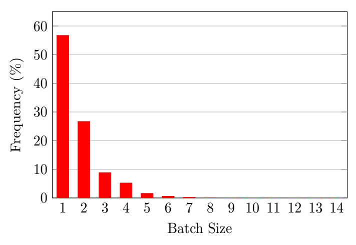
Figure 6 plots the results. Even a small window size of 5 increases performance by 4.5X average compared to the baseline with 100 window size! The highest performance is obtained for a window size of 100. Consequently, all our experiments for 10KB message size use a window size of 100. It is important to note that performance with window sizes of 500 or 1000 starts declining after 10 nodes, quite likely because the polling area increases considerably and large batches of application messages (if 200 messages of 10KB are sent in one RDMA write, total data size is a little less than 2MB) do not give good throughput with a simple multicast send scheme of SMC (sequential send). This suggests that applications should use a window size around 100 instead of pinning large buffers with RDMA.

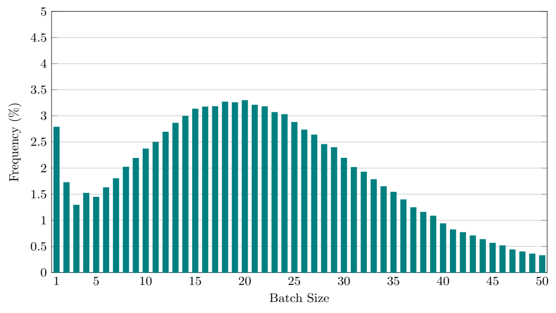
For the single subgroup case, the SST at each node consists of just two columns for the subgroup state $received\_num$ and $delivered\_num$ which takes 16 bytes of space per row, and the slots for the SMC. The total space for the slots at each node is

$$n * w * (m + 8)$$

where n is the number of nodes (rows), w and m are the window size and maximum message size (8 less than the size of the slot which also contains a counter) for the subgroup. For 16 members, 10KB message size, and w = 100, the total space per subgroup amounts to roughly 16MB. This suggests that applications can easily scale to tens of subgroups with the total memory allocated within few hundred MBs.

It is interesting to learn the batch sizes for different steps of the pipeline. Figure 7 plots the histograms for a window size of 100 for the single subgroup, 16 senders case. Messages are typically sent in small batches of less than 5, while most delivery batches are multiples of 16 suggesting that about 1-5 messages from each sender are typically delivered in a batch. Different mean batch sizes for send, receive, and delivery is further proof that a rigid batching scheme with fixed batch sizes is unlikely to work well in practice, especially


(a) Sends


(b) Receives

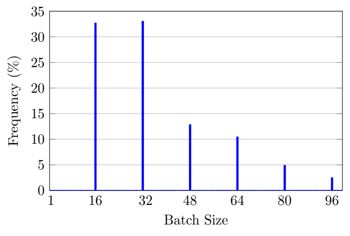

(c) Deliveries

Figure 7: Batching histograms for the three protocol stages. Receive merges data streams from all senders, forming larger batches. Delivery computes an extra level of stability over all members, forming even larger batches.

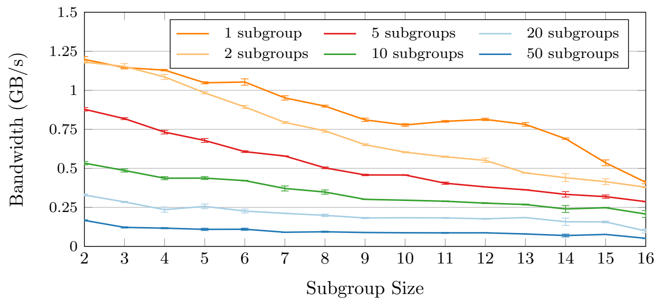


Figure 8: Performance of baseline for single active subgroup

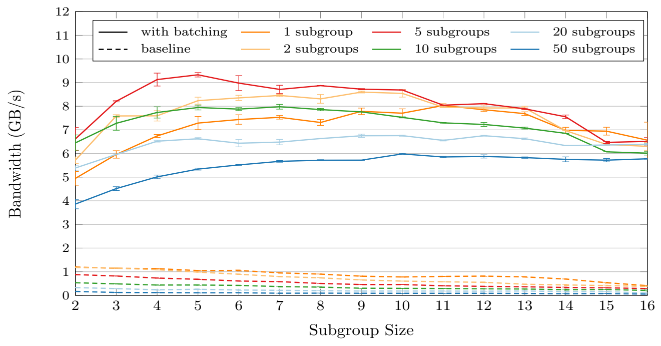


Figure 9: Performance with opportunistic batching for single active subgroup.

in heterogeneous environments where nodes are running at different speeds.

3) *Single active subgroup:* In this case, all nodes belong to all subgroups, but each node continuously sends 1M messages in just one of them. Our goal here is to expose inefficiencies inherent in the baseline which evaluates all subgroups' predicates fairly and show how opportunistic batching compensates for those inefficiencies.

In the baseline implementation (Figure 8), performance consistently decreases with increasing number of subgroups. Adding a single inactive subgroup degrades performance by 18% on average, while the performance with 50 subgroups is one-tenth of the performance with the sole active subgroup. On the other hand, Figure 9 plots the results for the optimized version. We see that adding more subgroups does not decrease performance invariably but even increases it in some cases. This is an artifact of batching: delays can sometimes lead to more efficient executions, due to larger average batch sizes. Clearly, there is a lot of potential in a more adaptive batching scheme that adjusts according to the circumstances. Even with 50 subgroups, the performance declines much more graciously compared to the baseline. This stability should help developers feel confident that a decision to use overlapping subgroups will not harm application performance.

For the baseline, for a sample run with 16 nodes, the percentage of time spent evaluating the active subgroup's predicates goes down from 54% for 2 subgroups to less than 15% for 50 subgroups. With opportunistic batching, this number is about 99% for 2 subgroups, 90% for 10 subgroups and 48% for 50 subgroups. The average batch sizes for sends, receives and deliveries increase from {1.72, 22.18, 35.19} for 1 subgroup (Figure 7) to {6.20, 49.36, 127.74}, {21.67, 79.15, 334.48} and {50.45, 207.46, 638.57} for 2, 10 and 50 subgroups, respectively. This shows the adaptability of opportunistic batching to real-time delays.

Opportunistic batching also vastly improves performance for the multiple active subgroups case, where multiple subgroups are actively sending messages. However, performance drops considerably with increasing number of subgroups. We infer that the predicate thread spends an increasing amount of time posting RDMA writes for the different subgroups, delaying timely sending of application messages. Our optimization of efficient thread synchronization resolves most of these overheads, hence, we evaluate this case in Section IV-C.

B. *Null-send scheme*

1) *Delayed sending:* In any real system, there may be unpredictable delays in sending. In this experiment, we simulate such a case for the all senders case by introducing a fixed delay after each send at either one or half of the senders. Senders that are not delayed send as fast as possible. We tried several different delays: 1) 1 µs, a minimal delay close to the network latency, 2) 100 µs, much larger than network latency, yet realistic for applications, 3) a lengthy delay. In each case, the delay is implemented with a busy-wait loop. We measure bandwidth after a fixed number of messages have been delivered. As detailed in Section III-B, the baseline

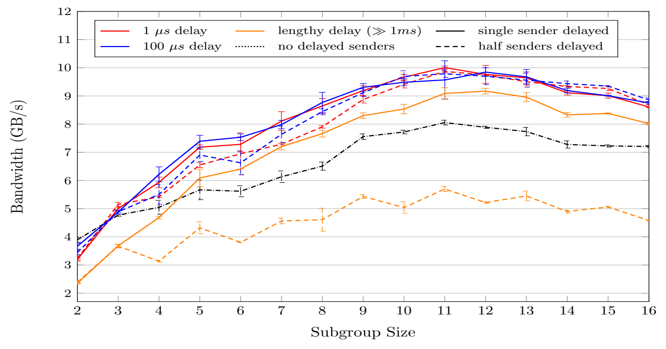


Figure 10: Data for sender delay test with null-sends.

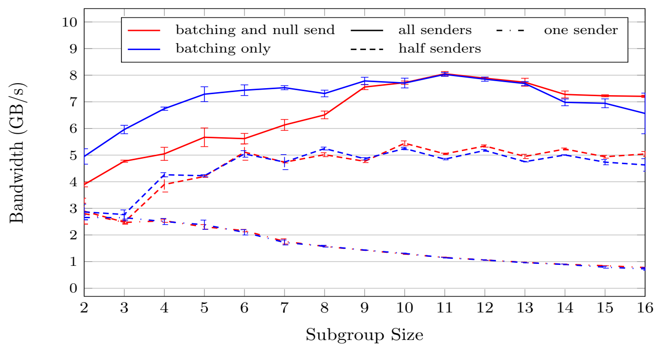


Figure 11: Impact of null-sends on continuous sending.

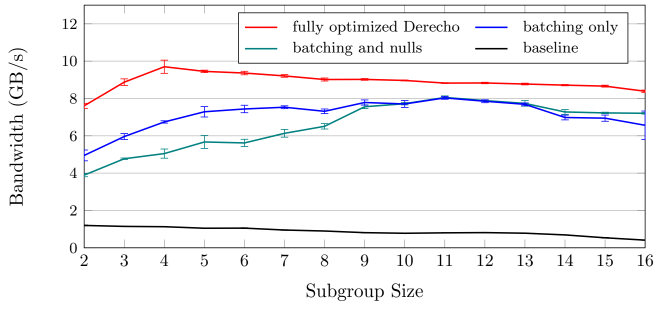


Figure 12: Impact of efficient synchronization.

protocol does nothing to adjust for these kinds of delays. This is the primary test for the null-send scheme.

Figure 10 plots the results, which are surprising. For every case other than where half senders are delayed indefinitely, performance increases, peaking at 10.0GB/s. This is because small delays lead to larger average batch sizes and large delays lead to more efficient bandwidth utilization by the remaining senders. This shows that the system adapts very well to real-time delays.

In case of 16 nodes sending 1M messages each with 1 sender delayed by 100 µs, the delayed sender sends one or more nulls in 517K iterations of the receiver predicate while a continuous sender sends them in only 189K iterations. The average inter-delivery time between consecutive messages from a continuous sender and a delayed sender comes down drastically and in fact, decreases from 3.779 µs for 2 nodes to 1.617 µs for 8 nodes and 1.192 µs for 16 nodes. This confirms that nulls accelerate delivery of application messages.

2) Continuous sending: Nulls may be inserted even when all senders are sending continuously because of inevitable small relative motion between the members in sending and receiving messages. This could potentially either reduce performance if nulls interfere with application messages or increase performance if nulls compensate for batching-induced delays. In a real setting, where sending patterns are more varied, null-sends will improve performance as established by the previous experiment.

In this experiment, we measure the impact of null-sends when all senders are sending continuously in a subgroup. We compare Derecho with only opportunistic batching against Derecho with null-sends on top of opportunistic batching.

Figure 11 compares performance. For all senders, performance is initially worse because there is less scope for improvement, and therefore nulls have a minor but deleterious impact. With larger subgroup sizes, small delays become more prominent. Here null-sends accelerate message delivery leading to improved performance. The drop for smaller nodes is significant for all senders (up to 25%) and almost negligible for half senders. No nulls can ever be sent for one sender; the graph confirms that no overhead is introduced.

3) Additional Null-Send experiments: We also conducted additional experiments that exposed the null-sending scheme to increasing complex and disruptive delays, such as by declaring all members of a shard as senders, but then having just one member do all the sends. For reasons of brevity, we omit details, but in all cases the mechanism successfully compensated, allowing the active senders to run at full speed, while filling any gaps caused by inactive senders. Null-sends are not always the entire solution: in Sec. III-B we mentioned a case in which an unfair C++ spin-lock caused a library to malfunction in a way that drastically slowed some senders. The null-send mechanism prevents such slowdowns from propagating to other senders, but doesn't fix the slowdown itself. Still, the resulting pattern highlighted the slow sender. This focused our attention, and ultimately enabled to track down the root cause, at which point we were able to modify the library in question to use a mutex lock, restoring full performance.

C. Efficient thread synchronization

We evaluate the effect of restructuring predicates to move RDMA writes to the end and release locks before issuing the writes. We evaluate the performance for the single subgroup, all senders and the multiple active subgroups cases.

Figure 12 plots the results for single subgroup. The optimization, on top of batching and nulls, improves performance considerably by about 1.4X average. The maximum network utilization of 77.6% is reached for 4 members which stays very stable all the way to 16 members.

Figure 13 plots the results for multiple active subgroups, comparing them with the baseline. The results show excellent scaling with the number of subgroups. The performance remains relatively stable for all subgroup sizes.

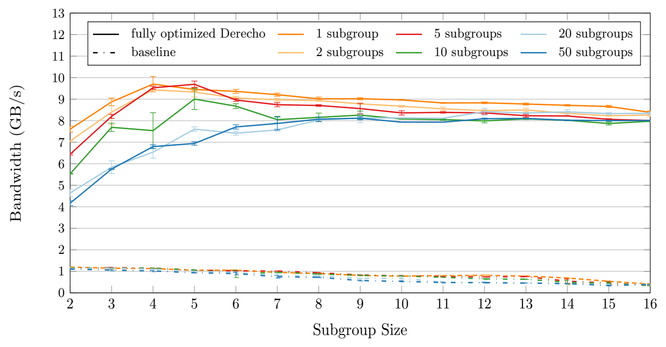


Figure 13: Final performance with all optimizations for multiple active subgroups.

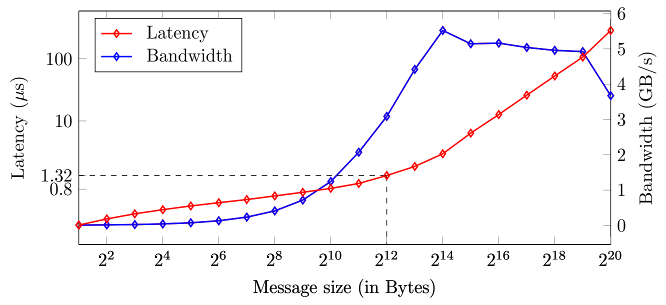


Figure 14: Performance of memcpy with data size.

D. Delays caused by memcpy

RDMA is based on the zero-copy idea: memory copy within a node is much slower than remote copy over a network. High RDMA speeds impose considerable strains on application memory management. In our optimized Derecho implementation, for instance, sends and delivery must finish quickly to stay close to the optimal performance. It may not be practical to avoid memory copy when generating a message in the ring buffers (for example, if the application receives data out of band from external clients) or to give up ownership of a message immediately after delivery. However, memory copy is not that expensive for small messages. Figure 14 measures the latency and bandwidth of memcpy on one of our machines. The latency remains low up to a few KBs, then quickly deteriorates for large message sizes.

For this reason, we evaluate a pragmatic approach where the application copies data from external buffers into library-provided slots before sending and copies data out of the ring buffers in delivery. We again evaluate the single subgroup case for 10KB messages. For smaller messages sizes of 1B and 128B, memcpy carries much less overhead.

Figure 15 shows that there is a decline for all senders, though the bandwidth still remains consistently around 7.5GB/s. Performance declines slightly for half senders, while there is almost no decline for the single sender case as the memcpy induced delays are likely absorbed into the coordination overheads. We also evaluated this case for the extreme case of 1B messages and observed no performance loss.

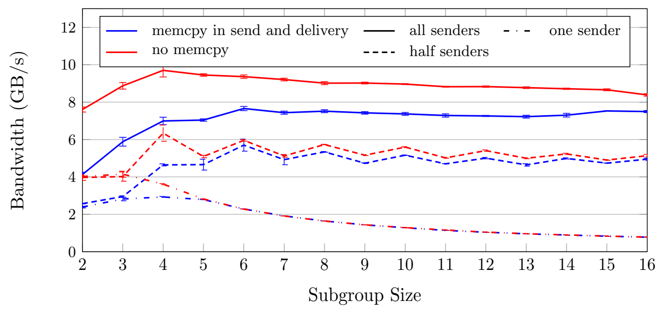


Figure 15: Performance with memcpy in send and delivery.

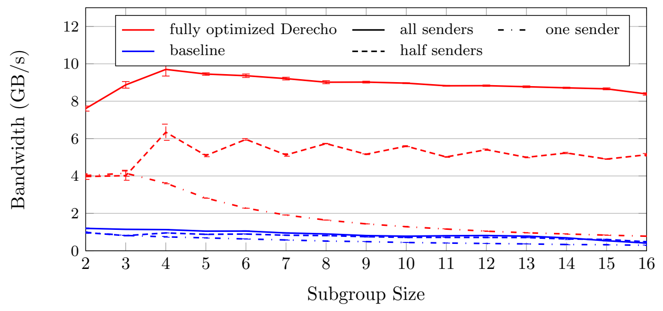


Figure 16: Final throughput numbers for a single subgroup.

E. Final results

Figure 16 shows the final numbers. As noted, although our optimizations were focused on throughput, Figures 5 and 17 both show substantial improvement in latency. Note that the logarithmic Y-axis scale magnifies error bars for small latency values.

F. DDS evaluation on Spindle + Derecho

We now consider how the Spindle optimizations affect a prototype DDS system we built with the purpose of getting a stronger form of consistency than what existing implementations can offer. We mapped the OMG DDS API, Data-Centric Publish-Subscribe (DCPS), to the underlying Derecho system by forming a single Derecho "top-level" group that includes all publishers and subscribers. Then, for each topic, we form subgroups containing only the processes that publish or subscribe to that topic (the actual Spindle DDS also supports "external clients" that connect to the DDS via TCP or RDMA, requiring an extra relaying step, but we did not evaluate that mode of use). The user then defines data types and publish and subscribe topics as replicated objects of those types. Importantly, the Spindle-DDS permits developers to construct messages "in place", and then mark them as ready to send. Had we used a model in which the application allocates space elsewhere to create its messages, the resulting overheads would have sharply reduced performance.

We tested performance for a single DDS topic with a single publisher and varying number of subscribers. We defined a *Sequence* data type, which represents a simple byte sequence, to be exchanged among the entities. The publisher continuously publishes 1 million topic samples of type Sequence, each of

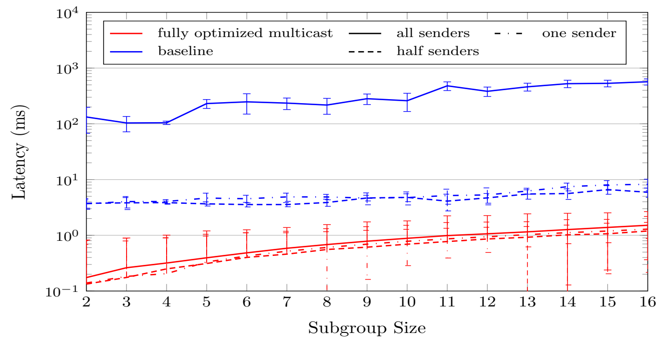


Figure 17: Final latency numbers for a single subgroup.

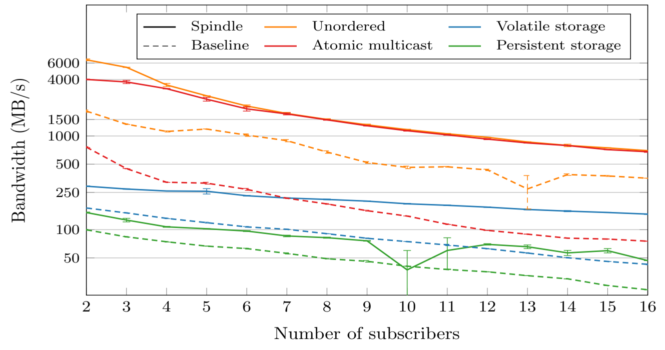


Figure 18: DDS performance improvements with Spindle optimizations for all 4 QoS levels.

10KB size. To stress the network performance, publishers and subscribers are all on different nodes.

An OMG DDS can offer different levels of consistency by combining different parameters (*QoS policies*). Our DDS has four: 1. **Unordered**: Data is delivered to the application without waiting for stability and discarded after delivery. This is relevant for applications that do not need any kind of ordering or reliability. 2. **Atomic multicast**: This maps directly to Derecho's atomic multicast, and data is discarded after the delivery upcall. 3. **Volatile storage**: Incoming data is copied and saved on the receiver node's memory (this allows a joining subscriber to catch up). 4. **Logged storage**: Data is additionally appended to a log file on SSD storage, and is used for debugging, and in applications that track the evolution of a reported measurement over time.

Figure 18 compares bandwidth for our baseline DDS implementation with one that uses the Spindle optimizations. We see that Spindle improves performance for all four cases relative to the baseline. Whereas Spindle-DDS has nearly the same performance for unordered and atomic multicast mode, notice that the pre-Spindle baseline's performance decreases considerably with each additional QoS level. This validates our effort to reduce communication overheads. Interestingly, Spindle's performance improvements even carry over to the volatile and persistent storage modes, despite the fact that these are limited by memory copying and disk I/O costs. The finding supports our hypothesis that whole stack optimization yields a steadier end-to-end data stream even when a variety of potential bottlenecks are present.

V. Related Work

There has been other work on using RDMA to accelerate state machine replication [4]–[6] and key-value storage [10], [11], [14], [15]. The separation of control from the data plane originated in BarrelFish and Arrakis [9], [16] and the iX μ-kernel [17] employs a similar separation of layers. Derecho introduced its own optimizations, but also incorporated older ideas, such as control-plane and data-plane separation and opportunistic delivery batching. Spindle goes much further, exploring and optimizing overheads stemming from layering real applications over the high-speed communication substrate. We would argue that only a full-stack perspective can yield a zero-copy lock-free and delay-free solution capable of running at the full capacity of a modern RDMA device.

Earlier we compared Spindle with the work of Kalia et. al. [11]. Other similar studies [18] tackle the challenge of systematically improve the performance of RDMA-based server interactions through a combination of different optimizations. Those works optimize RPC-style and one-to-one streaming applications for RDMA systems dominated by one-to-one interactions. However, our work revealed that more complex systems expose more subtle causes of inefficiencies that arise, for instance, from multiple software layering of applications over middleware over RDMA, and in some cases are triggered by background events that cause delays. There are also interesting similarities to μTune [19], a thread-level coordination package for low-latency, high-throughput gRPC-based μ-services: for example, authors recommend a single polling thread for message discovery, to avoid contention. This is an important assumption in Spindle too, and common in modern RDMA systems.

One of the Spindle optimizations involves sending null messages to avoid delays if a sender is not ready to send a new multicast when its turn arises. This idea was first explored in the Totem [20] and Transis [21] systems, and similar mechanisms have been used in modern Ring Paxos protocols [12], [13]. However, Spindle takes null-sending much further, first by identifying a step in the critical path where the need to send a null can efficiently be recognized, but also allowing early transmission of non-null messages to maximize wire utilization. When combined with opportunistic batching, we obtain far higher performance.

VI. Conclusion

We presented Spindle, a methodology for whole-stack optimization in complex, layered middleware designed to leverage RDMA communications. Key innovations include unified opportunistic batching, null sends, and zero-copy lock-free in-place message construction. Our methods give a speedup of up to 20X over a baseline implementation for a single replication subgroup and achieve up to 77.6% network utilization of a 100 Gbps network, while also dramatically reducing multicast latency. We applied Spindle to an OMG DDS, but our optimizations would be equally relevant in other kinds of systems.

Acknowledgments

The authors wish to express gratitude to the anonymous reviewers for their constructive comments and suggestions. This work was supported by AFRL under the SWEC program, Microsoft, Nvidia/Mellanox and Siemens.

References

[1] "OMG Data Distribution Standard." [Online]. Available: https://www.dds-foundation.org/omg-dds-standard

[2] K. P. Birman and T. A. Joseph, "Communication support for reliable distributed computing," in *Proceedings of the Asilomar Workshop on Fault-Tolerant Distributed Computing*. London, UK, UK: Springer-Verlag, 1990, pp. 124–137. [Online]. Available: http://dl.acm.org/citation.cfm?id=645425.652315

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, p. 299–319, dec 1990. [Online]. Available: https://doi.org/10.1145/98163.98167

[4] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman, "Derecho: Fast state machine replication for cloud services," *ACM Trans. Comput. Syst.*, vol. 36, no. 2, Apr. 2019. [Online]. Available: https://doi.org/10.1145/3302258

[5] M. Poke and T. Hoefler, "DARE: High-performance state machine replication on RDMA networks," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 107–118. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749267

[6] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "APUS: Fast and scalable Paxos on RDMA," in *Proceedings of the Eighth ACM Symposium on Cloud Computing*, ser. SoCC '17. Santa Clara, CA, USA: ACM, Sept. 2017. [Online]. Available: http://www.cs.hku.hk/research/techreps/document/TR-2017-03.pdf

[7] V. Gavrielatos, A. Katsarakis, and V. Nagarajan, "Odyssey: The impact of modern hardware on strongly-consistent replication protocols," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 245–260. [Online]. Available: https://doi.org/10.1145/3447786.3456240

[8] "Data plane development kit." [Online]. Available: https://www.dpdk.org

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 29–44. [Online]. Available: https://doi.org/10.1145/1629575.1629579

[10] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi\'c

[11] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 437–450. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[12] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1813654.1813656

[13] P. Jalili Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring paxos: A high-throughput atomic broadcast protocol," in *2010 IEEE-IFIP International Conference on Dependable Systems & Networks (DSN)*, June 2010.

[14] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 295–306. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626299

[15] ——, "Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 185–201. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[16] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015. [Online]. Available: https://doi.org/10.1145/2812806

[17] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "Ix: A protected dataplane operating system for high throughput and low latency," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, pp. 49–65.

[18] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "Darpc: Data center rpc," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–13. [Online]. Available: https://doi.org/10.1145/2670979.2670994

[19] A. Sriraman and T. F. Wenisch, "μTune: Auto-Tuned Threading for OLDI Microservices," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, 2018.

[20] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Commun. ACM*, vol. 39, no. 4, p. 54–63, Apr. 1996. [Online]. Available: https://doi.org/10.1145/227210.227226

[21] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Commun. ACM*, vol. 39, no. 4, p. 64–70, Apr. 1996. [Online]. Available: https://doi.org/10.1145/227210.227227