RDMA-ACCELERATED STATE MACHINE FOR CLOUD SERVICES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Sagar Jha

December 2022

RDMA-ACCELERATED STATE MACHINE FOR CLOUD SERVICES

Sagar Jha, Ph.D.

Cornell University 2022

With the advent of RDMA, networks that provide upward of 100Gbps data-transfer speeds and sub-microsecond latencies have become widely available. These high speeds and low delays are a disruptive departure from prior capabilities, and force a redesign of the systems and applications running above them. In this dissertation, we tackle a core question: optimal support for systems that employ data or state machine replication for strong consistency. Such systems are common and vital in the cloud, and state machine replication is a building block for stronger higher level guarantees such as the ones commonly sought in database applications. Advances thus have the potential for broad impact.

This dissertation focuses on the design and implementation of Derecho, a user-space library written in C++ that offers fault-tolerant state machine replication to application subgroups. We motivate Derecho's unusual design, and show that the system's in-memory replication performs close to the network capacity for a variety of application scenarios and cluster settings. We demonstrate that the system is reliable, both through proofs and experiments: replicas maintain the same state through crash failures and network partitions, and the system will only restart correctly, after first repairing any damaged durable state on persistent storage.

Derecho reexpresses Paxos protocols in a novel manner that allows replicas to agree on state updates asynchronously, resulting in continuous, non-blocking data paths. Derecho's in-memory replication outperforms traditional Paxos systems on TCP by a factor of 80X

and a modern Paxos system on RDMA by a factor of 25X.

Theoretical study reveals that Derecho's protocols are optimal under a model put forward by the theory community more than a decade ago, but never previously viewed as a practical goal that might be of value in deployed systems. Although the focus of the author has been quite practical throughout his work (a focus reflected in the choices of material included in this dissertation), this optimality confirms our intuition about the style of knowledge-based programming best matched to the shared state table (SST) abstraction proposed by this author, around which Derecho was built.

Specifically, Derecho expresses atomic multicast and Paxos protocols using a form of coding best understood as epistemic (temporal) logic programming, which in turn is expressed over a monotonic table encoding system state that is continuously extended in a lock-free, asynchronous, uncoordinated manner. For a variety of actions important in atomic multicast or Paxos, our protocols are designed to deduce the point at which those actions can be safely taken. We think of the shared system state as a monotonic knowledge representation in which new information extends but never invalidates past information.

Efficiency, for a protocol of this kind, centers on making such deductions as soon as it is safe to do so, and then ensuring that the protocols themselves impose what might be termed *knowledge-minimal* requirements: they should await the weakest condition adequate for safe progress. The SST design was carefully optimized to run protocols expressed this way as efficiently as possible, hence effort invested to tune and optimize protocols expressed over the SST leads to the same place that an effort to derive an optimal formulation would have arrived at. This fortuitous insight is simply that the shift in style compelled by RDMA, because of its specific latency and throughput properties, happens to favor the network model within which optimality becomes the most natural coding style!

Our goals in creating Derecho were primarily practical, and it is in some ways surprising to realize that simply by setting out to build a fast data replication tool for extremely fast network hardware, we arrived at a design point in which the most optimal protocol designs from a performance perspective also would turn out to be optimal in a more formal sense. We had not anticipated this when we started the project.

It is perhaps surprising that this alignment was not previously noticed by the theory community: the state machine replication as a model can be traced back to papers written 40 years ago, and there have been hundreds (if not thousands) of research papers and systems implementing the model. However, the same properties that make RDMA so disruptive for systems builders also are disruptive relative to classical ways of modeling and abstracting networks: RDMA forces us to think about shared memory models rather than purely in terms of message passing. Surely this explains why even the most efficient of the previous generation of Paxos solutions turns out to be one or more orders of magnitude slower than our Derecho protocols.

What we encountered in our work should be of relevance in many other kinds of systems, protocols and distributed algorithms. We hope that future researchers will pick up the thread and carry it even further: what we achieved for atomic multicast and state machine replication would surely be possible in many other settings.

BIOGRAPHICAL SKETCH

Sagar Jha hails from India. He began his PhD studies in 2015 after graduating with a bachelor's degree in computer science from the Indian Institute of Technology (IIT) Bombay, India.

He is interested in the study of efficiency and performance in complex distributed systems, an interest that is reflected in the focus of his PhD.

ACKNOWLEDGEMENTS

I am profoundly grateful to the many professors, students, and researchers who I interacted with during the course of my PhD. In particular, I would like to thank my advisor, Prof. Ken Birman and my collaborators, Weijia Song, Edward Tremel, Mae Milano, and Lorenzo Rosa for providing the help and support I needed to succeed in the endeavor. I have also had the opportunity to mentor and advise quite a few undergraduates (too many to list here), and found this experience highly satisfying and personally valuable. Learning to better communicate subtle ideas to others has been an unexpected but gratifying side-effect of my time at Cornell.

My PhD and the research work that forms part of this dissertation in general was supported through grants from Defense Advanced Research Project Agency (DARPA), National Science Foundation (NSF), Air Force Research Laboratory (AFRL), Mellanox (later acquired by NVIDIA), Huawei Innovation Systems Lab, and Microsoft.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Idea of zero-copy

Network speeds have risen dramatically over the past 10 years, with data-center and rack-scale networks currently achieving 200Gb/s, and 400Gb/s already available on Infiniband (optical Ethernet RoCE deployments tend to lag IB deployments by a few years). These rates are not expected to plateau until well beyond 1Tb/s, a speed we may reach surprisingly soon. Traditional distributed systems developed in the '90s and the 2000s were developed for older, much slower networks. The underlying designs reflected inefficiencies that do not create bottlenecks on slow networks, yet significantly impair performance when those same solutions are run on more modern networks. Thus there is growing pressure to develop systems whose performance matches the modern network speed as well as scales with the ever-increasing numbers.

One major reason why traditional systems are slow is because asymptotic data transfer rates are often an order of magnitude or more lower than the available "raw" network speed. The culprit proves to be the stack of software layers that support TCP, the de facto reliable, data-transfer protocol used by application processes to exchanges messages with each other. TCP is a universal solution at all scales today: wide-area Internet, local area network, data-center systems and WiFi. This standardization is unquestionably valuable, and it is nothing short of astonishing that TCP can work so well in so many settings. Yet TCP embodies assumptions that are problematic when trying to exploit a network that can run at speeds above about 25Gb/s.

To see why TCP is inefficient, let us understand how its socket API works on Linux (other platforms are very similar). The two application processes at the end of the connection each open a socket with the Linux kernel. A socket is nothing but a kernel data structure indexed by a small integer: what Linux refers to as a *file-descriptor*. Once the connection is established (this involves a “handshake”, the details of which are not relevant to us here), either side can send data to the other. To initiate a new send, the application process first must place the data into a buffer. It then provides this buffer to Linux’s *write* function. Write will pause until the send is underway, and then returns to the caller, which can immediately reuse the buffer memory for other purposes.

Unfortunately (from a performance perspective), the write call does not immediately send the bytes over the network, but may need to delay them for a short period of time. Accordingly, write simply copies the data from the buffer into a form of kernel buffer associated with the socket file descriptor. The data can then be sent asynchronously as network packets sometime later, when the TCP connection is ready. Notice that this feature compels copying: the buffer itself could be in user space, rather than the kernel, but the data must be buffered somewhere. Unfortunately, memory copy operations turn out to be the most dominant limiting factor when it comes to performance. Unlike network speeds, local memory copy speeds have stagnated at around 50Gb/s, and might be even lower if a page fault occurs (if the memory buffer holding application data is paged out). The fundamental bottleneck reflects NUMA hardware architecture considerations.

One can see many inefficiencies here. Some are historical: Linux only added a way to "pin" memory pages and prevent them from being paged out fairly recently. Linux also has newer, application-mode TCP implementations, notably *data-center TCP*. But not all systems support data-center TCP as it imposes slightly different semantics on the application. A new standard called the data plane developer's kit (DPDK) offers application-address space TCP, but is not fully compatible with container virtualization – a nearly universal property of modern data centers. Thus, while many applications are starting to explore the options for pinning message memory regions, and developers are certainly eager to use DCTCP or DPDK, there are so many legacy solutions and virtualization barriers that standard socket-based applications often gain no benefits even if those options are present. Moreover, copying occurs even with DCTCP and DPDK.

Another issue is that while fast networks gain speed by running data over wider and wider buses with more and more "wires" carrying bits concurrently, the link from a modern CPU to its memory bus is much harder to expand: such a link will typically be wide enough for one cache line, but the cache line size is a system-wide hardware constant and doubling it would have broad ramifications. As a result, even as the speed with which we can send data between nodes continues to double every two or three years, CPUs have locked into cache line sizes of 32 or 64 bytes. Indeed, some researchers suggest that with very wide cache lines, CPUs would begin to encounter circuit density, power, and heat dissipation concerns.

The process on the destination side of the socket is similar. As network data reaches the target node, it first will be written to kernel-managed memory. Only when the application calls Linux's *read* function, can data be copied from the kernel memory to the supplied application memory location – indeed, at the instant data arrives, the application may not yet have posted a memory buffer into which it will receive it. Thus, there are two local memory copies in the critical path of data transfer. And while the data traveling through the network is sent at impressively high speeds, the copies at the two ends of the transfer stand out as bottlenecks.

There is another implied drawback of TCP. Both read and write involve passing control to the kernel, or to the threads associated with DCTCP or DPDK. TCP also tends to be interrupt driven, in all situations. Both aspects entail context switches, which use precious CPU cycles. After an initial burst of NUMA parallelism a decade or so ago, the number of CPU cores has scaled far less quickly and now has largely stalled for commodity servers. There has been a growing trend in the recent research literature to offload computation to peripheral devices, such as GPUs, NICs, FPGAs, etc., to avoid burdening the CPU and process information at line rates.

Network hardware engineers confronted by this challenge introduced what has come to be known as a zero-copy methodology – performing operations such as moving data from one location to another while bypassing the kernel and avoiding local memory copies. In an ideal protocol, any single machine would have a single copy of any given data item, initialized just once, and sent just once to any peer that needs a copy (ignoring NUMA memory performance aspects, which may motivate certain applications to create extra copies of data to reduce average access delays for frequently-accessed data, as well as GPU or TPU considerations, which can require extra copying simply to move host data into accelerator memory). Often, zero-copy solutions are also designed to be totally lock-free and asynchronous; when this is not possible, we still strive to minimize the use of locks and to avoid any form of global or distributed coordination and locking.

RDMA is the current industry leader among hardware solutions that embody the zero-copy philosophy. It exists in two forms: one for the Infiniband networking standard (popular on HPC clusters), and a second in which RDMA shares an optical Ethernet connection with TCP (RoCE). The application perspective is identical irrespective of which case the hardware implements. Nothing in our work is wedded to any particular hardware choice as our protocols are quite general: we tested on both Infiniband and RoCE. We also experimented with them on many other networking transports, including standard TCP and DPDK (an industry standard for user-mode TCP hosted in the application's address space). Our work achieves exceptional performance across the board, even yielding substantial benefits when our protocols are compared with highly optimized ones on TCP networks – suggesting that today's data center TCP, running on the fastest data center hardware, is quite different from the TCP previous generations of systems designers optimized against.

1.2 RDMA concept

RDMA was created as a way to offload packet processing from the CPU to the advanced NIC (network interface card) hardware. RDMA-enabled NICs additionally can write incoming network data to application memory through a DMA write and send data stored in application memory over the network with a DMA read, therefore avoiding any memory copy. The technology offers many features; we will focus on the TCP-like "reliable connection" model here. We are not dismissing other options, such as the unreliable datagram configuration of RDMA, but full treatment would require a different approach, and protocols running on an unreliable RDMA datagram lead to very different design choices than the ones we will ultimately present here.

When used in the reliable mode of interest to us, RDMA has semantics similar to TCP: data is delivered in order, without loss or duplication or corruption, and flow control is automatic. The RDMA connection manages itself unless there is a hardware failure. With reliable RDMA connections, application protocols never need to check for data loss, duplication or out-of-order arrival. On the downside, RDMA lacks TCP's deep integration with switch and router congestion reporting mechanisms, which involve IP packet loss that will automatically cause TCP to slow down and resend any missing data. RDMA does have its own form of congestion control, but centers on a credit scheme that isn't coordinated relative to TCP congestion and that cannot tolerate packet loss.

To deploy RDMA, the data center operator will need to switch to versions of NICs, TOR switches and routers that never suffer from congestion when data is in transit. This property can be achieved in various ways: Infiniband is closest to a pure RDMA model, and implements a credit-based permission to send messages between two end-hosts. RoCE approximates this behavior using end-to-end congestion management mechanisms such as Google's TIMELY protocol or the Microsoft/NVIDIA-Mellanox DCQCN one, combined with a hardware congestion-notification mechanism called a priority pause frame; the PPF is used by an overloaded downstream device to request that an upstream device immediately pause transmission for a few microseconds.

Given a suitable control framework (for legacy reasons, called "IB Verbs"), both RDMA implementations enable zero-copy end-to-end solutions, and the performance of RDMA over RoCE has closely matched its performance on Infiniband in our settings. However, management of RoCE networks remains more of an art than a mature engineering discipline, and data center operators have been cautious about offering unlimited customer access to RDMA other than on small Infiniband clusters that are double-wired, with an RDMA network side by side with a standard Ethernet connection. RDMA is common in their own platform solutions, but difficult for cloud customers to access. Thus, RDMA is unquestionably emerging into widespread availability, but it hasn't been a quick or easy undertaking.

Our work uses RDMA through a very lightweight software shim called libfabric, created originally by the MPI community (MPI is a widely popular high performance computing framework). With libfabric, in C++ 17, with full optimization enabled, performance of RDMA is essentially identical to direct use of verbs, but we gain the option of reconfiguring our solutions to run on a TCP emulation of RDMA, on DPDK (still in progress but apparently coming soon), Intel OMNI fabric, and other connection technologies. This dissertation focuses on the RDMA case, but everything we present would work well on all of those options, and we actually do evaluate our work on TCP in Chapter 3.

RDMA supports multiple modes of data transfer. We cover the important ones in the next section.

1.3 RDMA unicast primitives

As briefly noted in the previous section, RDMA offers reliable as well as unreliable modes of data transfer. In the unreliable modes, there is no acknowledgment of message delivery and packets may be lost. In our replication-based systems, we opted for the reliable connection-based modes as this TCP-like guarantee simplifies reasoning about correctness in terms of how we built up end-to-end reliability for the system as a whole. Specifically, by avoiding unreliable modes altogether, we do not need to handle cases where network data is lost under heavy load. However, it is worthy of note that for some purposes, unreliable datagrams can actually be preferable. For example, a CMU research effort called FaSST [58, 60] created a form of distributed database based on sharded key-value APIs in which every node potentially talks to every other node in an all-to-all manner. The authors showed that with reliable RDMA connections, FaSST scales poorly. In contrast, with unreliable datagrams and a simple retransmission scheme, FaSST can scale to (at least) hundreds of nodes. We suspect that this was a consequence of the way FaSST is implemented: the research reported here was evaluated with similar traffic patterns but did not exhibit the kind of scalability issues the CMU team experienced even at large scales. Accordingly, we focus here on asynchronous and synchronous RDMA primitives on reliable connected channels.

1.3.1 One-sided RDMA

In the asynchronous mode, RDMA messages are sent or received without software involvement on the remote host. It comes in two variants: one-sided reads and one-sided writes.

For one-sided RDMA reads, after an exchange of access rights, the initiator node is permitted to read data from a remote node's memory directly into its own memory, without any software involvement from the remote system. One-sided RDMA writes do the opposite: the initiator node writes data directly to a remote node's memory from its own memory. Obviously, quite a bit of information is required for these to work, and in particular, the applications running on the two nodes need to be aware of the relevant memory addresses required for data transfer. This happens in the connection phase: When forming a connection (binding), the applications exchange maps of one-other's virtual memory addresses. Later, they reference these maps when exchanging data. This means that the application needs an out-of-band connection, typically a TCP one, to exchange this information.

One-sided RDMA operations are asynchronous in the sense that the remote node is oblivious to the data transfer. The data is directly read from or written to the remote application memory; no interrupt occurs on the remote node, and no notification is available for the remote system to know that the event occurred. To make this work, the RDMA NICs need to be able to do a DMA read or DMA write directly to application memory. This requires the NIC to be able to translate application virtual memory addresses to host physical memory addresses. RDMA-enabled NICs store a page table for this purpose. The application first needs to register the relevant application memory with its NIC which then pins those pages in memory so that the virtual to physical address mapping does not change. This also ensures that the application memory is never swapped out. All the needed setup occurs during the connection establishment phase, minimizing work to be done at run-time by the application or NIC.

Operations such as memory pinning and exchanging addresses can be costly, and it is important that they happen outside the critical path of data transfer. This makes the pattern in which one-sided RDMA operations are used relatively inflexible: all the memory for RDMA is preallocated, pinned into place, and then held in place while information is exchanged over long-standing connections. This is precisely how we use RDMA as well; 1. our state management system is built upon one-sided RDMA writes that updates state asynchronously in a lock-free manner over the entire failure-free run of the system. It is designed such that the state required for the system to run can be represented in a fixed memory format that can be determined prior to the start of the run, and, 2. our multicast system specialized for small messages uses one-sided RDMA writes over memory that is modeled as a ring buffer consisting of a fixed number of message slots. While these two steps ensure that the memory footprint of the application for small message multicast remains constant (important for making one-sided operations work as just detailed), a backlog could develop if messages in the ring buffer are not delivered in a timely fashion. Chapter 4 discusses how our Spindle optimizations deal with these issues.

Given that there are both read and write variants, it is reasonable to ask how they compare in different scenarios. We found that in our state management subsystem, where nodes take actions that update local copies of their states, then share this data using one-sided RDMA so that peers can observe it, one-sided writes significantly outperformed reads. The reason turned out to be a more fundamental difference between the pull and the push model. When data is written asynchronously, it becomes discoverable by the remote application immediately. While if the data is read instead, it takes a full round-trip for the updates to be available to the initiating node. Given that our main use cases for one-sided RDMA involve small message sizes for the state updates that add negligible latency to the minimum signal propagation time between the nodes, one-sided RDMA writes can drive system progression in just half the round-trip time. Thus, we exclusively used one-sided RDMA writes in situations that exchange smaller objects (up to a few KB), which arise often in our setting.

Finally, if the messages appear asynchronously with RDMA writes for the performance reasons we outlined, it is important to discuss how they should be discovered by the remote application. To scale performance critical applications that can make use of the ultra-low RDMA latencies, the time to detect new messages must not introduce a significant delay after the associated data becomes visible. For this reason, applications dedicate what is called a polling thread. Such a thread runs only when the protocols are active and will become idle if there is no work to be done. But when active, the polling thread continuously examines memory registered with one-sided RDMA connections, checking memory locations (variables) that are used by the sender side to signal events to the receiver side. This ensures that new data can be discovered in the order of microseconds, comparable to the network latency. In contrast, a TCP connection that involves passing control to the kernel for read and write would certainly have millisecond latencies, and even a user-mode DCTCP solution with the DPDK API would incur at least a millisecond of delay. Again, it is reasonable to ask whether we are really utilizing the CPU effectively with this scheme, as one of the selling points of RDMA is CPU-bypass. The answer to this is twofold: By passing control to the application in how it wants to poll its memory for new messages, the performance of the system is in application's hands. For high-performance applications where there is a lot going on, it makes sense to dedicate a single thread that ensures low response-latency. RDMA does offer the option of getting interrupts upon data transfer, although that only makes sense if performance needs to be sacrificed in-favor of low CPU usage. Second, if the system fluctuates through high loads and low loads, the CPU usage can be reduced when not a lot is going on. In our system, the polling thread goes to sleep for a millisecond if no new events have occurred in the last millisecond. After it wakes up, it checks again and goes to sleep or starts functioning as may be the case.

1.3.2 Two-sided RDMA

Two-sided RDMA is synchronous – both sides need to be aware of the data exchange, exactly as with TCP sockets (which, as noted earlier, are actually used to set up the two-sided RDMA connection). Given a two-sided connection, the receiver needs to post a receive buffer, meaning it needs to designate a memory location with the NIC for the upcoming transfer, before the sender sends the message. This enables the NIC to directly DMA the incoming message to the receiver's memory, achieving zero-copy. While a setup still is required, the sender does not need to know what memory addresses the receiver will be using, providing the receiver the flexibility to dynamically decide where to receive. Any memory used to send or receive still needs to be registered with the NIC and pinned so that the DMA reads and writes can happen seamlessly.

Our multicast library makes use of two-sided RDMA operations for sending large messages for several reasons. First, the memory pool of receive buffers could all be allocated non-contiguously which simplifies memory handling for large messages, particularly if the large message may have variable size that has to be sent as a kind of protocol advisory data side by side with the first chunk of bytes (two-sided RDMA supports a feature that is ideal for this purpose). Thus, a receiver can post a fixed size first receive buffer, the sender can send into it, and then the receiver can allocate and post a big buffer for the remainder of the data (moving the first buffer into the first pages of the big buffer, a task that can be done using copying because the chunk size is typically not large). In contrast, a one-sided RDMA based system for multicasting small messages has a single contiguous memory region organized as a round-robin buffer consisting of small message slots, which thus must have a fixed maximum size. For large messages, the small synchronization delay, in order to ensure that receive buffers are posted before sends, is negligible compared to the overall latency of sending the messages themselves.

As will be seen, we often mix two-sided and one-sided mechanisms. For example, all synchronization in our setting happens over a one-sided connection, even if the data is being transferred using two-sided RDMA.

1.4 Future developments

At the time of this writing, techniques for using RDMA, as well as more evolved high-speed data-center communication options to support remote memory, have grown to become a significant research area, with increasing numbers of products and proposed models. Many of these proposals weaken assumptions that our work makes, hence the results in this thesis might need to be generalized in the future to fully exploit emerging memory technologies. We limit ourselves to just a few words about the large trends and the factors that motivate this work, but then leave the topic as beyond our scope.

1.4.1 Disaggregated data centers

Perhaps the largest of the current trends is the push to move away from data center designs in which each computer is identically configured, and to instead envision a style of data center that can "assemble" a special-purpose system on demand, configured ideally for the purpose. Just as we currently can order computers with varying amounts of memory, various storage features, accelerators, etc, a *disaggregated* data center would allow an application to request a bespoke configuration that would be provided on demand. The model leads towards an approach in which the data center has racks of storage servers, racks of compute servers, racks of memory servers, etc. There are huge challenges here: we already find it difficult to fully leverage NUMA compute nodes because today's programming styles often compile to code that might incur unexpected and costly access to remote memory even within the NUMA machine itself. With a disaggregated model in which RDMA or some successor communication framework is used to connect the various components, latencies would be even higher. Nonetheless, the topic is a lively one and for some purposes, seems to be an excellent match that yields significantly more cost-effective platforms.

1.4.2 Far memory

A number of recent papers have explored what are being called "far memory" models. Typically, these are approaches in which the application is given an extremely large virtual memory, which is then implemented by paging from memory servers over an RDMA interconnect. Such designs lead to many questions about performance, use of the paging hardware, and integration of this model with the O/S, and addressing them has already revealed a great many technical challenges, which are being tackled by a lively research community. One could imagine extending our work to allow a multicast into far memory, although our work does not pursue that option and it would not be trivial.

1.4.3 One-sided RDMA with novel memory models

In very recent work, the memory community has begun to deploy novel forms of memory that implement so-called *weak atomic* memory semantics. The proper semantics for RDMA to enforce when running over such memories is a matter of debate, and beyond our scope here. All Intel processors implement what are called *total store order* semantics, meaning that the computer memory behaves as if there was a single system-wide order of memory writes at the level of cache line chunks of memory (recall that a cache-line would typically be 64 bytes wide). Moreover, Intel's TSO respects the order in which any single writer issued writes: if some thread does write A and then write B, any thread seeing write B will see write A. Our work adopts this model, which is expected to be universally supported even if other models also emerge. One-sided RDMA writes behave as writes issued by the NIC, and respect the order that the sender used. Thus we have this A-then-B guarantee, and can assume it in all of our protocols and proofs, at the cost of obtaining solutions that might not leverage the full power of next-generation systems in which weak atomics might afford some performance gains.

1.4.4 Programmable everything

Modern NICs, switches and routers are often programmable, offering functionality whereby a user familiar with the relevant domain-specific language can code logic that can then be pushed into the device and will execute at line rates directly in the fabric. Meanwhile, there is a trend to support massive FPGA arrays (or even arrays of GPUs or TPUs) in which programs running on the accelerator would incorporate parallel computing functionality, with message passing supported by RDMA directly between accelerator memories, or configured as a bump in the wire. We have had some inquiries about the possibility of using Derecho (our work) in such settings; the potential is intriguing, but the effort that would be required to port the system would not be small.

1.4.5 Embrace of unreliable RDMA datagrams

We noted that, led by the experiences in the FaSST project at CMU, there is growing interest in using unreliable datagram models with RDMA. We touched on this above, noting that the FaSST system has an architecture in which at scale, every system node (every machine) has roughly an i.i.d. likelihood of needing to interact with any random server. In effect, FaSST has a pattern of all-client to all-servers, leading to huge fanouts on both sides. This puts stress on the RDMA connection model supported by the hardware, leading to a situation in which a connectionless model becomes preferable even at the cost of lost reliability. FaSST is not the only system exploring this direction: some RDMA state machine replication solutions (Paxos) don't actually require reliability, and developers frequently reason that if there is a non-zero cost to establishing an RDMA connection (and there is!), why not just use a connectionless unreliable datagram approach? This work is of interest to us, but seems to result in solutions that are considerably slower than the approach taken in Derecho, primarily because the lack of assured reliability forces the protocol to work hard to overcome datagram loss, for example, by adopting quorum methods that might leave some receivers with gaps in their input, and forcing any queries to merge data from multiple receivers. In Derecho, we do not see any evidence that reliable RDMA is introducing concerning costs, and hence for our purposes, we did not explore unreliable RDMA options.

Similarly, FaSST and systems of a like nature encounter memory-management puzzles that seem not to arise in Derecho. Because FaSST supports a massive form of in-memory database (a so-called in-memory key-value model), the amount of memory that RDMA might try to access becomes immense. Researchers who created FaRM ([38], the first and most famous RDMA key-value database) noted that the NIC DMA engine can slow down in such cases because the size of the data structure used to track memory layouts becomes so large; they favored a "huge page" remedy, and FaSST takes this even further, employing not just huge pages but also a form of request inlining that only can be employed if the keys and objects in the store are fairly small (in total, the size of a cache-line, namely 64 bytes). Derecho can support massive key-value storage models too, but is a much more general technology with many other possible uses: for publish-subscribe in a data distribution system (a DDS), for message queuing support (there are many MQ products and models), for state replication and sharing within some form of distributed application or application-defined replicated object. As such, we did not feel that huge pages and inlined key-value data were of particular importance, and decided against optimizing the system in these ways.

Having completed the Spindle optimization methodology discussed in Chapter 4, we would argue that objectively, Derecho is not losing performance in this way, perhaps because the actual runtime distribution of requests in Derecho puts the majority of load on a small subset of network links (those between members of the same shard, when data in that shard is being updated frequently). This pattern of load stresses a subset of the possible all-to-all RDMA network connections, and matches the internal design choices made by hardware designers of modern RDMA NICs. In contrast, the key-value stores mentioned above tend to load all RDMA connections in a more uniform way, because they decide which peer to interact with using pseudo-random hashing.

1.5 System challenges when developing for RDMA

Even with our decision to focus on data replication and higher level primitives such as atomic multicast or replication of abstract data types, the mapping from the user model to RDMA poses a spectrum of challenges. Simply mapping standard protocols to RDMA will often yield substantially less than optimal performance. Totally general work – solutions that can solve any problem for any user – is never a realistic place to start when facing such an open-ended question. Accordingly, our work will be shaped by the broad goal of supporting replication, but also by the desire to optimally leverage the two primary RDMA technologies discussed above: reliable RDMA used with one-sided writes of smaller (a few KBs or less) objects, and reliable RDMA employed in a two-sided manner (potentially, with very large message sizes: a photo could easily be 1MB or more in size, and even a short video might be 100's of MB).

1.5.1 Redesigning application memory structure

As discussed in the previous section, application memory needs to be allocated and registered with the NIC prior to the critical path of data transfer. This means that the memory structure of the application needs to be re-imagined. Systems from two or more decades ago had seemingly unlimited message memory, but an application running at RDMA speeds is forced to work within this RDMA memory pool. Older protocols implicitly leveraged the buffering done by the kernel after a TCP message write was done, but before the data was really sent. RDMA protocols can't make use of this trick (although they do avoid the extra copying it entailed).

With one-sided RDMA, the memory layout shared by sender and receiver needs to be static for relatively large periods of time (to avoid memory reregistration and recreating the connections). For our distributed state management system, this meant being able to model the control state for replication in a fixed memory format. We came up with fixed size state variables that could be updated within the bounds of their sizes. Moreover, a ring-buffer based solution is also very popular for supporting certain system functionalities. We use it to support multicast for small messages. It has also been used in other systems such as key-value stores to send RPC stubs containing get or put requests. Naturally, those systems limit the size of both the keys and the values to ensure that memory bounds for any single request are not exceeded.

The second challenge in this category pertains to the asynchronous nature of communication. A single memory location could potentially be updated multiple times by a remote node prior to the update being discovered. Additionally, the polling thread could be reading a memory location simultaneously while a DMA write to it is underway (like any shared memory system, such a situation does provide cache-line atomicity and sequencing, but not stronger properties). This led us to design protocols that share state asynchronously, but in which state updates for new events will not invalidate or hide information of previous ones. For example, a counter representing the number of messages delivered can go up from 3 to 4 to 5, which just means that all the 5 messages have been delivered: the receiver thread may miss the value 4, but can infer that the sender must have incremented the counter to 3, then 4, then 5. The hardware-layer cache-line atomicity guarantee, together with Intel's total store order property, are enough to let us design solutions in which race conditions will not arise even if a reader is querying some state variable exactly when the remote system is updating it. This avoids the need for any form of locking (which is good, because locking is clumsy and extremely costly with one-sided RDMA and not available with two-sided RDMA).

For the two-sided RDMA operations, we built needed support to ensure that receive buffers are posted before a message is sent. This just involved building another one-sided acknowledgment-based layer on top of the multicast subsystem.

1.5.2 Efficient polling infrastructure

The updates that happen asynchronously through one-sided RDMA writes must be discovered very rapidly to take maximum advantage of high RDMA speeds. If an important update arrives while the polling thread is busy examining a region of memory that is not crucial, it can add significant latency and reduce performance sharply. Our polling system is designed to ensure that multiple objectives of discovering new messages, getting acknowledgments, delivering messages, etc., are balanced appropriately so that entire pipeline of sending a message all the way to delivering it performs smoothly. Our Spindle methodology, Chapter 4 in this dissertation, devotes a full paper to the topic.

1.5.3 Adjusting to the low RDMA latencies: reducing application-induced delays

Traditional systems worked on slow networks. Their design involved a lot of inefficiencies, particularly, delays based on round-trip communication where a thread waits to receive acknowledgment of its send before taking further action. The effect of these is magnified in the fast-paced setting of RDMA.

Our replication system separates the control plane from the data plane. This means that coordination on past messages happens concurrently with sending new messages. This ensures that the system is always busy in doing useful work.

For small messages, we noticed that the control plane operations were relatively more costly, and in particular, that applications with irregular sending rates faced a costly trade-off: an application with nothing to send could remove itself from the list of active senders, but the cost of doing so turns out to be fairly high. Adding itself back to that list later would be similarly costly. This made it appealing to add a feature whereby a process with nothing to send, noticing that others will be waiting for it, can send dummy null messages, eliminating the delay. Yet even this rather simple optimization turns out to require a careful design in order to not interfere with performance, and its own proof of correctness, and extensive testing.

1.5.4 Applying classic system optimizations effectively

Classic system optimizations such as batching need to be carefully designed and evaluated for RDMA since the parameters that affect performance are entirely different. The RDMA setting also leads to some interesting thread synchronization issues that we handled in our work. For example, in the Spindle work described in Chapter 5, we discovered that even holding a lock for a few instructions longer than necessary caused our entire system to slow down. Developers of shared-memory data structures have seen issues such as these on NUMA machines, but it is only with the advent of RDMA that distributed systems researchers have encountered them. They consumed a great deal of effort in all of our research.

1.6 An overview of RDMA-based systems

This section provides pointers to some notable RDMA-based systems that have been developed since RDMA-focused research became topical in the systems community. For a more detailed related work, see the related work sections of the subsequent chapters that provide the proper context of our work.

FaRM [38] builds a distributed memory cluster where remote memory is accessed through RDMA. FaRM was really the first major distributed system to leverage RDMA, and offered a key-value store as a proof of concept, including some basic atomic update guarantees. HERD [58] and Pilaf [72] came soon after; both also adopt a key-value storage model with different design choices intended to leverage RDMA to an even greater degree. FaSST [60] provides distributed, serializable in-memory transactions. MVAPICH [4] is an MPI library specialized to run over Infiniband and other high-end computing interconnects.

1.7 An overview of Paxos protocols

The main system that we developed, Derecho, supports state machine replication using a Paxos-based atomic multicast and durable logging, with a virtual synchrony framework used to manage the system membership. The development of this protocol is not my primary contribution; my advisor, Ken Birman, who has been working on virtual synchrony and related protocols for the past three decades deserves much of the credit.

A very basic definition of reliability in our context implies that any system state or data is not lost during the run of the system. For a distributed system to be reliable amidst possible crash failures (meaning that one or more of the processes on the physical machines stop executing) and network outages, replication is a proven technique. If multiple processes (replicas) have the same data or state, then losing access to a subset of them does not lead to the loss of reliability. State machine replication (SMR), first formalized by Leslie Lamport in [63], starts the replicas with the same initial state and process commands (or requests) in the same order in all the replicas. If the commands from any given state lead to a deterministic next state, then the replicas transition through the same sequence of states, maintaining the replication requirement. Fred Schneider's tutorial is a classic on this topic [86].

The central problem in solving SMR involves the replicas agreeing on which command to execute next. This is a version of the distributed consensus problem, one of the most fundamental problems in distributed systems. To give a flavor of the complexity of the problem, note that the replicas might be receiving requests concurrently from the users (which translate into commands that change the system state and generate a response). Thus, at a very minimum, reaching a consensus involves exchanging these requests as well as the acknowledgements of the receipts. In an asynchronous setting, where messages can take arbitrarily long to be sent and received, distinguishing a node failure from a slow network is theoretically impossible. Formally, the famous FLP theorem [43] states that even with one just one faulty process, guaranteeing that the (correct) processes reach consensus is impossible.

In practice, however, the precise sequence of continuous failures that denies progress is impossible to attain. The Paxos family of protocols are widely employed to solve consensus problems in distributed systems (that theoretically guarantee the correctness of consensus, but not progress). Paxos as a methodology was introduced by Leslie Lamport in 1998 ([64]), although work on these protocols first started about a decade earlier by at least three groups: Ken Birman and his research group at Cornell University (see [16,23,24]), Brian Oki and Barbara Liskov ([75]), and Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer ([39]). We focus our discussion on the first of these, the virtual synchrony protocol, that forms the basis of the Derecho system.

Virtual synchrony [22] was introduced in 1987 by Ken Birman and his research group (notably, in work he conducted with Thomas Joseph). In virtual synchrony, the system moves through epochs: Each epoch is a run of the system with a fixed ordered membership (called the "group"), known to all members. The members constitute what we referred to as replicas earlier. In an epoch, all group members propose updates (the Paxos terminology would be "commands") by sending them to the entire group via a per-member FIFO channel. Each member executes commands in the same order by cycling through the commands in the membership order. In other words, the first command executed is the first message by the first member of the group. It is followed by the first message of the second member, and so on. After one complete cycle, the next command executed is the second message by the first member. In a way, the problem of agreeing on which command to execute next is reduced down to agreeing on the membership of the system. A node failure or a new node join or leave event triggers a reconfiguration (change of epoch) where the old (surviving) nodes reach a consensus on the new membership. By ensuring that the correct processes reach an agreement on which command to stop at before the reconfiguration and that the new members receive the system state via state transfer, we get a solution that implements state machine replication correctly. Inductively, the replicas are started with the same initial state in the first epoch and every node that is part of the next epoch is guaranteed to have the same state as everybody else.

In the space of the Paxos protocols, virtual synchrony is best seen as a variant of vertical Paxos, presented in [66] by Leslie Lamport, Dahlia Malkhi, and Lidong Zhou.

APUS [99] is a leader-based Paxos protocol on RDMA, similar to Raft. DARE [81] implements state machine replication on an RDMA network.

1.8 Organization of the remainder of this dissertation

The rest of this dissertation is organized as chapters that each cover a paper, that our group published and I contributed majorly to, that form a part of the overarching goal of creating a reliable, performant replication system on RDMA.

The chapters contain the paper contents exactly as they were published in the journal/-conference they appeared in, with the exception of some typo fixes. In particular, because the pseudo-code in Chapter 3 was updated to be completely correct, the errata associated with the published version of that protocol are omitted here.

Chapter 2 covers our multicast system, RDMC, for large objects that launched our work on data replication on RDMA. RDMC, alongside SMC (the multicast system for small messages), forms the backbone of the replication system. The chapter describes how the binomial pipeline algorithm, a theoretically optimal data distribution protocol that involves exchanging chunks of data between pairs of nodes in multiple stages is mapped elegantly to a sequence of two-sided RDMA operations. RDMC was published in DSN 2018 in Luxembourg.

Chapter 3 presents Derecho, the replication library itself. Derecho provides state machine replication to multiple application subgroups each implementing part of the application functionality and manages the membership of the application as a whole. This paper is the core of the thesis presented here. It describes SMC (Small-message Multicast), the control plane, SST (Shared State Table) that orchestrates RDMC and SMC to build basic reliability, the group management system that handles crash failures and network partitioning using the SST, and the data persistent system for durable Paxos and many other things that make up the system. The paper also presents a comprehensive evaluation of the many different functionalities Derecho provides, including comparison with existing, competing systems. Derecho was published in the April, 2019 edition of the TOCS journal.

We found that Derecho performed close to network speeds when replicating large data sizes thanks to RDMC, but its performance for small data sizes left a lot to be desired. Hence, we worked on techniques specifically designed to optimize small-message replication. Chapter 4 presents Spindle, a bundle of such techniques. Spindle's optimizations include unified opportunistic batching, Derecho's null send logic, and efficient thread synchronization. Spindle was published in ICDCS 2022 in Bologna.

Chapter 5 of this document concerns itself with how a Derecho system can be booted and repaired after restart from a total failure – a case when a majority of the nodes fail during a run and the entirety of the system has to shut down. Total restart involves comparing and merging durable state spread across the restarting members as well as handling any failures that could happen during the process. The paper describes a correct approach particular to the constraints and assumptions of Derecho and in particular, handles the core issues around restarting multiple application subgroups and remapping the restarting members to satisfy natural and custom-defined node provisioning constraints to the different subgroups.

Finally, the last chapter concludes the work and presents some ideas for further exploration that could be of interest to future researchers.

CHAPTER 2

RDMC: A RELIABLE RDMA MULTICAST FOR LARGE OBJECTS

Notes:

The paper presented in this chapter originally appeared in DSN, 2018 in Luxembourg, Luxembourg, although the work on the RDMC system started much earlier, in 2014. The authors of this paper are Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. At the start of my own research, Jonathan and Ken were just defining the scope of this project. While Jonathan developed and tested most of the RDMC protocol, my contributions include a) porting RDMC to run on the Infiniband verbs API on our Cornell private cluster (the original system ran on the Intel PSM API on the Sierra cluster at Lawrence Livermore National Laboratory), b) undertaking a mathematical analysis of pipeline “slack” that explained the remarkable (and unexpected) robustness of RDMC’s binomial pipeline protocol in the face of software and network delays, and c) integrating RDMC with SST, the state management system of Derecho, to provide fault-tolerance and maintaining it over the years.

Within these results, it is notable that the original RDMC binomial pipeline protocol had been proposed for a systolic array network as a purely theoretical undertaking. The prevailing assumption was that such protocols are highly timing-dependent and that today’s computer clusters would not be able to host them. Thus, the slack analysis we present, which shows that in general a node receiving some block B at time step t will not need to transmit B until time step t+1 or later (leaving time for straggling data to show up before it is actually needed) not only helps to explain RDMC's steady performance under a wide variety of difficult conditions, but also represents a conceptual advance.

RDMC provides performance close to line rates for multicasting large application message sizes. At the time of the work (and still), support for a standardized multicast primitive for RDMA was lacking. All that existed was the RDMA unicast primitives, including both one-sided and two-sided modes, for fast data exchange between a pair of nodes. For a state machine replication system, the data that is committed by the replicas after stabilization needs to be received by everyone in the first place. This is the basic multicast pattern, where one of the replicas that has the data (received through an external client, e.g.) acts as the sender. If the application needs to move large files such as disk images or needs to replicate large-sized videos, the message sizes could be as high as a few hundred MBs. RDMC is the underlying system for large-message multicast in Derecho, the state machine replication library presented in the next chapter.

Abstract

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

2.1 Introduction

Datacenter loads are heavily dominated by data copying delays, often from a source node to two or more destinations. By 2011, distributed file systems like Cosmos (Microsoft), GFS (Google), and HDFS (Hadoop) handled many petabytes of writes per day (hundreds of Gb/s) [40], and the throughput is surely far higher today. Many files are replicated to multiple storage servers [47]. The latency of this process determines overall write performance for end-user applications. At Facebook, Hadoop traces show that for jobs with reduce phases, the transfer of data between successive phases represents 33% of total run time [32]. Google's Borg has a median task startup latency of around 25 seconds (about 80% devoted to package installation) with upwards of 10,000 tasks starting per minute in some cells [97]. In some cases, copying VM images and input files takes substantially more time than computation [88].

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

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

RDMC can also be extended to offer stronger semantics. In work reported elsewhere, we describe Derecho [55]: a new open-source software library layered over RDMC that supports atomic multicast as well as a classic durable Paxos. To gain these properties, Derecho introduces a small delay, during which receivers buffer messages and exchange status information. Delivery occurs when RDMC messages are known to have reached all destinations. No loss of bandwidth is experienced, and the added delay is surprisingly small.

The contributions of the present paper are as follows:

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

2.2 Background on RDMA

RDMA (remote direct memory access) is a zero-copy communication standard. It has been used for many years on Infiniband, but is now also working robustly on standard datacenter Ethernet [73, 104].

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

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

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

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

Today's RDMA NICs guarantee two forms of ordering: (1) requests enqueued on a single send or receive queue will be performed in FIFO order (2) a receive completion occurs only after the incoming transfer is finished. Mellanox's CORE-Direct [71] feature proposes a third form of request ordering: it is possible to enqueue an RDMA send that will wait both until the prior request has completed, as well as for completion of some other RDMA send or receive, possibly even on a different queue pair. In cases where a node Q needs to relay data received from P to another node R, this avoids the software delay at Q to issue the relay operation after the receive is complete. We believe that CORE-Direct is just one of what will eventually be a wide range of new RDMA NIC programmability features.

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

2.3 High level RDMC summary

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

This yields a framework that operates as follows:

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

```
bool create_group(int group_number,
                  vector<int> members,
                  function<memory_region(int size)> incoming_message_callback,
                  function<void(char* data, int size)> message_completion_callback);

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

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

Figure 2.1: RDMC library interface

2.4 System Design

2.4.1 External API

Figure 2.1 shows the RDMC interface, omitting configuration parameters like block size. The `send` and `destroy_group` functions are self-explanatory. The `create_group` function is called concurrently (with identical membership information) by all group members; we use the out-of-band TCP connections mentioned earlier to initiate this step. `create_group` takes two callback functions, which will be used to notify the application of events. The `incoming_message_callback` is triggered by receivers when a new transfer is started, and is also used to obtain a memory region to write the message into. Memory registration is expensive, hence we perform this step during startup, before any communication activity occurs.

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

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

2.4.2 Architectural Details

RDMC runs as a user-space library. Figure 2.2 shows an overview of its architecture.

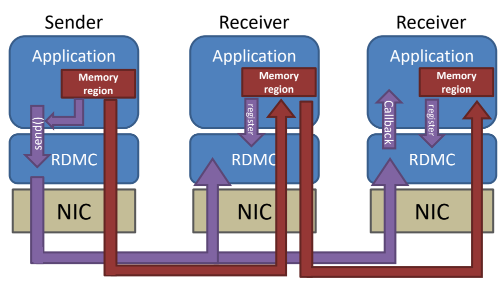


Figure 2.2: RDMC with a sender and 2 receivers.

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

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

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

The sender and each receiver treat the schedule as a series of asynchronous steps. In each step every participant either sits idle or does some combination of sending a block and receiving a block. The most efficient schedules are bidirectional: they maximize the degree to which nodes will send one block while concurrently receiving some other block. Given the asynchronous step number, it is possible to determine precisely which blocks these will be. Accordingly, as each receiver posts memory for the next blocks, it can determine precisely which block will be arriving and select the correct offset into the receive memory region. Similarly, at each step the sender knows which block to send next, and to whom.

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

2.4.3 Protocol

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

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

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

Chain Send. This algorithm implements a bucket-brigade, similar to the chain replication scheme described in [96]. After breaking a message into blocks, each inner receiver in the brigade relays blocks as it receives them. Relayers use their full bidirectional bandwidth, but the further they are down the chain, the longer they sit idle until they get their first block, so worst-case latency is high.

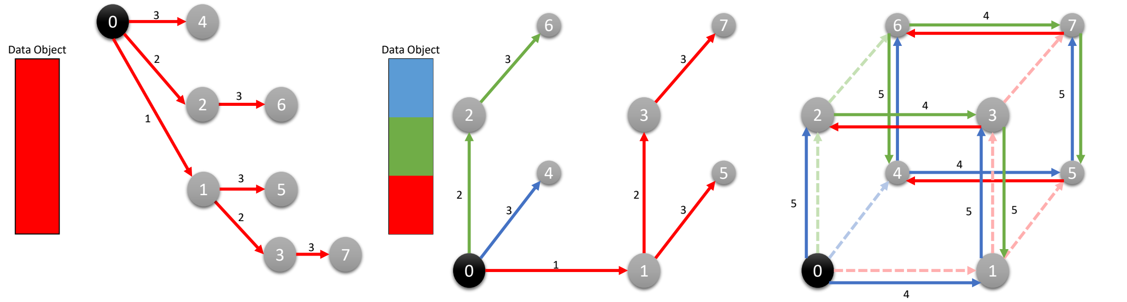


Figure 2.3: (Left) A standard binomial tree multicast, with the entire data object sent in each transfer. (Center) A binomial pipeline multicast, with the data object broken into three blocks, showing the first three steps of the protocol. In this phase, the sender sends a different block in each round, and receivers forward the blocks they have to their neighbors. (Right) The final two steps of the binomial pipeline multicast, with the earlier sends drawn as dotted lines. In this phase, the sender keeps sending the last block, while receivers exchange their highest-numbered block with their neighbors.

Binomial Tree. For large objects, better performance is possible if senders send entire messages, and receivers relay each message once they get it, as seen in Figure 2.3 (left). The labels on the arrows represent the asynchronous time step. Here, sender 0 starts by sending some message to receiver 1. Then in parallel, 0 sends to 2 while 1 sends to 3, and then in the final step 0 sends to 4, 1 sends to 5, 2 sends to 6 and 3 sends to 7. The resulting pattern of sends traces out a binomial tree, hence latency will be better than that for the sequential send, but notice that the inner transfers can't start until the higher level ones finish. For a small transfer, this would be unavoidable, but recall that RDMC aims at cases where transfers will often be very large. Ideally, we would wish to improve link utilization by breaking large transfers into a series of smaller blocks and pipelining the block transfers, while simultaneously minimizing latency by leveraging a binomial tree routing pattern.

Binomial Pipeline. By combining the Chain Send with the Binomial Tree, we can achieve both goals, an observation first made by Ganesan and Seshadri [45]. The algorithm works by creating a virtual hypercube overlay of dimension d, within which d distinct blocks will be concurrently relayed (Figure 2.3, middle, where the blocks are represented by the colors red, green and blue). Each node repeatedly performs one send operation and one receive operation until, on the last step, they all simultaneously receive their last block (if the number of nodes isn't a power of 2, the final receipt spreads over two asynchronous steps). The original work by Ganesan and Seshadri was theoretical, validated with simulations. Further, they assumed that the network is synchronous. We extended their approach to work in a fully asynchronous setting where a node is waiting for exactly one node to send a block. We also decoupled the send and receive steps so that a send step is only pending if the associated block hasn't been received. The resulting algorithm is exceptionally efficient because it reaches its fully-loaded transfer pattern quickly, ensuring that nodes spend as much time as possible simultaneously sending and receiving blocks.

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

2.4.4 Analysis

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

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

We refer to steps l to $l + k - 2$ as "steady" steps.

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

Ganesan and Seshadri provide the following characterization of the algorithm:

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

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

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

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

2.4.5 Robustness of RDMC's Binomial Pipeline

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

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

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

1. A delay ϵ in sending a block leads to a maximum delay of ϵ in the total time to send. If a block send takes about δ time, the total time without delay is $(l + k - 1)\delta$. Assuming $\epsilon = \mathcal{O}(\delta)$, the total time becomes $(l+k-1)\delta+\epsilon$. If the number of blocks is large, $(l + k - 1)\delta >> \epsilon$, and thus the effective bandwidth does not decrease by much.

2. Since a node cycles through its l neighbors for exchanging blocks, a link between two neighbors is traversed on just $1/l$ of the steps. Thus a slow link has a limited impact on performance. For example, if one link has bandwidth T' and other links have bandwidth T, with $T > T'$, rough calculations show the effective bandwidth to be at least a factor of $\frac{lT'}{T+(l-1)T'}$ of the bandwidth when each link is of bandwidth T. If $T' = T/2, n = 64$, this fraction is 85.6%. Contrast this to the chain replication scheme where each link is traversed by each block and the bandwidth is limited by the slowest link (T' in our example).

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

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

2.4.6 Insights from using RDMC

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

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

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

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

Systems seeking stronger guarantees can leverage RDMC too. For example, Derecho augments RDMC with a replicated status table implemented using one-sided RDMA writes [55]. On reception of an RDMC message, Derecho buffers it briefly. Delivery occurs only after every receiver has a copy of the message, which receivers discover by monitoring the status table. A similar form of distributed status tracking is used when a failure disrupts an RDMC group. Here, Derecho uses a leader-based cleanup mechanism (again based on a one-sided RDMA write protocol) to collect state from all surviving nodes, analyze the outcome, and then tell the participants which buffered messages to deliver and which to discard. Through a series of such extensions, Derecho is able to offer the full suite of Paxos guarantees, yet it can still transfer all messages over RDMC.

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

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

2.5 Experiments

2.5.1 Setup

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

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

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

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

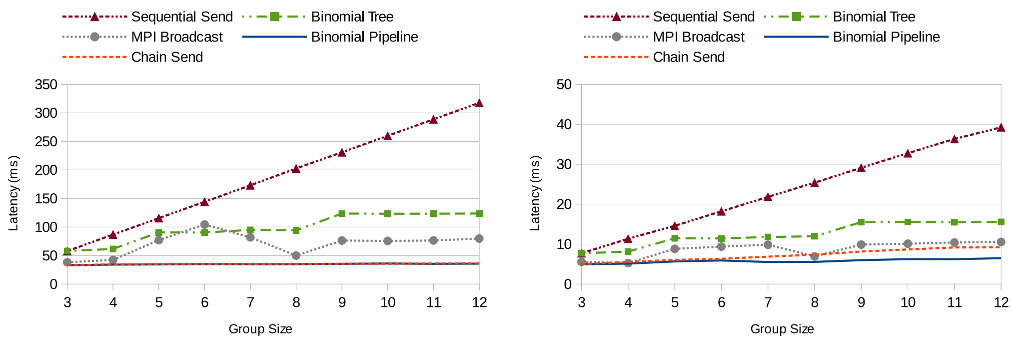
Apt Cluster. The EmuLab Apt cluster contains a total of 192 nodes divided into two classes: 128 nodes have a single Xeon E5-2450 processor with 16 GB of RAM, while 64 nodes have two Xeon E5-2650v2 processors and 64 GB of RAM. All have one FDR Mellanox CX3 NIC which is capable of 56 Gb/s.

Interestingly, Apt has a significantly oversubscribed TOR network that degrades to about 16 Gb/s per link when heavily loaded. This enabled us to look at the behavior of

RDMC under conditions where some network links are much slower than others. Although the situation is seemingly ideal for taking the next step and experimenting on hybrid protocols, this proved to be impractical: Apt is batch-scheduled like Sierra, with no control over node placement, and we were unable to dynamically discover network topology.

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

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



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

(b) 8 MB multicasts.

Figure 2.4: Latency of MPI (MVAPICH) and several RDMC algorithms on Fractus. Group sizes include the sender, so a size of three means one sender and two receivers.

2.5.2 Results

Figure 2.4 compares the relative performance of the different algorithms considered. For comparison, it also shows the throughput of the heavily optimized `MPI_Bcast()` method from MVAPICH, a high-performance computing library that implements the MPI standard on Infiniband networks (we measured this using a separate benchmark suite). As anticipated, both sequential send and binomial tree do poorly as the number of nodes grows. Meanwhile chain send is competitive with binomial pipeline, except for small transfers to large numbers of nodes where binomial pulls ahead. MVAPICH falls in between, taking from 1.03× to 3× as long as binomial pipeline. Throughout the remainder of this paper we primarily focus on binomial pipeline because of its robust performance across a range of settings, however we note that chain send can often be useful due to its simplicity.

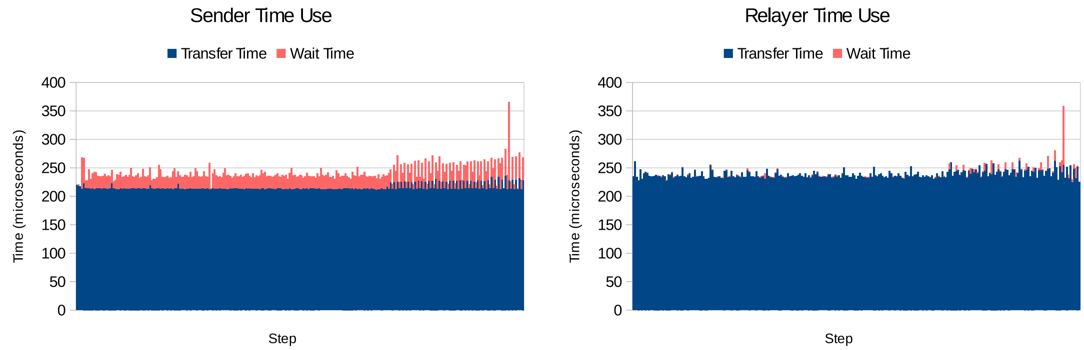


Figure 2.5: Breakdown of transfer time and wait time of two nodes taking part in the 256 MB transfer. The majority of time is spent in hardware (blue), but the sender (left) incurs a higher CPU burden (orange) than the receiver (right). Offloading RDMC fully into the hardware would eliminate this residual load and reduce the risk that a long user-level scheduling delay could impact overall transfer performance.

Microbenchmarks

In Table 2.1 we break down the time for a single 256 MB transfer with 1 MB blocks and a group size of 4 (meaning 1 sender and 3 receivers) conducted on Stampede. All values are in microseconds, and measurements were taken on the node *farthest* from the root. Accordingly, the Remote Setup and Remote Block Transfers reflect the sum of the times

Table 2.1: Time (microseconds) for key steps in a transfer.

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

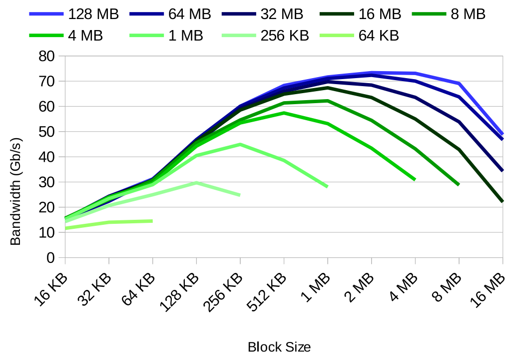


Figure 2.6: Multicast bandwidth (computed as the message size divided by the latency) on Fractus across a range of block sizes for messages between 16 KB and 128 MB, all for groups of size 4.

taken by the root to send and by the first receiver to relay. Roughly 99% of the total time is spent in the Remote Block Transfers or Block Transfers states (in which the network is being fully utilized) meaning that overheads from RDMC account for only around 1% of the time taken by the transfer.

Figure 2.5 examines the same send but shows the time usage for each step of the transfer for both the relayer (whose times are reported in the table) and for the root sender. Towards the end of the message transfer we see an anomalously long wait time on both instrumented nodes. As it turns out, this demonstrates how RDMC can be vulnerable to delays on individual nodes. In this instance, a roughly 100 μs delay on the relayer (likely caused by the OS picking an inopportune time to preempt our process) forced the sender to delay on the following step when it discovered that the target for its next block wasn't ready yet. The CORE-Direct functionality would mitigate this.

In Figure 2.6, we examine the impact of block size on bandwidth for a range of mes-

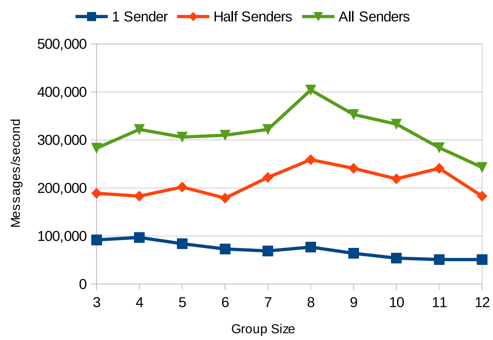


Figure 2.7: 1 byte messages/sec. (Fractus)

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

Finally, Figure 2.7 measures the number of 1 byte messages delivered per second using the binomial pipeline, again on Fractus. Note, however, that the binomial pipeline (and indeed RDMC as a whole) is not really intended as a high-speed event notification solution: were we focused primarily on delivery of very small messages at the highest possible speed and with the lowest possible latency, there are other algorithms we could have explored that would outperform this configuration of RDMC under most conditions. Thus the 1-byte behavior of RDMC is of greater interest as a way to understand overheads

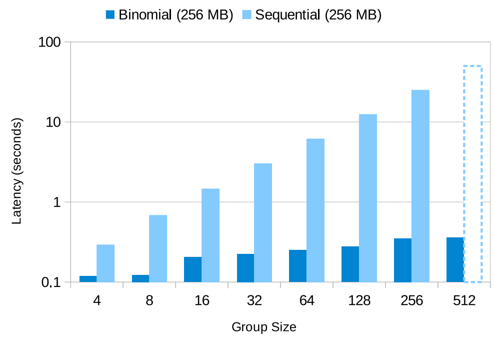


Figure 2.8: Total time for replicating a 256MB object to a large number of nodes on Sierra.

than for its actual performance.

Scalability

Figure 2.8 compares scalability of the binomial pipeline on Sierra with that of sequential send (the trend was clear and Sierra was an expensive system to run on, so we extrapolated the 512-node sequential send data point). While sequential send scales linearly in the number of receivers, binomial pipeline scales sub-linearly, which makes an orders of magnitude difference when creating large numbers of copies of large objects. This graph leads to a surprising insight: with RDMC, *replication can be almost free:* whether making 127, 255 or 511 copies, the total time required is almost the same.

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

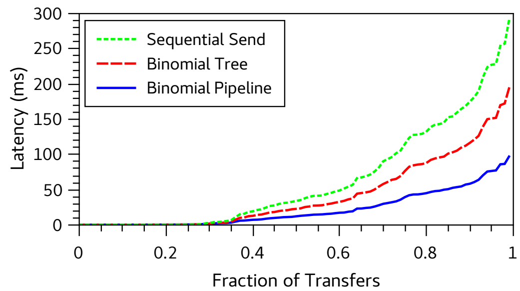
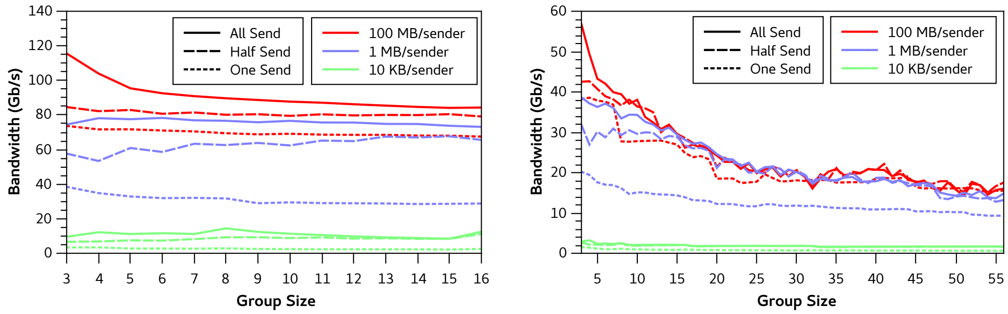


Figure 2.9: Distribution of latencies when simulating the Cosmos storage system replication layer.

leave the whole system idle waiting for one node to finish. In contrast, the linear degradation of sequential send is also associated with high skew. This highlights the very poor performance of the technology used in most of today's cloud computing frameworks: not only is copy-by-copy replication slow, but it also disrupts computations that need to wait for the transfers to all finish, or that should run in loosely synchronized stages.

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

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



(a) Fractus (b) Apt Cluster

Figure 2.10: Aggregate bandwidth of concurrent multicasts on Fractus and the Apt cluster for cases in which we varied the percentage of active senders in each node-group (in a group with k senders, we used k overlapped RDMC groups with identical membership). The Apt cluster has an oversubscribed TOR; our protocols gracefully adapt to match the available bandwidth.

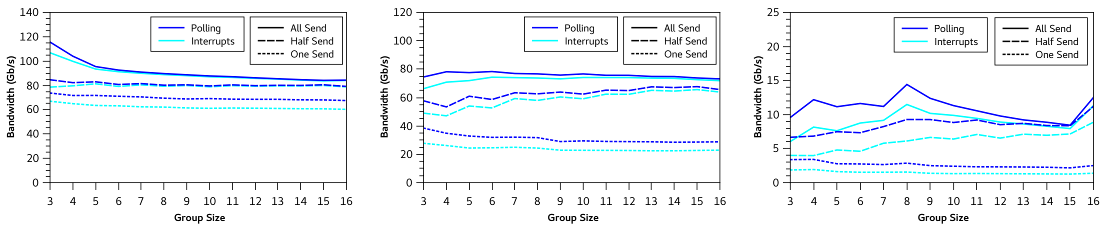
objects filled with random content, of the same sizes as seen in the trace, then replicating them by randomly selecting one of the possible 3-node groupings as a target (the required 455 RDMC groups were created beforehand so that this would be off the critical path). Figure 2.9 shows the latency distribution for 3 different send algorithms. Notice that binomial pipeline is almost twice as fast as binomial tree and around three times as fast as sequential send. Average throughput when running with binomial pipeline is around 93 Gb/s of data replicated, which translates to about a petabyte per day. We achieve nearly the full bisection capacity of Fractus, with no sign of interference between concurrent overlapping transfer. The RDMC data pattern is highly efficient for this workload: no redundant data transfers occur on any network link.

A second experiment looked at group overlap in a more controlled manner with a fixed multicast message size. In Figure 2.10, we construct sets of groups of the size given by the X-axis label. The sets have identical members (for example, the 8-node case would always have the identical 8 members), but different senders. At each size we run 3 experiments, varying the number of senders. (1) In the experiment corresponding to the solid line, all members are senders (hence we have 8 perfectly overlapped groups, each with the same members, but a different sender). (2) With the dashed line, the number of overlapping groups is half the size: half the members are senders. (3) Finally, the dotted line shows performance for a single group spanning all members but with a single sender. All senders run at the maximum rate, sending messages of the size indicated. Then we compute bandwidth by measuring the time to transfer a given sized message to *all* of the overlapping groups, and dividing by the message size times the number of groups (i.e. the total bytes sent).

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

Resource Considerations

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



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

Figure 2.11: Comparison of RDMC's normal hybrid scheme of polling and interrupts (solid), with pure interrupts (dashed). There is no noticeable difference between pure polling and the hybrid scheme. All ran on Fractus.

that this was not measurably faster than the hybrid.

Next, as shown in Figure 2.11, we compared RDMC in its standard hybrid mode with a version running using pure interrupts, so that no polling occurs. For the latter case, CPU loads (not graphed) are definitely lower: they drop from almost exactly 100% for all runs with polling enabled, to around 10% for 100 MB transfers and 50% for 1 MB transfers. With 10 KB transfers, there was only a minimal difference since so much time was spent processing blocks. Despite the considerable improvement in CPU usage, the bandwidth impact is quite minimal, particularly for large transfers. A pure-interrupt mode may be worthwhile for computationally intensive workloads that send large messages, provided that the slightly increased transfer delay isn't a concern.

On hardware that supports CORE-Direct we can offload an entire transfer sequence as a partially-ordered graph of asynchronous requests. Here, our preliminary experiments were only partially successful: a firmware bug (a NIC hardware issue) prevented us from testing our full range of protocols. Figure 2.12 shows results for chain send, where the

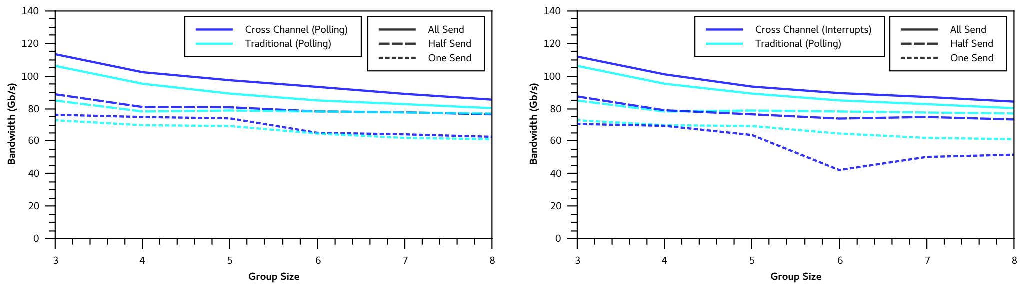


Figure 2.12: CORE-Direct experiment using a chain multicast protocol to send a 100 MB message. The left is a run using hybrid polling/interrupts; on the right is a run with purely interrupts. Both experiments were on Fractus.

request pattern is simple and the bug did not occur. The left graph uses a hybrid of polling and interrupts, while the right graph uses pure interrupts. As seen in the graphs, cross-channel generally provides a speedup of about 5%, although there is one scenario (a single sender transmitting in groups of size 5-8, in polling-only mode) in which our standard RDMC solution wins.

2.5.3 Future Work: RDMC on TCP

When Ganesan and Seshadri first explored multicast overlay topologies, they expressed concern that even a single lagging node might cause cascading delay, impacting every participant and limiting scalability [45]. This led them to focus their work on dedicated, synchronous, HPC settings, justifying an assumption that nodes would run in lock-step and not be exposed to scheduling delays or link congestion.

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

The observation has an interesting practical consequence: it suggests that RDMC might work surprisingly well over high speed datacenter TCP (with no RDMA), and perhaps even in a WAN network. In work still underway, we are porting RDMC to access RDMA through LibFabrics from the OpenFabrics Interface Alliance (OFI) [78]. LibFabrics is a mature solution used as the lowest layer of the message passing interface (MPI) library for HPC computing. The package uses a macro expansion approach and maps directly to RDMA as well as to other hardware accelerators, or even standard TCP. When the port is finished, we plan to closely study the behavior of RDMC in a variety of TCP-only settings.

2.6 Related Work

Replication is an area rich in software libraries and systems. We've mentioned reliable multicast, primarily to emphasize that RDMC is designed to replicate data, but is not intended to offer the associated strong group semantics and multicast atomicity. Paxos is the most famous state machine replication (consensus) technology. Examples of systems in this category include the classical Paxos protocol itself, our Derecho library, libPaxos, Zookeeper's ZAB layer, the head-of-log mechanism in Corfu, DARE, and APUs [2, 9, 55, 56, 64, 81, 99]. Derecho demonstrates that RDMC can be useful in Paxos solutions, but also that additional mechanisms are needed when doing so: RDMC has weaker semantics than Paxos.

We are not the first to ask how RDMA should be exploited in the operating system. The early RDMA concept itself dates to a classic paper by Von Eicken and Vogels [98], which introduced the zero-copy option and reprogrammed a network interface to demonstrate its benefits. VIA, the virtual interface architecture then emerged; its "Verbs" API extended the UNet idea to support hardware from Infiniband, Myrinet, QLogic and other vendors. The Verbs API used by RDMC is widely standard, but other options include the QLogic PSM subset of RDMA, Intel's Omni-Path Fabric solution, socket-level offerings such as the Chelsio WD-UDP [3] embedding, etc.

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

Broadcast is also important between CPU cores, and the Smelt library [57] provides a novel approach to address this challenge. Their solution is not directly applicable to our setting because they deal with tiny messages that don't require the added complexity of being broken into blocks, but the idea of automatically inferring reasonable send patterns is intriguing.

Although our focus is on bulk data movement, the core argument here is perhaps closest to the ones made in recent operating systems papers, such as FaRM [38], Arrakis [80] and IX [13]. In these works, the operating system is increasingly viewed as a control plane, with the RDMA network treated as an out of band technology for the data plane that works best when minimally disrupted. Adopting this perspective, one can view RDMC as a generic data plane solution well suited to out-of-band deployments. A recent example of a database optimized to use RDMA is Crail [93].

2.7 Conclusions

Our paper introduces RDMC: a new reliable memory-to-memory replication tool implemented over RDMA unicast. RDMC is available for download as a free, open-source library, and should be of direct use in O/S services that currently move objects either one by one, or over sets of side-by-side TCP links. The protocol can also be used as a component in higher level libraries with stronger semantics.

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

Acknowledgements

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

CHAPTER 3

DERECHO: FAST STATE MACHINE REPLICATION FOR CLOUD SERVICES

Notes:

This paper, published in the April, 2019 edition of the TOCS journal, is the cornerstone of my dissertation. The authors are Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew (Mae) Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Ken Birman. My advisor, Ken Birman, had the original idea for building Derecho – briefly, that it could center on a state management system, SST, that would use RDMA to share state, combined with the fast RDMC data path, all integrated in a non-blocking manner. At that time, our motivation was purely practical: we sought to better align the style of protocols used to implement atomic multicast and state machine replication (Paxos) with the timing properties and bandwidth of RDMA networks, building on the insights Ken and Jonathan and I had derived from our work on RDMC. In the abstract and introduction, we noted that this decision proved fortuitous, in that it aligned our performance optimization goals with what turned out to be theoretical optimality goals for the protocols themselves. However, we did not anticipate that outcome when we started the work.

I took on the lead role in transforming this from idea to system, and implemented the entirety of the SST and many aspects of the remainder of the infrastructure supporting RDMA-based point to point messaging, as well as the SMC small-message multicast. However, others made important contributions without which the system could never have been realized. Together with my colleagues, Edward Tremel and Weijia Song, I also created the needed group management system (view-change logic based on virtual synchrony), the message delivery pipeline (message sends, receives, acknowledgments, and delivery for strong consistency in multiple independent subgroups), and an infrastructure to support external clients.

There are some Paxos libraries in which the protocol itself is central and the implementation, minimal. Derecho has the opposite balance: the protocols were fairly mature when work began, but because we sought to express them in a completely new way, had to be “refactored”, implemented, and then tested and debugged. RDMA itself is not as fully specified as one might wish, and understanding the correct use of the needed RDMA interactions led us deeply into the hardware supporting the technology. Seemingly minor, standard, design choices turned out to have overheads that often overwhelmed all other aspects of performance. Thus, the work to create Derecho was a major undertaking and involved countless hours of coding and debugging.

The finished system is unique among Paxos-based infrastructures: optimal with respect to the protocol itself, ideally mapped to the RDMA hardware (and quite impressive on TCP as well, with early evidence that it will also dominate other options on DPDK). Derecho’s in-memory replication outperforms traditional Paxos systems on TCP by a factor of 80X and a modern Paxos system on RDMA by a factor of 25X. Derecho opens the door to a wide range of follow-on projects, such as the Cascade effort that Weijia Song and Ken Birman are pursuing, the DerechoDDS system Lorenzo Rosa created at the University of Bologna in Italy, and the Skyborg project at AFRL Dayton, which is using Derecho at the core of a new open-source infrastructure for advanced avionics. Other early adopters are likely to include Siemens, which is now prototyping a new form of "dashboard" for IIoT using Derecho as its communication layer and Microsoft, where a number of potential use cases have emerged.

None of these applications of Derecho is simple, and several would have been impossible to undertake without the support provided by the Derecho framework.

Beyond these projects, Derecho has potential uses in formal methods settings. I, alongside Ken Birman and Mae Milano, completed a protocol verification using the IVy prover (a first order logic that can be model-checked using Z3). We were guided by Orr Tamir, a PhD student at the Tel-Aviv university at the time. Currently, Mae Milano is leading a project to explore formal verification of Derecho, jointly with PL proof researchers at Princeton and New Jersey Institute of Technology. Goals for those formal spinoff activities center on the idea that at the core of Derecho is a formal "nugget" of significant interest and value: a potentially provable data replication "type" that could be extracted, proved, and ultimately reused in many contexts.

The optimality of the Derecho protocols has been a topic of ongoing interest, particularly for Ken Birman. With Gregory Chockler, Ken was able to evaluate the costs of the Derecho protocols using a methodology and optimality bounds posed by Keidar and Schraer. To their surprise, the match was exact (a point touched on in the 2019 paper). In ongoing but unpublished work, Ken has found that the progress condition for the protocol is a form of eventual strong failure sensing that (in other work by Keidar) is apparently also optimal. Let me say, however, that I am personally much more drawn to the practical and engineering challenges of exploiting the world's fastest communication technologies in modern data center computing settings.

The chapter produced here contains small edits that fix errors present in the pseudo-code in Appendix A of the original published version from ACM TOCS. The issues involved errors in the formal specification, developed at the time of writing the paper. The errors were identified by Annie Liu's DistAlg project that began to test our specification using a framework they created specifically as a debugging tool for this purpose. We are grateful to Professor Liu and her students for their help. It is important to note that these errors in the specification were not present in the system, which was already up and running at the time, passing increasingly aggressive unit tests, and even in use by real practitioners. In effect, we created and proved our system first, then later created the formal specification, and typos crept in at that final step because our formal specification departed at some points from what we had implemented and even from our own proofs. The majority of these mistakes involved the formal description of the code used to compute the current length of the filled portion of the ring-buffer used in the SMC protocol (they were "off by one" mistakes).

Abstract

Cloud computing services often replicate data and may require ways to coordinate distributed actions. Here we present Derecho, a library for such tasks. The API provides interfaces for structuring applications into patterns of subgroups and shards, supports state machine replication within them, and includes mechanisms that assist in restart after failures. Running over 100Gbps RDMA, Derecho can send millions of events per second in each subgroup or shard and throughput peaks at 16GB/s, substantially outperforming prior solutions. Configured to run purely on TCP, Derecho is still substantially faster than comparable widely used, highly-tuned, standard tools. The key insight is that on modern hardware (including non-RDMA networks), data-intensive protocols should be built from non-blocking data-flow components.

3.1 Introduction

There is a pervasive need for cloud-hosted services that guarantee rapid response based on the most up-to-date information, scale well, and support high event rates. Yet today's cloud infrastructure focuses primarily on caching with limited consistency guarantees [26, 84]. A consequence is that if data were captured from IoT source such as cameras and videos, and immediately processed at the edge, errors could occur. Instead, incoming data is typically stored into a global file system and processed later in batches, consuming significant resources and introducing long delays.

A Derecho is an intense storm characterized by powerful straight-line winds that overwhelm any obstacle. Our work views data replication similarly: Derecho moves data over non-stop pipelines, then superimposes a state machine replication model [86] on these flows using out-of-band logic. At each step, the developer can customize event handling, leveraging consistent replicated state. Examples include data compression, discarding uninteresting data, extracting higher level knowledge from sensor inputs, initiating urgent responses, and so forth.

State machine replication can be implemented in many ways. Derecho uses virtual synchrony [23] for dynamic membership tracking. The update path uses a version of Paxos [64], and is independent of the path used for queries, which exploits a new form of temporally-accurate, strongly-consistent, snapshot isolation. Unlike classic snapshot isolation, which in fact weakens consistency, this new approach runs queries on a strongly consistent snapshot, which is additionally deterministic (the same indexing operation will always return the same result) and temporally accurate (up to clock synchronization limits). Full details can be found in [91].

Many cloud computing architects have been critical of strongly consistent replication [26], citing concerns about speed and scalability. In prior solutions such issues can often be tracked to pauses associated with two-phase commit or similar interactive data exchanges. While the leader is waiting, backlogs form and this triggers congestion control. A central innovation in Derecho is that our protocols exchange protocol-control information through a lock-free distributed shared memory that we call a shared-state table (SST). All protocol participants learn of progress directly from their peers through updates to the

SST. A form of monotonic deduction allows them to independently deduce when batches of messages are safe to deliver and in what order. Thus data is moved in non-blocking, high-rate flows; control information is exchanged through non-blocking one-way flows; and the update and query paths are separated so that neither blocks the other.

The developers of the Bloom streaming database system noted that distributed systems can preserve consistency and execute asynchronously until an event occurs that requires consensus. Consensus however cannot be implemented without blocking [33]. This insight carries over to state machine replication. Classical Paxos protocols run a two-stage protocol in which each stage uses a form of two-phase commit on every update, whereas Derecho's non-blocking pipelined protocols leverage the full throughput of the network, discovering this same commit point asynchronously and in receiver-defined batches. Doing so allows Derecho to avoid pausing on each update. Derecho is still forced to pause when reconfiguring: virtual synchrony membership agreement requires a two-phase majority information exchange to ensure that logical partitions (split-brain behavior) cannot arise. But the delay is brief: typically, less than 200ms.

For example, when running over TCP on 100G Ethernet, Derecho is at least 100x faster than today's most widely-used Paxos libraries and systems, and it can support system scales and data sizes that trigger collapse in prior solutions. If available, RDMA permits an additional 4x performance improvement, sharply reduces latencies, and offloads so much work to the NIC that CPU overheads drop to near zero. On a machine where the C++ `memcpy` primitive runs at 3.75GB/s for non-cached data objects, Derecho over 100Gbps RDMA can make 2 replicas at 16GB/s and 16 replicas at 10GB/s: far faster than making even a single local copy. The slowdown is sublinear as a function of scale: with 128 replicas, Derecho is still running at more than 5GB/s. Moreover, the rate of slowdown as a function of scale is logarithmic, hence very large deployments continue to give high performance. Latencies from when an update is requested to when it completes can be as low as 1.5μs, and recipients receive them simultaneously, minimizing skew for parallel tasks. When persisting data to SSD, the replication layer is so much faster than the NVM write speed that we can peg the full speed of the storage unit, with no loss of performance even when making large numbers of persisted copies.

This new point in the cloud-performance space enables new options for cloud infrastructure design. First, by offering consistency at high speed in the edge, it becomes reasonable to talk about taking immediate action when critical events occur, or using the edge as an intelligent data filtering layer that can draw on the most current knowledge possessed by the cloud for decision-making. Second, simply because it can make remote copies so quickly, Derecho enables a kind of parallelism that would traditionally have been impossible.

The remainder of the paper is organized as follows. Section 3.2 focuses on the application model, the Derecho API and the corresponding functionality. Section 3.3 discusses the system architecture, protocols and implementation details. Section 3.4 focuses on Derecho's performance, but also includes side-by-side comparisons with LibPaxos, Zookeeper and APUS. Section 3.5 reviews prior work. Appendix 3.7 shows pseudo-code for some key protocol steps, and Appendix 3.8 presents the full protocol suite in detail. Appendix 3.9 discusses the conditions under which Derecho is able to make progress,

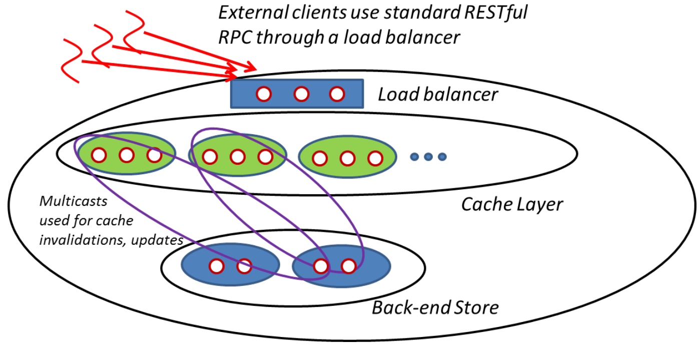


Figure 3.1: Derecho applications are structured into subsystems. Here we see 16 processes organized as 4 subsystems, 3 of which are sharded: the cache, its notification layer, and the back-end. A process can send 1-to-1 requests to any other process. State machine replication (atomic multicast) is used to update data held within shards or subgroups. Persisted data can also be accessed via the file system, hence a Derecho service can easily be integrated into existing cloud computing infrastructures.

comparing these with the conditions under which classic Paxos can make progress.

3.2 Application Model and Assumptions Made

3.2.1 Use cases

We target systems hosted in a single cloud-based data center, composed of some set of microservices (μ-services) each implemented by a pool of processes (Figure 3.1). Today, such services generally interact with external clients via RESTful RPC, but use message queuing (rendezvous or pub-sub) for internal communication. This architecture is widely popular both in the cloud edge and in back-end layers, and is seen in platforms such as the Apache infrastructure, Spark/Hadoop, Amazon AWS, Azure, Google Cloud, and IBM

WebSphere. Key-value sharding permits scalability.

Derecho could be used to create a new generation of faster, more scalable μ-services offering standard APIs but running at higher speeds. Obvious candidates include configuration management (Zookeeper), pub-sub message queuing (Kafka, SQS), file storage (HDFS, Ceph), tabular data storage (BigTable), distributed shared memory, etc. We intend to explore these possibilities, since the mix of better performance with stronger consistency, all packaged for backward compatibility, would appeal to a large existing community. On the other hand, existing applications evolved in a setting lacking ultra-fast consistent data replication, hence simply porting the underlying infrastructure to run over Derecho would make limited use of the system's power.

We conjecture that the most exciting possibilities will involve time-critical applications arising in the IoT domain: services to support smart homes or power grids, smart highways, self-driving cars, etc. Here application developers could work directly with our library to perform machine learning tasks close to sensors and actuators, for example to filter uninteresting images while tagging important ones for urgent action. These are examples in which the standard cloud edge is unable to provide the needed mix of consistency and real-time responsiveness, hence Derecho would enable genuinely new functionality.

3.2.2 Process groups

Derecho adopts a *process group* computing style that was popular for several decades starting in the 1980's. Process group software libraries fell into disuse as cloud computing emerged, reflecting perceived scalability and performance issues, limitations that Derecho overcomes. Process groups are dynamic, with members joining and leaving while the application is active. In accordance with the virtual synchrony model, membership evolves through a single sequence of views, with no partitioning [21]. The system-managed view can easily be augmented with parameters or other forms of application-defined metadata.

3.2.3 Programming model

Our programming model assumes that a single service (a single "top-level group") will consist of a collection of identical application processes. A top-level group might be large (hundreds or thousands of members), but would often be structured into a set of subsystems with distinct roles, as seen in Figure 3.1. In this example, the μ-services include a load balancer, a sharded cache layer, a sharded back-end data store, and a pattern of shard-like subgroups used by the back end to perform cache invalidations or updates. Each μ-service is implemented by an elastic pool of application instances. Notice that some μ-services consist of just a single (perhaps large) subgroup while others are sharded, and that shards typically contain 2 or 3 members. Overlap is not permitted between the shards of any single subgroup, but by creating two subgroups with the same membership, it is easy to implement overlapped shards.

The capabilities of a μ-service are defined by a C++ class of type `replicated<T>`. Each application instance runs identical source code, hence possesses all such definitions. However, an application instance instantiates only objects associated with roles it actually plays. A mapping function is used to compute this role assignment: each time a new view of the top-level group is installed, it maps the membership to a set of subgroups. If a subgroup is sharded, the mapper additionally lays the shards out in accordance with user preferences for replication level.

Non-members interact with a subgroup or a shard by invoking point-to-point handlers, using a *handle* object to obtain type signatures for the available operations. At present, this option is available only for members of the top-level group, but we plan to extend the system so that external processes with compatible memory layouts could use the API as well, permitting them to benefit from our strongly typed RDMA operations. Of course, external processes can also use standard RPC mechanisms such as REST, WCF, JNI, the OMG IDE, etc., but such tools often have heavy marshalling and copying overheads.

Members hold replicated state, and can access it directly: data is fully replicated, and all have identical content, which can be read without locks. Members can also initiate updates through a multicast called `ordered_send` that can be configured to behave as a what we call an "unreliable mode" 1-to-many data transfer (lowest cost, but very weak semantics), an atomic multicast, or a durable Paxos update. Figures 3.3–3.6 illustrate a few of the communication patterns that a subgroup or shard might employ.

3.2.4 Restarts and failures

When a group is initially started or a new member is added to an already-active system, Derecho orchestrates the loading of persisted state, repairs logs that contain uncommitted

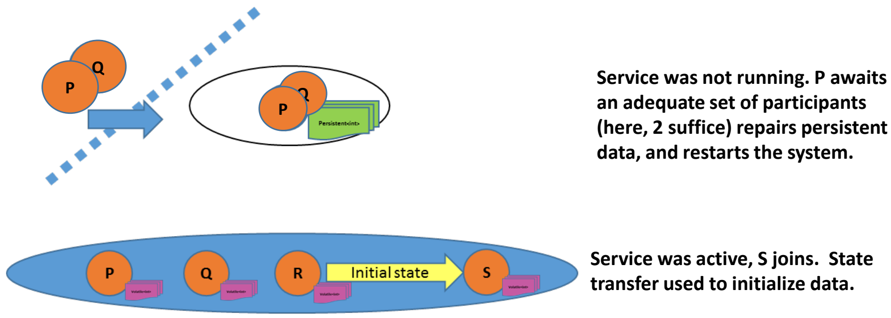


Figure 3.2: When launched, a process linked to the Derecho library configures its Derecho instance and then starts the system. On the top, processes P and Q restart a service from scratch, reloading persisted state; below, process S joins a service that was already running with members {P,Q,R}. As the lowest-ranked process, P "leads" during these membership reconfigurations; were P to fail, Q would take over, and so forth. Under the surface, the membership management protocol itself is also a version of Paxos.

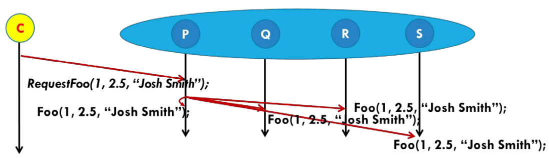


Figure 3.3: Multicasts occur within subgroups or shards and can only be initiated by members. External clients interact with members via P2P requests.

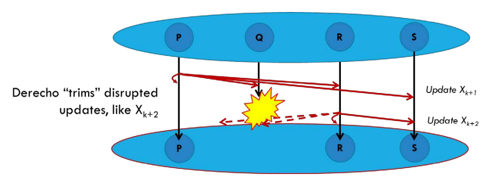


Figure 3.4: If a failure occurs, cleanup occurs before the new view is installed. Derecho supports several delivery modes; each has its own cleanup policy.

versions or omit committed ones, coordinates state-transfer to the joining process, and so forth (see Figure 3.2). As a consequence, whenever a group becomes active, every shard will be adequately populated, all data will be replicated to the desired degree, and every replica will be in the identical state.

Derecho can tolerate two forms of failure: *benign crash failures* and *network partitioning* (see Appendix C for more details). We believe that such failures will occur relatively

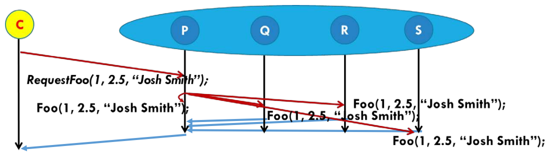


Figure 3.5: A multicast initiated within a single subgroup or shard can return results. Here, the handler would run on the most current version.

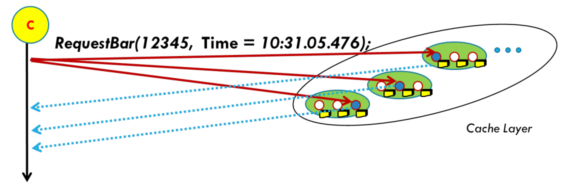


Figure 3.6: Read-only queries can also occur via P2P requests that access multiple subgroups or shards. Derecho's temporal indexing is lock-free, yet strongly consistent: a new form of *temporal snapshot isolation.*

infrequently for even moderately large data center services (ones with thousands of processes). At Google, Jeff Dean measured reliability for several such services, and found that disruptive failures or reconfigurations occurred once per 8 hours on average [87]. In contrast, Derecho needs just a few hundred milliseconds to recover from failure.

The transports over which Derecho runs all deal with packet loss[1], hence Derecho itself never retransmits. Instead, unrecoverable packet loss results in a failure detection even if both endpoints are healthy. The communication layer notifies Derecho, which initiates reconfiguration to drop the failed endpoint.

A network failure that partitions a group would manifest as a flurry of "process failure" events reported on both "sides" of the failed link. However, just one side could include a majority of the current view. Accordingly, virtual synchrony requires that any service that loses access to a majority shut down. This is in contrast with what Eric Brewer put forward as the CAP methodology, in which consistency is weakened to permit availability

[1]RDMA requires a nearly perfect network, but does have a very primitive packet retransmission capability.

even during partitioning failures [26, 48]. In effect, Derecho favors consistency over availability: the minority halts so that the majority can safely continue[2].

Even if the top-level group never partitions, all the members of a shard could fail. In Derecho, the mapping function treats such a state as inadequate and forces the application to wait until more processes recover and all shards can be instantiated. Indeed, Derecho pauses even if some durable shard simply has fewer than its minimum number of members: here, if we allowed updates we might violate the developer's desired degree of replication.

Accordingly, after a failure Derecho waits for an *adequate* next view: one that includes a majority of members of the prior view, and in which all durable shards have a full complement of members. If needed, Derecho then copies state to joining members, and only then does the system permit normal activity to resume.

3.2.5 The `replicated<T>` class

Each Derecho application consists a set of classes of type `replicated<T>`. The developer can specify the operations and replicated state that the class will support. Event handlers are polymorphic methods tagged to indicate whether they will be invoked by point to point calls or multicast. The constructor of `replicated<T>` configures the group to select the desired semantics for updates: unreliable mode multicast, atomic multicast or durable Paxos.

[2]Derecho targets data center use cases. In a WAN, partitioning is more frequent and this tactic would not be appropriate.

Derecho groups know their own membership. An application process that does not belong to a subgroup or shard would select a proxy, often the 0-ranked member. Then it can issue a point-to-point request. In these examples, `who` designates the desired proxy:

```
ExternalCaller<MemCacheD>& cache = g.get_nonmember_subgroup<MemCacheD>(k)
```

```
auto outcome = cache.p2p_send<RPC_NAME(request_put)>(who, "John Smith", 22.7);
```

```
auto result = cache.p2p_query<RPC_NAME(get)>(who, "Holly Hunter");
```

The multicast and Paxos options are limited to communication from a member of a subgroup or shard to the other members:

```
replicated<MemCacheD>& cache = g.get_subDerechoGroup<MemCacheD>(k);
auto outcome = cache.ordered_send<RPC_NAME(put)>("John Smith", 22.7);
```

Here, `ordered_send` was used: a 1-to-N multicast. Failures that disrupt an `ordered_send` are masked: the protocol cleans up, then reissues requests as necessary in the next adequate membership view, preserving sender ordering. The `outcome` object tracks the status of the request, indicating when the operation has completed. If the method returns results, the outcome object can also be used to iterate over responses as they arrive

3.2.6 Versioned Data and Persistent Logs

Derecho services are fault-tolerant and parallel by virtue of using state machine replication. But where does the data live? One option is to simply declare some variables within the class, and to register them for state transfer: when a process joins an active group the variable will be included in the initialization state of the joining member. Here, it suffices to perform updates using `ordered_send`. Members will perform updates in the same order, and if any performs an update, every non-failed members will do so.

In some applications it is more natural to save state in a log. A log could implement a versioned variable (in which each update yields a new version), function as the block store for a versioned file system, or even "Parliamentary decrees" as in Lamport's original Paxos scenario. In support of these cases, we offer an atomic log append.

Derecho performs atomic appends in two steps. On arrival of the update, as an atomic multicast, the system issues an upcall to the relevant event handler. This runs, and the system learns the new versions of the associated state variables (Figure 3.7). Each new version builds on the prior one, hence the event handlers are simple to code and do updates in a natural way. For example, if counter is a part of the replicated state of some shard, operations like counter++ work normally. Where a version can more compactly be represented by a delta, the option of doing so is available. If no change occurs, no version is created.

At this initial stage, the new versions are considered to be pending, but not yet committed[3]. This is illustrated in Figure 3.7, with older and more stable data on the left, and new updates on the right (the newest are colored red).

[3]Derecho has multiple levels of update *stability*. The ones relevant to the present paper are version commit and temporal stability. We define the latter to be the time such that no new update could have an earlier timestamp. This is identical to the way Spanner [34] delays updates until it is certain that no update with a prior timestamp could still be in the network.

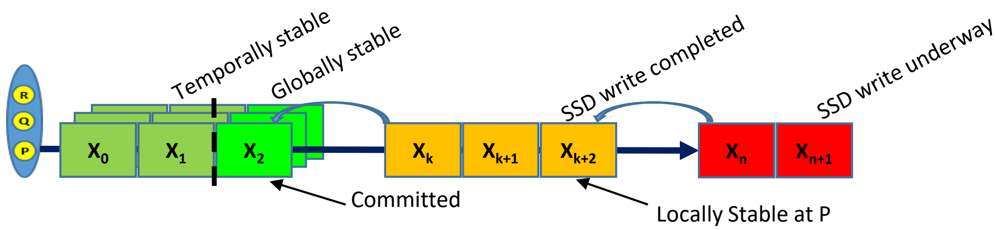


Figure 3.7: Derecho applications store data in replicated version vectors. Versions pass through a series of stages: they are created (red), logged locally (orange) and then become stable (green). Within the stable versions, Derecho distinguishes between basic Paxos durability (all green versions) and temporal stability (darker green), which adds an additional guarantee that no update with an earlier real-time timestamp could occur.

.

Next, Derecho writes the new data to the end of the respective log replicas: each of P, Q and R has a local file that it uses for this purpose. In the figure, these updates are shown in yellow: they are locally logged but not yet globally stable.

Finally, when enough copies are safely logged, Derecho can consider the update to have committed (green versions), and read-only queries can be permitted to access the updated values. If some failure disrupts this sequence, Derecho will back the updates out by truncating the logs, then reissue them.

Since this sequence is subtle, Derecho packages it as a *version-vector subsystem*. A replicated version vector is an extensible array of type `volatile<T>` or `persistent<T>`; the developer defines the underlying byte-serializable class. Version vectors have an in-form (the array API), but also an out-form: they can also be accessed through a file system layer like HDFS or Ceph, via a file name that the user supplies in the constructor for the vector object. Our version-vector implementation is actually based on the Freeze Frame File System (FFFS) [91], and can support extremely large amounts of data, using its own caching layer to balance performance against memory footprint. The subsystem will retrieve data in a way that is lock free, deterministic, out of band from the update path, as temporally precise as possible within clock synchronization limits, and causally consistent [30, 91].

A Derecho-based "file server" is fully customizable: all operations are ultimately performed by event-handlers that the developer customizes. Thus we end up with a new kind of temporal file system that can perform tasks such as data compression, deduplication, transformation, and so forth directly on the update and query code pathways. Moreover, the file system will scale through sharding, and is both fault-tolerant and consistent.

In Figure 3.6 we saw an example that used time-indexed read-only versioned queries. Such a query can arise from a file-system access in HDFS or Ceph, in which case we obtain the temporal index from the file name (we overload a POSIX notation used for file-system checkpoints). Alternatively, the query could arise inside the Derecho-based C++ application itself, in which case the developer would specify the timestamp. The indexing operation looks like any indexed vector read, but the index will be of type `time_t`. If the requested time is in the future, or if the data is not yet committed it must await temporal stability, as seen in Figure 3.7 (dark green). Then the read can occur. For queries that access data spread across multiple shards, we obtain a powerful new model: *stable, temporally precise, and causally consistent temporal snapshot isolation.*

3.3 Design and Implementation Considerations

In this section we turn to the question of how we created the actual Derecho protocols.

The overarching trend that drives our system design reflects a shifting balance that many recent researchers have highlighted: RDMA networking is so fast that to utilize its full potential, developers must separate data from control, programming in a pipelined, asynchronous style [13, 80].

To appreciate the issue, consider the most widely discussed Paxos protocol (often called the "Synod" protocol). The goal of the protocol is to accept requests from external clients, persist them into a a log, and then offer a way to learn the contents of the log [64]. The technology of the 1990's shaped Lamport's problem statement and choice of solution: I/O was very slow compared to computation, and servers were single-core machines prone to long scheduling and workload-related delays. Machines had assigned roles: the set of servers running the application was known, even if at a particular moment some might be unavailable.

We see this in Lamport's description of a community governed by politicians who wander in and out of the Senate, desiring progress on new legislation, but unwilling to wait for a quorum on the Senate floor. In the Synod protocol, each proposal has a leader. It prepares the proposal (perhaps, a batch of end-user requests), then competes to gain ownership of a slot in the log, using a series of ballots that each require two-phase commit. When the leader succeeds in obtaining majority consent to its ballot, a second two-phase operation commits the action.

Suppose that we used this protocol to perform updates on today's NUMA machines and RDMA networks, that the request being transmitted contains 10KB of data, and that there is just one active leader. With a 100Gbps RDMA network that has a 1.5us round-trip latency, a leader successful in the first Paxos ballot proposal will still need to send the 10KB proposal to each of the Paxos group members, wait for a majority to respond, verify that all are still prepared to commit, and then commit. With a group of 5 members, we might see 5 RDMA send operations, a delay to collect at least 3 responses, 5 smaller RDMA sends, a further delay to collect responses, and then a final commit. Thus in the best case the latency will be at least $5 * 1.5\mu$s, plus 1μs for data transfer: 8.5μs. We would need 115,000 concurrently active Paxos operations to keep the network busy![4]

In a server group with 5 members, it might make more sense to have all be leaders, each owning a disjoint sets of slots (a standard Paxos optimization). But 23,000 Paxos log-append proposals would need to be running in each leader: prohibitive if each runs separately. Accordingly, leaders must batch requests: perhaps 5,000 at a time, so that each Paxos operation carries 50MB of data. Doing so has the disadvantage of forcing new request to wait, but resource stress is reduced. The query side of the system also faces steep overheads. In Lamport's classic Paxos, an update only needs to reach a majority of logs, but queries therefore have to read a majority of logs and combine them, to ensure that the reader learns complete set of committed updates. Thus any single read operation

[4]Many protocols, including some variations of Paxos, run in a single round, but still use a two-phase or request-response pattern during that round. The reduced number of rounds would reduce the backlogs and delays, yet would still support the identical conclusion.

needs to perform an RPC to multiple servers, and await a read-quorum of responses: a costly remote operation.

Despite these drawbacks, the design just described is present in most modern RDMA Paxos protocols, such as DARE and APUS [81, 99], which have leader-based structures with the potential for pauses on the critical path (examples of exception include AllConcur, which uses a leaderless approach [82] and NOPaxos, which leverages datacenter network components to order events [68]).

But how might one implement Paxos if the goal, from the outset, were to leverage RDMA? The central challenge is to shift as many decisions off the runtime data path as possible.

3.3.1 Monotonic Deduction on Asynchronous Information Flows

The points just made lead to a model called *virtually synchronous Paxos*, which combines a membership-management protocol that was created as part of the Isis Toolkit [23] with a variation of Paxos. The virtually synchronous Paxos model was originally suggested by Malkhi and Lamport at a data replication workshop in Lugano. Later, the method was implemented in a distributed-systems teaching tool called Vsync, and described formally in Chapter 22 of [21].

The virtual synchrony model focuses on the evolution of a process group through a series of *epochs*. An epoch starts when new membership for the group is reported (a *new* *view* event). The multicast protocol runs in the context of a specific epoch, sending totally ordered multicasts to the full membership and delivering messages only after the relevant safety guarantees have been achieved. These include total ordering, the guarantee that if any member delivers a message then every non-failed member will do so[5], and (if desired) durable logging to non-volatile storage. An epoch ends when some set of members join, leave, or fail. This could occur while a multicast is underway. Such a multicast must be finalized either by delivering it, or by reissuing it (preserving sender ordering) in the next epoch.

In keeping with the goal that no component of Derecho unnecessarily wait for action by any other component, we transformed Paxos into a version in which all receivers continuously and asynchronously learn about the evolving state of an underlying data stream. Key to expressing Paxos in this manner is the insight that we can track the state of the system through all-to-all information exchanges through a table of *monotonic* variables: single-writer, multiple-reader counters that advance in a single direction. We should emphasize that in transforming Paxos this way, we did not change the underlying knowledge-state achieved by the protocol. We simply remapped the pattern of information passing to better match the properties of the network. An analogy to a compiler optimization that pipelines operations while preserving correctness would not be inappropriate.

When aggregation is performed on monotonic data, and a predicate is defined over the aggregate, we obtain a logical test that will be stable in the sense that once the predicate

[5]The original Isis formulation added an optional early-delivery feature: multicasts could be delivered early, creating an *optimistic* mode of execution. A stability barrier (`flush`), could then be invoked when needed. Derecho omits this option.

holds, it continues to hold (even as additional data is exchanged). But such predicates have a further, and particularly useful properly: they are themselves monotonic, in that they can be designed to cover a batch of events rather than just a single event. A monotonic predicate is one with the property that *if the predicate holds for message* k*, it also holds for messages* $0...(k-1)$.

Monotonic predicates permit discovery of *ordered deliverability* or *safety* for sets of messages, which can then be delivered as a batch. Notice that this batching occurs on the receivers, and will not be synchronized across the set: different receivers might discover safety for different batches. The safety deductions are all valid, but the batch sizes are accidents of the actual thread scheduling on the different receivers.

In contrast, many Paxos protocols use batching, but these batches are formed by a leader. A given leader first accumulates a batch of requests, then interacts with the acceptors (log managers) to place the batch into the Paxos log. Batches are processed one by one, and incoming requests must wait at various stages of the pipeline.

Monotonic protocols help Derecho achieve high efficiency. Yet notice that the core question of safety is unchanged: Derecho still uses a standard Paxos definition. In effect, we have modified the implementation, but not the logical guarantee.

3.3.2 Building Blocks

Derecho is comprised of two subsystems: one that we refer to as RDMC [12], which provides our reliable RDMA multicast, and a second called the SST, which implements a shared memory table that supports the monotonic logic framework within which our new protocols are coded, and a bare bones multicast for small messages that we refer to as SMC. In this section, we provide overviews of each subsystem, then focus on how Derecho maps Paxos onto these simpler elements.

RDMC and SST both run over RDMA using reliable zero-copy unicast communication (RDMA also offers several unreliable modes, but we do not use them). For hosts P and Q to communicate, they establish RDMA endpoints (*queue pairs*) and then have two options:

1. "Two-sided" RDMA. This offers a TCP-like behavior in which the RDMA endpoints are *bound*, after which point if P wishes to send to Q, it enqueues a send request with a pointer into a pinned memory region. The RDMA hardware will perform a zero-copy transfer into pinned memory designated by Q, along with a 32-bit immediate value that we use to indicate the total size of a multi-block transfer. Sender ordering is respected, and as each transfer finishes, a completion record becomes available on both ends.
2. "One-sided" RDMA, in which Q grants permission for P to remotely read or write regions of Q's memory. P can now update that memory without Q's active participation; P will see a completion, but Q will not be explicitly informed.

RDMA is reliable and success completion records can be trusted. Should an endpoint crash, the hardware will sense and report this. For portability, Derecho accesses RDMA through LibFabric, an industry-standard layer that maps directly to RDMA if the hardware is available, but also offers a standard TCP simulation of RDMA functionality, so that we can also run on platforms that lack RDMA capabilities. LibFabric can also support other remote DMA technologies, such as Intel's OMNI Path.

To achieve the lowest possible latency, RDMA requires continuously polling for completion events, but this creates excessive CPU usage if no RDMA transfers are taking place. As a compromise, Derecho dedicates a thread to polling completions while transfers are active but switches to sleeping when inactive, reporting this through a field in its SST row. When the next transfer occurs, the initiator of the RDMA transfer sees that the target is sleeping, and uses an RDMA feature that triggers an interrupt to wake up the thread.

3.3.3 RDMA Multicast: SMC and RDMC

Derecho moves data using a pair of zero-copy reliable multicast abstractions, SMC and RDMC, both of which guarantee that messages will be delivered in sender order without corruption, gaps or duplication, but lack atomicity for messages underway when some member crashes. Later we will see how Derecho senses such situations and cleans up after a failure. SMC and RDMC are both single-sender, multiple receiver protocols. A group with multiple senders would instantiate a collection of multicast sessions, one for each potential sender.

Small messages. Derecho's *small message multicast protocol*, SMC, runs over one-sided writes. Recall that Derecho's subgroups can be sharded, but that shards *within a single subgroup* cannot overlap. To implement SMC, we build on this observation: for each subgroup or shard, we identify each potential multicast sender with an SMC session. Then each subgroup or shard member allocates a ring buffer for incoming messages from that potential sender. To send a new SMC multicast, the sender loops through the membership and for each member (1) waits until there is a free slot, (2) writes the message and (3) increments a counter of available messages. A receiver waits for an incoming message, consumes it, then increments a free-slots counter. This approach dates (at least) to Unix pipes, but was first applied to RDMA by systems such as U-Net [98], BarrelFish [11] and Arrakis [80].

SMC costs rise linearly in subgroup/shard size, message size, and in the number of subgroups or shards within which each process is a potential sender. Accordingly, we use SMC only with small fanouts, for messages no more than a few hundred bytes in size, and only if processes have a small number of "sender roles."

Large messages. For cases that exceed the limits supported by SMC, we use RDMC [12]. This protocol supports arbitrarily large messages which it breaks into 1MB chunks, then routes on an overlay network (the actual network is fully connected, so this overlay is purely an abstraction). A number of chunk-routing protocols are supported, all of which are designed so that the receiver will know what chunk to expect. This allows the receiver to allocate memory, ensuring that incoming data lands in the desired memory region. Once

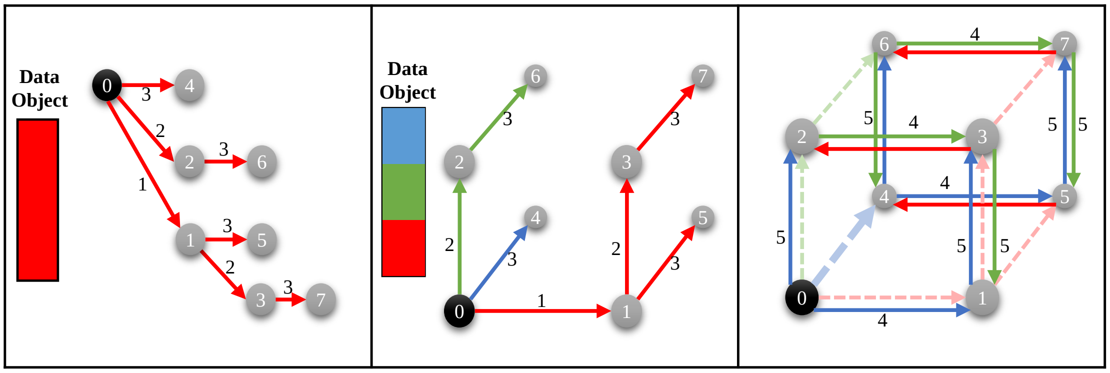


Figure 3.8: Our RDMA multicast protocol, RDMC [12], breaks large messages into chunks, then forwards them chunk by chunk over a network overlay of reliable FIFO channels. Here the left diagram illustrates a binomial copying protocol, with processes represented by circles and data transfer steps represented by numbered arrows; the same number can appear on multiple arrows if transfers are concurrent. The middle and right diagrams illustrate the idea of running d binomial protocols simultaneously on a d-dimensional hypercube, created as a network overlay within our RDMA network (which allows all-to-all connectivity). For the binomial pipeline, transfers occur in sets of d chunks: 3 blocks colored red, green and blue in this example. When the pipeline is fully active, every node has one block to send and one block to receive at each logical timestep, somewhat like in BitTorrent, but with a fully deterministic pattern.

all chunks are received, the message is delivered via upcall either to a higher-level Derecho protocol (for a group in durable totally ordered mode or totally ordered (atomic multicast) mode) or directly to the application (for a group in unreliable mode).

In all cases evaluated here, RDMC uses a *binomial pipeline*, which we adapted from a method originally proposed for synchronous settings [45] and illustrate in Figure 3.8. Chunks of data disseminate down overlaid binomial trees such that the number of replicas with a given chunk doubles at each step. This pattern of transfers achieves very high bandwidth utilization and minimizes latency. If the number of processes is a power of 2, all receivers deliver simultaneously; if not, they deliver in adjacent protocol steps.

An obvious concern that arises with relaying is that scheduling delay at a single slow member could potentially ripple downstream, impacting the whole group. To our surprise, experiments on RDMC revealed no such issue even when we were very aggressive about injecting network contention, overlapping data flows, and scheduler hiccups. It turns out that the binomial pipeline has a considerable degree of *slack* in the block transfer schedule. Although in most steps a process relays a chunk received during the immediately prior step, every now and then a process switches to relaying some other chunk that was received much earlier, and there are some steps in which a process has no chunk to relay at all. These events offer opportunities for a slightly delayed process to catch up, and where a delayed incoming block would not impact downstream performance.

Hybrid RDMC protocols. Although not used in the experiments reported here, RDMC includes additional protocols, and a datacenter owner can compose them to create hybrids. An interesting possibility would be to use the Binomial Pipeline twice: once at the top-of-rack (TOR) level, then again within each rack. Another option would be to use the *chain pipeline* protocol at the TOR level. This forms a bucket brigade: each message is chunked and then sent down a chain. Each TOR process would then be concurrently receiving one chunk while forwarding a prior one. A TOR chain would minimize the TOR load but have higher worst-case latency than a TOR instance of the binomial pipeline.

Failures. When RDMC senses a failure, it informs the higher level using upcalls and then wedges, accepting no further multicasts and ceasing to deliver any still in the pipeline. This can leave some multicasts incomplete (they may not have been delivered at all, or may have been delivered to some destinations but not to others).

3.3.4 Shared State Table: The SST

Reliable multicast is a powerful primitive, but requires that the application track membership and arrange for the endpoints to simultaneously set up each needed session, select a sender, and coordinate to send and receive data on it. With multiple senders to the same group of receivers, these protocols provide no ordering on concurrent messages. When a failure occurs, a receiver reports the problem, stops accepting new RDMC or SMC messages, and "wedges," but takes no action to clean up disrupted multicasts.

To solve these kinds of problems, Derecho uses protocols that run on a novel replicated data structure that we call the *shared state table,* or SST. The SST offers a tabular distributed shared memory abstraction. Every member of the top-level group holds its own replica of the entire table, in local memory. Within this table, there is one identically-formatted row per member. A member has full read/write access to its own row, but is limited to read-only copies of the rows associated with other members.

This simple model is quite powerful: it eliminates write-write contention on memory cells, because any given SST row only has a single writer. To share data using the SST, a process updates its local copy of its own row, then *pushes* the row to other group members by enqueuing a set of asynchronous one-sided RDMA write requests. We also support pushing just a portion of the row, or pushing to just a subset of other processes.

Even though any given SST cell has just one writer, notice that a sequence of updates to a single SST cell will overwrite one another (RDMA is order-preserving, hence the last value written will be the final one visible to the reader). If writes occur continuously, and the reader continuously polls its read-only copy of that cell, there is no guarantee that they will run in a synchronized manner. Thus a reader might see the values jump forward, skipping some intermediary values.

The SST guarantees that writes are atomic at the granularity of *cache lines*, typically 64 bytes in size. The C++ 14 compiler aligns variables so that no native data type spans a cache line boundary, but this means that if an entire vector is updated, the actual remote updates will occur in cache-line sized atomic chunks. Accordingly, when updating multiple entries in a vector, we take advantage of a different property: RDMA writes respect the sender's FIFO ordering, in that multiple verbs are applied at the target node sequentially. Thus, we can *guard* the vector within the SST with a counter, provided that we update the vector first and then the counter in a separate verb. When a reader sees the guard change, it is safe for it to read the guarded vector elements. It can then acknowledge the data, if needed, via an update to its own SST row.

In the most general case, an SST push transfers a full row to $N - 1$ other members. Thus, if all members of a top-level group were actively updating and pushing entire rows in a tree-structured network topology, the SST would impose an N^2 load on the root-level RDMA switches. Derecho takes a number of steps to ensure that this situation will not be common. Most of our protocols update just a few columns, so that only the modified bytes need to be pushed. RDMA scatter-gather is employed to do all the transfers with a single RDMA write: an efficient use of the hardware. Furthermore, these updates are often of interest to just the members of some single shard or subgroup, and hence only need to

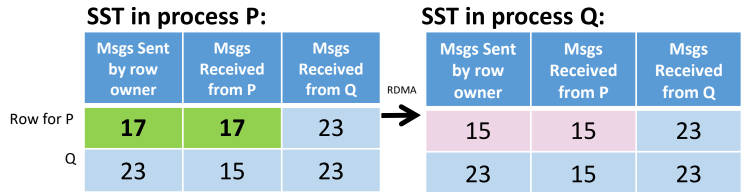


Figure 3.9: SST example with two members: P and Q. P has just updated its row, and is using a one-sided RDMA write to transfer the update to Q, which has a stale copy. The example, discussed in the text, illustrates the sharing of message counts and confirmations.

be pushed to those processes. Thus the situations that genuinely involve all-to-all SST communication most often involve just 2 or 3 participants. We have never seen a situation in which the SST was a bottleneck.

3.3.5 Programming with the SST

The SST is a flexible abstraction. Earlier we mentioned SMC; this is implemented directly on the SST's API. Stability tracking is also a simple SST protocol: in a subgroup or shard, as each multicast is received, members report receipt by incrementing a per-sender counter in their row. By computing the minimum, every member can track messages that every other member has received. The SST can even implement barrier synchronization with the Filter version of Peterson's Algorithm, or with Lamport's Bakery Algorithm.

Many protocols use aggregation. For example, suppose that a protocol needs to compute the minimum value within some SST column. If the column contained a strictly increasing counter, the minimum would be a value v such that every counter c_p in every process P satisfies $c_p \geq v$. This example illustrates *stable* deduction, in the sense that once the predicate becomes true, it remains true. Combining these ideas yields *monotonic* deductions: stable formulas $F(x)$ such that if $F(v)$ holds, then not only will $F(v)$ remain true, but we can also infer that $\forall v' < v : F(v')$. In other words, learning that F holds for v implies that F holds for every value from $0...v$. This generalizes the receiver-batched style of reasoning discussed earlier.

The SST framework layers high-level tools for logic programming over the basic functionality. The simplest of these is the *projector*, a wrapper type which allows programmers to project a value from a row. Generally, projectors just access some field within the row, although it may perform more complex reasoning (for example, indexing into a vector). These projectors are functions with the type $\mathsf{Row} \rightarrow \mathsf{T}$; they offer a convenient place to support variable-length fields and to implement any needed memory barriers.

The second (and more powerful) SST tool is the *reducer* function, SST's primary mechanism for resolving shared state. A reducer function's purpose is to produce a summary of a certain projector's view of the entire SST; intuitively, it is run over an entire SST column. One can think of these functions as serving a similar role to "merge" functions often found in eventual consistency literature; they take a set of divergent views of the state of some datum and produce a single summary of those views. Aggregates such as min, max, and sum are all examples of reducer functions.

By combining reducer functions with projectors, our Derecho protocols can employ complex predicates over the state of the entire SST without reasoning directly about the underlying consistency. The functionality of a reducer function is in fact higher order;

rather than simply running a projector f : $\mathsf{Row} \rightarrow \mathsf{T}$ over an SST column directly, it takes f, allocates a new field in the SST of type T, and returns a new function f' : $\mathsf{Row} \rightarrow \mathsf{T}$ which, when run on a row, sets the new field in that row to the result of f when run over the SST, finally returning the newly-set value. In this way, the reducer function actually has type $\mathsf{projector} \rightarrow \mathsf{projector}$, allowing reducer functions to be arbitrarily combined, somewhat like formulas over a spreadsheet.

Let's look at an example.

```
struct SimpleRow {int i;};
int iget(const volatile SimpleRow& s){
  return s.i;
}
bool proj(){
  return (Min(as_projector(iget)) > 7 ) || (Max(as_projector(iget)) < 2);
}
```

Here, function `proj` converts the function `iget` into a projector, calls the reducers `Min` and `Max` on this projector, then uses the boolean operator reducers to further refine the result.

We can do far more with our new projector, `proj`. The first step is to name the SST cell in which `proj`'s output will be stored[6]. Additionally we can also register a *trigger* to fire when the projector has attained a specific value. Extending our example:

[6]In practice, our implementation includes the obvious optimization of not actually storing the value unless other group members, distinct from the process that computes the projector, will use the value.

```
enum class Names {Simple};
SST<T> build_sst(){
  auto predicate = associate_name(Names::Simple, proj());
  SST<T> sst = make_SST<T>(predicate);
  std::function<void (volatile SST<T>&)> act = [](...){...};
  sst->registerTrigger(Names::Simple, act);
  return sst;
}
```

Here we have associated `proj` with the name `Simple` chosen from an `enum class Names`, allowing us to register the trigger `act` to fire whenever `proj` becomes true. As we see here, a trigger is simply a function of type `volatile SST<T>& -> void` which can make arbitrary modifications to the SST or carry out effectful computation. In practice, triggers will often share one important restriction: they must ensure monotonicity of registered predicates. If the result of a trigger can never cause a previously-true predicate to turn false, reasoning about the correctness of one's SST program becomes easy. Using this combination of projectors, predicates, and triggers, one can effectively program against the SST at a nearly-declarative high level, proving an excellent fit for protocols matching the common pattern "when everyone has seen event X, start the next round."

Encoding knowledge protocols in SST. SST predicates have a natural match to the *logic of knowledge* [52], in which we design systems to exchange knowledge in a way that steadily increases the joint "knowledge state." Suppose that rather than sharing raw data via the SST, processes share the result of computing some predicate. In the usual knowledge formalism, we would say that if *pred* is true at process P, then P *knows* pred, denoted $K_P(pred)$. Now suppose that all members publish the output of the predicate as each learns it, using a bit in their SST rows for this purpose. By aggregating this field using a reducer function, process P can discover that *someone knows* $pred$, that *everyone knows* $pred$, and so forth. By repeating the same pattern, group members can learn $K^1(pred)$: every group member *knows* that every other member $knowspred$. Using confirmations, much as with our simple multicast protocols, we can then free up the column used for $pred$ so that it can be reused for some subsequent predicate, $pred'$. For distributed protocols that run as an iterative sequence of identical rounds, this allows the protocol to run indefinitely using just a small number of SST fields.

Stable and monotonic predicates. Earlier, we defined a monotonic predicate to be a stable predicate defined over a monotonic variable v such that once the predicate holds for value v, it also holds for every $v' \leq v$. Here we see further evidence that we should be thinking about these protocols as forms of knowledge protocols. Doing so gives a sharp reduction in the amount of SST space required by a protocol that runs as a sequence of rounds. With monotonic variables and predicates, process P can repeatedly overwrite values in its SST row. As P's peers within the group compute, they might see very different sequences of updates, yet will still reach the same inferences about the overwritten data.

For example, with a counter, P might rapidly sequence through increasing values. Now, suppose that Q is looping and sees the counter at values 20, 25, 40. Meanwhile, R sees

Figure 3.10: SST example with three members, showing some of the fields used by our algorithm. Each process has a full replica, but because push events are asynchronous, the replicas evolve asynchronously and might be seen in different orders by different processes.

11, then 27, 31. If the values are used in monotonic predicates and some deduction was possible when the value reached 30, both will make that deduction even though they saw distinct values and neither was actually looking at the counter precisely when 30 was reached. If events might occur millions of times per second, this style of reasoning enables a highly pipelined protocol design.

Fault tolerance. Crash faults introduce a number of non-trivial issues specific to our use of the SST in Derecho. We start by adopting a very basic approach motivated by monotonicity. When a failure is sensed by any process, it will:

1. Freeze its copy of the SST row associated with the failed group member (this breaks its RDMA connection to the failed node);
2. Update its own row to report the new suspicion (via the "Suspected" boolean fields seen in Figure 3.10);
3. Push its row to every other process (but excluding those its considers to have failed). This causes its peers to also suspect the reported failure(s).

Derecho currently uses hardware failure detections as its source of failure suspicions, although we also support a user-callable interface for reporting failures discovered by the software. In many applications the SST itself can be used to share heartbeat information by simply having a field that reports the current clock time and pushing the row a few times per second; if such a value stops advancing, whichever process first notices the problem can treat it as a fault detection.

Thus, if a node has crashed, the SST will quickly reach a state in which every non-failed process suspects the failed one, has frozen its SST row, and has pushed its own updated row to its peers. However, because SST's push is implemented by N separate RDMA writes, each of which could be disrupted by a failure, the SST replicas might not be identical. In particular, the frozen row corresponding to a failed node would differ if some SST push operations failed midway.

Were this the entire protocol, the SST would be at risk of logical partitioning. To prevent such outcomes, we shut down any process that suspects a majority of members of the Derecho top-level group (in effect, such a process deduces that it is a member of a minority partition). Thus, although the SST is capable (in principle) of continued operation in a minority partition, Derecho does not use that capability and will only make progress so long as no more than a minority of top-level group members are suspected of having failed.

Stable, Fault-Tolerant Monotonic Reasoning. A next question to consider is the interplay of failure handling with knowledge protocols. The aggressive epidemic-style propagation of failure suspicions transforms a suspected fault into monotonic knowledge that the suspected process is being excluded from the system: P's aggressive push ensures that P will never again interact with a member of the system that does not know of the failure, while Derecho's majority rule ensures that any minority partition will promptly shut itself down.

With this technique in use, the basic puzzle created by failure is an outcome in which process P discovers that $pred$ holds, but then crashes. The failure might freeze the SST rows of failed processes in such a way that no surviving process can deduce that $pred$ held at P, leaving uncertainty about whether or not P might have acted on $pred$ prior to crashing.

Fortunately, there is a way to eliminate this uncertainty: before acting on $pred$, P can share its discovery that $pred$ holds. In particular, suppose that when P discovers $pred$, it first reports this via its SST row, pushing its row to all other members *before* acting on the information. With this approach, there are two ways of learning that $pred$ holds: process Q can directly deduce that $pred$ has been achieved, but it could also learn $pred$ indirectly by noticing that P has done so. If P possessed knowledge no other process can deduce without information obtained from P, it thus becomes possible to learn that information either directly (as P itself did, via local deduction) or indirectly (by obtaining it from P, or from some process that obtained it from P). If we take care to ensure that information reaches a quorum, we can be certain that it will survive even if, after a crash, the property itself is no longer directly discoverable! With stable predicates, indirect discovery that a predicate holds is as safe as direct evaluation. By combining this behavior with monotonic predicates, the power of this form of indirection is even greater.

Notice also that when Derecho's majority rule is combined with this fault tolerant learning approach, P either pushes its row to a majority of processes in the epoch, then can act upon the knowledge it gained from $pred$, or P does not take action and instead crashes or throws a partitioning fault exception (while trying to do the push operation). Since any two majorities of the top-level group have at least one process in common, in any continued run of the system, at least one process would know of P's deduction that $pred$ holds. This will turn out to be a powerful tool in what follows.

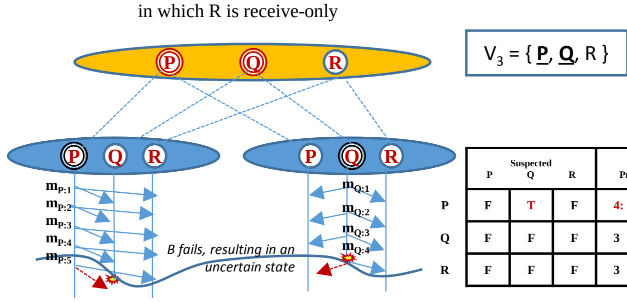
3.3.6 Derecho Protocols

To avoid burdening the reader with excessive detail, we limit ourselves to a brief overview. Derecho's protocols can be found in Appendix 3.7, which uses a pseudo-code notation to show the key steps in a high-level format, and Appendix 3.8, which walks through the full protocol suite.

Derecho's core structure can be seen in Figure 3.11. We map the top-level group to a set of subgroups, which may additionally be sharded. Here we see one subgroup. For each active sender, Derecho needs an SMC or RDMC session that will be used to stream multicasts reliably and in sender-order to the full group; in the figure, two such sessions are in use, one from sender P and one from sender Q. The group view is seen on the top right, and below it, the current SST for the group.

The columns in the SST are used by group members to share status. From the left, we see a vector of booleans denoting failure suspicions (in the example shown, Q has just failed, and P is aware of the event and hence "suspects" Q, shown in red). Eventually this

Derecho group with members {P, Q, R}
in which R is receive-only



Current view, showing senders

RDMC groups for senders P and Q. If Q fails, the group wedges along the ragged blue line, disrupting $\mathbf{m}_{P:5}$ and $\mathbf{m}_{Q:4}$.

	Suspected P	Suspected Q	Suspected R	Proposal	Acked	nCommit	nReceived P	nReceived Q	Final | Rank	Wedged
P	F	T	F	4: -Q	4	3	5	3	F | ⊥	T
Q	F	F	F	3	3	3	4	4	F | ⊥	F
R	F	F	F	3	3	3	5	4	F | ⊥	F

The leader (P)'s SST after sensing Q's failure. P is proposing that Q be removed. Once healthy members have wedged, P will use nReceived to compute and propose a final *trim* resolving the status of $\mathbf{m}_{P:5}$ and $\mathbf{m}_{Q:4}$.

Figure 3.11: Each Derecho group has one RDMC subgroup per sender (in this example, members P and Q) and an associated SST. In normal operation, the SST is used to detect multi-ordering, a property defined to also include data persistence. During membership changes, the SST is used to select a leader. It then uses the SST to decide which of the disrupted RDMC messages should be delivered and in what order; if the leader fails, the procedure repeats.

will trigger a new view in which Q will have been removed.

Next we see view-related SST columns, used by the top-level group leader to run a Paxos-based protocol that installs new views. We'll discuss this in moment.

To the right, we see a set of columns labelled "nReceived." These are counts of how many messages each group member has received from each sender. For example, in the state shown, R has received 5 multicasts from P, via RDMC. To minimize unnecessary delay, Derecho uses a simple round-robin delivery order: each active sender can provide one multicast per delivery cycle, and the messages are delivered in round-robin order. Derecho has built-in mechanisms to automatically send a null message on behalf of a slow sender, and will reconfigure to remove a process from the sender set if it remains sluggish for an extended period of time. Thus in the state shown, P and Q are both active, and messages are being delivered in order: P:1, Q:1, P:2, Q:2, etc.

Derecho delivers atomic multicasts when (1) all prior messages have been delivered, and (2) all receivers have reported receipt of a copy, which is determined as an aggregate over nReceived. Notice the monotonic character of this delivery rule: an example of receiver-side monotonic reasoning. For durable Paxos, Derecho delivers in two steps. As soon as a message can be properly ordered relative to prior messages, Derecho calls the appropriate update handler, which creates a new version of the associated data. This is then persisted to storage, as explained in Section 3.2.6, and then the persist action is reported via the same nReceived logic. Thus the same monotonic deduction used for atomic multicast delivery can now be used to commit the update.

In our example, messages can be delivered up to P:4, but then an issue arises. First, P is only acknowledging receipt of Q's messages through Q:3. Thus messages up to P:4 can be delivered, but subsequent ones are in a temporarily unstable state. Further, the failure is causing the group to *wedge*, meaning that P has noticed the failure and ceased to send or deliver new messages (wedged bit is true on the far right). Soon, R will do so as well. Q's row is ignored in this situation, since Q is suspected of having crashed. Once the group is fully wedged by non-faulty members, the lowest-ranked unsuspected process (P in this view) will propose a new view, but will also propose a final delivery "count" for messages, called a "ragged trim." P itself could fail while doing so, hence a new leader first waits until the old leader is suspected by every non-faulty group participant. Then it scans the SST. Within the SST, we include columns with which participants echo a proposed ragged trim, indicate the rank of the process that proposed it, and indicate whether they believe the ragged trim to have committed (become *final*). The sequence ensures that no race can occur: the scan of the SST will only see rows that have been fully updated.

This pattern results in a form of iteration: each successive leader will attempt to compute and finalize a ragged trim, iterating either if some new member failure is sensed, or the leader itself fails. This continues until either a majority is lost (in which case the minority partition shuts down), or eventually, some leader is not suspected by any correct member, and is able to propose a ragged trim, and a new view, that a majority of the prior members acknowledge. The protocol then commits. The ragged trim is used to finalize multicasts that were running in the prior view, and Derecho can move to the next view. The property just described is closely related to the weakest condition for progress in the Chandra / Toueg consensus protocol, and indeed the Derecho atomicity mechanism is very close to the atomicity mechanism used in that protocol [29].

One thing to keep in mind is that the SST shown is just one replica, specifically the copy held by P. Q and R have copies as well, and because SST push operations are performed as a series of one-sided RDMA writes, those other copies might not look identical. Rows do advance in monotonic order based on actions by the row owner, but at any instant in time, they can differ simply because one process has seen more updates than another. Our monotonic deductive rules are therefore designed to only take safe actions, even when participants evaluate them and take actions in an uncoordinated way. This is the underlying mechanism that results in our receiver-side batching.

In Appendix 3.7, the reader will see most elements of these steps expressed through pseudo-code in a style similar to the internal coding style used by Derecho for its SST protocols. Comments show the corresponding distributed knowledge statement, where relevant. Appendix 3.8 covers the same protocols but explains them in English and also offers correctness justifications. We should again emphasize that the protocols themselves are isomorphic to well known virtual synchrony membership protocols, and well-known variants of Paxos for use with virtually synchronous group membership, and have been proved correct in prior work.

3.3.7 Optimizations

The central innovation of Derecho is that we have expressed all our protocols as a composition of high-speed data-flow components that stream information in a lock-free manner designed to eliminate round-trip interactions or other sources of blocking. This offers the potential for very high performance, but to fully realize that opportunity we also needed to remove any unnecessary "speed bumps" on the primary data paths. The optimizations that follow describe some of the main delay sources we identified when tuning the system, and how we removed them. It is important to realize that all of these optimizations are secondary to that main protocol transformation: if Derecho's protocols including blocking steps, these optimizations would have little impact on system performance. But given that Derecho is highly asynchronous, the optimizations become quite important to overall system speed (they jointly gave us about a 1.5-2x speedup relative to the first versions of the system, which lacked them).

RDMA writes with no completions. In early work on the Derecho protocols, we tended to think of the SST as an optimized message-passing layer, and just as with many messaging systems, nodes would initiate I/O, and then check completion outcomes. We soon discovered that this led to a backlog of completion processing, resulting in delays in the critical path. Accordingly, we modified our protocol to eliminate waits for request completions. The current version of the SST operates without completions, but introduces a separate failure detection thread, which periodically writes a single byte to a remote node (on the same connection used by the SST), and waits for it to complete. If successful, the single completion is treated as a batch completion for all prior SST writes; if unsuccessful, this thread notifies the membership protocol, which terminates the epoch. The new scheme yielded significant performance gains.

SMC for small messages. SMC exposes tradeoffs of a different kind, also tied to the

RDMA hardware model. High speed devices, such as 100Gbps Ethernet or Infiniband, move data in side-by-side lanes that carry 64 bits of data plus some error-correction bits in each network-level cycle. Our Mellanox 100Gbps RDMA networks transmit 80 bytes per cycle (different devices would have different minimal transfer sizes). Additionally, the first network-level transfer for a particular object must carry IP or IB headers (26 bytes or more), as well as a NIC-to-NIC RDMA protocol header (28 bytes). Frames subsequent to the one containing the header will be filled with data. A single packet will thus be a series of frames containing headers, data and padding, limited in size by the MTU: 8KB for a "jumbo" network packet.

Thus a one-sided RDMA write containing 1 byte of data would travel with 54 bytes or more of overheads, padded to an 80-byte total size. The 100Gbps network will have been be reduced to an effective speed of just 1G! In contrast, suppose we were to send large objects on the same 100Gbps 10-lane hardware. 8KB rounds up to 8240 because of this 80-byte requirement, leaving room for 8KB of data as well as both headers.

In effect, the 100Gbps RDMA network runs at its full rated speed if we send large data objects, but is 80x slower for one-sided RDMA writes of single bytes. Clearly it is of interest to accumulate small objects into larger transfers.

We are currently experimenting with *opportunistic sender-side batching* for SMC. The basic idea is to never deliberately delay data, but shift the SST push used by SMC from the multicast path to occur in the same (single) thread used for SST predicate evaluations. The effect is that SMC would perform one RDMA write at a time, leaving time for a few updates to accumulate. A further step uses scatter-gather to send all the changed regions (the head and tail of the circular buffer, and the counter) as a single action.

No locks in the critical code. We've stressed that Derecho aims for a lock-free data acquisition pipeline. One way we achieved this was by designing protocols in which nodes independently deduce that messages are stable and deliverable, eliminating the need for consensus on the critical path. However, the issue of thread-level locking also arises. At the highest message rates, any delay in the core Derecho threads can result in a significant performance loss. Accordingly, we implemented our critical path to avoid use of locks in the critical path. The resulting logic is slightly more complex, but the performance benefit was dramatic.

Action batching. As noted earlier, by leveraging monotonicity, we can often design predicates that will cover multiple operations in each action. This allows Derecho to leverage receiver-side batching, enabling our multicast protocols to catch up even when a significant scheduling delay impacts some process. Within the system, batching is a widely used design pattern. For example, when messages are received, we can hand them off in batches to the higher-level protocol logic, not just one at a time. As noted earlier, when RDMA completions occur, Derecho can do a batch of cleanup actions. There are several additional such cases.

The benefit of a batched system architecture is that code paths that sense that an event or action is ready to trigger are traversed once, but then the associated action may fire multiple times, amortizing the code path cost over the actions.

3.4 Performance Evaluation

Our experiments seek to answer the following questions:

- How do the core state machine replication protocols perform on modern RDMA hardware, and on data-center TCP running over 100Gbps Ethernet? We measure a variety of metrics but for this section report primarily on bandwidth and latency.
- If an application becomes large and uses sharding heavily for scale, how will the aggregate performance scale with increasing numbers of members? Here we explore both Derecho's performance in large subgroups and its performance with large numbers of small shards (2 or 3 members, with or without overlap; for the overlap case, we created 2 subgroups over the same members). These experiments ran on a shared network with some congested TOR links, and in the overlapping shards case, the test itself generated overlapping Derecho subgroups. Thus any contention-triggered collapse would have been visible (we saw no problems at all, and this is also supported by other experiments, not shown here, in which we deliberately exposed large Derecho deployments to a variety of network and node stress).
- When using version-vector storage persisted to NVM (SSD), how fast is our logging solution, and how does it scale with increasing numbers of replicas?
- Looking at the end-to-end communication pipeline, how is time spent? We look at API costs (focusing here on polymorphic method handlers that require parameter marshalling and demarshalling; Derecho also supports unmarshalled data types, but

of course those have no meaningful API costs at all), delivery delay, stabilization delay in the case of persisted (durable Paxos) version-vector updates, and finally delay until temporal stability occurs.

- Latency for updates versus latency for queries, again in the case of versioned data. Here the queries exercise our temporally-consistent snapshot isolation functionality, accessing data spread over multiple shards.
- How does the performance of the system degrade if some members are slow?
- How long does the system require to reconfigure an active group?
- How does Derecho compare with APUS, LibPaxos and ZooKeeper?

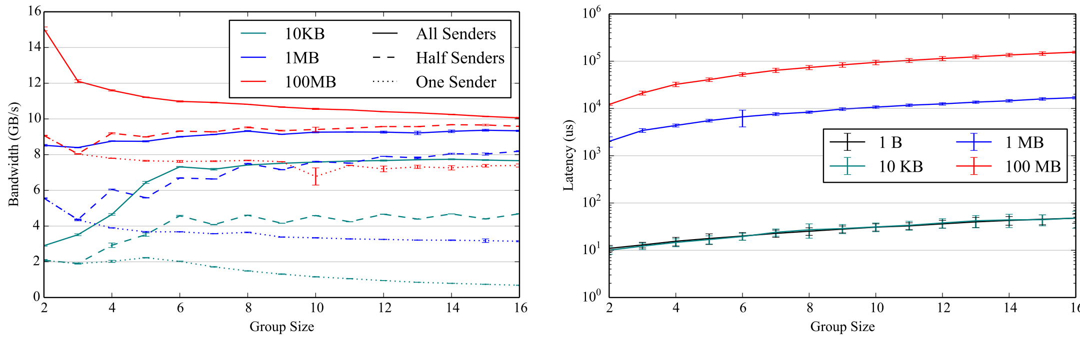
In what follows, the small-scale experiments were performed on our local cluster, Fractus. For larger experiments, we used Stampede 1, a supercomputing cluster in Texas. Fractus consists of 16 machines running Ubuntu 16.04 connected with a 100Gbps (12.5 GB/s) RDMA InfiniBand switch (Mellanox SB7700). The machines are equipped with Mellanox MCX456AECAT Connect X-4 VPI dual port NICs. Fractus is also equipped with a 100Gbps RoCE Ethernet switch, but we did not repeat our full set of IB experiments on RoCE (we did test a few cases, and obtained identical results). Stampede contains 6400 Dell Zeus C8220z compute nodes with 56G (8 GB/s) FDR Mellanox NIC, housed in 160 racks (40 nodes/rack). The interconnect is an FDR InfiniBand network of Mellanox switches, with a fat tree topology of eight core-switches and over 320 leaf switches (2 per rack) with a 5/4 bandwidth oversubscription. Nodes on Stampede are batch scheduled with no control over node placement. Node setup for our experiments consists of about 4 nodes per rack. Although network speeds are typically measured in bits per second, our bandwidth graphs use units of GB/s simply because one typically thinks of objects such as web pages, photos and video streams in terms of bytes.

3.4.1 Core protocol performance

Figure 3.12 measures Derecho performance on 2 to 16 nodes on Fractus. The experiment constructs a single subgroup containing all nodes. Each of the sender nodes sends a fixed number of messages (of a given message size) and time is measured from the start of sending to the delivery of the last message. Bandwidth is then the aggregated rate of sending of the sender nodes. We plot the throughput for totally ordered (atomic multicast) mode.

We see that Derecho performs close to network speeds for large message sizes of 1 and 100 MB, with a peak rate of 16 GB/s. In unreliable mode, Derecho's protocol for sending small messages, SMC, ensures that we get high performance (close to 8 GB/s) for the 10 KB message size; we lose about half the peak rate in our totally-ordered atomic multicast. As expected, increasing the number of senders leads to a better utilization of the network, resulting in better bandwidth. For the large message sizes, the time to send the message dominates the time to coordinate between the nodes for delivery, and thus unreliable mode and totally ordered (atomic multicast) mode achieve similar performance. For small message sizes (10 KB), those two times are comparable. Here, unreliable mode has a slight advantage because it does not perform global stability detection prior to message delivery.

Not shown is the delivery batch size; at peak rates, multicasts are delivered in small



(a) Atomic multicast throughput (RDMA) (b) Atomic multicast latency (RDMA)

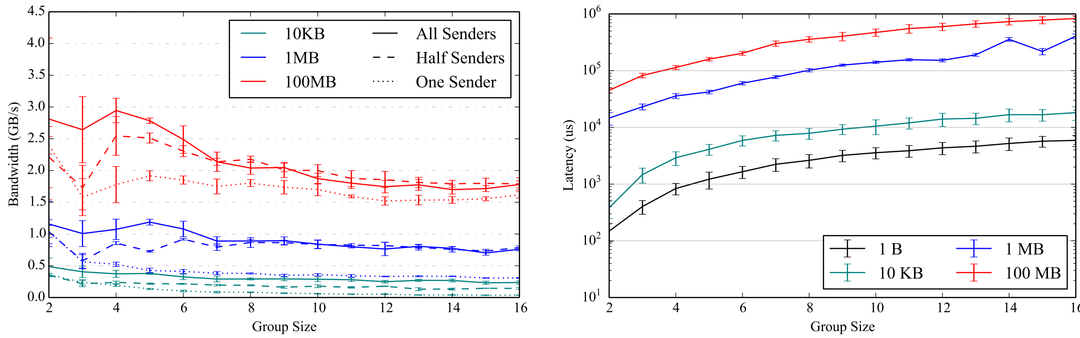
Figure 3.12: Derecho's RDMA performance with 100Gbps InfiniBand

batches, usually the same size as the number of active senders, although now and then a slightly smaller or larger batch arises. Since we use a round-robin delivery order, the delay until the *last* sender's message arrives will gate delivery, as we will show momentarily, when explaining Figure 3.20b.

Notice that when running with 2 nodes at the 100MB message size, Derecho's peak performance exceeds 12.5 GB/s. This is because the network is bidirectional, and in theory could support a data rate of 25GB/s with full concurrent loads. With our servers, the NIC cannot reach this full speed because of limited bandwidth to the host memory units.

3.4.2 Large subgroups, with or without sharding

Earlier, we noted that while most communication is expected to occur in small groups, there will surely also be some communication in larger ones. To explore this case, we ran the same experiment on up to 128 nodes on Stampede. The resulting graph, shown in



(a) Atomic multicast throughput (TCP)

(b) Atomic multicast latency (TCP)

Figure 3.13: Derecho performance using TCP with 100Gbps Ethernet

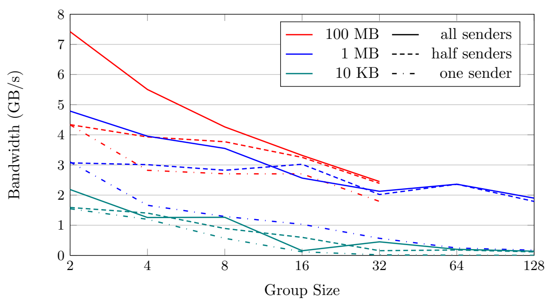


Figure 3.14: Totally ordered (atomic multicast) mode

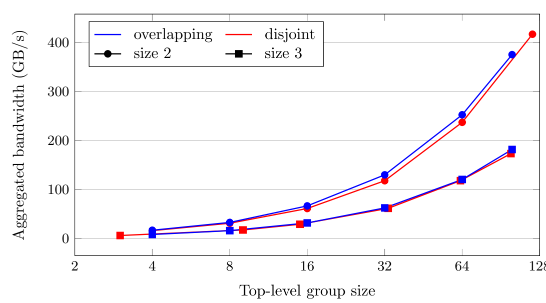


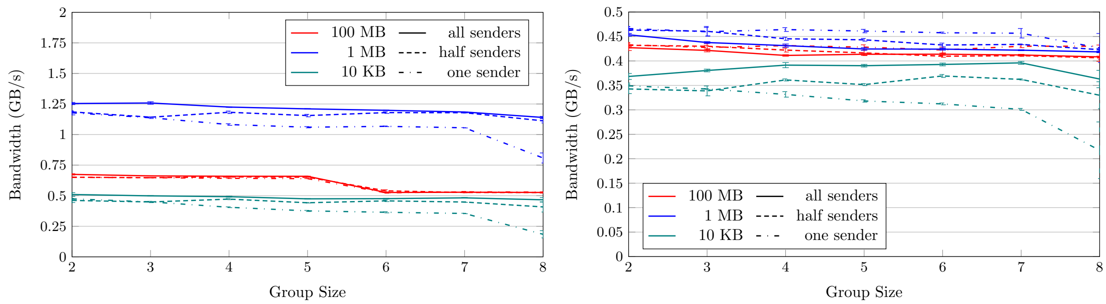
Figure 3.15: Derecho performance for sharded groups using RDMC with 100MB messages.

Figure 3.14, shows that Derecho scales well. For example we obtain performance of about 5GB/s for 1MB-all-senders on 2 nodes, 2.5 GB/s on 32 nodes, and 2 GB/s on 128 nodes: a slowdown of less than 3x. Limitations on experiment duration and memory prevented us from carrying out the same experiment for the 100 MB case on 64 and 128 nodes. This also explains the absence of error bars: each data point shown corresponds to a single run of the experiment. Note, however, that the performance obtained is similar to that seen on Fractus for small groups.

Next, we evaluate performance in an experiment with a large sharded group. Here, the interesting case involves multiple (typically small) subgroups sending messages simultaneously, as might arise in a sharded application or a staged computational pipeline. We formed two patterns of subgroups of fixed size: disjoint and overlapping. For a given set of n nodes, assume unique node ids from 0 to $n-1$. Disjoint subgroups partition the nodes into subgroups of the given size. Thus, disjoint subgroups of size s consist of n/s subgroups where the i^{th} subgroup is composed of nodes with ids $s*i, s*i+1, \ldots, s*(i+1)-1$. Overlapping subgroups of size s, on the other hand, place every node in multiple (s) subgroups. They consist of n subgroups where the the i^{th} subgroup is composed of nodes $i, i+1, ..., i+s-1$ (wrapping when needed).

We tested with overlapping and disjoint subgroups of sizes 2 and 3. All nodes send a fixed number of messages of a given message size in each of the subgroups they belong in. The bandwidth is calculated as the sum of the sending rate of each node. Figure 3.15 shows that for large messages (100 MB), the aggregated performance increases linearly with the number of nodes for all subgroup types and sizes. This is as expected; the subgroup size is constant, and each node has a constant rate of sending, leading to a linear increase in aggregated performance with the number of nodes.

We do not include data for small messages (10 KB) because this particular case triggered a hardware problem: the "slow receiver" issue that others have noted in the most current Mellanox hardware [51]. In essence, if a Mellanox NIC is used for one-sided I/O operations involving a large address region, RDMA read and write times increase as a function of the actual memory address being accessed in a remote node. Derecho encoun-



(a) Volatile version-vectors (Linux RamDisk). (b) Persistent version-vectors (SSD).

Figure 3.16: Derecho performance with marshalled method invocations, and handlers that use version-vector storage.

tered this with small messages: in this case, the system routes data through SMC, creating one SMC session per subgroup or shard. Thus, as the number of subgroups increase, the amount of space allocated for the SST row increases proportionally. We end up with very long SST rows, triggering this Mellanox issue. Although Mellanox is planning to fix the slow receiver problem, we are currently implementing a work-around, whereby disjoint SMC subgroups will share SST columns. This should significantly reduce the size of our SST rows.

3.4.3 RPC with versioned storage, with and without NVM logging

Our next experiments focus on the costs of versioned storage. Whereas the data shown above was for applications that invoke a null message handler, figures 3.16a and 3.16b look at a full code path that includes marshalling on the sender side (required only if the data being sent is not suitable for direct RDMA transmission), demarshalling for delivery, and then storage into volatile or persistent version-vectors, respectively. The latter incur a further marshalling cost, followed by a DMA write, while the former entails allocating memory and copying (this occurs within the Linux "ramdisk" subsystem). Not surprisingly, Derecho's multicast is far faster than the copying and DMA write costs, hence we see flat scaling, limited by the bandwidth of the relevant data path. For the SSD case, our experiment pegs our SSD storage devices at 100% of their rated performance.

We were initially surprised by the relatively low bandwidth for volatile storage of large objects: we had expected the 100MB speed to be higher than the 1MB numbers, yet as seen in Figure 3.16a, they are tied. We tracked the phenomenon to slower performance of the `memcpy` primitive when the source object is not cached. Whereas `memcpy` achieves single-core bandwidth of 8GB/s on our machines if the source object is in the L2 cache, bandwidth drops to just 3.75GB/s for objects too large to cache. In our experiment, this slowdown effect is amplified because several copying operations occur.

In the best case, with neither volatile storage nor NVM logging, Derecho's RPC mechanisms incur a single copy to send RPC arguments and no copy on receipt. Derecho's RPC framework has no statistically significant overheads beyond the cost of this copy. The initial copy is necessary to move RPC arguments into Derecho's allocated sending buffers; we plan to eliminate even this copy operation by permitting RDMA from the entirety of the Derecho process memory, instead of a dedicated set of buffers. Derecho supports user-defined serialization and arbitrary datastructures, but tries whenever possible to use the same object representation both in memory and on the wire. User-defined serialization and serialization of certain datatypes (like STL containers) may incur additional copies (and thus additional overhead).

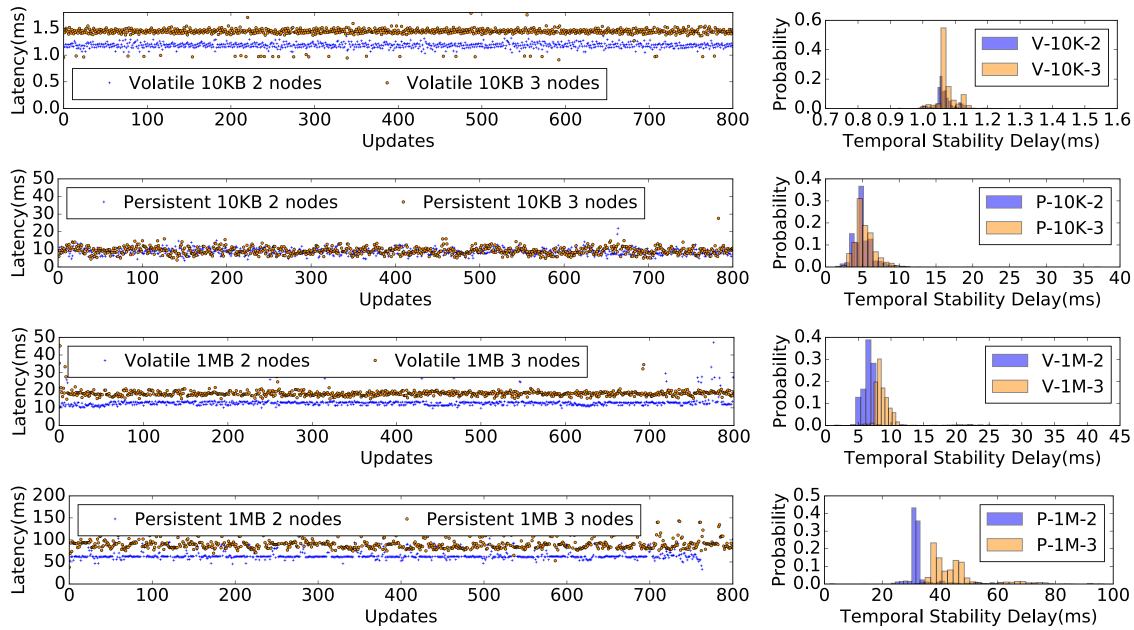


Figure 3.17: End to end performance. On the left, time from when an update was initiated until the new version is temporally stable. On the right, minimum delay for a query to access data at time "now".

3.4.4 End-to-End performance

In Figure 3.17, we see Derecho's end-to-end performance, measured in two ways. The graphs on the left are for an experiment that initiates all-to-all multicasts in a subgroup of 2 nodes (blue) or 3 nodes (orange), which store the received objects into version vectors, showing 10KB objects (top two graphs) and 1MB objects (bottom two). We measured time from when the update was initiated until the new version is temporally stable and the `ordered_send` has completed.

To obtain the histograms on the right, we launched an additional thread within the same

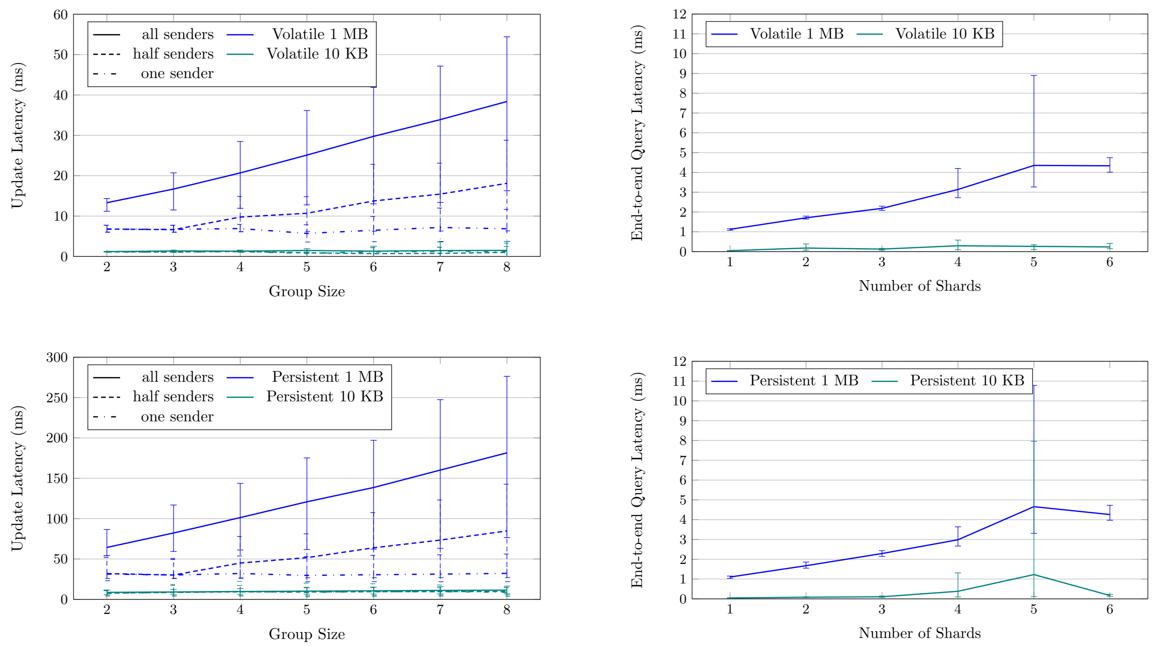


Figure 3.18: Left: Latency as a function of group size for updates. Right: Latency for a temporal query issued to a single member each, for varying numbers of shards.

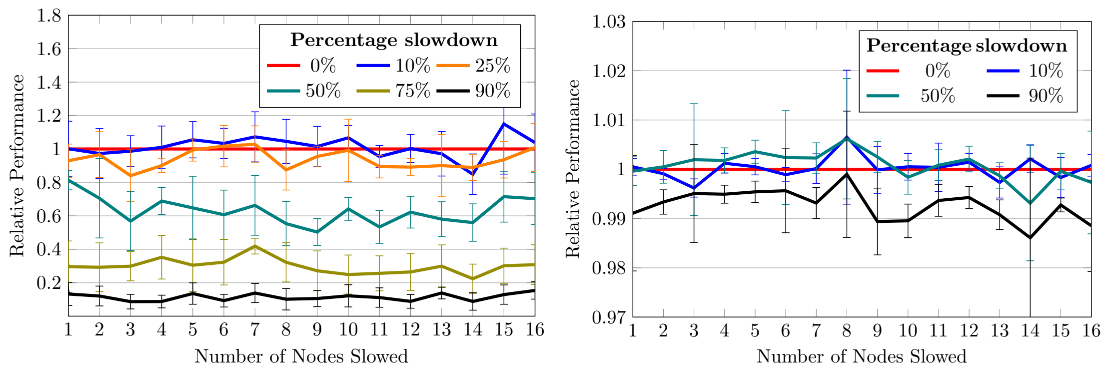
processes; it loops. During each iteration, our test samples the local clock, then indexes into the local instance of the versioned object at the corresponding time, measuring the delay until the data is available. This yields a pure measurement of delay on the query path used to retrieve information from a temporal snapshot (recall that in Derecho, updates and temporal queries use different code paths, as illustrated in Figure 3.6 and discussed in Section 3.2).

In Figure 3.18, we repeat this experiment, but now vary the size of the group, looking at the trends. Again, the update path is shown on the left, exploring various patterns: all-to-all sending (solid lines), half-to-all (dashed) and one-to-all (dotted), and object sizes 10KB and 1MB. On the right we looked at how Derecho's temporal query delays grow when increasing numbers of shards are accessed in the query. For this experiment, we used 13 Fractus nodes. 12 nodes are structured into a set of 2-member shards, holding data in version-vectors. We pre-populated the vectors with objects of size 10KB or 1MB. The 13^{th} node queries a single member each within 1 to 6 shards, using Derecho's `p2p_query` API, which issues the P2P requests concurrently, then collects replies in parallel. The temporal indices for these requests were picked to be far enough in the past so that the data would be temporally stable, yet not so old that data would no longer be cached in memory. We graph the time from when the request was made until the full set of replies is available.

High variance is to be expected for the update path with multiple senders: some multicasts will inevitably need to wait until their turn in the round-robin delivery schedule. However, the spike in variance for 5-shard queries is an oddity. We traced this to Linux, and believe it is either caused by an infrequent scheduling anomaly, or by a pause in the TCP stack (for example to wait for kernel netbufs to become available). We are reimplementing `p2p_send` and `p2p_query` to use unicast RDMA, which should improve the absolute numbers and eliminate this variability.

3.4.5 Resilience to contention

The experiments shown above were all performed on lightly loaded machines. Our next investigation explores the robustness of control plane/data plane separation and batching techniques for Derecho in a situation where there might be other activity on the same



(a) 10 KB

(b) 100 MB

Figure 3.19: Derecho performance for various values for efficiency and number of slow nodes, as a fraction of the no-slowdown case.

nodes. Recall that traditional Paxos protocols degrade significantly if some nodes run slowly [69]. The issue is potentially a concern, because in multitenancy settings or on busy servers, one would expect scheduling delays. In Derecho, we hoped to avoid such a phenomenon.

Accordingly, we designed an experiment in which we deliberately slowed Derecho's control infrastructure. We modified the SST predicate thread by introducing artificial busy-waits after every predicate evaluation cycle. In what follows, we will say that a node is working at an efficiency of $X\%$ (or a slowdown of $(100 - X)\%$), if it is executing X predicate evaluation cycles for every 100 predicate evaluation cycles in the normal no-slowdown case. The actual slowdown involved adding an extra predicate that measures the time between its successive executions and busy-waits for the adjusted period of time to achieve a desired efficiency. In contrast, we did not slow down the data plane: RDMA hardware performance would not be impacted by end-host CPU activity or scheduling. Further, our experiment sends messages without delay.

In many settings, only a subset of nodes are slow at any given time. To mimic this in our experiment, for a given efficiency of $X\%$, we run some nodes at $X\%$ and others at full speed. This enables us to vary the number of slow nodes from 0 all the way to all the nodes, and simplifies comparison between the degraded performance and the no-slowdown case. The resulting graph is plotted in Figure 3.19.

The first thing to notice in this graph is that even with significant numbers of slow nodes, performance for the large messages is minimally impacted. This is because with large messages, the RDMA transfer times are so high that very few control operations are needed, and because the control events are widely spaced, there are many opportunities for a slowed control plane to catch up with the non-slowed nodes. Thus, for 90% slowdown, the performance is more than 98.5% of the maximum while for 99% slowdown (not shown in the graph), it is about 90%.

For small messages (sent using SMC), the decrease in performance is more significant. Here, when even a single node lags, its delay causes all nodes to quickly reach the end of the sending window and then to wait for previous messages to be delivered. Nonetheless, due to the effectiveness of batching, the decrease in performance is less than proportional to the slowdown. For example, the performance is 70% of the maximum in case of 50% slowdown and about 15% for a 90% slowdown.

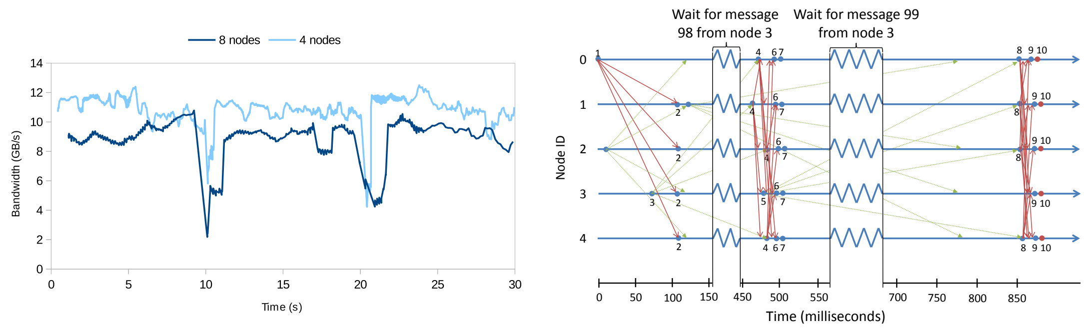
Notice also that performance does not decrease even as we increase the number of slow nodes. In effect, the slowest node determines the performance of the system. One can understand this behavior by thinking about the symmetry of the Derecho protocols, in which all nodes independently deduce global stability. Because this rule does not depend on a single leader, all nodes proceed independently towards delivering sequence of messages. Further, because Derecho's batching occurs on the receivers, not the sender, a slow node simply delivers a larger batch of messages at a time. Thus, whether we have one slow node or all slow ones, the performance impact is the same.

From this set of experiments, we conclude that Derecho performs well with varying numbers of shards (with just minor exceptions caused by hardware limitations), that scheduling or similar delays are handled well, and that the significant performance degradation seen when classic Paxos protocols are scaled up are avoided by Derecho's novel asynchronous structure.

Next, we considered the costs associated with membership changes.

3.4.6 Costs of membership reconfiguration

In Figure 3.20a we see the bandwidth of Derecho multicasts in an active group as a join or leave occurs. The three accompanying figures break down the actual sequence of events that occurs in such cases, based on detailed logs of Derecho's execution. Figure 3.20b traces a single multicast in an active group with multiple senders. All red arrows except the first set represent some process putting information into its SST row (arrow source) that some other process reads (arrow head); the first group of red arrows, and the background green arrows, represent RDMC multicasts. At ① process 0 sends a 200MB message: message (0,100). RDMC delivers it about 100ms later at ②, however, Derecho must buffer it until it is multi-ordered. Then process 3's message (3,98) arrives (③-④) and



(a) Derecho multicast bandwidth with 200MB messages. A new member joins at 10s, then leaves at 20s.

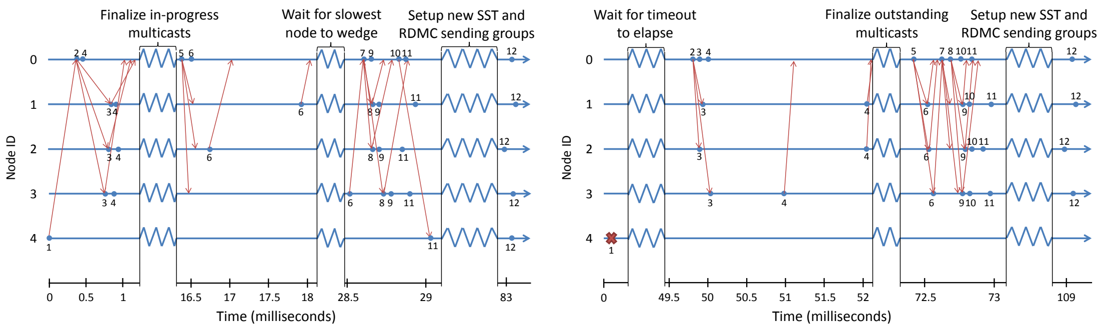
(b) Tracking the events during the multicast of a 200MB message in a heavily-loaded 5-member group.

Figure 3.20: Multicast bandwidth (left), and a detailed event timeline (right).

the SST is updated (4, 6), which enables delivery of a batch of messages at 7. These happen to be messages (2,98)...(1,99). At 8 process 3's message (3,99) arrives, causing an SST update that allows message 100 from process 0 to finally be delivered (9-10) as part of a small batch that covers (2,99) to (1,100). Note that this happens to illustrate the small degree of delivery batching predicted earlier.

In Figure 3.21a we see a process joining: 1 it requests to join, 2-6 are the steps whereby the leader proposes the join, members complete pending multicasts, and finally wedge. In steps 7-9 the leader computes and shares the trim; all processes trim the ragged edge at 9 and the leader sends the client the initial view (10). At 11 we can create the new RDMC and SST sessions, and the new view becomes active at 12. Figure 3.21b shows handling of a crash; numbering is similar except that here, 4 is the point at which each member wedges.

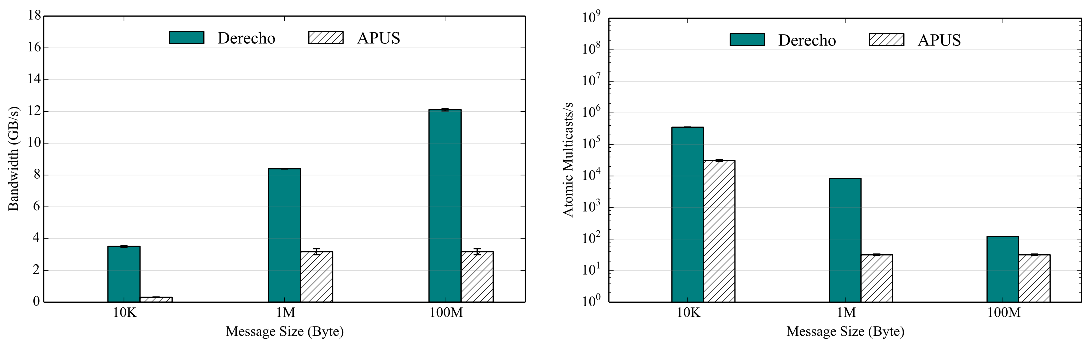
For small groups sending large messages, the performance-limiting factor involves



(a) Timeline for joining an active group.

(b) Crash triggers exclusion from an active group.

Figure 3.21: Timeline diagrams for Derecho.

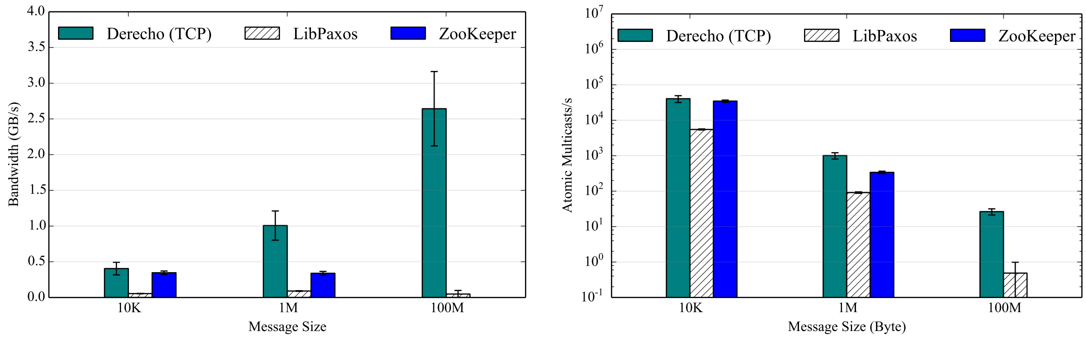


(a) Throughput

(b) Multicast Rate

Figure 3.22: Derecho vs APUS with Three Nodes over 100Gbps InfiniBand

terminating pending multicasts and setting up the new SST and RDMC sessions, which cannot occur until the new view is determined. This also explains why in Figure 3.20a the disruptive impact of a join or leave grows as a function of the number of active senders: the number of active sends and hence the number of bytes of data in flight depends on the number of active senders. Since we are running at the peak data rate RDMA can sustain, the time to terminate these sends is dominated by the amount of data in flight. In experiments with 20MB messages the join and leave disruptions are both much shorter.



(a) Throughput (b) Multicast Rate

Figure 3.23: Derecho vs LibPaxos and ZooKeeper with Three Nodes over 100Gbps Ethernet

3.4.7 Comparisons with Other Systems

Using the same cluster on which we evaluated Derecho, we conducted a series of experiments using competing systems: APUS, LibPaxos, ZooKeeper. All three systems were configured to run in their atomic multicast (in-memory) modes. APUS runs on RDMA, hence we configured Derecho to use RDMA for that experiment. LibPaxos and ZooKeeper run purely on TCP/IP, so for those runs, Derecho was configured to map to TCP.

The comparison with APUS can be seen in Figure 3.22. We focused on a 3-member group, but experimented at other sizes as well; within the range considered (3, 5, 7) APUS performance was constant. APUS does not support significantly larger configurations. As seen in these figures, Derecho is faster than APUS across the full range of cases considered. APUS apparently is based on RAFT, which employs the pattern of two-phase commit discussed earlier, and we believe this explains the performance difference.

Comparison of Derecho atomic multicast with the non-durable configurations of LibPaxos and ZooKeeper are seen in Figure 3.23 (Zookeeper does not support 100MB writes, hence that data point is omitted). Again, we use a 3-member group, but saw similar results at other group sizes. LibPaxos employs the Ring Paxos protocol, which is similar to Derecho's protocol but leader-based. Here the underlying knowledge exchange is equivalent to a two-phase commit, but the actual pattern of message passing involves sequential token passing on a ring.

The comparison with ZooKeeper is of interest, because here there is one case (10KB writes) where ZooKeeper and Derecho have very similar performance over TCP. Derecho dominates for larger writes, and of course would be substantially faster over RDMA (refer back to Figure 3.13b). On TCP, the issue is that Derecho is somewhat slow for small writes, hence what we are seeing is not so much that ZooKeeper is exceptionally fast, but rather that the underlying communications technology is not performing terribly well, and both systems are bottlenecked.

More broadly, Derecho's "sweet spot," for which we see its very highest performance, involves large objects, large replication factors, and RDMA hardware. The existing systems, including APUS, simply do not run in comparable configurations and with similar object sizes.

As a final remark, we should note that were we to compare Derecho's peak RDMA rates for large objects in large groups with the best options for implementing such patterns in the prior systems (for example, by breaking a large object into smaller chunks so that ZooKeeper could handle them), Derecho would be faster by factors of 30x or more. We omit such cases because they raise apples-to-oranges concerns, despite the fact that modern replication scenarios often involve replication of large objects both for fault-tolerance (small numbers of replicas suffice) and for parallel processing (here, large numbers may be needed).

3.4.8 Additional Experimental Findings

Above, we described just a small fraction of the experiments we've run. We omit a series of experiments that explored very large configurations of as many as 500 group members, because the machine on which we carried those out had a different architecture than the platforms on which we report. We also omitted experiments on RoCE, focusing instead on datacenters equipped with RDMA on IB, because datacenter use of RoCE is too much of a moving target for us to identify a consensus deployment model. We studied Derecho performance with contention, but on yet a different platform. The complication for those experiments was that the background loads were imposed by a scheduler not under our control, and as a result the exact nature of the contention was hard to quantify.

We also studied API costs. Say that a multicast that sends just a byte array is "uncooked," while a multicast that sends a polymorphic argument list is "cooked." We measured the cost of the cooked RPC framework on simple objects without persistence, but discovered it added no statistically significant difference in round-trip time compared to the uncooked case (discounting the initial copy of the RPC arguments into Derecho's sending buffers, a step which can sometimes be avoided for an uncooked send when the data is already in memory, for example after a DMA read from a camera). The primary finding in this work was that copying of any kind can lead to significant delays. Thus applications seeking to get the best possible performance from Derecho should use "zero copy" programming techniques as much as feasible. Within our API, we avoid copying arguments if the underlying data format is suitable for direct transmission, and on receipt, we don't create copies of incoming objects so long as the event handler declares its argument to be of C++ "reference type." For example, in C++ an argument of type `int` must be copied to the stack, but an argument of type `int&` is treated as a reference to the parameter object (an alias). For even better performance, handlers should declare arguments to be `const`, for example `const int&`. A const argument is one that the user's code will not mutate; knowing this, C++ can perform a number of compiler optimizations, reducing costs on Derecho's critical path.

None of those experiments uncovered any signs of trouble: if anything, they support our belief that Derecho can run at extremely large scale, on a wide variety of platforms, and in multitenant environments.

3.5 Prior work

Paxos. While we did not create Derecho as a direct competitor with existing Paxos protocols, it is reasonable to compare our solution with others. As noted in our experimental section, we substantially outperform solutions that run on TCP/IP and are faster than or "tied with" solutions that run on RDMA. Existing RDMA Paxos protocols lack the asynchronous, receiver-batched aspects of our solution. As a result, Derecho exhibits better scalability without exhibiting the form of bursty behavior observed by Jallali [69].

With respect to guarantees offered, the prior work on Paxos is obviously relevant to our paper. The most widely cited Paxos paper is the classic Synod protocol [64], but the version closest to ours is the virtually synchronous Paxos described by Birman, Malkhi, and van Renesse [21]. A number of papers how suggested ways to derive the classic Paxos protocol with the goal of simplifying understanding of its structure [28, 35, 67, 70, 83, 95]

Among software libraries that offer high speed Paxos, APUS [99] has the best performance. APUS implements a state-machine replication protocol related to RAFT, and developed by Mazieres [70]. APUS is accessed through a socket extension API, and can replicate any deterministic application that interacts with its external environment through the socket. A group of n members would thus have 1 leader that can initiate updates, and $n - 1$ passive replicas that track the leader.

Corfu [9] offers a persistent log, using Paxos to manage the end-of-log pointer,[7] and chain-replication as a data-replication protocol [96]. In the Corfu model, a single log is shared by many applications. An application-layer library interacts with the Corfu service to obtain a slot in the log, then replicates data into that slot. Corfu layers a variety of higher level functionalities over the resulting abstraction. Our version-vectors are like Corfu logs in some ways, but a single application might have many version vectors, and each vector holds a single kind of replicated data. Our RDMC protocol is more efficient than the Corfu

[7] Corfu has evolved over time. Early versions offered a single log and leveraged RDMA [9], but the current open-source platform, vCorfu [100] materializes individualized "views" of the log and has a specialized transactional commit mechanism for applications that perform sets of writes atomically.

replication protocol at large scale, but Corfu is not normally used to make large numbers of replicas. More interesting is the observation that with one original copy and two replicas, we outperform the chain replication scheme used in Corfu: RDMC never sends the same data twice over any RDMA link. With two replicas, the leader (P) sends half the blocks to replica Q, which then forwards them to R, and the other half to S, which forwards them to R, and the resulting transfer completes simultaneously for Q and R almost immediately (one block-transfer) after the leader has finished sending: potentially, twice as fast as any protocol that either uses a chain of participants, or where P sends separately to both Q and to R.

Round-robin delivery ordering for atomic multicast or Paxos dates to early replication protocols [31]. Ring Paxos is implemented in LibPaxos [2], and Quema has proven a different ring Paxos protocol optimal with respect to its use of unicast datagrams [50], but Derecho substantially outperforms both. The key innovation is that by re-expressing Paxos using asynchronously-evaluated predicates, we can send all data out-of-band. Section 3.4.7 compares performance of LibPaxos with Derecho.

RAFT [77] is a popular modern Paxos-like protocol; it was created as a replication solution for RamCloud [79] and was the basis of the protocol used in APUS. Microsoft's Azure storage fabric uses a version of Paxos [27], but does not offer a library API. NetPaxos is a new Paxos protocol that leverages features of SDN networks. It achieves very high performance, but just for a single group, and lacks support for complex, structured applications [36]. DARE [81] looks at state machine replication on an RDMA network. RDMA-Paxos is an open-source Paxos implementation running on RDMA [5].

NOPaxos [68] is an interesting new Paxos protocol that uses the SDN network switch to order concurrent multicasts, but it does not exploit RDMA. None of these libraries can support complex structures with subgroups and shards, durable replicated storage for versioned data, or consistent time-indexed queries. They all perform well, but Derecho still equals or exceeds all published performance measurements.

Atomic multicast. The virtual synchrony model was introduced in the Isis Toolkit in 1985 [20], and its gbcast protocol is similar to Paxos [7]. Modern virtual synchrony multicast systems include JGroups [10] and Vsync [6], but none of these maps communication to RDMA, and all are far slower than Derecho. At the slow network rates common in the past, a major focus was to batch multiple messages into each send [44]. With RDMA, the better form of batching is on the receiver side.

Monotonicity. We are not the first to have exploited asynchronous styles of computing, or to have observed that monotonicity can simplify this form or protocol, although Derecho's use of that insight to optimize atomic multicast and Paxos seems to be a new contribution. Particularly relevant prior work in this area includes Hellerstein's work on eventual consistency protocols implemented with the Bloom system [33]. The core result is the *CALM* theorem, which establishes that logically monotonic programs are guaranteed to be eventually consistent. The authors shows that any protocol that does not require distributed synchronization has an asynchronous, monotonic implementation (and conversely, that distributed synchronization requires blocking for message exchange). This accords well with our experience coding Derecho, where the normal mode of the system is asynchronous and monotonic, but epoch (view) changes require blocking for consensus, during the protocol Derecho uses to compute the ragged trim.

DHTs. Transactional key-value stores have become widely popular in support of both the NoSQL and SQL database models. Derecho encourages key-value sharding for scalability, and offers strong consistency for read-only queries that span multiple subgroups or shards. However, at present Derecho lacks much of the functionality found in full-fledged DHT solutions, or DHT-based databases such as FaRM [38], HERD [58] and Pilaf [72]. Only some of these DHTs support transactions. FaRM offers a key-value API, but one in which multiple fields can be updated atomically; if a reader glimpses data while an update is underway, it reissues the request. DrTM [101] is similar in design, using RDMA to build a transactional server. Our feeling is that transactions can and should be layered over Derecho, but that the atomicity properties of the core system will be adequate for many purposes and that a full transactional infrastructure brings overheads that some applications would not wish to incur.

3.6 Conclusions

Derecho is a new software library for creating structured services of the kind found in today's cloud-edge. The system offers a simple but powerful API focused on application structure: application instances running identical code are automatically mapped to subgroups and shards, possibly overlapping, in accordance with developer-provided guidence. Consistency is achieved using the virtual synchrony model of dynamic membership management side by side with a new suite of Paxos-based protocols that offer atomic multicast and durable state machine replication. Unusually, Derecho can perform updates and read-only queries on disjoint lock-free data paths, employing either TCP (if RDMA is not available) or RDMA for all data movement. Derecho is 100x faster than comparable packages when running in identical TCP-based configurations, and a further 4x faster when RDMA hardware is available. The key to this performance resides in a design that builds the whole system using steady data flows with minimal locking or delays for round-trip interactions.

Appendix

3.7 Pseudo-Code for Key Protocol Steps

3.7.1 Notation

SST

```
column_name −> string | string[int] // e.g. wedged or latest_received_index[3]
sst_row −> sst[row_rank]
row_rank −> int
sst_column −> sst[∗].column_name
```

```
sst_entry −> sst_row.column_name // e.g. sst[0].stable_msg_index[0]
```

A reducer function, for example, Min(sst_column) represents the minimum of all the entries of the column. Count(sst_column, value) counts the number of entries that are equal to value. MinNotFailed(sst_column) is a Min(NotFailed(sst_column)) where NotFailed is just a filtering function that removes the rows that are suspected, from the column.

A rank of a member is the index of its row in the SST. The code shown below is run by every process, but each has a distinct rank (referred to as my_rank).

Message Ordering

The group is represented by G. The failed node set is denoted by F, $F \subseteq G$.

Message : $M(i, k)$ represents a message with i as the sender rank and k as the sender index. For example, the zeroth message by sender number 2 is $M(2, 0)$. We have a round robin ordering imposed on messages. $M(i_1, k_1) < M(i_2, k_2) \iff k_1 < k_2 || (k_1 == k_2 \wedge i_1 < i_2)$.

The global index of $M(i, k)$, $gi(M(i, k))$ is the position of this message in the round-robin ordering. So $M(0, 0)$ has a global index of 0, $M(1, 0)$ has a global index of 1 and so on.

It is easy to see that,

$$gi(M(i,k)) = i + |G| * k$$

Conversely, if $M(i,k) = g$, then $i = g \ mod \ |G|, k = g/|G|$.

Knowledge

$\mathcal{P}$: a predicate.

$K_{me}(\mathcal{P})$: This process ("me") knows that $\mathcal{P}$ is true.

$K_S(\mathcal{P})$: Every process in set S knows that $\mathcal{P}$ is true.

$K^1(\mathcal{P})$: Every process knows that $\mathcal{P}$ is true i.e. $K_G(\mathcal{P})$.

$K^2(\mathcal{P})$: Every process knows that every process knows that $\mathcal{P}$ is true i.e. $K_G(K_G(\mathcal{P}))$.

$\Diamond\mathcal{P}$: Eventually, either $\mathcal{P}$ holds, or a failure occurs and the epoch termination protocol runs.

Note that all of these are completely standard with the exception of $\Diamond\mathcal{P}$: in prior work on knowledge logics, there was no notion of an epoch termination and new-view protocol. This leads to an interesting line of speculation: should epoch-termination be modelled as a "first order" behavior, or treated as a "higher order" construct? Do the Derecho membership views function as a form of common knowledge, shared among processes to which the view was reported? We leave such questions for future study.

3.7.2 SMC

In what follows, we begin by presenting the SST multicast (SMC), which implements a ring-buffer multicast. In combination with the atomic multicast delivery logic and the membership management protocol that follows, we obtain a full Paxos. RDMC could be similarly formalized, but is omitted for brevity.

SST Structure

SMC uses two fields, slots and received_num. slots is a vector of window_size slots, each of which can store a message of up to max_message_size characters. The index associated with a slot is used to signal that a new message is present in that slot: For example, if a slot's index had value k and transitions to $k+1$, a new message is available to be received. The vector received_num holds counters of the number of messages received from each node.

Initialization

```
for i in 1 to n {
  for j in 1 to n {
    sst[i].received_num[j] = −1;
  }
```

```
    for k in 1 to window_size {
        sst[i].slots[k].buf = nullptr;
        sst[i].slots[k].index = 0;
    }
}

// helper variable
sent_num = −1;
```

Sending

First the sending node reserves one of the slots:

```
char* get_buffer(msg_size) {
    assert(msg_size <= max_msg_size);
    // a slot can be reused if the previous message in that slot was received by everyone.
    // Combine it with the FIFO ordering of messages
    completed_num = Min{sst[∗].received_num[my_rank]};
    if (sent_num − completed_num >= window_size) {
        return nullptr;
    }
    slot = (sent_num + 1) % window_size;
    sst[my_rank].slots[slot].size = msg_size;
```

```
    return sst[my_rank].slots[slot].buf;
}
```

After get_buffer returns a non-null buffer, the application writes the message contents in the buffer and calls send:

```
void send() {
    slot = (sent_num + 1) % window_size;
    sst[my_rank].slots[slot].index++;
    sent_num++;
}
```

Receiving

```
always {
    for i in 1 to n {
        // the next message from node i will arrive in this slot
        slot = (sst[my_rank].received_num[i] + 1) % window_size;
        if (sst[s_i].slots[slot].index == (sst[my_rank].received_num[i] + 1)/window_size +
                1) {
            ++sst[my_rank].received_num[i];
            recv(M(i, sst[my_rank].received_num[i]));
```

```
        }
    }
}
```

3.7.3 Atomic Multicast Delivery in the Steady State

Receive

```
on recv(M(i,k)) {
    // ⊨ K_me(Received M(i,k))
    // store the message to deliver later
    msgs[gi(M(i,k))] = M(i,k);
    sst[my_rank].latest_received_index[i ] = k;
    // calculate global index of the message in the global round−robin ordering
    (min_index_received, lagging_node_rank) =
        (min, arg min)_i sst[my_rank].latest_received_index[i];
    sst[my_rank].global_index = (min_index_received + 1) * |G| + lagging_node_rank
        − 1;
    // ⊨ K_me(Received all messages M(i,k) s.t. gi(M(i,k)) ≤
        sst[my_rank].global_index)
}
```

Stability and Delivery

```
always {
  stable_msg_index = Min{sst[∗].global_index}
  // ⊨ K_me(∀p ∈ G : K_p("Received all messages M(i, k) s.t. gi(M(i, k)) ≤
      sst[my_rank].global_index"))
  for (msg : msgs) {
    if (msg.global_index <= stable_msg_index) {
      deliver_upcall(msg);
      msgs.remove(msg.global_index);
    }
  }
  sst[my_rank].latest_delivered_index = stable_msg_index
}
// ⊨ K_me(Delivered all messages ≤ sst[my_rank].latest_delivered_index)
```

3.7.4 View Change Protocol

Failure Handling and Leader Proposing Changes for Next View

```
every 1 millisecond {
    post RDMA write with completion to every SST row that is not frozen
    if (no completion polled from row r) {
        sst.freeze(r);
        report_failure(r);
    }
}
```

```
report_failure (r) {
    // local node suspects node that owns row r
    sst[my_rank].suspected[r] = true;
    total_failed = Count(sst[∗].suspected, true);
    if (total_failed >= (num_members + 1)/2) {
        throw derecho_partitioning_exception;
    }
}
```

```
find_new_leader (r) {
    // returns the node that r believes to be the leader, on the basis of the current sst.
        Notice that
    // because of asynchrony in the sst update propagations, callers other than r itself
        might be using
    // old version’s of r’s suspicion set, in which case the caller could obtain an old
        leader belief of r’s.
```

```
    // This is safe because leader beliefs are monotonic (they are ascending, in rank
        order).
    for (int i = 0; i < curr_view.max_rank; ++i){
        if (sst[r].suspected[i]) continue;
        else return i;
    }
}
```

```
always {
//Waits until all non−suspected nodes consider me to be the leader. This implies that a
    new leader
//takes action only after every healthy node has pushed final nReceived data to it.
new_leader = find_new_leader(my_rank);
if (new_leader != curr_view.leader_rank && new_leader == my_rank){
    //so long as I continue to believe I will be leader
    while(find_new_leader(my_rank) == my_rank) {
        bool all_others_agree = true;
        //check if everyone else agrees I am leader
        for (r : SST.rows){
            if (sst[my_row].suspected[r] == false)
                all_others_agree &&= (find_new_leader(r) == my_rank)
        }
        if (all_others_agree){
```

```
            //and set myself to leader
            curr_view.leader_rank = my_rank;
            break;
        }
    }
}
// ∃leader_rank s.t. ∀node_id' < leader_rank,
// ∀j ∈ SST, SST.suspects[j][node_id'] ∨ SST.suspects[leader_rank][j]
}
```

```
always {
  for (every row r and s) {
    if (sst[r].suspected[s] == true) {
      // failure propagation − the local node also suspects s
      sst[my_rank].suspected[s] = true;
    }
  }

  for (s = 0; s < num_members; ++s) {
    // if s is newly suspected
    if (sst[my_rank].suspected[s] == true and curr_view.failed[s] == false) {
      freeze(s)
      report_failure(s);
```

```
            // mark s as failed in the current view
            curr_view.failed[s] = true;
            // ⊨ s ∈ F
            // removes predicates defined in section 1 so that no new message can be sent or
                delivered
            curr_view.wedge();
            sst[my_rank].wedged = true;
            if (curr_view.leader_rank == my_rank and sst[my_rank].changes.contains(s) ==
                false) {
              next_change_index = sst[my_rank].num_changes − sst[my_rank].
                  num_installed;
              sst[my_rank].changes[next_change_index] = id of node owning s
              sst[my_rank].num_changes++;
              // ⊨ proposed a new membership change and wedged the current view
            }
          }
        }
    }
```

Terminating old view and installing new view after wedging

```
when (sst[leader_rank].num_changes > sst[my_rank].num_acked) {
  // ⊨ leader proposed a new change
  if (curr_view.leader_rank ≠ my_rank) {
    sst[my_rank].num_changes = sst[leader_rank].num_changes;
    // copy the entire changes vector from the leader's row
    sst[my_rank].changes = sst[leader_rank].changes;
    sst[my_rank].num_committed = sst[leader_rank].num_committed;
    curr_view.wedge();
    sst[my_rank].wedged = true;
    // ⊨ acknowledged leader's proposal and wedged the current view
  }
}

when (curr_view.leader_rank == my_rank and
          MinNotFailed(sst[∗].num_acked) > sst[my_rank].num_committed) {
  // ⊨ K_{U\F}( acknowledged a new proposal )
  sst[my_rank].num_committed = MinNotFailed(sst[∗].num_acked);
  // ⊨ committed acknowledged proposals
}

when (sst[my_rank].num_committed[leader_rank] > sst[my_rank].num_installed[
    my_rank]) {
  // ⊨ leader committed a new membership change
```

```
    curr_view.wedge();
    sst[my_rank].wedged = true;
    when (LogicalAndNotFailed(sst[*].wedged) == true) {
      // ⊨ $K_{U \setminus F}$( current view is wedged)
      terminate_epoch();
    }
}

terminate_epoch() {
  // calculate next view membership
  committed_count = sst[leader_rank].num_committed − sst[leader_rank].
      num_installed;
  next_view.members = curr_view.members;
  for (change_index = 0; change_index < committed_count; change_index++) {
    node_id = sst[my_rank].changes[change_index];
    // if node already a member, the change is to remove the node
    if (curr_view.contains(node_id) == true) {
      new_view.members.remove(node_id);
    }
    // otherwise the change is to add the node
    else {
      next_view.members.append(node_id);
    }
```

```
    }
    if (leader_rank == my_rank) {
        leader_ragged_edge_cleanup();
    }
    else {
        when (sst[leader_rank].ragged_edge_computed == true) {
            non_leader_ragged_edge_cleanup();
        }
    }
    curr_view = next_view;
    // ⊨ New view installed
}
```

```
leader_ragged_edge_cleanup() {
    if (LogicalOr(sst[∗].ragged_edge_computed) == true) {
        Let rank be s.t. sst[rank].ragged_edge_computed is true
        // copy min_latest_received from the node that computed the ragged edge
        for (n = 0; n < |G|; ++n) {
            sst[my_rank].min_latest_received[n] = sst[rank].min_latest_received[n];
        }
        sst[my_rank].ragged_edge_computed = true;
    }
    else {
```

```
        for (n = 0; n < |G|; ++n) {
            sst[my_rank].min_latest_received[n] = Min(sst[∗].latest_received_index[n]);
            // ⊨ K_me( sst[my_rank].min_latest_received[n] number of messages from n are
                safe for delivery)
        }
        sst[my_rank].ragged_edge_computed = true;
    }

    deliver_in_order();
}

non_leader_ragged_edge_cleanup() {
    // copy from the leader
    for (n = 0; n < |G|; ++n) {
        sst[my_rank].min_latest_received[n] = sst[leader_rank].min_latest_received[n];
    }
    sst[my_rank].ragged_edge_computed = true;
    deliver_in_order();
}

deliver_in_order() {
    curr_global_index = sst[my_rank].latest_delivered_index;
    max_global_index = max over n of (sst[my_rank].min_latest_received[n] ∗ |G| + n);
```

```
    for (global_index = curr_global_index + 1; global_index <= max_global_index; ++
        global_index) {
      sender_index = global_index / |G|;
      sender_rank = global_index % |G|;
      if (sender_index <= sst[my_rank].min_latest_received[sender_rank]) {
        deliver_upcall(msgs[global_index]);
      }
    }
  }
```

3.8 Detailed Design of Derecho's Protocols

Derecho is implemented over the building blocks and programming model described in Section 3. Appendix 3.7 illustrated key steps in the protocol in a higher level formalism. Here, we walk through the entire protocol suite in greater detail, describing all aspects of the solution, and offering a correctness justification. As noted earlier, the protocols are transformed versions of a standard virtual synchrony membership protocol, within which we run a standard Paxos-based atomic multicast and durable state machine replication protocol. All three protocols are well known, and were proved correct in Chapter 22 of [21].

3.8.1 The Derecho Membership Protocol

One set of protocols is concerned with membership management. For this purpose Derecho uses a partition-free consensus algorithm based on the protocol described in Chapter 22 of [19], modified to use the SST for information passing. The key elements of the membership subsystem are:

- When a new process is launched and tries to join an application instance, if Derecho was already active, active group members use the SST associated with epoch k to agree on the membership of the top-level group during epoch $k + 1$.
- Conversely, if Derecho was not active, Derecho forms an initial membership view that contains just the set of recovering processes.
- When a process fails Derecho terminates the current epoch and runs a "new view" protocol to form a membership view for the next epoch. These steps are normally combined, but if Derecho lacks adequate resources to create the new view a notification to the application warns it that Derecho activity has been temporarily suspended.
- If multiple joins and failures occur, including cascades of events, these protocol steps can pipeline. Moreover, multiple joins and failures can be applied in batches. However, in no situation will the system transition from a view k to a view $k + 1$ unless the majority of members of k are still present in view $k + 1$, a constraint sufficient to prevent logical partitioning ("split-brain" behavior).

The corresponding protocols center on a pattern of two-phase commit exchanging information through the SST. We designate as the *leader* the lowest-ranked member of the SST that is not suspected of having failed. Notice that there can be brief periods with multiple leaders: if P is currently the lowest-ranked member, but Q suspects P of having failed, then R might still believe P to be the leader, while Q perhaps believes itself to be the leader. Such a situation would quickly converge, because Derecho aggressively propagates fault suspicions. In what follows protocol-state shared through the SST is always tagged with the rank of the current leader[8]. If a succession of leaders were to arise, each new leader can identify the most recently disseminated prior proposal by scanning the SST and selecting the membership proposal with the maximum rank.

The epoch termination protocol runs as soon as any failure is sensed. The leader will collect information, decide the outcome for pending multicasts, and inform the surviving members.

The new-view protocol runs as soon as an adequate set of processes is available. Often, this protocol is combined with epoch termination: both employ the same pattern of information exchange through the SST, and hence the actions can piggyback. The leader proposes a change to the view, through a set of SST columns dedicated for this purpose (we can see such a protocol just as it starts in Figure 3.11). In general, the change proposal is a list of changes: the next view will be the current view, minus process Q, plus process S, and so forth. Non-leaders that see a proposal copy it into their own proposal columns, then acknowledge it through a special SST column set aside for that purpose. When all

[8]Readers familiar with the Paxos Synod protocol [95] may find it helpful to think of this rank number as the equivalent of a Paxos ballot number. However, Paxos can perform an unbounded number of 2-phase exchanges for each ballot, whereas Derecho would not switch leaders more than $N/2$ times before pausing. We revisit this issue, which bears on conditions for progress, in Appendix 3.9.

processes have acknowledged the proposal, it commits.

If a new failure is sensed while running the two-phase commit to agree upon a new view, the leader extends the proposal with additional requested changes and runs another round. Recall that if any process ever suspects a majority of members of the current view of having failed, it shuts itself down; this is also true for the leader. It follows that progress only occurs if at most a minority of members of the prior view have failed. Agreement on the next view is reached when (1) every process has acknowledged the proposal, or has been detected as faulty; (2) the set of responsive members, plus the leader, comprise a majority of the current membership; (3) no new failures were sensed during the most recent round. This is the well-known group membership protocol of the virtual synchrony model [20, 22, 23] and has been proven correct [19]. We will not repeat the proof; passing data through the SST rather than in messages does not change the fundamental behavior.

Next, Derecho uses the new top-level view to compute the membership of subgroups and shards. Specifically, as each new top-level view becomes defined, Derecho runs the mapping function described in Section 3.2 to generate a list of subviews for the subgroups and shards. The new membership can now trigger object instantiation and initialization, as detailed momentarily. Given a new top-level view, for each process P, Derecho examines the subgroup and shard views. If P will be a member of some subgroup or shard that it is not currently a member of, a new instance of the corresponding object type is created and initialized using constructor arguments that were supplied via the constructor call that created the top-level group.

Initialization of newly created objects is carried out as follows:

- For each subgroup or shard that is restarting from total failure (that is, those with no members in the prior epoch, but one or more members in the new epoch), a special cleanup and recovery algorithm is used to retrieve the most current persisted state from logs. Our algorithm is such that no single failure can ever prevent recovery from total failure, but if a subset of the system is trying to restart while a number of processes are still down, several logs may be missing. In such states, recovery might not be possible, in which case the view is marked as inadequate and Derecho waits for additional recoveries before full functionality is restored.
- Conversely, if a process is joining an already-active subgroup or shard, state transfer is performed as a two-step process. In the first step, a process that is restarting is informed of the future view, and which existing active member has the data needed to initialize it. The restarting process will pull a copy of that data and load it while still "offline". In the second step, it pulls copies of any updates that occurred while the first step was running, updates its versions, and installs the new view. The joining member is now fully operational.

State transfers from an already-active subgroup or shard include both fields of the `replicated<T>` object that are included in its serialization state as well as data associated with its `volatile<T>` and `persistent<T>` version vectors. In contrast, on restart from a total failure, only `persistent<T>` data is recovered; volatile version vectors are initialized by the corresponding constructors.

Next, Derecho creates a new SST instance for the new epoch and associates an RDMC session with each sender for each subgroup or shard (thus, if a subgroup has k senders it will have k superimposed RDMC sessions: one per sender).

The epoch is now considered to be *active*.

3.8.2 Atomic Multicast and Paxos

A second set of protocols handle totally ordered atomic multicast (vertical Paxos) or two-phase message delivery to a subgroup or shard handling persistent data. These protocols take actions in asynchronous batches, a further example of Derecho's overarching pipelined behavior.

A sender that needs to initiate a new multicast or multi-query first marshalls the arguments, then hands the marshalled byte vector to RDMC (or, if small, to SMC) for transmission. As messages arrive, Derecho buffers them until they can be delivered in the proper order, demarshalls them, and then invokes the appropriate handler. When data can be transmitted in its native form, a scatter gather is used to avoid copying during marshalling, and references into the buffer are used to avoid copying on delivery.

Although Derecho has several modes of operation, unreliable mode simply passes RDMC or SMC messages through when received. Totally ordered (atomic multicast) mode and durable totally ordered mode employ the same protocol, except that message delivery occurs at different stages. For the totally ordered (atomic multicast) mode Derecho uses the SST fields shown as `nReceived` in Figures 3.10 and 3.11. Upon receipt, a message is buffered and the receiver increments the `nReceived` column corresponding to the sender (one process can send to multiple subgroups or shards, hence there is one column per sender, per role). Once all processes have received a multicast, which is detected by aggregating the minimum over this column, delivery occurs in round-robin order. Interestingly, after incrementing `nReceived`, the SST push only needs to write the changed field, and only needs to push it to other members of the same subgroup or shard. This is potentially a very inexpensive operation since we might be writing as few as 8 bytes, and a typical subgroup or shard might have just 2 or 3 members.

For durable totally ordered mode, recall that Derecho uses a two-stage multicast delivery. Here, the first-phase delivery occurs as soon as the incoming RDMC message is available and is next in the round-robin order (hence, the message may not yet have reached all the other members). This results in creation of a new pending version for any `volatile<T>` or `persistent<T>` variables. After persisting the new versions, `nReceived` is incremented. Here, commit can be inferred independently and concurrently by the members, again by aggregating the minimum over the `nReceived` columns and applying the round-robin delivery rule.

All three modes potentially deliver batches of messages, but in our experiment only very small batches were observed. Moreover, unless the system runs short on buffering space, the protocol is non-blocking, and can accept a continuous stream of updates, which it delivers in a continuous stream of delivery upcalls without ever pausing.

3.8.3 Epoch Termination Protocol

The next set of protocols are concerned with cleanup when the system experiences a failure that does not force a full shutdown but may have created a disrupted outcome, such as the one illustrated in Figure 3.4. The first step in handling such a failure is this: as each process learns of an event that will change membership (both failures and joins), it freezes the SST rows of failed members, halts all RDMC sessions and then marks its SST row as wedged before pushing the entire SST row to all non-failed top-level group members. Thus, any change of membership or failure quickly converges to a state in which the epoch has frozen with no new multicasts underway, all non-failed members wedged, and all non-failed processes seeing each-other's wedged SST rows (which will have identical contents).

Derecho now needs to finalize the epoch by cleaning up multicasts disrupted by the crash, such as multicasts $m_{Q:4}$ and $m_{P:5}$ in Figure 3.11. The core idea is as follows: we select a leader in rank order; if the leader fails, the process of next-highest rank takes over (the same leader as is used for membership changes, and as noted previously, a single pattern of information exchange will carry out both protocols if the next proposed view is already known at the time the current epoch is terminated). As leader, a process determines which multicasts should be delivered (or committed, in the two-stage Paxos case) and updates its nReceived fields to report this. Then it sets the "Final" column to true and replaces the bottom symbol shown in Figure 3.11 with its own leader rank. We refer to this as a *ragged trim*. Thus one might now see a ragged trim in the nReceived columns and

"T|r" in the column titled "Final", where r would be a rank value such as 0, 1, etc. Other processes echo the data: they copy the nReceived values and the Final column, including the leader's rank.

The ragged trim itself is computed as follows: for each subgroup and shard, the leader computes the final deliverable message from each sender using the round-robin delivery rule among active senders in that subgroup or shard. This is done by first taking the minimum over the nReceived columns, determining the last message, and then further reducing the nReceived values to eliminate any gaps in the round-robin delivery order. Since messages must be delivered in order, and Derecho does no retransmissions, a gap in the message sequence makes all subsequent messages undeliverable; they will be discarded and the sender notified. Of course the sender can still retransmit them, if desired. This is much simpler than the classic Paxos approach, in which logs can have gaps and readers must merge multiple logs to determine the appropriate committed values.

For example, in Figure 3.4 the round-robin delivery order is $m_{P:1}, m_{Q;1}, m_{P:2}, m_{Q,2}, m_{P:3}, m_{Q:3}, m_{P:4}$, $m_{Q:4}, m_{P:5}, m_{Q:5}$, and so forth. However, the last message from Q to have reached all healthy processes was Q:3, because the crash prevented Q:4 from reaching process P. Thus the last message that can be delivered in the round-robin delivery order was message P:4. Any process with a copy of Q:4 or P:5 would discard it (if the message was an atomic multicast) or would discard the corresponding version-vector versions (if the message was a durable totally ordered mode message and the first-phase delivery had already occurred).

The leader would thus change its own nReceived columns[9] to show P:4, Q:3. Then

[9]Reuse of columns may seem to violate monotonicity, because in overwriting the nReceived column, P

it would set T|0 into the final field, assuming that P is the leader doing this computation. Other processes see that Final has switched from false to true, copy the ragged trim from the leader into their own nReceived columns, and copy T|0 to their own Final column. Then they push the SST row to all other top-level members.

A process that knows the final trim and has successfully echoed it to all other top-level group members (excluding those that failed) can act upon it, delivering or discarding disrupted multicasts in accord with the trim and then starting the next epoch by reporting the new membership view through upcall events.

Although Figure 3.11 shows a case with just a single subgroup or shard, the general case will be an SST with a block structure: for each subgroup or shard, there will be a set of corresponding columns, one per sender. During normal execution with no failures, only the members of the subgroup or shard will actively use these columns and they push data only one one-another; other top-level members will have zeros in them. When the epoch ends and the row wedges, however, every process pushes its row to every other process. Thus, the leader is certain to see the values for every subgroup and shard. At this point, when it computes the ragged trim, it will include termination data for all subgroups and shards. Every top-level member will learn the entire ragged trim. Thus, a single top-level termination protocol will constitute a ragged trim for active multicasts within every subgroup and shard, and the single agreement action covers the full set.

---

may decrease values. No issue arises because the nReceived columns are no longer needed at this point and the SST message delivery actions that monitor those columns are disabled. In effect, these columns have been reassigned them to new roles.

3.8.4 Recovery From Full Failure

To deal with full shutdowns we will need a small amount of extra help. We will require that every top-level group member maintain a durable log of new views as they become committed, and also log each ragged trim as it is proposed. These logs are kept in non-volatile storage local to the process. Our protocol only uses the tail of the log.

With this log available, we can now describe the protocols used on restart from a full shutdown. These define a procedure for inspecting the persisted Derecho state corresponding to version vectors that may have been in the process of being updated during the crash and cleaning up any partial updates. The inspection procedure requires access to a majority of logs from the last top-level view and will not execute until an adequate set of logs is available. It can then carry out the same agreement rule as was used to compute the ragged trim. In effect, the inspector is now playing the leader role. Just as a leader would have done, it records the ragged trim into the inspected logs, so that a further failure during recovery followed by a new attempt to restart will re-use the same ragged trim again. Having done this, Derecho can reload the state of any persisted version vectors, retaining versions that were included in the ragged trim, and discarding any versions that were created and persisted, but did not commit (were excluded from the ragged trim). The inspector should record a ragged trim with "top" shown as its leader rank: in a given epoch, the inspector is the final leader.

3.8.5 Failure of a Leader During the Protocol, or of the Inspector During Restart

Now we can return to the question of failure by considering cases in which a leader fails during the cleanup protocol. Repeated failures can disrupt the computation and application of the ragged trim, causing the "self-repair" step to iterate (the iteration will cease once a condition is reached in which some set of healthy processes, containing a majority of the previous top-level view, stabilizes with no healthy process suspecting any other healthy process and with every healthy process suspecting every faulty process). When the process with rank r sees that every process ranked below r has failed, and that all other correct processes have discovered this fact, it takes over as leader and scans its SST instance to see if any ragged trim was proposed by a prior leader. Among these, it selects and reused the ragged trim proposed by the leader with the highest rank. If it finds no prior proposal, it computes the ragged trim on its own. Then it publishes the ragged trim, tagged with its own rank.

3.8.6 Special Case: Total Failure of a Shard

One remaining loose end remains to be resolved. Suppose that a subgroup or shard experiences a total failure. In the totally ordered (atomic multicast) mode, this does not pose any special problem: in the next adequate view, we can restart the subgroup or the shard with initially empty state. But suppose that the subgroup was running in the durable totally ordered mode and using objects of type `persistent<T>`. Because no member survived the crash, and because nReceived is written by a subgroup member only to the other subgroup members, the leader has no information about the final commit value for the subgroup (had even a single member survived long enough to wedge its SST row and push it to the leader, then any committed version would be included into nReceived and the leader would include it in the ragged trim, but because *all* members crashed, the leader sees only 0's in the nReceived columns for the subgroup's senders).

This is a specific situation that the leader can easily sense: for such a subgroup, every member is suspected of failure, and the leader lacks a wedged SST row from any member. It can then record a special value, such as -1, for the corresponding ragged trim columns. Now, before allowing the subgroup to recover, we can reconstruct the needed state by inspection of the persisted state of *any* subgroup member, treating the state as inadequate if the persisted data for a shard is not accessible. Suppose that the state is adequate, the leader is able to inspect the state of subgroup member Q, and that it discovers a persisted vector containing versions $0..k$. Clearly, this vector must include any committed data for the subgroup because (prior to the failure) any commit would have had to first be persisted by every member, including Q. Furthermore, we know that subsequent to the crash, the leader did not compute a ragged trim for this subgroup.

Conversely, the version vector could contain additional persisted versions that are not present in any log except for Q's log. These versions are safe to include into the ragged trim because they were generated by delivery of legitimate multicasts and reflect a delivery in the standard round-robin ordering. Thus, we can include them in the ragged trim. When the subgroup recovers in the next view, these versions will be part of the initial state of the new members that take over the role of the prior (crashed) members. Accordingly, we copy the version vector from Q to each process that will be a member of the subgroup in the next epoch, record the ragged trim, and resume activity in the new epoch. Notice that this approach has the virtue that no single failure can ever prevent progress. While we do need to access at least one log from the subgroup, it is not important which log we use, and because copies are made (for the new epoch) before we start execution in the new epoch, a further failure exactly at this instant still leaves the system with multiple copies of the log that was used. This property can be important: in a large system, one certainly would not want a single crash to prevent progress for the entire system.

3.8.7 Discussion

Our approach can be understood as a new implementation of older protocols, modified to preserve their information structure but favor an asynchronous, pipelined style of execution. In effect, we reimplement both virtually synchronous membership management and Paxos as a collection of concurrently active components, each optimized for continuous data flow. These include our use of RDMA itself, if available, for reliable communication, the RDMC (or SMC) for 1-to-N messaging, streams of lock-free SST updates in an N-to-N pattern (here N is the shard or subgroup size: probably just 2 or 3), and queries that run in a lock-free way on temporally precise snapshots. Receiver side batching ensures that when a Derecho receiver is able to make progress, it does as much work as possible. Monotonicity lets us maximize the effectiveness of this tactic.

Derecho's replication model, in which all replicas are updated and reads can access just one instance eliminates the usual Paxos quorum interactions, and simplifies the associated commit logic: it becomes Derecho's notion of distributed stability.

Yet the usual Paxos "Synod" messaging pattern is still present, notably in the membership protocol after a failure occurs. Here, consensus is required and the usual iteration of two-stage commit protocols has been replaced by a pattern of information exchanges through the SST that corresponds to the fault tolerant K1 concept, discussed in Section 3.3.5. The key challenge turns out to be consensus on the ragged trim, and the key idea underlying our solution is to ensure that before any process could *act* on the ragged trim, the trim itself must have reached a majority of members of the top-level view and been logged by each. Thus in any future execution, a ragged trim that might have been acted upon is certain to be discovered and can be reused.

It should now be clear why our protocol logs the proposed ragged trim, tagged by the rank of the leader proposing it, at every member of the top-level group. Consider a situation in which a partially recorded ragged trim has reached less than a majority of top-level members, at which point a failure kills the leader. When this failure is detected, it will cause the selection of a new leader, which might in turn fail before reaching a majority. With our scheme, a succession of ragged trim values would be proposed, each with a larger rank value than was used to tag the prior one. This is shown in Figure 3.4 as a column "Final" where values are either F, with no leader (bottom), or T, in which case the nReceived values report the ragged trim, and the rank of the leader that computed the ragged trim would replace the bottom symbol: 0, 1, etc. The key insight is that if any ragged trim actually reached a majority, it will be learned by any leader taking over because the new leader has access to a majority of SST rows, and at least one of those rows would include the ragged trim in question. Thus, in the general case, the ragged trim used to clean up for the next epoch will either be learned from the prior leader, if that trim could have reached a majority, or it will be computed afresh, if and only if no prior value reached a majority.

This is precisely the pattern that arises in the Synod protocol [95] (the slot-and-ballot protocol at the core of Paxos). The main difference is that classic Paxos uses this protocol for every message, whereas Derecho uses the Synod protocol only when a membership change has occurred. Nonetheless, this pattern is really the standard one, with the leader rank playing the role of ballot number.

Although the Derecho protocol is expressed very differently in order to take full advantage of the SST, it can be recognized as a variant of a protocol introduced in the early Isis Toolkit, where it was called Gbcast (the *group multicast*) [20]. The protocol is correct because:

- If any message was delivered (in totally ordered (atomic multicast) mode) or committed (in durable totally ordered mode), the message was next in round-robin order and was also received (persisted) by all group members, and every prior message must also have been received (delivered) by all group members. Thus, the message is included into the ragged trim.
- If the ragged trim includes a message (or version), then before the ragged trim is

applied, it is echoed to at least a majority of other members of the top-level group. Therefore, the ragged trim will be discovered by any leader taking over from a failed leader, and that new leader will employ the same ragged trim, too.

- Upon recovery from a total failure, the inspector runs as a single non-concurrent task and is able to mimic the cleanup that the leader would have used. Moreover, the inspector can be rerun as many times as needed (if a further failure disrupts recovery) until the cleanup is accomplished.

A further remark relates to the efficiency of Derecho. Keidar and Schraer developed lower bounds on the information exchanges required to achieve stable atomic broadcast (they use the term Uniform Agreement) [61]. By counting exchanges of data as information flows through the SST, one can show that Derecho achieves those bounds: Derecho's distributed decision making ensures that as soon as a process can deduce that a message can safely be delivered, it will be delivered. Because control information is shared through the SST and transmitted directly from the member at which an event occurs to the members that will perform this deductive inference, the pattern of information flow is clearly optimal.

With respect to the efficiency of data movement, note first that for small objects Derecho uses SMC, which simply sends messages directly from source to receivers. RDMC's binomial pipeline has logarithmic fan-out, once "primed" with data, all processes make full use of their incoming and outgoing NIC bandwidth, and every byte transfers traverses a given NIC just once in each direction: the best possible pattern for block-by-block unicast data movement.

3.9 Progress

A full comparison of Derecho to classic Paxos requires two elements. Setting aside subgroup and sharding patterns, consider a single Derecho group in which all members replicate a log either in-memory (virtually synchronous atomic multicast, also known as "vertical Paxos") or on disk (via Derecho's durable version vectors). For both cases, Derecho's protocols guarantee that the Paxos ordering and durability safety properties hold in any state where progress is permissible. Indeed, one can go further: any "log" that a classic Paxos protocol could produce can also be produced by Derecho, and vice-versa (they *bisimulate*).

But progress raises a more complex set of issues. Note first that the Fischer, Lynch and Patterson impossibility result establishes that no system capable of solving consensus can guarantee progress [43]. This limitation clearly applies to Derecho and also to any Paxos implementation. This is a well known observation, dating to the 1990's, and not at all surprising. No system of this kind can guarantee "total correctness", in the sense of being both safe and live.

The good news is that the scenario studied in the FLP result is very unlikely in real systems (FLP requires an omniscient adversary that can examine every message in the network, selectively delaying one message at a time now and then). Thus, FLP does not teach that consensus is impossible. On the contrary, we can build practical systems, and they will work in the target environments. We simply must be aware that they cannot *guarantee* progress under all possible failure and message delay scenarios.

Accordingly, one talks about a form of conditional liveness by treating failure detection as a black box. These black boxes can then be categorized, as was done by Chandra and Toueg [29]. Relevant to our interests is a failure detector called P, the "perfect" failure detector. A perfect failure detector will report every real failure that occurs and will never misreport a healthy process as failed.

P is normally defined in the context of a perfect network that never drops packets, even because of router or switch failures. This is unrealistic: Any real system does experience network failures, hence P cannot be implemented. This leaves us with two options. We could ignore the question of realism and simply ask "Yes, but would progress occur with P"? This is the essential technique used by Chandra and Toueg. We end up with an interesting but impractical conclusion, for the reason noted. For our purposes here, we will not say more about this form of analysis.

Alternatively, we can work around the limitation by accepting that networks do fail, and then treating unreachable processes as if they had failed. To prevent logical partitioning, we employ a majority-progress rule. This is how Derecho is implemented: rather than insisting on accurate failure detections, Derecho concerns itself entirely with unreachability, but uses virtually synchronous membership to prevent "split brain" behavior.

Thus, as seen earlier, Derecho will make progress if (1) no more than a minority of the current view fails, and (2) the failure detector only reports a failure if the process in question has actually failed *or has become unreachable*. Classic Paxos protocols have a similar liveness guarantee, but not identical. For classic Paxos, any two update quorums must overlap, hence Paxos requires that a majority of the processes be accessible. Classic

Paxos doesn't actually "care" about failures. It simply waits for processes to respond.

We could take these to be equivalent progress conditions. Indeed, and this is why we noted that the P detector is of interest, Keidar and Schraer argue that P can be generalized to cover all such behaviors (the work is spread over several papers, but see [61]).

However, a subtle issue now arises: it turns out that there are conditions under which Derecho can make progress where a classic Paxos protocol would have to pause. As an example, consider a group of initial size N in which half the members fail, but gradually, so that Derecho can repeatedly adapt the view. Recall that Derecho's majority progress requirement applies on a view-by-view basis. Thus, for example, we could start with $N = 25$ members, then experience a failure that drops 5 members. Now $N = 20$. Both Derecho and Paxos can continue in this mode. Next, suppose 9 of the remaining processes crash. Derecho can adapt again: 9 is a minority of 20, hence Derecho forms a view containing 11 processes, and continues. Classic Paxos, in contrast, finds that 14 of the original 25 processes have become inaccessible. Updates cease until some processes restart.

But the situation gets more complex if all 11 now fail, so that the entire system is down, and then some restart. Classic Paxos will resume updates as soon as 13 members are up and running: any majority of the original 25 suffices. Derecho cannot recover in this state without 6 of the 11 members of that final view: otherwise, it deems the view to be inadequate. In effect, sometimes Paxos is live when Derecho is wedged. But sometimes Derecho is live when Paxos is wedged. And yet both always respect the Paxos *safety* properties.

3.10 Acknowledgements

This work was supported in part by grants from the NSF-sponsored Institute for Computational Sustainability at Cornell University, NSF, DARPA, ARPA-e, NYSERDA, Huawei, and Mellanox. Cornell's Dave Lifka and Chris Hemple at the UT Stampede 1 computing center were extremely helpful in providing testbeds used to evaluate Derecho. Heming Cui and the APUS developers helped us compare Derecho with APUS. We are particularly grateful to the anonymous TOCS reviewers for their detailed and constructive suggestions. In addition, a number of our colleagues reviewed early drafts of this paper and made helpful comments. These include: Marcos Aguilera, Mahesh Balakrishnan, Eddie Bortnikov, Miguel Castro, Gregory Chockler, Hana Chockler, David Cohen, Manuel Costa, Aleksandar Dragojević, Joseph Gonzalez, Guy Goren, Qi Huang, Roy Friedman, Alexy Gotsman, Joe Izraelevitz, Idit Keidar, Lev Korostyshevsky, Liran Liss, Dahlia Malkhi, David Maltz, Pankaj Mehra, Dahlia Malkhi, Yoram Moses, Dushyanth Narayanan, Oded Padon, Fernando Pedone, Noam Rinetsky, Kurt Rosenfeld, Ant Rowstron, Mooley Sagiv, Ashutosh Saxena, Robert Soule, Ion Stoica, Orr Tamir, Paulo Verissimo, and Matei Zaharia.

CHAPTER 4

SPINDLE: TECHNIQUES FOR OPTIMIZING ATOMIC MULTICAST ON RDMA

Notes:

This paper was published in ICDCS 2022, in Bologna, Italy. The authors are Sagar Jha, Lorenzo Rosa, and Ken Birman. While working on the Derecho system, we realized that its performance for small message replication was unexpectedly low compared to the network speed. This led to a variety of optimizations aimed at improving small message performance, distilled in a few major themes in this paper, optimizations that we believe can be beneficial to a multitude of RDMA-based systems. My contributions include a) designing the optimizations, and b) designing a comprehensive set of evaluations to show the effectiveness of the optimizations. I also advised Lorenzo Rosa on this project. Lorenzo assisted in implementing the ideas, participated equally in formulating some of the receiver-side batching optimizations, and carried out the entire evaluation of the system over the multiple iterations of the work.

This paper, in many ways, represents my core interests in distributed systems. I believe in coming up with clean, elegant protocols that translate efficiently to the actual code performance in practical settings. I find myself frequently reasoning about the performance of the systems I work on, not theoretically but logically. In my future work in the industry, I intend to continue to make systems work and perform close to their potential. I am particularly proud of the null-send protocol I devised as part of this work, where the four desired properties of sender-invariance, low-overhead, correctness, and quiescence are interesting purely from a performance perspective. Working at high RDMA speeds, where sending a 1-byte null message is comparable in latency to sending a few KBs of application data, designing what might seem like a trivial task ends up needing careful attention and forethought. This also sheds a light on how delicate the performance is with RDMA's sub-microsecond latencies – small application-level delays in the application can result in much larger performance degradation.

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.

4.1 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) [76], 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 [17] 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 $200\,\mathrm{Gbps}$ and one-way latency as low as $0.75\,\mu\mathrm{s}$.

Prior research applied RDMA to strongly consistent fault-tolerance models such as State Machine Replication (SMR) [86], yielding RDMA libraries for atomic multicast [55, 81, 99]. 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 [55]. The approach is general, and should be of value for the entire class of strongly consistent replication systems [46], as well as for other forms of middleware and for other high-speed networking technologies, e.g., DPDK [1]. 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 [11] and optimizing the match between RDMA and data movement [38, 59]. 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.

4.2 Background

4.2.1 Derecho atomic multicast protocol

Derecho implements a virtually synchronous membership model within which it offers atomic multicast and persistent replication using variants of Paxos [17, 55]. 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.

Table 4.1: Sample SST state at node 0 for 5 application nodes and 3 subgroups

(a) State for atomic multicast

	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

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

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

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.

4.2.2 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 [55].

In Table 4.1a 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 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.

4.2.3 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 4.1b, 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.

4.2.4 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 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.

4.3 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.

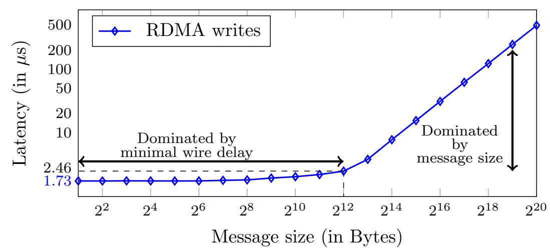


Figure 4.1: Latency vs data size on $100\,\mathrm{Gbps}$ Mellanox RDMA. Latency is nearly constant for up to 4KB message size.

4.3.1 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 4.1 plots RDMA write latency for different message sizes. Latency for small messages does not increase appreciably with the data size, increasing only marginally from $1.73\,\mu\mathrm{s}$ for 1-byte data to $2.46\,\mu\mathrm{s}$ for 4KB data.

The predicates described in Section 4.2.4 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 $\sim 1\,\mu\mathrm{s}$. 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 4.1 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.

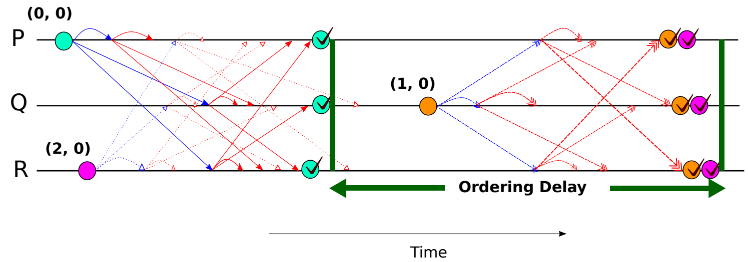


Figure 4.2: Delays at a sender can impact performance of the system by slowing down other senders.

Accordingly, we expanded the batching architecture into a unified technique that covers all stages of atomic multicast. 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 4.2.4 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. [59] 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.

4.3.2 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 4.2.1), 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 4.2. 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 4.1), 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 [50, 54]. We will return to this point in Section 5.6.

4.3.3 Efficient thread synchronization

Efficient thread synchronization is crucial to high performance. For systems that rely on polling (see Section 4.2.4), 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.

4.3.4 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 4.4.4.

4.4 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.

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.

4.4.1 Opportunistic batching

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 4.3 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 maxi-

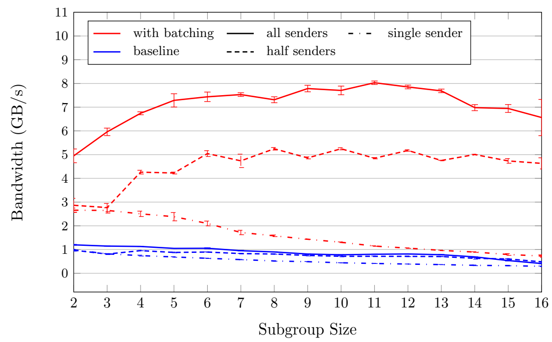


Figure 4.3: Performance for single subgroup with opportunistic batching. Performance improves by up to 16X.

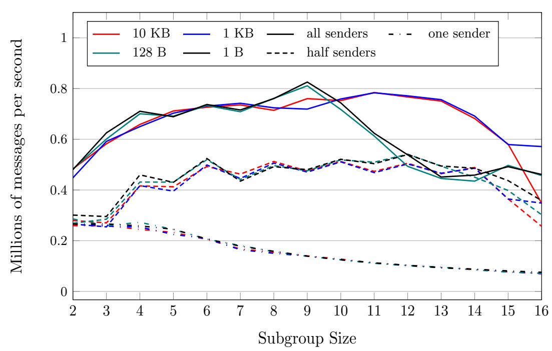


Figure 4.4: Rate of delivery for single subgroup with opportunistic batching. Derecho has a second communication larger, RDMC, for very large subgroups or messages. Although RDMC was not evaluated in our work, shifting to it might be advisable for subgroups with more than 12 members.

mum 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 4.1, performance is proportional to the data size for both the baseline and the optimized version. Hence, the number of messages delivered per second

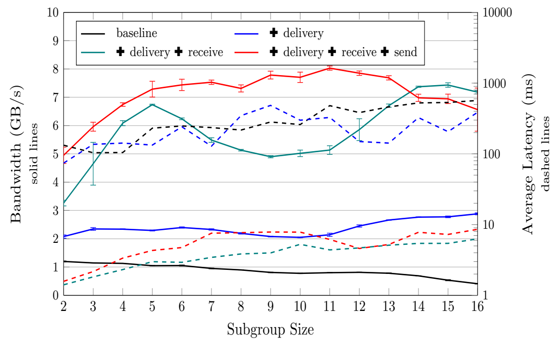


Figure 4.5: Performance gains with batching applied to successively more stages of the pipeline for all senders. Throughput (left Y-axis) is shown by solid lines, and latency (right Y-axis) using dashed lines. Both metrics show significant improvements relative to our baseline system.

remains about the same for different small message sizes. Figure 4.4 confirms this observation for the optimized version. As such, all subsequent experiments only show data for the 10KB case.

It is interesting to learn the impact of batching at different stages of the protocol. Figure 4.5 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

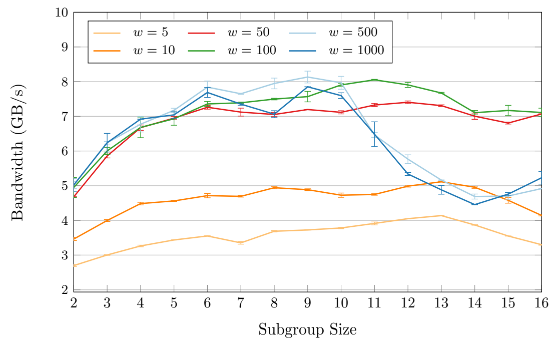


Figure 4.6: Performance with different window sizes when all nodes are sending messages continuously

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).

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 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.

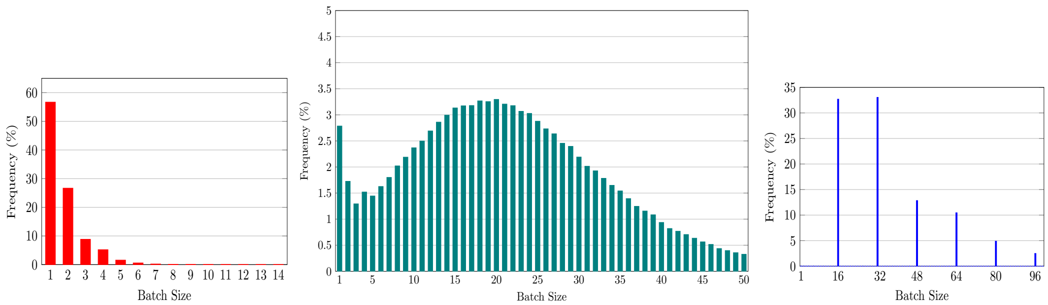
Figure 4.6 plots the results. Even a small window size of 5 increases performance by 4.5X average compared to the baseline with 100 window size! The highest performance is obtained for a window size of 100. Consequently, all our experiments for 10KB message size use a window size of 100. It is important 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 4.7 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 in heterogeneous environments where nodes are running at different speeds.



(a) Sends (b) Receives (c) Deliveries

Figure 4.7: 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.

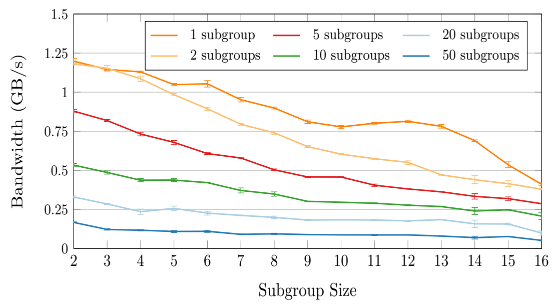


Figure 4.8: Performance of baseline for single active subgroup

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 4.8), performance consistently decreases with

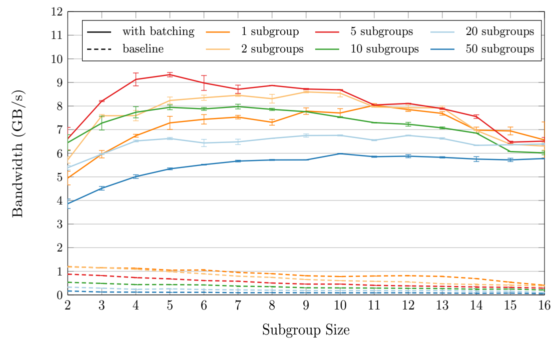


Figure 4.9: Performance with opportunistic batching for single active subgroup.

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 4.9 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 4.7)

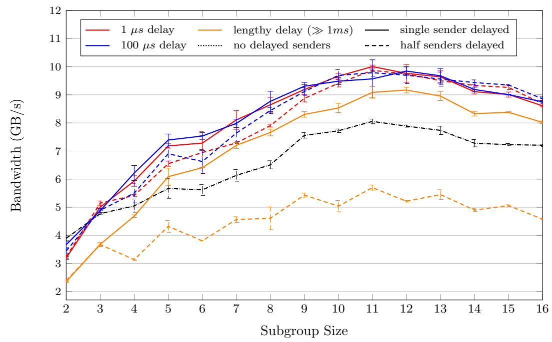


Figure 4.10: Data for sender delay test with null-sends.

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 4.4.3.

4.4.2 Null-send scheme

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

Figure 4.10 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

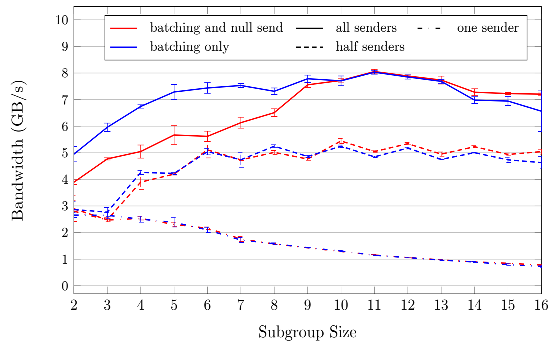


Figure 4.11: Impact of null-sends on continuous sending.

drastically and in fact, decreases from $3.779\,\mu s$ for 2 nodes to $1.617\,\mu s$ for 8 nodes and $1.192\,\mu s$ for 16 nodes. This confirms that nulls accelerate delivery of application messages.

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 4.11 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.

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. 4.3.2 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.

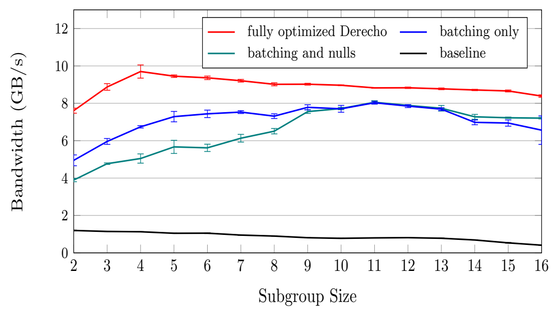


Figure 4.12: Impact of efficient synchronization.

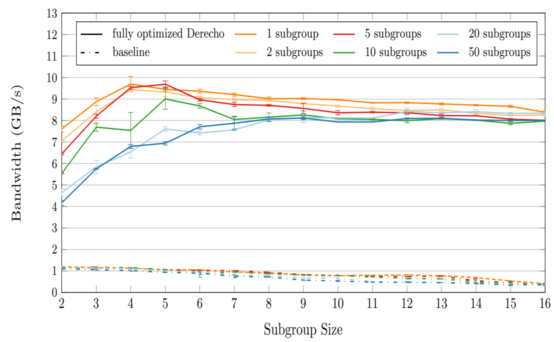


Figure 4.13: Final performance with all optimizations for multiple active subgroups.

4.4.3 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 4.12 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.

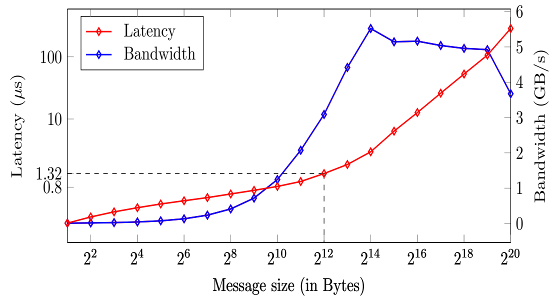


Figure 4.14: Performance of memcpy with data size.

Figure 4.13 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.

4.4.4 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 4.14 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.

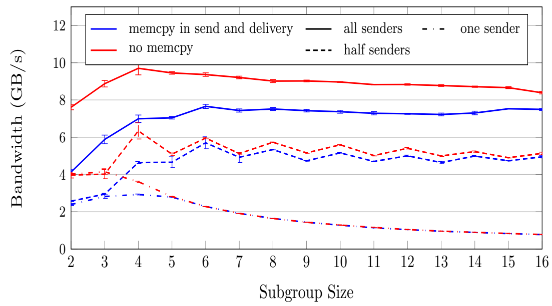


Figure 4.15: Performance with memcpy in send and delivery.

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 4.15 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.

4.4.5 Final results

Figure 4.16 shows the final numbers. As noted, although our optimizations were focused on throughput, Figures 4.5 and 4.17 both show substantial improvement in latency. Note that the logarithmic Y-axis scale magnifies error bars for small latency values.

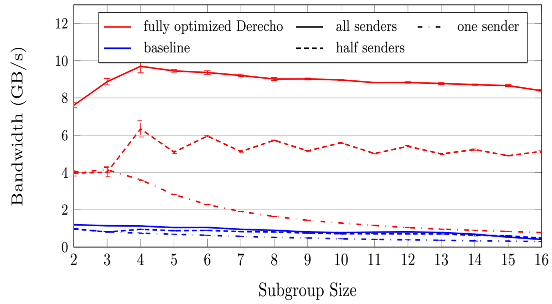


Figure 4.16: Final throughput numbers for a single subgroup.

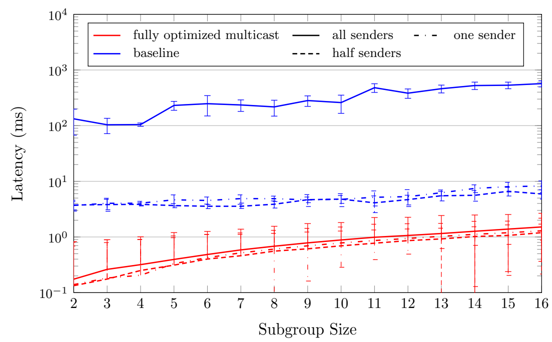


Figure 4.17: Final latency numbers for a single subgroup.

4.4.6 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 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 4.18 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

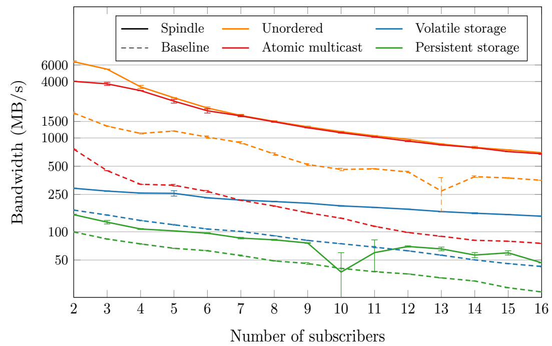


Figure 4.18: DDS performance improvements with Spindle optimizations for all 4 QoS levels.

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.

4.5 Related Work

There has been other work on using RDMA to accelerate state machine replication [55, 81, 99] and key-value storage [38, 58–60]. The separation of control from the data plane originated in BarrelFish and Arrakis [11, 80] and the iX μ-kernel [13] 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. [59]. Other similar studies [94] 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 [92], 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 [74] and Transis [37] systems, and similar mechanisms have been used in modern Ring Paxos protocols [50, 54]. 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.

4.6 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 $100\,\mathrm{Gbps}$ 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.

CHAPTER 5

RELIABLE, EFFICIENT RECOVERY FOR COMPLEX SERVICES WITH REPLICATED SUBSYSTEMS

Notes:

This paper was published in DSN 2020, held remotely. The authors are Edward Tremel, Sagar Jha, Weijia Song, David Chu, and Ken Birman. A replication group can fail if all nodes lose power, and also needs to shut down in cases when a majority of the nodes fail so as to avoid split-brain behavior (two or more partitions of the system which each believe that they constitute the entirety of the system). When the nodes are available to reboot the system again, the restart protocols ensure that the system state is reconstructed correctly from the data on the disks that was persisted at those nodes in the run prior to the failure. This work was led by Edward Tremel, who implemented the needed restart logic. My contributions include verifying the theoretical analysis of correctness, the recognition of the problem of assigning restarting nodes to subgroups as a min-cost flow problem, and the subtle integration required with Weijia Song's persistent storage layer.

It bears emphasizing that system restart problems for state machine replication services have been extensively studied in the literature. The novelty of our work lies in the uniqueness of the system state distributed across multiple subgroups that contain just a subset of the group membership and in our in-depth analysis of correctness that involves reasoning through failures that could occur during the restart. Also unusual is the integration of the sharded layout of the system with the larger virtual-synchrony group membership of the entire top-level group. Here, we find that while a quorum policy is needed at the top level, to avoid risk of partitioning, we also need f+1 members in each shard to tolerate f possible crash failures. Through careful analysis, we arrive at a provably correct solution to the problem of deciding when and how to restart.

Abstract

Applications with internal substructure are common in the cloud, where many systems are organized as independently logged and replicated subsystems that interact via flows of objects or some form of RPC. Restarting such an application is difficult: a restart algorithm needs to efficiently *provision* the subsystems by mapping them to nodes with needed data and compute resources, while simultaneously guaranteeing that replicas are in distinct failure domains. Additional failures can occur during recovery, hence the restart process must itself be a restartable procedure. In this paper we present an algorithm for efficiently restarting a service composed of sharded subsystems, each using a replicated state machine model, into a state that (1) has the same fault-tolerance guarantees as the running system, (2) satisfies resource constraints and has all needed data to restart into a consistent state, (3) makes safe decisions about which updates to preserve from the logged state, (4) ensures that the restarted state will be mutually consistent across all subsystems and shards, and (5) ensures that no committed updates will be lost. If restart is not currently possible, the algorithm will await additional resources, then retry.

5.1 Introduction

We are seeing a shift from a *query-dominated cloud* in which most operations are read-only and use data acquired out-of-band, to a *real-time control cloud*, hosting increasingly complex online applications, in which near-continuous availability is important. Such needs arise in stream processing for banking and finance, IoT systems that monitor sensors and control robots or other devices, smart homes, smart power grids, smart highways, and cities that dynamically manage traffic flows, etc. These applications often have multiple subsystems that interact, and that bring safety requirements which include the need for fault-tolerance and consistency in the underlying data-management infrastructure.

Traditional transactional database methods scale poorly if applied naively [49]. Our work adopts a *state machine replication* model, using *key-value sharding* for scaling. Such models are relatively easy to program against and hence increasingly popular, but pose challenges when crashes occur.

To maintain the basic obligations of the state machine replication methodology, updates must be applied to replicas exactly once, in the same order, and should be durable despite damage a failure may have done. For a given replication factor the system should also guarantee recoverability if fewer than that number of crashes occur. Subsystems may being further constraints: numbers of cores, amounts of memory, etc. A further consideration is that datacenter hardware can exhibit correlated failures due to shared resource dependencies. To ensure high availability, replicas must be placed into distinct *failure correlation sets*.

Performance considerations further shape the design of modern cloud systems, which often migrate artificially intelligent behavior into the edge [18]. This may entail use of machine learned models for decision-making or event classification, as well as real-time learning in which new models are trained from the incoming data stream. For example, a smart highway might need to learn the behavior of vehicles, and adapt the acquired

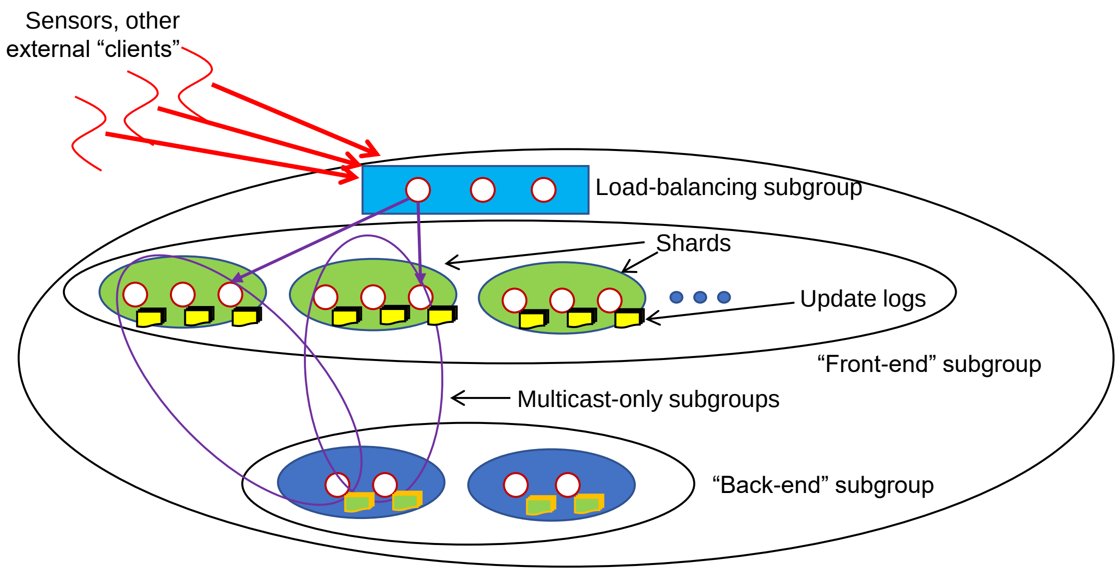


Figure 5.1: A Derecho service spanning 16 (or more) machines and containing several subsystems that employ a mix of point-to-point RPC and multicast. The ovals represent subgroups and shards within which data is replicated. Independent use of state machine replication isn't sufficient: after a shutdown, components must restart in a *mutually-consistent* state.

models as vehicles change their behavior over time. The large data sizes (photos, videos, radar, lidar) and intense time pressure (guidance is of little value if based on stale data) compel the use of accelerators, such as RDMA (which offloads data transport to hardware and achieves zero-copy transfers), NVM (which offers durable memory-mapped storage), GPU and TPU, and FPGA, without which applications would often be unable to meet performance demands [55, 68, 99].

The Derecho library [55] was created to support this new class of demanding edge applications. Derecho models the application as a collection of subgroups where each subgroup is partitioned into shards (subgroups can overlap, but shards of the same subgroup are disjoint). Each shard is a replicated state machine. The membership of the entire system is managed in a top-level group, which consists of all the nodes in the system. Figure 5.1 shows an example application. Derecho makes several key design decisions that are necessary to achieve high performance:

- Consensus off the critical path: Derecho adopts a virtual synchrony approach [23]. The top-level group membership moves through epochs (or views) where each epoch is a failure-free run of the system. Each failure triggers a reconfiguration (or view change) of the group membership. The view change involves agreement on pending updates and recomputation of the membership of each shard.
- Update all, read any single replica: An update is only committed in a shard if it has been logged at every non-failed member. Every replica has full state, enabling fast single-replica queries that do not interfere with updates. In this model, a shard can survive the failure of all but one member without losing any committed updates. This is in contrast to quorum-based protocols [65], where it suffices to update a majority of replicas, but where a query or a restart involves merging state from multiple replicas. Moreover, Derecho pipelines updates, such that each log consists of a prefix of committed updates followed by a suffix of pending updates. A reconfiguration results in a distributed log cleanup where updates that cannot be committed are discarded.
- Distributed logs: For safety, each shard member needs to log updates before they are committed. In this class of services, the "state of the application" is decentralized.

Services sometimes shut down and must later be restarted, for example when the application is migrated to different nodes, software is updated or the datacenter as a whole is serviced. Clearly we must recover each individual SMR subgroup or shard, but notice that the recovered states also need to correspond to a state the service *as a whole* could have experienced, while also preserving every committed update. This obligation is not unique to Derecho: systems like vCorfu [100] (the multi-log version of Corfu [9]) and Ceph [102] also have multiple subsystems that use sharding. Nonetheless, the problem has not previously been studied. For example, although the Derecho paper is detailed, it focuses on the efficiency of its protocols, their mapping to RDMA, and the resulting performance.

There are several factors that make restarting non-trivial:

- Failures during restart complicate the problem. We need to ensure safety under all circumstances and restore the system to a consistent, running state, equivalent to the last committed state before total failure.
- Some nodes that were once part of the system may never recover. Moreover, some restarting nodes may have failed in a view preceding the last view before the restart, in which case they will not be aware of the last membership of the top-level group. We need to determine the conditions under which a restart is possible and reconcile incomplete logs stored by shard members.
- We need to satisfy application constraints related to deployment. For example, shards may require that the members belong to different failure regions of the datacenter, impose a minimum on the number of members, and specify hardware configurations (such as number of cores, amount of memory, GPUs, etc).

The restart process should also be highly efficient to minimize application downtime. Thus we need to minimize the data transferred during restart and optimize data movement.

In this paper, we describe our restart algorithm for such systems, with configurable parameters as follows. Our algorithm requires the restarting service to designate a restart leader; it can be any restarting node. We model the failure characteristics of the nodes by organizing them into *failure correlation sets*. The application specifies the minimum number of failure correlation sets that the members of a shard should come from, for each shard of every subgroup. The application provides mappings from nodes to failure correlation sets through configuration files, making the process highly flexible; it can choose to distinguish nodes that belong to different racks or different regions of the datacenter altogether.

Our paper makes the following contributions:

1. Characterization of the state recovery problem for services composed of stateful subsystems, including a definition of correct recovery for replicated state machines that share a configuration manager.
2. An algorithm for restarting such a system from durable logs, including reasoning that argues why the algorithm is safe in the presence of any number of crashes, and live as long as any quorum of the last live configuration eventually restarts.
3. An algorithm that provably assigns nodes to shards in a way that satisfies deployment constraints and minimizes state transfer.
4. An experimental evaluation showing that a structured service can be recovered quickly

and efficiently using this algorithm.

An implementation is available in the Derecho system.

In section 5.2, we describe the restart problem at length, discussing our desirable goals for any algorithm that solves it. In section 5.3, we discuss our restart algorithm and the accompanying algorithm for assigning nodes to shards while satisfying deployment constraints. In section 5.4, we reason about the correctness of the restart algorithm and prove the node assignment algorithm correct. We show the feasibility of our approach in section 5.5 and discuss related work in section 5.6. Finally, we summarize our findings and conclude in section 5.7.

5.2 Problem Description

The essence of our problem is that independent recovery of state-machine replicated components is not sufficient. SMR guarantees that a service with $2f+1$ members can tolerate f crash failures. However a complex service with multiple subsystems and shards has many notions of f. For the service as a whole, Derecho's virtually synchronous membership protocol requires that half the members remain active from one view to the next; this prevents "split brain" behavior. But notice that in Figure reffig:derecho-services some shards have as few as 2 members. A log instance could be lost in a crash, hence such a shard must not accept updates if even a single member has crashed.

We can distinguish two cases. One involves continued activity of a service that experiences some failures, but in which many nodes remain operational. This form of *partial* failure has been treated by prior work, including the Derecho paper. In summary, if the partial failure creates a state in which safety cannot be maintained, the service must halt ("wedge") and cannot install a new view or accept further updates until the damage has been repaired.

The second case is our focus here: a full shutdown, which may not have been graceful (the service may not have been warned). To restart, we must first determine the final membership of the entire service, and the mapping of those nodes to their shard memberships in the restarted service. Then we must determine whether all the needed durable state is available, since recovery cannot continue if portions of the state are lacking, even for a single shard. Furthermore, intelligent choices must be made about the mapping of nodes to shard roles in the restarted service. On the one hand, this must respect constraints. Yet to maximize efficiency it is also desireable to minimize "role changes" that entail copying potentially large amounts of data from node to node.

In what follows, we will describe the restart problem and our algorithm for its solution in terms of a more generic system, with the hope that our techniques will be useful even in systems where Derecho is not employed.

5.2.1 System Setup

We consider a distributed system of nodes (i.e. processes) organized into subgroups partitioned into shards, in which each shard implements a virtually synchronous replicated state machine. In general, we will refer to a shard without mentioning which subgroup it belongs to, unless the distinction is important for clarity. Each shard maintains a durable log of totally ordered updates to its partition of the system state, and an update is considered committed once it is logged at every replica in the current view. As is standard in the virtual synchrony approach [23], each update records the view in which it was delivered. Also, each reconfiguration (view change) event requires every node to commit to an *epoch termination* decision which must contain, at a minimum, the highest update sequence number that can be committed in each shard, as well as the ID of the view that it terminates.

We believe this model to be quite general. Obviously, it is a natural fit for services implemented using Derecho, but it can also be applied to the materialized stream abstraction in vCorfu [100]. A vCorfu stream abstracts the action of applying a sequence of updates to a single replicated object (what we would call a shard). Moreover, vCorfu has multiple subsystems: it stores the system's configuration in a separate layout server, rather than having replicas store their own configuration. Turning to the Ceph file system [102], we find a meta-data service, a cluster mapping service, and a sharded SMR-replicated object store (RADOS). Again, the requirements are analogous to the ones we described for Derecho, with the cluster map playing the role of the view. To our knowledge, neither vCorfu nor Ceph currently addresses the issue of consistency across different shards and subsystems in the event of a full shutdown; our methods would thus strengthen the recoverability guarantees offered by these systems.

5.2.2 The Restart Problem

Our task is to ensure that the committed state of this system can be recovered in the event that every node in the system crashes in a transient way. This could be the result of a power failure or network disconnection, or an externally-mandated shutdown caused by datacenter management policies. When the system begins restarting after such a failure, we can assume that most of the nodes that crashed will resume functioning and can participate in the restart process. However, some nodes may remain failed. The system should be able to restart as long as enough of its former members participate in the restart process to guarantee that its state has been correctly restored.

Specifically, we need to restore the system to a consistent, running state, that is equivalent to its last committed state before the total failure. The restarted system must also have the same fault tolerance guarantees as it did before. This means that the restarted distributed service must (1) include every update in every shard that had reached a durably-committed state before the crash, (2) adopt a configuration that is the result of a valid view-change procedure from the system's last installed configuration, and (3) assign nodes to shards such that each shard meets its constraint of having nodes from different failure correlation sets.

The service must also be resilient against failures during the restart process, since the same transient crashes that caused it to stop can also occur during restart. It must tolerate the failure of any node in the system, detect it, and revert the system to a safe state until recovery can continue. Recovery must be able to continue from any intermediary state.

We assume that some simple external process triggers the restart procedure, such as a datacenter-management system that re-runs each interrupted program after a shutdown event. As a preliminary design choice, we will also assume that the restart procedure will be leader-based. The restarting system's first task, then, is to choose a *restart leader*. While we could elect a restart leader using standard techniques, we found it simpler and just as effective to use a preconfigured list of restart leaders installed on all nodes in the system (e.g. through a settings file). We have designed our protocol such that any node that was a member of the system at any time can serve as the restart leader, so the choice of restart leader is arbitrary and does not depend on the state of the system at the time of the total failure. As we will see in section 5.4.2, this also means that it is easy for another node to take over for the restart leader if it fails during recovery.

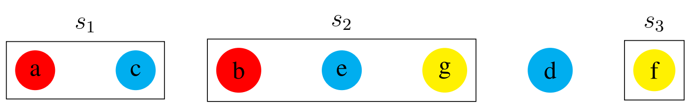
In order to restart to a consistent state, several subproblems must be addressed. First, when the restart leader starts up, it does not know whether it was a member of the last installed configuration, or whether it crashed much earlier but was nonetheless set as the restart leader; thus, its logs of both system state and the group membership could be arbitrarily out of date. Second, when the restart leader communicates with other restarting nodes, it must determine whether those nodes' configuration and state data is newer or older than its own, and whether it represents the last known state of the system, without knowing in what order the other nodes crashed. Third, for each node that restarts and has logged state updates, the restart leader must determine which updates in that log might have been externally visible and acted upon, and which were still in-progress and might never have reached a majority of replicas. Answering this question requires knowing what configuration was active at the time the update was logged, and what configuration was active at the time the system crashed. Finally, during the restart process any node could experience another transient crash, including the restart leader itself, and these crashes should not result in the system restarting in an inconsistent state or prevent the system from restarting when it has a sufficient number of healthy replicas.

The restarted system must also install a configuration that meets each shard's fault-tolerance constraints. To avoid shard shutdowns due to correlated failures, each shard is statically configured to require a minimum number of nodes from different failure correlation sets. Here, a distinction between shards of different subgroups is important, since only shards of the same subgroup are disjoint. Given a number of restarted nodes and their failure correlation sets, the restart leader must not only partition them between each subgroup's shards, but it must also (1) satisfy the minimum number of nodes required from different failure correlation sets for each shard, (2) assign as many nodes as possible to their original shards, in order to minimize the number of state transfers between nodes, and (3) compute the new assignment in a timely manner. Section 5.2.3 gives a detailed example of what is required.

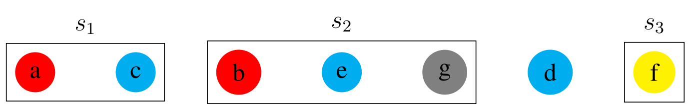
The log-recovery system we describe here addresses all of these concerns, and restarts the system as efficiently as possible by allowing each shard to complete state transfer operations in parallel.

5.2.3 Failure-Domain-Aware Assignment

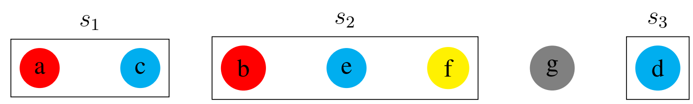
Suppose that a system has failure correlation sets f_1, f_2, f_3, such that f_1 contains nodes a and b, f_2 contains nodes c, d, and e, and f_3 contains nodes f and g. It has just one subgroup with three shards s_1, s_2, s_3, which require 2, 3, and 1 node(s) from different failure correlation sets respectively. A valid initial configuration for this system would be $s_1 = \{a, c\}$, $s_2 = \{b, e, g\}$, $s_3 = \{f\}$, leaving d unassigned to any shard. This can be represented in the following diagram, in which colors correspond to failure correlation sets:



Now suppose a shutdown occurs and all nodes except g restart. Shard s_2 is no longer in a valid configuration because it has 1 less node than it requires, but it would not suffice for the restart leader to simply add the unassigned node d to the shard because d is from the same failure correlation set as c.



An optimal reassignment is to move f from s_3 to s_2, and add d to s_3, resulting in the post-restart configuration $s_1 = \{a, c\}$, $s_2 = \{b, e, f\}$, $s_3 = \{d\}$. This reassigns only 2 nodes to new roles, which is the minimum that can be achieved while satisfying each shard's requirements.



5.3 Restart Algorithm

Having established the parameters of the restart problem, we now present our algorithm for solving it. At a high level, this algorithm has seven steps:

1. Find the last-known view by inspecting persistent logs
2. Wait for a quorum of this view to restart
3. Find the longest replicated state log for each shard
4. Compute new shard assignments and complete epoch termination from the last view, if necessary
5. Trim shard logs with conflicting updates
6. Update replicas with shorter logs
7. Install the post-restart view

However, this is not a linear process, because failures at any step after 2 can force the algorithm to return to step 2 if the quorum is lost. Also, in practice, steps 1-3 are executed concurrently by the restart leader, because it can gather information about the longest update log available for each shard while it is waiting to reach a restart quorum.

In order for log recovery to be possible, we must add a few requirements to the system described in Section 5.2.1. First, during a reconfiguration, all nodes which commit to a new view must log it to nonvolatile storage before installing it. Furthermore, in order to ensure that no updates are used in the restarted state of the system that would have been aborted by the epoch termination process, live nodes must log each epoch termination decision to persistent storage before acting upon it. Before committing to a new view, the new members of each shard must download and save the epoch termination information for the prior view in addition to the logged updates that they download during the state-transfer process.

The pseudocode for our algorithm is shown in Algorithms 1, 2, 3, and 4, where Algorithms 1 and 2 show the code that runs on the restart leader, Algorithm 3 shows the code that runs on a non-leader node, and Algorithm 4 shows the STATE_TRANSFER function that is common to both nodes. For brevity, we have factored out the leader's failure-handling code into a macro called HANDLE_FAILURE, which should be inserted verbatim wherever it is named.

In our pseudocode's syntax, the dot-operator accesses members of a data structure by name, and the bracket operator accesses members of a map by key, as in C++ or Java. Note that there are three kinds of integer identifiers: node IDs or NIDs, shard IDs or SIDs, and view IDs or VIDs. Each node has a globally unique node ID, and, as is common in virtual synchrony, view IDs are unique and monotonically increasing. Shard IDs are unique identifiers assigned to each shard (globally, across all subgroups) of the system. In the following sections, we will explain the details of the algorithm, which should make the

Algorithm 1 The restart leader's behavior, part 1

$V_c \leftarrow$ READ(view_log)
$restarted \leftarrow \{nid_{me}\}$
$u_e \leftarrow$ READ(update_log)$.end$
$LL \leftarrow \{V_c.my\_sid \rightarrow (nid_{me}, u_e.seqno)\}$
$ET \leftarrow$ READ(epoch_termination_log)
while $\neg$ QUORUM($V_c, restarted$) **do**
  $(V_i, nid_n, sid, seqno) \leftarrow$ RECEIVE from n
  $restarted \leftarrow restarted \cup \{nid_n\}$
  if $V_i.vid > V_c.vid$ **then**
    $V_c \leftarrow V_i$
    WRITE(view_log, V_c)
    $ET \leftarrow \{\}$
  if $LL[sid].seqno < seqno$ **then**
    $LL[sid] \leftarrow (nid_n, seqno)$
  $et \leftarrow$ RECEIVE from n
  if $et \neq \{\} \wedge et.vid = V_c.vid$ **then**
    $ET \leftarrow et$
    WRITE(epoch_termination_log, ET)
$V_r \leftarrow$ CHANGE_VIEW($V_c, restarted$)
if $ET = \{\}$ **then**
  $ET.vid \leftarrow V_c.vid$
  for all $s \in V_r.subgroups$ **do**
    $ET.last[s.sid] \leftarrow LL[s.sid].seqno$
$sent \leftarrow \{\}$
for all $s \in V_r.subgroups$ **do**
  for all $nid_n \in s.members$ **do**
    $success \leftarrow$ SEND($V_r, ET, LL[s.sid].nid$) to n
    if $\neg success$ **then**
      HANDLE_FAILURE($nid_n, sent$)
    $sent \leftarrow sent \cup \{nid_n\}$

Algorithm 2 The restart leader's behavior, part 2

```
if ET.vid = u_e.vid then
    success ← SEND(∅) to LL[V_r.my_sid].nid
    trim_seqno ← ET.last[V_r.my_sid]
else
    success ← SEND(u_e.vid) to LL[V_r.my_sid].nid
    trim_seqno ← RECEIVE from LL[V_r.my_sid].nid
if ¬success then
    HANDLE_FAILURE(LL[V_r.my_sid].nid, restarted)
TRUNCATE(update_log, trim_seqno)
success ← STATE_TRANSFER(
    LL[V_r.my_sid].nid, nid_me, V_r)
if ¬success then
    HANDLE_FAILURE(LL[V_r.my_sid].nid, restarted)
sent ← {}
for all nid_n ∈ V_r.members do
    success ← SEND("Prepare") to n
    if ¬success then
        HANDLE_FAILURE(nid_n, sent)
for all nid_n ∈ V_r.members do
    SEND("Commit") to n
WRITE(view_log, V_c)

procedure HANDLE_FAILURE(nid, notify_set)
    restarted ← restarted − {nid}
    for all nid_m ∈ notify_set do
        SEND("Abort") to m
    if ¬ QUORUM(V_c, restarted) then
        goto 6
    else
        goto 19
```

pseudocode more clear.

5.3.1 Awaiting Quorum

The restart leader's first operation is to read its logged view, which becomes the first "current" view, V_c, and its logged epoch termination information, which becomes the currently-proposed epoch termination, ET. It then begins waiting for other nodes to restart and contact it; non-leader nodes will contact the preconfigured restart leader as soon as they restart and discover that they have logged system state on disk.

When a non-leader node contacts the leader, it sends a copy of its logged view, V_i, its node ID, the ID of the shard it was a member of during V_i, and the sequence number of the latest update it has on disk. The joining node may optionally then send a logged epoch termination structure, if it has one that is as new as its logged view. The leader updates V_c and possibly ET if the client's view and epoch termination are newer, and uses data structure LL (a map from shard IDs to pairs of node IDs and update sequence numbers) to keep track of the location of the longest log for each shard. Note that sequence numbers from later views are always ordered after sequence numbers from earlier views.

After each node restarts, the leader checks to see if it has a *restart quorum*. A restart quorum consists of a majority of the members of the system in the last known view that includes at least one member of every shard from that view. In addition, the restart leader must be able to install a new post-restart view in which the entire group has at least $f + 1$ replicas to meet the overall fault-tolerance threshold, and each shard is populated by nodes

Algorithm 3 A non-leader node's behavior

$V_c \leftarrow$ READ(view_log)
$et \leftarrow$ READ(epoch_termination_log)
$u_e \leftarrow$ READ(update_log)$.end$
SEND($V_c, nid_{me}, V_c.my\_sid, u_e.seqno$) to leader
if $et \neq \{\} \wedge et.vid = V_c.vid$ **then**
  SEND(et) to leader
$commit \leftarrow \perp$
while $\neg commit$ **do**
  $(V_r, ET, nid_\ell) \leftarrow$ RECEIVE from leader
  $et \leftarrow ET$
  if $et.vid = u_e.vid$ **then**
    $success \leftarrow$ SEND($\varnothing$) to ℓ
    $trim\_seqno \leftarrow et.last[V_c.my\_sid]$
  else
    $success \leftarrow$ SEND($u_e.vid$) to ℓ
    $trim\_seqno \leftarrow$ RECEIVE from ℓ
  if $\neg success$ **then**
    continue
  TRUNCATE(update_log, $trim\_seqno$)
  $success \leftarrow$ STATE_TRANSFER(nid_ℓ, nid_{me}, V_r)
  if $\neg success$ **then**
    continue
  $p \leftarrow$ RECEIVE from leader
  if $p =$ "Prepare" **then**
    $d \leftarrow$ RECEIVE from leader
    $commit \leftarrow (d =$ "Commit"$)$
$V_c \leftarrow V_r$
WRITE(view_log, V_c)

that meet its failure-correlation requirements. Note that the post-restart view can add new members that were not part of the last known view, since nodes that failed in an earlier view but restarted after the system-wide failure can still participate in the recovery process.

Once the leader has reached a restart quorum, if the newest epoch termination structure it has discovered is from an older view than V_c, it makes its own decision about how to terminate V_c's epoch. Specifically, it synthesizes an epoch termination structure by taking the sequence number of the latest update for each shard, and marking it with the same VID as V_c. It then computes V_r, the next view to install after restarting.

In practice, the leader waits for a short "grace period" after a quorum is achieved to allow nodes that restarted at a slightly slower rate to be included in V_r. This makes it less likely that V_r will require many node reassignments (and hence state transfers), and has only a minor effect on restart time.

5.3.2 Assigning Nodes to Shards

When testing for a restart quorum and computing V_r, the leader must determine an optimal assignment from nodes to shards. Since the shards of a subgroup must be disjoint, it can consider each subgroup individually. For each subgroup, the leader computes the assignment of nodes to shards in V_r by solving an instance of the min-cost flow problem [8].

It first creates a bipartite graph from shards to failure correlation sets as follows: For each shard there is a vertex s_i, and for each failure correlation set (FCS) there is a vertex

fcs_j. There is one "shard" vertex u representing unassigned nodes, one source vertex, and one sink vertex. If m_i is the required number of nodes from different failure correlation sets for shard i, then there is an edge from the source vertex to s_i with cost 0 and capacity m_i. An edge with cost 0 and capacity 0 extends from the source vertex to u. An edge extends from each shard vertex s_i to each FCS vertex fcs_j, with cost 0 if shard i contained a node from FCS j in V_c, cost 1 otherwise, and capacity 1. For vertex u, these edges always have cost 0 and capacity 1. Finally, there is an edge from each FCS vertex fcs_j to the sink vertex with cost 0 and capacity equal to the number of nodes in FCS j in V_r.

The leader solves min-cost flow on the generated bipartite graph, increasing flow along augmenting paths until all shard vertices s_i have at least m_i flow and a solution is generated, or no augmenting path can be generated for the graph. If a solution is generated, then the leader translates that solution into a node assignment, where shard i is assigned one node from failure correlation set j if an edge contains flow between vertices s_i and fcs_j. If min-cost flow halts without a solution, then there is no solution that satisfies m_i for all shards, and there is not yet a restart quorum.

5.3.3 Completing Epoch Termination

For each shard, the restart leader sends to each node that will be a member in V_r the identity of the node on which the latest update for that shard resides (denoted node ℓ), as well as V_r itself and the epoch termination information.

When sending this information to node n, the restart leader might discover that n has

Algorithm 4 The state-transfer function

function STATE_TRANSFER(nid_ℓ, nid_{me}, V_r)
  if $nid_\ell = nid_{me}$ **then**
    $UL \leftarrow$ READ(update_log)
    for all $n \in V_r.shards[V_r.my\_sid]$ **do**
      $vid_n \leftarrow$ RECEIVE from n
      if $vid_n \neq \varnothing$ **then**
        $seqno_n \leftarrow$ FIND_MAX(UL, vid_n)$.seqno$
        $succ_1 \leftarrow$ SEND($seqno_n$) to n
      $seqno_e \leftarrow$ RECEIVE from n
      $succ_2 \leftarrow$ SEND($\{UL[seqno_e], \ldots UL.end\}$) to n
      if $\neg succ_1 \vee \neg succ_2$ **then**
        return $\bot$
  else
    $u_e \leftarrow$ READ(update_log)$.end$
    $success \leftarrow$ SEND($u_e.seqno$) to ℓ
    if $\neg success$ **then**
      return $\bot$
    $\{u_{e+1}, u_{e+2}, \ldots, u_\ell\} \leftarrow$ RECEIVE from ℓ
    APPEND(update_log, $\{u_{e+1}, u_{e+2}, \ldots, u_\ell\}$)
  return $\top$

crashed because it does not respond to the leader's connection attempts (we assume TCP-like semantics for our network operations). In this case, the leader removes n from the set of restarted nodes, sends an "Abort" message to all the nodes that have already received its message, and recomputes whether it has a restart quorum. If there is still a restart quorum, the leader recomputes V_r and starts over at sending ET and V_r to each live node. If not, it returns to step 2 and waits for additional nodes to restart.

Meanwhile, when a non-leader node receives V_r, ET, and nid_ℓ, it compares ET's view ID to the view ID associated with its last logged update. If these IDs match, the node completes epoch termination by deleting from its update log any updates with a sequence number higher than the last commit point for its shard. If the epoch termination structure is from a later view, though, all the updates in the node's log are from an earlier view that might have had its own epoch termination. In order to ensure that it also trims any updates that were aborted by the earlier epoch termination, the node contacts node ℓ and sends it the VID of its last logged update. Node ℓ, upon receiving this message, inspects its update log and finds the last update with that VID, then replies with that update's sequence number. The sending node then deletes from its log any updates with a higher sequence number. (Node ℓ's behavior in this exchange is implemented in the STATE_TRANSFER function).

5.3.4 Transferring State

Once each node, including the leader, has truncated from its log any updates that would have aborted, it must download any committed updates that are not in its log. Each node that has been designated as the location of the longest log must, conversely, listen for connections from the other nodes that will be members of its shard in V_r and send them the updates they are missing. This is shown in the `STATE_TRANSFER` function in Algorithm 4. In this phase, a non-leader node may discover that the node with the longest log has failed when it attempts to contact it. In that case, the node can conclude that the V_r it has received from the leader will not commit, and return to waiting to receive a new V_r and longest-log location from the leader.

5.3.5 Committing to a Restart View

When a non-leader node finishes its state transfer operations, it awaits a "prepare" message from the leader. Meanwhile, when the leader has finished its own state transfer operations, it begins sending "prepare" messages to each node. If it discovers while sending these that a node has crashed (because the connection is broken), it sends an "abort" message to all nodes that it has already sent "prepare" messages to, and recomputes the post-restart view to exclude the crashed node. The leader might then discover that it no longer has a sufficient quorum for restart without the crashed node, in which case it returns to step 2 and waits for additional nodes to restart. If it still has a quorum, however, the leader can return to step 3, calculating the new shard membership and sending the new V_r and longest log location to all nodes. Once the leader has successfully sent "prepare" messages to all nodes in V_r, it can send a "commit" message to all of them confirming that this view can be installed. Once a non-leader node receives the leader's commit message, it can install V_r and begin accepting new messages and committing new updates. At this point, the restart leader no longer has a leader role, and all future failures and reconfigurations can be handled by the normal view-change protocol for a running system.

5.4 Analysis

We will now prove that this protocol satisfies the goals we set out in section 5.2.2. We first show that the protocol is correct in the case where there are no failures during the restart process, and then show that failures of any node do not affect its correctness.

Regardless of which view the restart leader has logged on disk when it first starts up, it is guaranteed to discover the last view that was installed in the pre-crash system before it exits the await-quorum loop, because a restart quorum requires a majority of nodes from the current view V_c to contact it. The view-change protocol in virtual synchrony requires a majority of the members of the current view to be members of the next view, which means that if the restart leader starts with some obsolete view V_k, a majority of members of V_k were also members of V_{k+1}, and the restart leader will discover at least one member of V_{k+1} by waiting for a majority of members of V_k. When a member of V_{k+1} restarts, it will send V_{k+1} to the leader, which will then use V_{k+1} as V_c and begin waiting for a majority of V_{k+1}'s members. If V_{k+1} is not the latest view, then by the same logic, the leader is guaranteed to discover V_{k+2} on at least one of the members of this majority. Thus, the leader must have discovered and installed the last known view V_ℓ by the time it has satisfied the quorum condition of contacting a majority of V_c.

Furthermore, by the time the leader exits the await-quorum loop, it is guaranteed to discover at least one log containing all committed updates for each shard in the system. This is because an additional condition of a restart quorum is that the leader must contact at least one member of each shard according to V_c. As we have just shown, V_c must equal V_ℓ before the majority condition of the quorum can be satisfied, so the leader will contact at least one member of each shard in V_ℓ. Since updates that commit in a view are by definition logged on every member of a shard in that view, any node that was a member of a shard in V_ℓ will have a log containing all committed updates for that shard up to the point of the total crash. Thus, every shard will have a designated longest-log location that contains all of its committed updates by the time the leader exits the await-quorum loop. Recovery into a mutually consistent state follows because membership epochs are totally ordered with respect to SMR events in shards or subgroups: the end of each epoch is a *consistent cut* [30].

The epoch termination decision ET that the leader sends out after achieving quorum is guaranteed to preserve any decision made by the group prior to the crash, and to include only updates that were safe to commit. Since the system's epoch termination process (as augmented in section 5.3) requires all members of a view to log the epoch termination decision before acting on it, by the time the leader reaches a majority of V_ℓ, it must find at least one copy of the epoch termination information that was computed for V_ℓ if any node acted upon it. Using this epoch termination structure as ET preserves the decision made by V_ℓ's reconfiguration leader about which updates to include. Conversely, if the leader does not find any epoch termination information for V_ℓ, then no node had yet delivered or aborted any updates that were in-progress at the time of the crash. This means it is safe for the restart leader to construct ET using the longest sequence of updates that is available on at least one node in each shard, and unilaterally decide to commit any pending updates at the tail of that log. Before any node installs a view in which those updates are committed (V_r), the state transfer process ensures that any pending updates are fully replicated to all members of their shard. Thus, for each shard, every update up to the last commit point in ET will be present on all members of that shard in the new view, which is the same guarantee provided by the epoch termination process during a normal run of the system.

Finally, the post-restart view V_r that the leader installs is guaranteed to have the same stability and durability guarantees as any other view in the running system. As we just showed, all nodes that will be members of a shard in V_r will have the exact same update log for that shard before V_r is installed, which means that the updates committed in V_r are just as fault-tolerant as updates in any prior view. V_r itself is also durable, and guaranteed to be recovered by a future restart leader during the recovery process, because a majority of members of V_ℓ are also members of V_r.

5.4.1 Tolerance of Failures of Non-Leaders

Our approach to failed non-leader nodes is to treat them as nodes that have not yet restarted. Upon detecting a failure at any point after reaching a restart quorum, the leader removes the failed node from its $restarted$ set, and recomputes both V_r and whether it has a restart quorum. By sending an "abort" message to all other nodes that may already have received V_r, the leader ensures that they will return to waiting for V_r and the epoch termination information. Regardless of how many times nodes fail and restart during the restart process, the leader still cannot proceed past the await-quorum loop until it has reached a restart quorum, which means it must reach at least one node from each shard that has all the committed updates for that shard. Since nodes never truncate updates from their logs that had actually committed in V_ℓ (due to the correctness of the epoch termination procedure), and committed updates were present on every member of their shard in V_ℓ, this will always be possible as long as enough members of V_ℓ eventually restart.

It is safe for the nodes that received ET and V_r from the leader before it detected a failure to begin the epoch termination and state transfer process, because at the point the leader started sending V_r it had reached a restart quorum. This means that ET only included updates that were safe to commit, and only excluded updates that had definitely aborted. Although V_r will change whenever there is a failure, the only way that ET could change after a failure is to include or exclude a different number of pending-but-uncommitted updates at the tail of a shard's log, and that will only happen if the node that failed was the location of the longest log for a shard. In that case, the new ET may include fewer of the uncommitted updates at the tail of the shard's log, but it is equally safe to abort these updates, since they had not yet committed at the end of V_ℓ. Nodes that had downloaded some of these updates at the time of the failure will simply truncate them when they re-run the epoch termination process.

The two-phase commit at the end of the state-transfer process ensures that all of the nodes in V_r are still live and have finished state transfer before any of them can commit to V_r. This ensures that no node can begin acting on V_r until all of the updates committed by ET are fully replicated.

5.4.2 Tolerance of Failure of the Leader

Much of our restart protocol seems to depend on correct operation of the restart leader, but in fact it can tolerate the failure of the restart leader: a subsequent restart leader would always select a state that is a safe extension of the state of the original leader (in fact it will be the identical state if the original leader's proposal might have been acted upon, and otherwise will be a safe choice with respect to the state the system was in when it crashed). One caveat is that our solution is correct only with a single leader running at a time. Since no fault-tolerant configuration management system is yet in place while the system is restarting, choosing a restart leader with an election protocol would be quite difficult. However, a small amount of manual configuration can to be used both to choose the initial leader and to select one to take over if the initial one fails. This can be accomplished by, for example, specifying both a default restart leader and an ordered list of fallback restart leaders in a configuration file. Handling the failure of the leader in an efficient manner may also require some manual intervention, specifically in the case where the leader fails during the await-quorum loop, because non-leader nodes can expect to wait a rather long time for the leader to reach a quorum (depending on how long it takes nodes in the system to restart after a total crash). They can eventually conclude that the leader has failed if it does not send ET after a suitably long timeout, but the restart process can complete faster if a system administrator or other outside process forcibly restarts them if the leader fails while awaiting a quorum. Failures of the leader during the 2-phase commit are easier to detect, because the leader should send the prepare and commit messages shortly after sending the ragged trim information, so the non-leader nodes can safely use much shorter timeouts on these messages.

When non-leader nodes detect that the leader has failed, they restart the recovery process using the new restart leader. This means that the new restart leader receives all of the same view, epoch-termination, and update-log information as the previous restart leader, and will reach the same conclusions. It will still wait for majority of members of each view it discovers to restart, meaning it must discover the last known view before it concludes that it has a restart quorum. If the previous restart leader was in fact required to achieve a quorum (because, for example, it was the only member of some shard in V_ℓ), then the new restart leader must wait for it to restart and rejoin the system as a non-leader.

5.4.3 Correctness of Node Assignment to Shards

Next, we prove that our min-cost flow algorithm finds a node assignment that satisfies each shard's required number of nodes from different failure correlation sets, given that a node assignment exists that obeys this constraint. We also show that our algorithm is optimal, generating an assignment where a minimal number of nodes are moved to shards they were not previously a part of. Thus, we minimize time spent on state transfer between old and new members of each shard.

We prove correctness by reduction to min-cost flow. Our solution is correct if it finds a feasible node assignment given that one exists. Given capacities of edges from the source vertex to shard vertices, shard vertices to failure correlation set vertices, and failure correlation set vertices to the sink vertex, any feasible flow in the graph can be translated into a feasible assignment of nodes to shards. Each shard receives exactly the number of nodes from different failure correlation sets it requires, because that is the capacity of the edge from the source to the shard vertex. No node from any failure correlation set is assigned to more than one shard, because the capacity of the edge from the failure correlation set vertex to the sink is number of nodes in that failure correlation set. All nodes assigned to any one shard are from different failure correlation sets, because the capacity of the edges from shard vertices to failure correlation set vertices is always 1. Thus a solution to min-cost flow is a solution to the node assignment problem. In fact, any solution to the node assignment problem can also be translated into a flow.

Furthermore, the solution to min-cost flow represents an optimal node assignment. We defined optimality above; a solution is optimal if it minimizes the number of nodes whose shard membership changes. By definition, the solution to min-cost flow is a flow that minimizes the cost along all its edges. Costs along edges are 0 except for edges from shard vertices to failure correlation set vertices, where no member from the failure correlation set belonged in the shard in the previous view. That is the definition of optimality. Thus any solution to min-cost flow is optimal.

Note that we opted to reduce to min-cost flow, which can be solved in polynomial time, instead of integer linear programming, which can be used to satisfy more generic constraints but might not find a solution efficiently.

5.4.4 Efficiency and Generality

Our restart protocol is designed for a particular form of state machine replication (the one implemented by Derecho), which allows us to take advantage of some efficiencies built into this SMR protocol. Specifically, Derecho's SMR enforces a read quorum of 1 within each shard, which means that reading the log of one up-to-date replica is sufficient to learn the entire committed state of that shard. Thus, the restart quorum only requires a single member of each shard from the last known view, and when new or out-of-date replicas are added to a shard during restart, they only need to contact and transfer state from a single up-to-date member. Furthermore, uncommitted updates only occur at the tail of a log, and there are no "holes" in the committed prefix of the log because updates are only aborted during a reconfiguration (which also trims them from the log). This allows us to easily make the correct decision about whether to accept these updates during recovery: they can safely be committed unless a logged epoch termination decision is found that proves they will be aborted.

Nevertheless, our protocol could be applied to other forms of SMR with a few relaxations of these optimizations. For example, a read quorum > 1 would merely increase the size of the restart quorum, as long as reconfiguration was still handled via virtual synchrony. In a system with a per-shard read quorum of r_i, the restart leader would need to contact at least r_i members of shard i in the current view in order to ensure it found both the next view (if one exists) and the longest sequence of committed updates in shard i; the restart quorum would include a read quorum of every shard in the last known view. Any nodes added to a shard in the restart view would also need to contact all the members of the most-recent read quorum in order to complete state transfer.

Some SMR systems, such as vCorfu [100], separate configuration information from the replicated state itself, using a separate "layout" service and "data" service. In this case, our protocol would need to explicitly separate step 1 (finding the last configuration) from step 3 (finding the longest log), rather than executing them concurrently. The restart leader would first need to contact a quorum of the layout service in order to find the last active configuration, then use that configuration to compute and wait for a restart quorum of the data service.

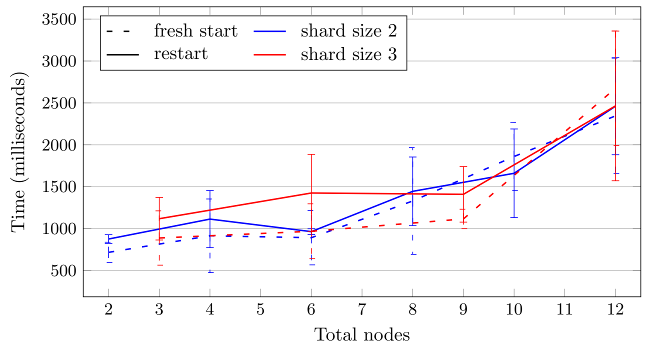


Figure 5.2: Total time to start or restart a service. Error bars represent 1 standard deviation.

5.5 Experiments

We have implemented our restart algorithm as part of the Derecho library, and in this section we measure its performance when restarting sample Derecho applications. All experiments were carried out on our local cluster, which contains 12 servers running Ubuntu 16.04, using SSD disks for storage. In summary, we found that our recovery logic scales well, and adds only a small delay compared to the costs of process launches and initial Derecho platform setup.

Our first experiment was a straightforward end-to-end benchmark. We used our algorithm to restart a simple Derecho service with a single subgroup and shards of 2 or 3 nodes each, after an abrupt crash in which all nodes failed near the same time, and measured the time from when the restart leader launched to when the first update could be sent in the recovered service. For comparison, we also measured the time required to start a fresh instance of the same service, with no logged state to recover. Figure 5.2 shows the results.

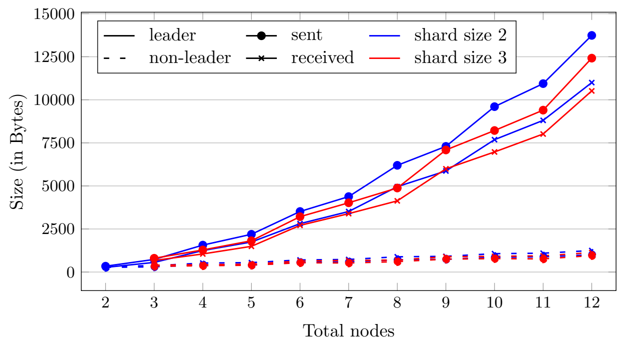


Figure 5.3: Total metadata sent/received during the restart process.

We find that the restart algorithm adds only minimal overhead compared to the fresh-start case, and that the assignment of nodes into more or fewer shards does not have a noticeable effect on restart time, owing to the polynomial run time of min-cost flow. In both cases, the time to launch the service increases as the system scales up due to the fixed costs of initializing more distributed processes. For example, there is an increasingly variable delay in the time it takes each server to actually start the Derecho process after being given a command to do so.

Next, we measured the amount of metadata that was exchanged between the restart leader and the non-leader nodes in order to complete the restart algorithm, using the same setup as the experiment in Figure 5.2. (Metadata includes everything sent during the restart process except for the missing updates sent during state transfer). In Figure 5.3, we see that the restart leader sends and receives more metadata as the size of the overall group increases, increasing at an approximately linear rate. This is because the restart leader must contact every restarting node, both to receive its logged information and to send out the proposed restart view. However, the non-leader nodes exchange a nearly-constant amount

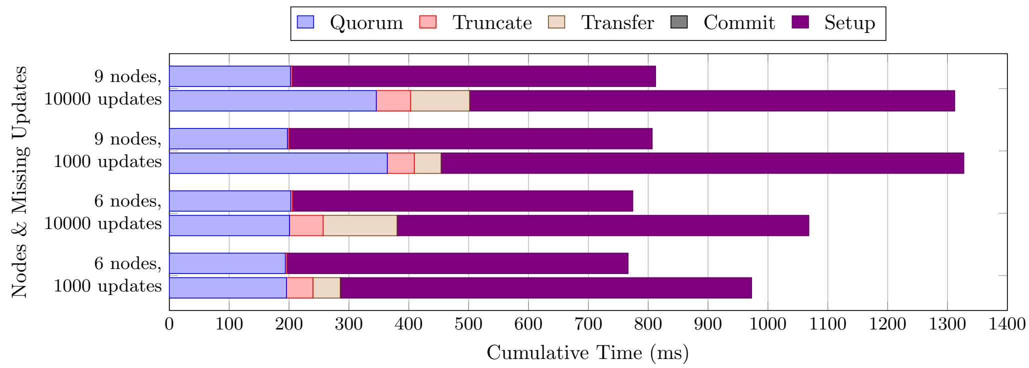


Figure 5.4: Breakdown of time spent in each phase of starting or restarting a service, when 1 node per shard is out of date upon restart. Upper bars show fresh start, lower bars show restart.

of data regardless of the size of the group, since they only need to contact the leader and wait for its response. Note, also, that even at the largest group sizes, the leader only needs to receive a few kilobytes of data, aggregated over all of the restarting nodes.

In our next series of experiments, we evaluated the costs of restarting a system with one or more significantly out-of-date replicas (i.e. nodes whose logs are missing many committed updates). To do this, we created a Derecho service organized into shards of 3 nodes each, and allowed two out of three replicas in each shard to continue committing updates for some time after one replica had crashed. We then crashed the rest of the replicas, and restarted all of the nodes at once. Each update in this service contained 1KB of data.

Figure 5.4 shows a detailed breakdown of the amount of time spent in the four major phases of the restart algorithm in this situation: (1) awaiting quorum, (2) truncating logs to complete epoch termination, (3) transferring state to out-of-date nodes, and (4) waiting

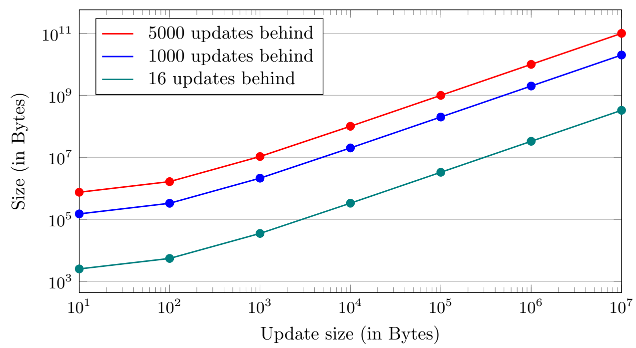


Figure 5.5: Data downloaded by each out-of-date node, in a system with 3 shards of 3 members each.

for the leader to commit a restart view. It also shows a fifth phase, which is the time spent in the setup process of the Derecho library before the first update can be sent; this includes operations such as pre-allocating buffers for RDMA multicasts. For comparison, we also measured the breakdown of time spent in a fresh start of the same service, which has only two phases: Awaiting quorum (i.e. waiting for all the processes to launch) and setting up the Derecho library.

This experiment shows even more clearly that our restart process is quite efficient compared to the normal costs of starting a distributed service. Even when one replica in each shard is missing 10000 committed updates, state transfer accounts for at most 120 ms, a small fraction of the overall time. It also shows the benefits of allowing each shard to complete state transfer in parallel: The 3-shard service spent no more time on state transfer than the 2-shard service, even though there were an additional 1000 or 10000 updates to send to an out-of-date node.

We also measured the number of bytes of data received by each out-of-date replica

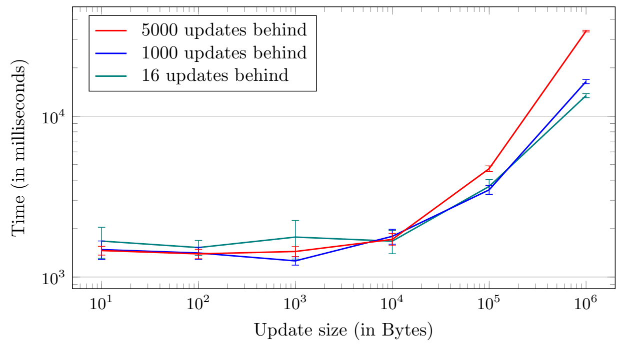


Figure 5.6: Time to restart a service with 3 shards of 3 members each, with 1 out-of-date node per shard. Error bars represent 1 standard deviation.

during the state-transfer process, and varied the amount of data contained in each update as well as the number of missing updates. The results are shown in Figure 5.5, and are fairly straightforward: the amount of data transferred to each out-of-date replica increases linearly with the size of an update, and with the number of updates that the out-of-date replica has missing from its log. Moreover, it is almost exactly equal to the number of missing updates multiplied by the size of each update, because the node did not need to download and merge logs from multiple other replicas. It is also important to note that this data is sent in parallel for each shard, so unlike the metadata in Figure 5.3, there is no difference in how much data any one node must send as the number of shards increases.

Finally, we measured the amount of time required to restart a service with out-of-date replicas as the size of each update scales up, shown in Figure 5.6. We found that for updates of sizes below 1 MB, neither the size of the update nor the number of missing updates on the out-of-date replicas had much of an effect on the restart time. For update sizes of 1MB and larger, the increasing amount of data that needed to be transferred to the out-of-date replicas had the expected effect of slowing down the restart process.

5.6 Related Work

The algorithms implemented by Derecho combine ideas first explored in the Isis Toolkit [20, 23] with the Vertical Paxos model [66]. Other modern Paxos protocols include NOPaxos [68] and APUS [99]. Recent systems that offer a more durable form of Paxos, such as Spinnaker [85] and Gaios [25], include mechanisms for restarting failed nodes using their persistent logs. However, these papers generally do not consider the case in which every replica must be restarted at once. "Paxos Made Live" [28] explores a number of practical challenges (including durability) seen in larger SMR systems, a motivation shared by our work.

Bessani et al. looked at the efficiency of adding durability to SMR in [14], including the problem of minimizing state transfer during replica recovery. They provided a solution for recovering a non-sharded service in a Byzantine setting, and also showed how to lower the runtime overhead of logging and checkpointing. Their work did not look at services with complex substructure, which was a primary consideration here.

Corfu [9] is a recent implementation of SMR that uses a different approach from classic Paxos, distributing the command log across shards of storage-only nodes. Clients use Paxos to reserve a slot, then replicate data using a form of chain replication [96]. vCorfu [100] extends this by offering virtual sublogs on a per-application basis. However, if multiple subsystems use Corfu separately, recovery of the Corfu log might not recover the application as a whole into a consistent state. As we mentioned in sections 5.2.1 and 5.4.4, our protocol could be adapted to vCorfu to ensure that a quorum of replicas from each sublog of the last known layout is contacted before the system is restarted. Other replicated cloud services, such as Hadoop [89], Zookeeper [53], and Spark [103], employ an alternative approach to durability by ensuring that any state lost due to an unexpected failure can always be recomputed from its last checkpoint, but this is not an option in our setting.

Our work is inspired by a long history of distributed checkpointing and rollback-recovery protocols, many of which are summarized in [41], but updates these principles to the modern setting of replicated services and SMR. Rather than rely on an explicitly coordinated global checkpoint, as in [62] and [42], or attempt to record a dependency graph between locally-recorded checkpoints, as in [15], our system incorporates the dependency information already recorded in SMR updates to derive a globally consistent system snapshot from local logs.

Recovery of the final state of a single process group was first treated in Skeen's article "Determining the Last Process to Fail" [90]. Our scenario, with potentially overlapping subgroups, is more complex and introduces an issue of joint consistency they did not explore.

5.7 Conclusion

Modern datacenter services are frequently complex, and may employ SMR mechanisms for self-managed configuration, membership management, and sharded data replication.

In these services, application data will be spread over large numbers of logs, and recovery requires reconstruction of a valid and consistent state that preserves all committed updates. We showed how this problem can be solved even if further crashes occur during recovery, implemented our solution within Derecho, and evaluated the mechanism to show that it is highly efficient.

Acknowledgments

This work was supported, in part, by a grant from AFRL Wright-Patterson.

CHAPTER 6

CONCLUSION

In this dissertation, we presented our work on Derecho, a C++ library that enables the development of reliable applications designed to fully leverage fast, modern networks for high performance. We described:

- **RDMC**: Derecho's data plane, which provides high-speed software-level multicast functionality.
- **SST**: Derecho's control plane, enabling asynchronous, lock-free coordination between application processes. As part of this we described SST multicast (SMC), Derecho's multicast protocol for small messages.
- Derecho's design and implementation, which efficiently combines RDMC, SST, SMC, and other functionality such as automated membership management, P2P communication, persistent storage with versioning, etc.
- **Spindle**: Our set of optimization techniques that improve Derecho's performance for SMC and other cases involving small messages.
- Techniques for automated self-repair and automated restart of a Derecho application using state stored on disk after a damaging crash or a full shutdown.

Although not discussed in this dissertation, Derecho is already in use as an infrastructure to support other research efforts, as the lower layer of an extensible storage layer for cloud intelligence under time-pressure or with real-time constraints, and in an OMG-compatible publish-subscribe system.

This chapter provides a summary of Derecho's features and performance benefits, presents an overview of current research based on Derecho, and suggests ideas for related future work.

6.1 Features summary

Derecho is aimed at supporting complex cloud applications that need to process large amounts of data in order to take fast and consistent actions. The application model is similar to the micro-services architecture – application components are modeled as subgroups of the overall (top-level) group consisting of all application nodes. Each subgroup is further partitioned into one or more disjoint shards. Changes in the top-level group membership due to new nodes joining or existing nodes leaving or failing result in a reassignment of nodes to the shards. This is done to ensure that each subgroup shard meets any fault-tolerance requirements, e.g., having a minimum number of nodes. The assignment function can either be supplied by the application for customized behavior or automated by Derecho. For the latter, the application can specify the minimum number of failure correlation sets each shard must cover. For the initial assignment and every reassignment henceforth, Derecho then computes a feasible assignment minimizing the amount of nodes that need to move between the shards. This is aimed to minimize the data and state that needs to be transferred during a reconfiguration of the system. When such a feasible assignment does not exist, Derecho simply waits for more nodes to be added to the system.

Each shard's state can be represented entirely in memory (in-memory mode) or persisted on disk. Additionally, Derecho can keep track of the history of states in what is called a version vector. This allows Derecho to respond to queries that seek a state corresponding to a version number or a time in the past.

Derecho offers state machine replication to each application shard. The application developer supplies a C++ class for each subgroup. An instance of this class is replicated at every member of each shard of the subgroup. Operations to this state are modeled as the member functions of the class. A subset of the nodes of a given shard can be designated as senders. Each sender can then send a message in the shard that contains an RPC stub specifying the function and the arguments. Derecho delivers these messages safely in the same order at all the shard members, when it is guaranteed that the message will not be lost amidst failures. This requires that all nodes acknowledge the message receipt to everyone in the shard. The state machine operations are run when the corresponding message is delivered. Optionally, this leads to the logging of the state in persistent storage. A shard can also be used to replicate raw data, without any state machine operations. Additionally, Derecho calls back into the application notifying it of the delivery through a preregistered callback.

When a failure (or a node join or leave) disrupts the system, Derecho pauses the message delivery pipeline (send, receive, delivery) across all shards in order to reconfigure. During reconfiguration, it cleans up the messages in the system. This means that stable messages are delivered and others are discarded. A discarded message can be delivered after the reconfiguration given that the sender of the message is still a member of the shard. A node that joins a shard receives the existing state of the shard and (optionally) a log of past states, through a state transfer process.

When a majority of nodes in the top-level group fail, Derecho is forced to shut down the remaining nodes in order to avoid split-brain behavior. In other cases too, when the entire application needs to shut down, for example, to update the application code, Derecho offers reliable, automated recovery. The recovery process requires a majority of the nodes from the last configuration to restart with access to state stored on disk before the shutdown. This process is resilient to failures during recovery and reassigns nodes to shards respecting application's assignment constraints.

Derecho also supports querying a shard's state through a top-level group member or an "external" client. This makes the state available to nodes that are not members of the shard, enabling aggregation of states across multiple shards, for example. Similarly, it is possible to initiate state operations from non-members. For instance, a sensor with data to send (external client) can send it to one of the members which can then replicate the data in its shard.

When strong consistency is not desired, Derecho shards can be used in the unreliable mode. Here, the messages are delivered as soon as they are received and are not guaranteed to be delivered to zero or all members (i.e., no atomicity) when failures occur.

Finally, Derecho supports both RDMA and TCP networks through its use of the libfabric [78] library. Libfabric offers a consistent RDMA interface on many different network providers, making it possible to extend Derecho in the future to other network environments such as Intel's omni-path. We have tested Derecho exclusively with TCP and Mellanox's verbs fabric provider.

Cluster	Fractus (100Gbps RDMA, 100Gbps TCP), Stampede (56Gbps RDMA)
Message size	Small (10KB), Large (1MB, 100MB)
Number of subgroups	1-128
Number of senders	one, half, all
Persistent levels	In-memory (no persistence), Volatile (Linux RamDisk), Persistent (SSD)

Table 6.1: Experiment Settings

6.2 Performance summary

We tested Derecho extensively under a variety of settings. Table 6.1 provides a summary. Most results cited below are produced on 16 nodes of the Fractus cluster with 12.5GB/s (100Gbps) RDMA unless otherwise indicated. The focus of this section is exclusively on Derecho's in-memory performance for raw data replication which I contributed the most to. Other results, including performance with logging state on disk and sending RPC stubs as messages (involving marshaling and demarshaling of arguments), can be found in the evaluation section of Chapter 3. Additionally, the results below are presented for subgroups that contained just a single shard. Therefore, we use the term *subgroup* below to refer to the single shard of the subgroup. Derecho's performance characteristics do not differ when subgroups are composed of multiple shards and hence, all our conclusions below apply in those cases as well.

Derecho's atomic multicast performance in a single subgroup consistently stays above 10GB/s (80% network utilization) for 100MB messages. With 10KB messages, it performs at over 8GB/s, reaching a peak of 10GB/s. Derecho outperforms APUS, a modern

Paxos system on RDMA by a factor of 25X. On 12.5GB/s TCP network, the performance is capped by 3GB/s, owing to TCP's memory copy overheads. This still outperforms traditional Paxos systems, notably LibPaxos and Zookeeper, by a factor of 80X.

For in-memory replication in very large groups, relevant to applications that need to support highly distributed querying of the data, Derecho's scaling is impressive. We tested group sizes of 2 to 128 on the Stampede cluster that has 7GB/s RDMA network. Performance with 1MB messages on 128 nodes is 1.9GB/s, about 40% of the performance on 2 nodes.

Derecho scales well with multiple subgroups. When all subgroups have the same members, the predicate thread needs to scale with the increasing area of memory it needs to poll to detect new events for the many subgroups. Derecho's performance with small messages in this case leads to a degradation of only about 15% on average, going from 1 to 50 subgroups. Next, we tested disjoint and partially overlapping subgroups of size 2 and 3 for number of subgroups ranging from 1 to 128. We found that the aggregate performance, computed across all the subgroups, increased proportionally to the number of subgroups.

To test Derecho's resilience, we measured its performance in the presence of all sorts of delays. For large messages, the time to transmit data takes up most of the time to deliver the message. We proved theoretical results showing that RDMC (data-plane for large messages) adapts well to delays in both software and network. We validated this through experiments on the Apt cluster that contained an oversubscribed switch resulting in delays in sending data across nodes on different racks. RDMC's performance with 56 nodes was just one-third of its performance with 2. Memory copy in the data path for large messages has a devastating effect. Switching to TCP instead of RDMA slashes the performance in more than half. On the other hand, delays in the control plane did not impact performance for large messages significantly. For example, when we slowed down the predicate thread, performance with 100MB messages declined by only 1.5% when the predicate thread was running at an efficiency of just 10%. With small messages, the decline in performance was larger but still much less compared to the slowdown. Therefore, other such experiments focused only on Derecho's small message performance.

Derecho's atomic multicast protocol requires every sender node to continuously send messages. In the event that a node does not have any data to send, Derecho sends automated null-messages on its behalf. Delaying message sends by $1 - 100\mu s$ at one or half of the nodes surprisingly improves Derecho's performance, owing to more efficient batched processing and better bandwidth utilization by the fast senders. Memory copy in the case of small messages has a much diminished impact – performance remains consistently around 7.5 GB/s for the all senders case.

6.3 Research projects based on Derecho

This dissertation has focused on prior work and the author's own work. However, as the lead developer of a system that sets performance records in its domain, the author has also been consulted by a number of potential and actual users for assistance and advice, and through those interactions, learned quite a lot about work others are doing using Derecho. As noted, there are a number of projects underway that do employ Derecho: some are built upon the actual software, others are studying its protocols using formal methods in various ways, and still others are viewing Derecho more as an roadmap for leveraging RDMA in their own, independently created, applications and systems. While it would be tempting to include a list and to describe all of this work, it seems inappropriate for this author to offer any kind of preview of future publications that other authors are still preparing. Moreover, some of the deployed uses of Derecho involve companies or organizations that have not publicly discussed their efforts. With this in mind, we have decided to omit any such list.

6.4 Suggested future work

In this section, we will discuss possible extensions of our work that could be explored as research projects.

6.4.1 Smart polling

Our Spindle work on optimizing Derecho's small message performance uncovered further optimization opportunities. Our opportunistic batching scheme involves processing all events of a particular type (send, receive, or delivery) that are available. For example, the predicate thread receives all new messages that have arrived in the system. This ensures that we don't risk keeping the network idle in an attempt to form larger batches.

A natural question arises: is there an even better trade-off between wait times and processing times such that the overall performance is even higher? The potential for improvement becomes apparent when we look at the results of the experiment where one or half of the senders are delayed for a small period ($1\mu s$) before each message they send. We find that delaying actually improves the throughput as it results in the predicate thread forming larger, more efficient batches. Another case in point is the experiment when all nodes were part of multiple subgroups, up to 50. Each node continuously sent messages in all subgroups. While the performance remained high even with 50 subgroups, there still was a slight decline with increasing load on the predicate thread. This does not have to be the case – with multiple data streams, we have the potential to improve network utilization. In fact, we found that the performance with 2 subgroups was already higher than that with 1 subgroup.

For further exploration, we see the most value in improving the polling logic. Currently, the predicate thread goes through each subgroup in cyclic order and within a subgroup, it evaluates each predicate (send, receive, delivery) one after the other. If instead, it could prioritize evaluating predicates that could give us the most value for the time, we would see much improved results. Realizing this idea involves an understanding of how the predicate evaluations directly impact throughput. Specifically, we would want answers to questions such as: a) Is running the receive predicate more valuable than the delivery predicate? and b) Should subgroup number 3's predicates be evaluated more because it is more active? It is important to note here that the current design of the code involves holding locks when the predicates run and thus precludes evaluating them in parallel. And in any case, dedicating a thread per subgroup does not seem feasible given the typically limited CPU resources in data centers.

At the same time, we need to ensure that any smart polling logic does not interfere with the speed of predicate evaluations as that is crucial to a more streamlined processing. Ideally, such a scheme should involve making batched decisions periodically. We could also consider using ML techniques to direct the selection of predicates to evaluate. Using ML techniques in systems has become popular recently, so such an endeavor can benefit from relevant related research in that area.

With the increasing variety of applications that seek to leverage RDMA using polling threads (polling is more efficient than interrupts, at least while such systems are actively executing), such questions will arise in many settings. We encourage interested researchers to look into RDMA-based database systems, key-value stores, IoT applications and the like to find similar opportunities where performance is directly impacted by the efficacy of the underlying polling infrastructure.

6.4.2 Power of SST's asynchronous model

SST's lock-free, asynchronous style of computation works wonderfully for a coordination-heavy system like Derecho. It allows for the data path to run in a non-blocking manner and enables batched decision-making on monotonic data. Naturally, one could wonder whether the SST would be useful in other settings in which it is possible to reexpress the corresponding protocols to run in an asynchronous manner.

Leveraging asynchrony in distributed protocols is often challenging. We realized this for Paxos protocols 30 years after Paxos was invented. We were also interested in implementing a version of the parallel stochastic gradient descent algorithm over SST, however, we ran into multiple roadblocks. The underlying root of the issues was our limited understanding of the effects of asynchrony on the correctness (in terms of convergence of the model) of the protocol. We were unable to ascertain under what conditions of asynchrony would the protocol still converge. In the same vein, one could imagine exploring asynchronous executions of other distributed ML protocols. That is, the SST model might be of direct use in creating new kinds of distributed AI and distributed ML solutions.

With the increased popularity of peripheral devices such as GPUs and FPGAs too, concurrent computation with multiple memory systems is gaining prominence. We believe that new memory systems such as the SST will continue to find increasing applications in many modern systems.

Also of interest would be a more formal treatment of the SST model and programming methodology from a programming languages perspective. Although outside my own direct interests, I noted the work we undertook using the IVy theorem prover, as part of our verification of the Derecho protocols. Far more could be done: IVy is limited to properties that can be expressed in a decidable first order logic (the one supported by the Z3 solver), but there are more powerful provers today, particularly those designed for higher-order logic using Coq as their base language.

Whereas the IVy proofs we completed (but did not publish) turned out to be fairly straightforward, they were at the same time limited: because we could not express higher order properties, we were unable to specify or derive progress conditions for Derecho (an example of a higher-order property), and our approach to modeling the overall sequence of membership views didn't fully capture the concept of a single primary partition, central to the virtual synchrony model. With a higher-order prover, questions of these kinds could be addressed, and one could also explore issues arising when an application or protocol interacts with the SST.

Pushing even further, some provers, such as the Princeton VST infrastructure layered over Coq, should be able to modularize such proofs, separating them into an abstraction and proofs for the SST, yielding formal software characterization of the SST programming model, which could then be composed with code for the protocol running on the SST, such as the SMC protocol in Derecho and our extension of it into an atomic multicast and a version of Paxos. Work along these lines would ultimately yield a proved-correct and constructive (implemented) version of Derecho's core that could be used safely in mission-critical settings, such as to control a drone or a self-driving car. While my own work will not follow that path, it does strike me as a particularly promising one.

6.5 Closing

To conclude this dissertation, we hope Derecho's many features and exceptional performance numbers is of value to applications that seek to leverage high performance networks, in the present and the coming future. We also hope that the development of Derecho inspires the developement of high performance systems in other settings.

BIBLIOGRAPHY

[1] Data plane development kit.

[2] LibPaxos: Open-source Paxos. http://libpaxos.sourceforge.net/.

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

[4] Mvapich: Mpi over infiniband, omni-path, ethernet/iwarp, and roce.

[5] RDMA-Paxos: Open-source Paxos. https://github.com/wangchenghku/RDMA-PAXOS.

[6] Vsync reliable multicast library. http://vsync.codeplex.com/, November 2011.

[7] Gbcast protocol. https://en.wikipedia.org/wiki/Gbcast, July 2012.

[8] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[9] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, December 2013.

[10] Bela Ban. JGroups reliable multicast library. http://jgroups.org/, November 2002.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP ’09, pages 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Behrens, Jonathan and Birman, Ken and Jha, Sagar and Tremel, Edward. RDMC: A Reliable RDMA Multicast for Large Objects. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '18, pages 1–12, Washington, DC, USA, 2018. IEEE Computer Society.

[13] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard 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*, OSDI'14, pages 49–65, USA, 2014. USENIX Association.

[14] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proc. 2013 USENIX Annual Technical Conference*, pages 169–180, San Jose, CA, USA, June 2013. USENIX Association.

[15] Bharat Bhargava and Shu-Renn Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems – an optimistic approach. In *Proc. 7th Symp. Reliable Distributed Systems*, pages 3–12, October 1988.

[16] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[17] K. P. Birman and T. A. Joseph. Communication support for reliable distributed computing. In B. Simons and A. Spector, editors, *Fault-tolerant Distributed Computing*, pages 124–137. Springer-Verlag, London, UK, UK, 1990.

[18] Ken Birman, Bharath Hariharan, and Christopher De Sa. Cloud-hosted intelligence for real-time IoT applications. *SIGOPS Oper. Syst. Rev.*, 53(1):7–13, July 2019.

[19] Kenneth Birman. *Guide to Reliable Distributed Systems*. Number XXII in Texts in Computer Science. Springer-Verlag, London, 2012.

[20] Kenneth P. Birman. Replication and Fault-tolerance in the Isis System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, 1985. ACM.

[21] Kenneth P. Birman. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Springer Verlag Texts in Computer Science, New York, NY, USA, 2012.

[22] Kenneth P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[23] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[24] Kenneth P. Birman, Thomas A. Joseph, and Frank B. Schmuck. Isis documentation: Release 1. Technical report, Cornell University, Ithaca, NY, USA, 1987.

[25] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. 8th USENIX Conf. Networked Systems Design and Implementation*, pages 141–154, Boston, MA, USA, 2011. USENIX Association.

[26] Eric Brewer. A certain freedom: Thoughts on the CAP theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 335–335, New York, NY, USA, 2010. ACM.

[27] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[28] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[29] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[30] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[31] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[32] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

[33] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM.

[34] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[35] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 105–120, New York, NY, USA, 2015. ACM.

[36] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 5:1–5:7, New York, NY, USA, 2015. ACM.

[37] Danny Dolev and Dalia Malki. The Transis Approach to High Availability Cluster Communication. *Commun. ACM*, 39(4):64–70, April 1996.

[38] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[39] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

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

[41] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[42] E. N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[43] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[44] Roy Friedman and Robbert van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. In *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC '97).* Also available as *Technical Report 95-1527, Department of Computer Science, Cornell University.* , 1997.

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

[46] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 245–260, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[48] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[49] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA, 1996. ACM.

[50] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Trans. Comput. Syst.*, 28(2):5:1–5:32, July 2010.

[51] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[52] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *J. ACM*, 37(3):549–587, July 1990.

[53] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[54] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE-IFIP International Conference on Dependable Systems & Networks (DSN)*, June 2010.

[55] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, April 2019.

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

[57] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 33–48, GA, 2016. USENIX Association.

[58] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[59] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association.

[60] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 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)*, pages 185–201, Savannah, GA, November 2016. USENIX Association.

[61] Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 169–178, New York, NY, USA, 2006. ACM.

[62] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, SE-13(1):23–31, January 1987.

[63] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, April 1984.

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

[65] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[66] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 312–313, New York, NY, USA, 2009. ACM.

[67] Butler Lampson. The ABCD's of Paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 13–, New York, NY, USA, 2001. ACM.

[68] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA*, 2016.

[69] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea, and Kenneth P. Birman. The performance of Paxos in the cloud. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, SRDS '14, pages 41–50, Washington, DC, USA, 2014. IEEE Computer Society.

[70] D. Mazieres. Paxos Made Practical. In *Technical report. http://www. scs. stanford. edu/dm/home/papers* , 2007.

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

[72] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[73] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel,

Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[74] 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*, 39(4):54–63, April 1996.

[75] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[76] OMG Data Distribution Standard.

[77] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

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

[79] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[80] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[81] Marius Poke and Torsten Hoefler. DARE: High-performance state machine repli-

cation on RDMA networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, New York, NY, USA, 2015. ACM.

[82] Marius Poke, Torsten Hoefler, and Colin Glass. AllConcur: Leaderless concurrent atomic broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, New York, NY, USA, 2017. ACM.

[83] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pages 111–125, London, UK, UK, 1997. Springer-Verlag.

[84] Dan Pritchett. Base: An ACID alternative. *Queue*, 6(3):48–55, May 2008.

[85] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4(4):243–254, January 2011.

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

[87] Steve Shankland. Google's Jeff Dean Spotlights Data Center Inner Workings. *C|Net Reviews*, May 2008.

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

[89] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proc. 26th IEEE Symp. Mass Storage Systems and Technologies*. IEEE Computer Society, 2010.

[90] Dale Skeen. Determining the last process to fail. *ACM Trans. Comput. Syst.*, 3(1):15–30, February 1985.

[91] Weijia Song, Theo Gkountouvas, Ken Birman, Qi Chen, and Zhen Xiao. The

Freeze-Frame File System. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 307–320, New York, NY, USA, 2016. ACM.

[92] Akshitha Sriraman and Thomas F Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, 2018.

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

[94] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[95] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

[96] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[97] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[98] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.

[99] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of the Eighth ACM Symposium on Cloud Computing*, SoCC '17, Santa Clara, CA, USA, Sept. 2017. ACM.

[100] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. vCorfu: A cloud-scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 35–49, Boston, MA, 2017. USENIX Association.

[101] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. ACM.

[102] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[103] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[104] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.