Derecho: Fast State Machine Replication for Cloud Services

SAGAR JHA, Cornell University, USA
JONATHAN BEHRENS, Cornell University, USA and MIT, USA
THEO GKOUNTOUVAS, MATTHEW MILANO, WEIJIA SONG, EDWARD TREMEL, ROBBERT VAN RENESSE, SYDNEY ZINK, and KENNETH P. BIRMAN,
Cornell University, USA

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.

CCS Concepts: • **Computer systems organization** → *Dependable and fault tolerant systems and networks*; • **Software and its engineering** → **Cloud computing**;

Additional Key Words and Phrases: Cloud computing, RDMA, non-volatile memory, replication, consistency

ACM Reference format:
Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2, Article 4 (March 2019), 49 pages.
https://doi.org/10.1145/3302258

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 [15, 47]. 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 are typically stored into a global file system and processed later in batches, consuming significant resources and introducing long delays.

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.
Authors' addresses: S. Jha, T. Gkountouvas, M. Milano, W. Song, E. Tremel, S. Zink, K. P. Birman, and R. van Renesse, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY 14850, USA; emails: srj57@cornell.edu, {tedgoud, milano, ws393, edward}@cs.cornell.edu, sydney.zink@gmail.com, {ken, rvr}@cs.cornell.edu; J. Behrens, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY 14850, USA, and MIT, Department of Electrical Engineering and Computer Science, 77 Massachusetts Ave, Cambridge, MA, 02139, USA; email: behrensj@mit.edu.

0734-2071/2019/03-ART4 $15.00
https://doi.org/10.1145/3302258

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 and then superimposes a state machine replication model [48] on these flows using out-of-band logic. At each step, the developer can customize event handling, leveraging a 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 [14] for dynamic membership tracking. The update path uses a version of Paxos [35] 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 Reference [50].

Many cloud computing architects have been critical of strongly consistent replication [15], 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 are 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 [21]. 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 100× 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, then RDMA permits an additional 4× 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.

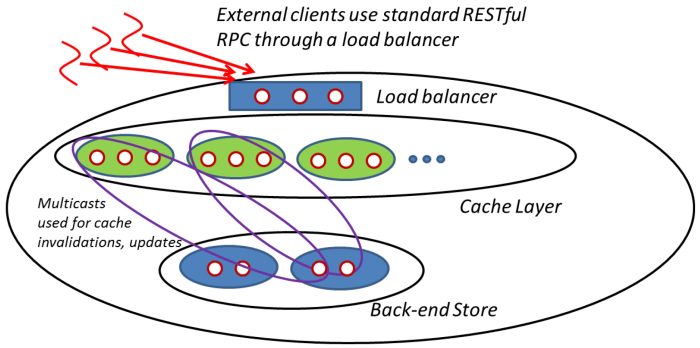


Fig. 1. Derecho applications are structured into subsystems. Here we see 16 processes organized as 4 subsystems, 3 of which are sharded: the cache, its notification layer, and the back-end. A process can send one-to-one requests to any other process. State machine replication (atomic multicast) is used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

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 article is organized as follows. Section 2 focuses on the application model, the Derecho API, and the corresponding functionality. Section 3 discusses the system architecture, protocols, and implementation details. Section 4 not only focuses on Derecho's performance but also includes side-by-side comparisons with LibPaxos, Zookeeper and APUS. Section 5 reviews prior work. Appendix A shows pseudo-code for some key protocol steps, and Appendix B presents the full protocol suite in detail. Appendix C discusses the conditions under which Derecho is able to make progress, comparing these with the conditions under which classic Paxos can make progress.

2 APPLICATION MODEL AND ASSUMPTIONS MADE

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 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, and so on. 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. However, 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, and so on. 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.

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 [12]. The system-managed view can easily be augmented with parameters or other forms of application-defined metadata.

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 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 two or three 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 and 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, then 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, and so on, but such tools often have heavy marshalling and copying overheads.

Members hold replicated state and can access it directly: Data are 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"' one-to-many data transfer (lowest cost, but very weak semantics), an atomic multicast, or a durable Paxos update. Figures 2–6 illustrate a few of the communication patterns that a subgroup or shard might employ.

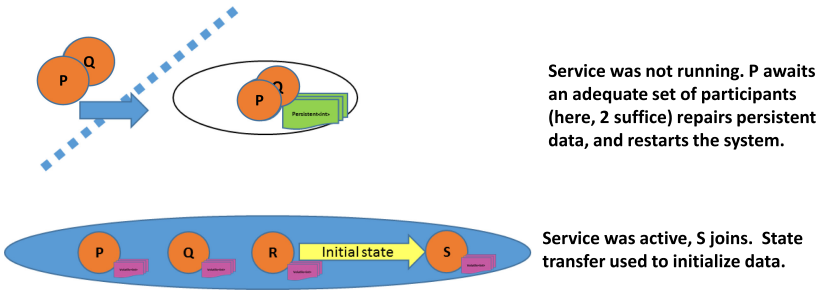


Fig. 2. When launched, a process linked to the Derecho library configures its Derecho instance and then starts the system. On the top, processes P and Q restart a service from scratch, reloading persisted state; below, process S joins a service that was already running with members {P,Q,R}. As the lowest-ranked 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.

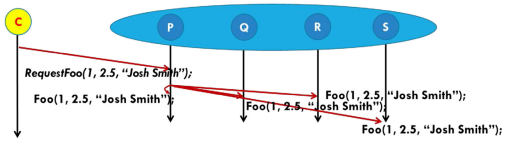


Fig. 3. Multicasts occur within subgroups or shards and can only be initiated by members. External clients interact with members via P2P requests.

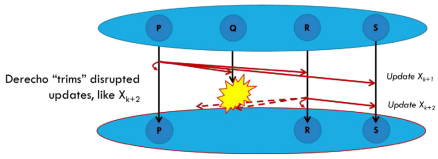


Fig. 4. If a failure occurs, then cleanup occurs before the new view is installed. Derecho supports several delivery modes; each has its own cleanup policy.

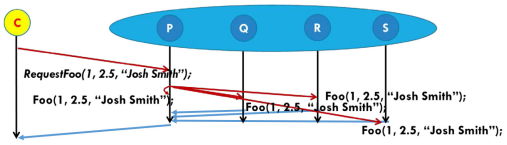


Fig. 5. A multicast initiated within a single subgroup or shard can return results. Here, the handler would run on the most current version.

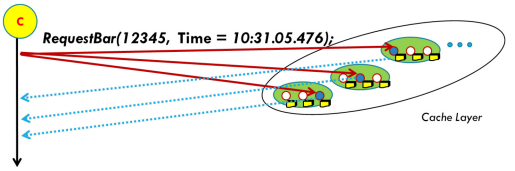


Fig. 6. Read-only queries can also occur via P2P requests that access multiple subgroups or shards. Derecho's temporal indexing is lock-free yet strongly consistent: a new form of *temporal snapshot isolation.*

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 versions or omit committed ones, coordinates state-transfer to the joining process, and so forth (see Figure 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 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 [49]. 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] and 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 even during partitioning failures [15, 29]. 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, then 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.

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.

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 one-to-N multicast. Failures that disrupt an `ordered_send` are masked: The protocol cleans up and then reissues requests as necessary in the next adequate membership view, preserving sender ordering. The `outcome` object tracks the status of the request,

[1] RDMA requires a nearly perfect network but does have a very primitive packet retransmission capability.
[2] Derecho targets data center use cases. In a WAN partitioning is more frequent and this tactic would not be appropriate.

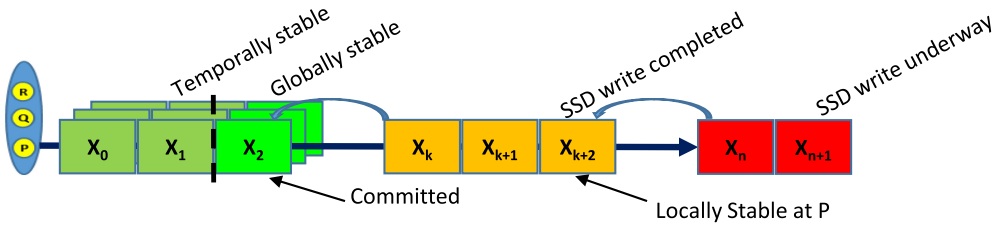


Fig. 7. Derecho applications store data in replicated version vectors. Versions pass through a series of stages: They are created (red), logged locally (orange), and then become stable (green). Within the stable versions, Derecho distinguishes between basic Paxos durability (all green versions) and temporal stability (darker green), which adds an additional guarantee that no update with an earlier real-time timestamp could occur.

indicating when the operation has completed. If the method returns results, then the outcome object can also be used to iterate over responses as they arrive

2.6 Versioned Data and Persistent Logs

Derecho services are fault tolerant and parallel by virtue of using state machine replication. But where do 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, then 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 7). Each new version builds on the prior one, and 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, then 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, then 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 7, with older and more stable data on the left and new updates on the right (the newest are colored red).

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, then Derecho will back the updates out by truncating the logs and then reissue them.

[3]Derecho has multiple levels of update *stability*. The ones relevant to the present article 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 [22] delays updates until it is certain that no update with a prior timestamp could still be in the network.

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) [50] 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 [19, 50].

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 on, 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 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 are not yet committed, it must await temporal stability, as seen in Figure 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 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 [9, 43].

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 log, and then offer a way to learn the contents of the log [35]. 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) and 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 1.5μs 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 $1\mu s$ for data transfer: $8.5\mu s$. We would need 115,000 concurrently active Paxos operations to keep the network busy[4]!

In a server group with five 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 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 [44, 54], which have leader-based structures with the potential for pauses on the critical path (examples of exception include AllConcur, which uses a leaderless approach [45] and NOPaxos, which leverages data-center network components to order events [37]).

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.

3.1 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 [14] 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 Reference [12].

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 waits 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 datastream. 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

[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.

[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.

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 holds, it continues to hold (even as additional data are exchanged). But such predicates have a further, and particularly useful, property: 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, then it also holds for messages* $0 \ldots (k-1)$.

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 and 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.

3.2 Building Blocks

Derecho is composed of two subsystems: one that we refer to as RDMC [8], 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 and 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 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. In Sec. 3.6, 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, and 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 [53], BarrelFish [7], and Arrakis [43].

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 [8]. This protocol supports arbitrarily large messages that it breaks into 1MB chunks and 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 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 [28] and illustrate in Figure 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, then all receivers deliver simultaneously; if not, then 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

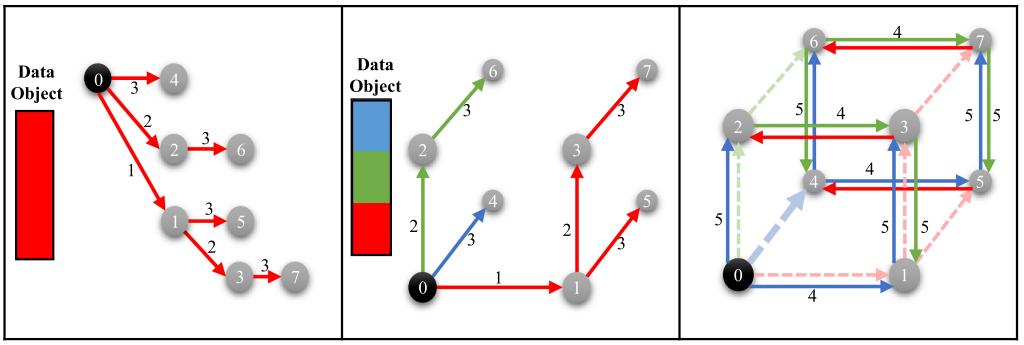


Fig. 8. Our RDMA multicast protocol, RDMC [8], breaks large messages into chunks and then forwards them chunk by chunk over a network overlay of reliable FIFO channels. Here the left diagram illustrates a binomial copying protocol, with processes represented by circles and data transfer steps represented 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, three 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.

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 data-center 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 and 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).

3.4 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,

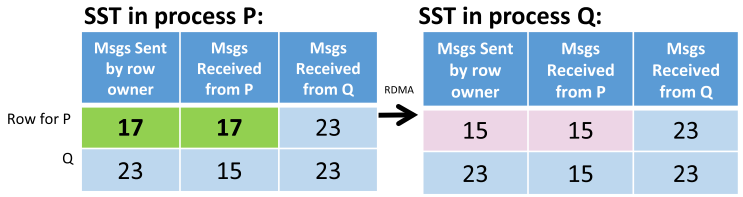


Fig. 9. SST example with two members: P and Q. P has just updated its row and is using a one-sided RDMA write to transfer the update to Q, which has a stale copy. The example, discussed in the text, illustrates the sharing of message counts and confirmations.

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 (illustrated in Figure 9).

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

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, then 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 $0 \ldots v$. 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 that 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 Row $\rightarrow$ 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 : Row $\rightarrow$ 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' : Row $\rightarrow$ T that, 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 projector $\rightarrow$ projector, allowing reducer functions to be arbitrarily combined, somewhat like formulas over a spreadsheet.

Let's look at an example.

```
1 struct SimpleRow {int i;};
2 int iget(const volatile SimpleRow& s){
3   return s.i;
4 }
5 bool proj(){
6   return (Min(as_projector(iget)) > 7 ) || (Max(as_projector(iget)) < 2);
7 }
```

Here, function `proj` converts the function `iget` into a projector, calls the reducers `Min` and `Max` on this projector, and 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` that 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, then 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* [32], 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 11 and then 27, 31. If the values are used in monotonic predicates and some deduction was possible when the value reached 30, then 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, then 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:

Fig. 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.

(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 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, then whichever process first notices the problem can treat it as a fault detection.

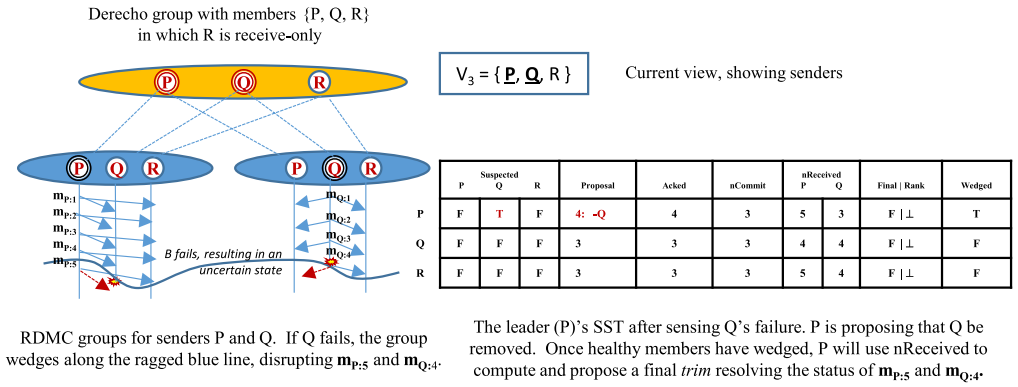
Thus, if a node has crashed, then the SST will quickly reach a state in which every non-failed process suspects the failed one, has frozen its SST row, and has pushed its own updated row to its peers. However, because SST's push is implemented by N separate RDMA writes, each of which could be disrupted 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



	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

Fig. 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, then the procedure repeats.

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, then 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, then 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 and 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.

3.6 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 A, which uses a pseudo-code notation to show the key steps in a high-level format, and Appendix B, which walks through the full protocol suite.

Derecho's core structure can be seen in Figure 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 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 five 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, and so on.

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 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 [18].

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 A, 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 B 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.

3.7 Optimizations

The central innovation of Derecho is that we have expressed all our protocols as a composition of high-speed dataflow 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 1.5–2× 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, then 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 80× 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 have 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.

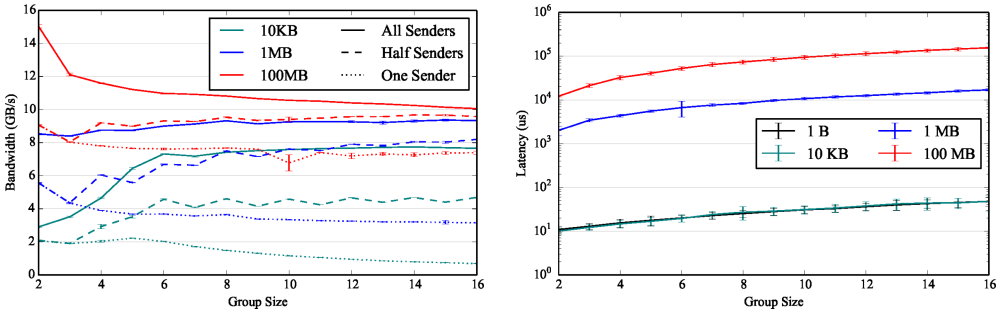
Action batching. As noted earlier, by leveraging monotonicity, we can often design predicates that will cover multiple operations in each action. This allows Derecho to leverage receiver-side batching, enabling our multicast protocols to catch up even when a significant scheduling delay impacts some 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.

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, then 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 (two or three members, with or without overlap; for the overlap case, we created two 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.



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

Fig. 12. Derecho's RDMA performance with 100Gbps InfiniBand.

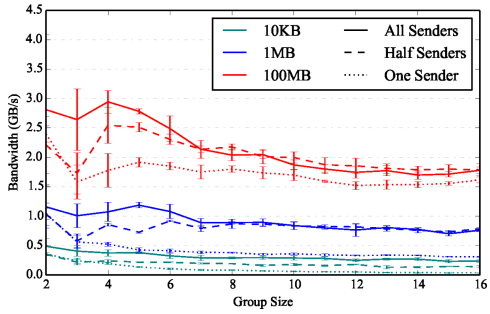
- Latency for updates versus latency for queries, again in the case of versioned data. Here the queries exercise our temporally consistent snapshot isolation functionality, accessing data spread over multiple shards.
- How does the performance of the system degrade if some members are slow?
- How long 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 (12.5GB/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 (8GB/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.

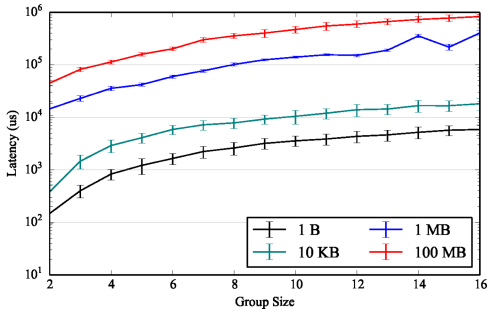
4.1 Core Protocol Performance

Figure 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 1MB and 100MB, with a peak rate of 16GB/s. In unreliable mode, Derecho's protocol for sending small messages, SMC, ensures that we get high performance (close to 8GB/s) for the 10KB 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



(a) Atomic multicast throughput (TCP)



(b) Atomic multicast latency (TCP)

Fig. 13. Derecho performance using TCP with 100Gbps Ethernet.

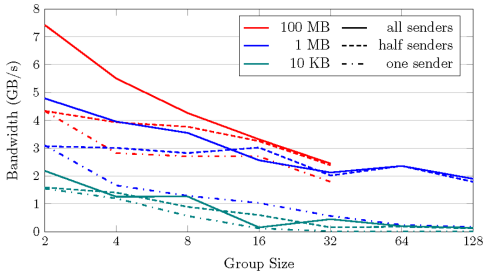


Fig. 14. Totally ordered (atomic multicast) mode.

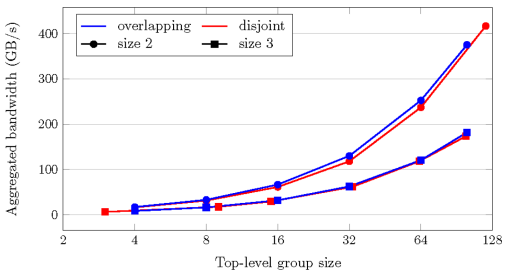


Fig. 15. Derecho performance for sharded groups using RDMC with 100MB messages.

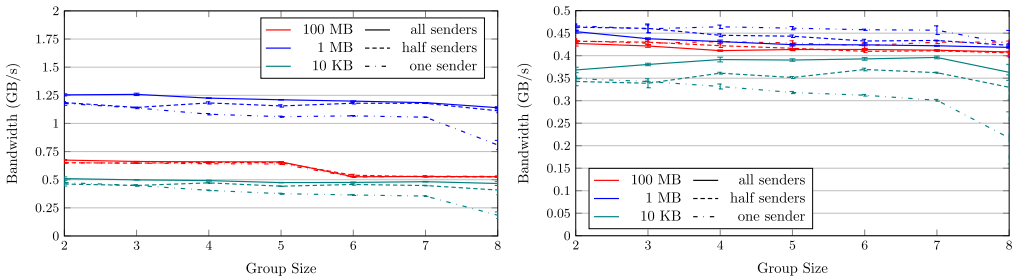
nodes for delivery, and thus unreliable mode and totally ordered (atomic multicast) mode achieve similar performance. For small message sizes (10KB), those two times are comparable. Here, unreliable mode has a slight advantage, because it does not perform global stability detection prior to message delivery.

Not shown is the delivery batch size; at peak rates, multicasts are delivered in small 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 20(b).

Notice that when running with two nodes at the 100MB message size, Derecho's peak performance exceeds 12.5GB/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.

4.2 Large Subgroups with or Without Sharding

In Sec. 2.3, 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 Figure 14, shows that Derecho scales well. For example, we obtain performance of about 5GB/s for 1MB-all-senders on 2 nodes, 2.5GB/s on 32 nodes, and 2GB/s on 128 nodes, a slowdown of less than 3×. Limitations on experiment duration and memory prevented us from carrying out the same experiment for the 100MB case on 64 and 128 nodes. This also explains the absence of error bars: Each data point



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

Fig. 16. Derecho performance with marshalled method invocations, and handlers that use version-vector storage.

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 ith 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 ith subgroup is composed of nodes $i, i+1, \ldots, i+s-1$ (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 15 shows that for large messages (100MB), 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 (10KB), because this particular case triggered a hardware problem: the "slow receiver" issue that others have noted in the most current Mellanox hardware [31]. 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 encountered 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.

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, Figure 16(a) and (b) looks at a full code path that includes marshalling on the sender side (required only if the data being sent are 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

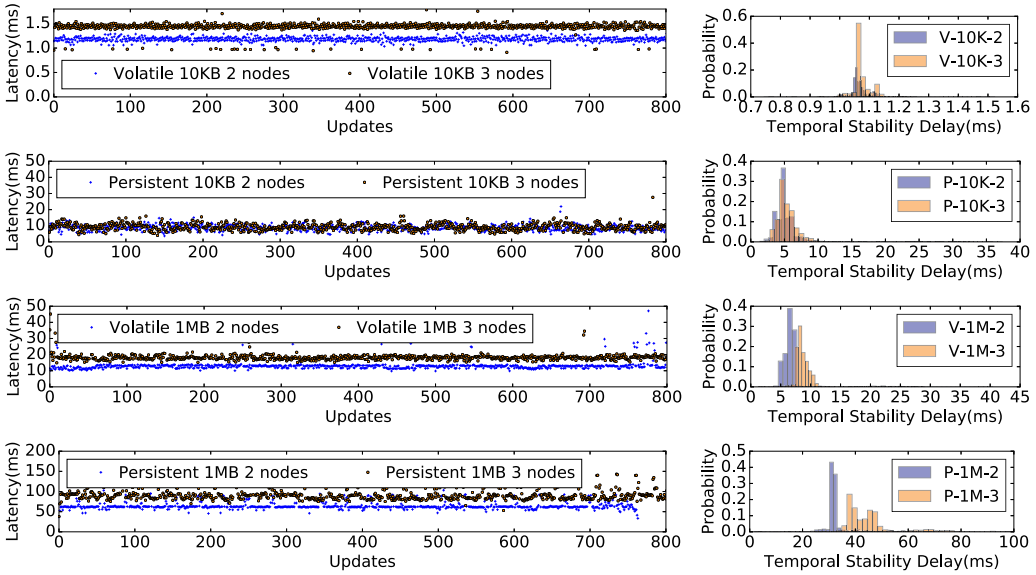


Fig. 17. End-to-end performance. On the left, time from when an update was initiated until the new version is temporally stable. On the right, minimum delay for a query to access data at time "now."

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 16(a), 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 data structures but tries whenever possible to use the same object representation both in memory and on the wire. User-defined serialization and serialization of certain data types (like STL containers) may incur additional copies (and thus additional overhead).

4.4 End-to-End Performance

In Figure 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 two nodes (blue) or three 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.

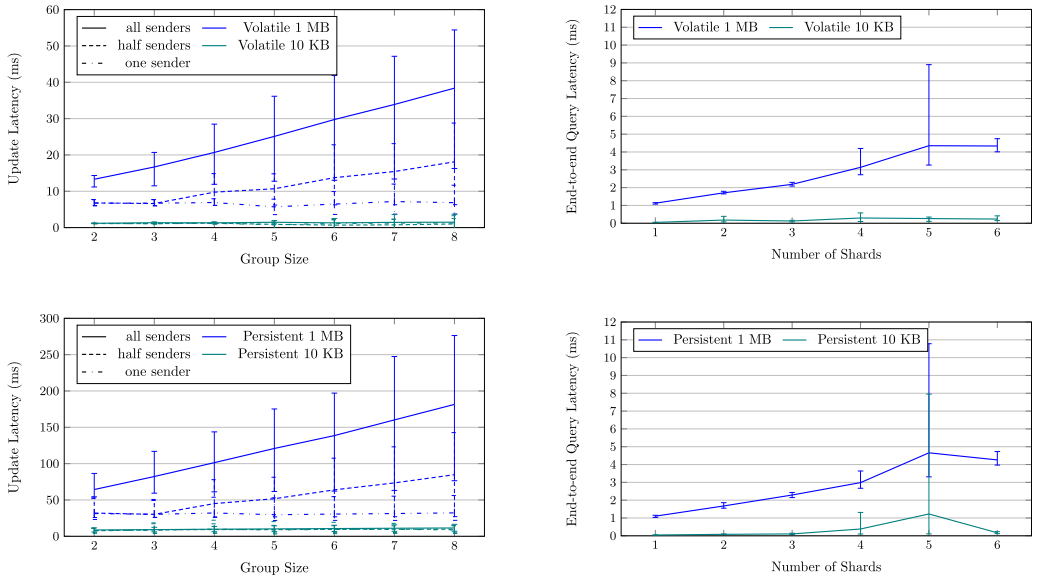


Fig. 18. Left: Latency as a function of group size for updates. Right: Latency for a temporal query issued to a single member each, for varying numbers of shards.

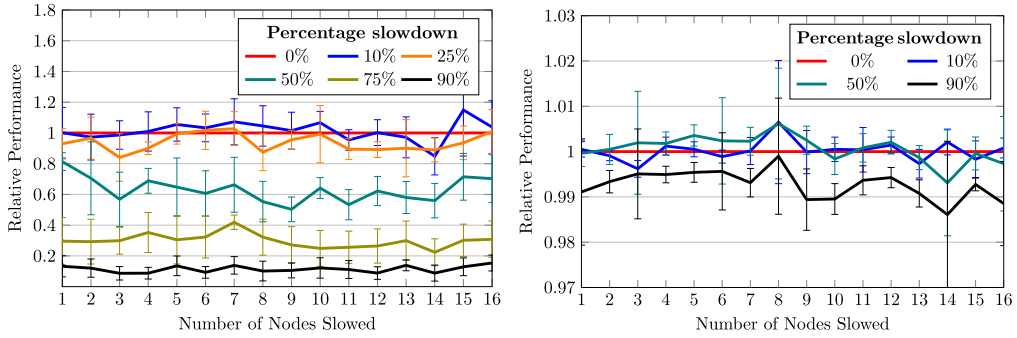
To obtain the histograms on the right, we launched an additional thread within the same processes; it loops. During each iteration, our test samples the local clock and then indexes into the local instance of the versioned object at the corresponding time, measuring the delay until the data are available. 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 6 and discussed in Section 2).

In Figure 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 13th node queries a single member each within 1 to 6 shards, using Derecho's `p2p_query` API, which issues the P2P requests concurrently and 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 five-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.

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



(a) 10 KB (b) 100 MB

Fig. 19. Derecho performance for various values for efficiency and number of slow nodes as a fraction of the no-slowdown case.

Derecho in a situation where there might be other activity on the same nodes. Recall that traditional Paxos protocols degrade significantly if some nodes run slowly [38]. 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.

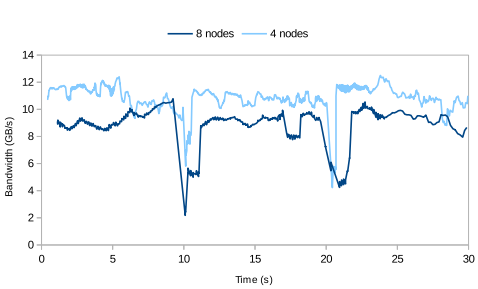
Accordingly, we designed an experiment in which we deliberately slowed Derecho's control infrastructure. We modified the SST predicate thread by introducing artificial busy-waits after every predicate evaluation cycle. In what follows, we will say that a node is working at an efficiency of $X\%$ (or 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 19.

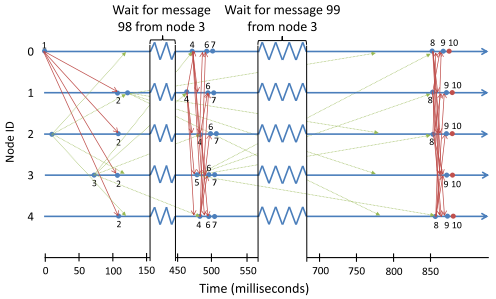
The first thing to notice in this graph is that even with significant numbers of slow nodes, performance for the large messages is minimally impacted. This is because with large messages, the RDMA transfer times are so high that very few control operations are needed and because the control events are 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



(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.

Fig. 20. Multicast bandwidth (left), and a detailed event timeline (right).

deduce global stability. Because this rule does not depend on a single leader, all nodes proceed independently toward 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.

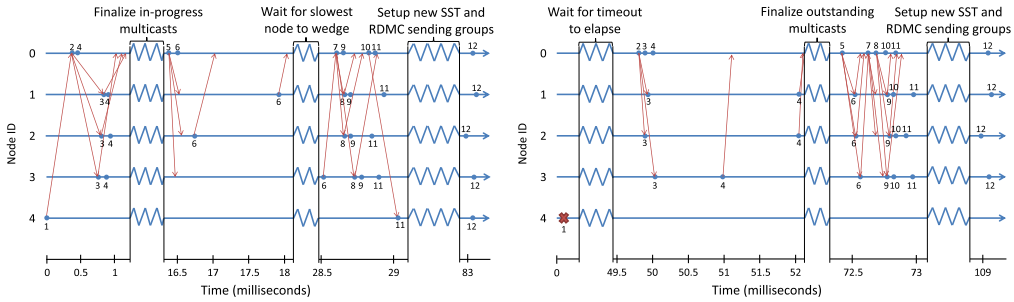
Next, we considered the costs associated with membership changes.

4.6 Costs of Membership Reconfiguration

In Figure 20(a), 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 20(b) 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 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.

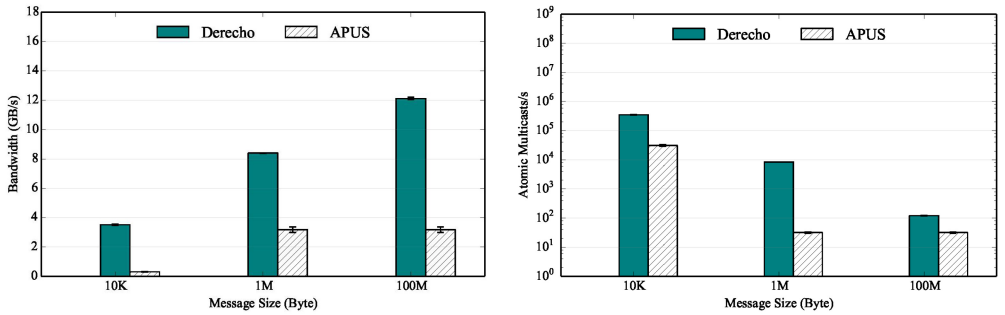
In Figure 21(a), we see a process joining: ① it requests to join, ②-⑥ are the steps whereby the leader proposes the join, and members complete pending multicasts and finally wedge. In steps ⑦-⑨, the leader computes and shares the trim; all processes trim the ragged edge at ⑨ and the leader sends the 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 21(b) 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 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 20(a) the disruptive impact of a join or



(a) Timeline for joining an active group. (b) Crash triggers exclusion from an active group.

Fig. 21. Timeline diagrams for Derecho.



(a) Throughput (b) Multicast Rate

Fig. 22. Derecho vs. APUS with three nodes over 100Gbps InfiniBand.

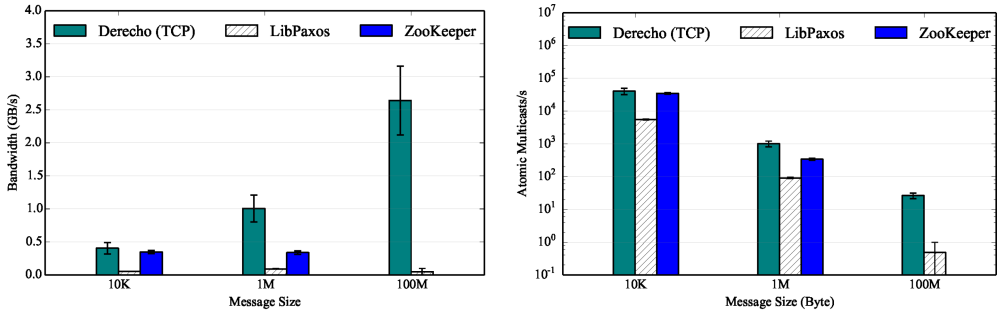
leave grows as a function of the number of active senders: the number of active sends and hence the number of bytes of data in flight depends on the number of active senders. Since we are running at the peak data rate RDMA can sustain, the time to terminate these sends is dominated by the amount of data in flight. In experiments with 20MB messages, the join and leave disruptions are both much shorter.

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, and 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 22. We focused on a three-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 23 (Zookeeper does not support 100MB writes and hence that data point is omitted). Again, we use a three-member group but saw similar results at other group sizes.



(a) Throughput (b) Multicast Rate

Fig. 23. Derecho vs. LibPaxos and ZooKeeper with three nodes over 100Gbps Ethernet.

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 13(b)). 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 30× 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).

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 data centers equipped with RDMA on IB, because data-center 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 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 that can sometimes be avoided for an uncooked send when the data are 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 do not 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.

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 [38].

With respect to guarantees offered, the prior work on Paxos is obviously relevant to our article. The most widely cited Paxos paper is the classic Synod protocol [35], but the version closest to ours is the virtually synchronous Paxos described by Birman, Malkhi, and van Renesse [12]. A number of articles how suggested ways to derive the classic Paxos protocol with the goal of simplifying understanding of its structure [17, 23, 36, 39, 46, 51]

Among software libraries that offer high speed Paxos, APUS [54] has the best performance. APUS implements a state-machine replication protocol related to RAFT and developed by Mazieres [39]. 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 [5] offers a persistent log, using Paxos to manage the end-of-log pointer[7] and chain-replication as a data-replication protocol [52]. 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 and 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 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

[7]Corfu has evolved over time. Early versions offered a single log and leveraged RDMA [5], but the current open source platform, vCorfu [55], materializes individualized "views" of the log and has a specialized transactional commit mechanism for applications that perform sets of writes atomically.

(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 [20]. Ring Paxos is implemented in LibPaxos [1], and Quema has proven a different ring Paxos protocol optimal with respect to its use of unicast datagrams [30], 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 4.7 compares performance of LibPaxos with Derecho.

RAFT [41] is a popular modern Paxos-like protocol; it was created as a replication solution for RamCloud [42] and was the basis of the protocol used in APUS. Microsoft's Azure storage fabric uses a version of Paxos [16] 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 [24]. DARE [44] looks at state machine replication on an RDMA network. RDMA-Paxos is an open-source Paxos implementation running on RDMA [2]. NOPaxos [37] 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 [11], and its gbcast protocol is similar to Paxos [4]. Modern virtual synchrony multicast systems include JGroups [6] and Vsync [3], 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 [27]. 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 [21]. 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 [25], HERD [33], and Pilaf [40]. 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, then it reissues the request. DrTM [56] 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.

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 100× faster than comparable packages when running in identical TCP-based configurations, and a further 4× faster when RDMA hardware is available. The key to this performance resides in a design that builds the whole system using steady dataflows with minimal locking or delays for round-trip interactions.

APPENDIX

A PSEUDO-CODE FOR KEY PROTOCOL STEPS

A.1 Notation

A.1.1 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).

A.1.2 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 $i = g \ mod \ |G|, k = g/|G|$.

A.1.3 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.

A.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.

A.2.1 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 $k + 1$, then a new message is available to be received. The vector received_num holds counters of the number of messages received from each node.

A.2.2 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;
```

A.2.3 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++;
}
```

A.2.4 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]));
    }
  }
}
```

A.3 Atomic Multicast Delivery in the Steady State

A.3.1 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)
}
```

A.3.2 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)
```

A.4 View Change Protocol

A.4.1 *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){
    bool all_others_agree = true;
    //so long as I continue to believe I will be leader
    while(find_new_leader(my_rank) == my_rank) {
        //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){
        //Scan SST to learn prior proposals, then become 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
      }
    }
  }
}
```

A.4.2 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\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 {
```

```
11      for (n = 0; n < |G|; ++n) {
12        sst[my_rank].min_latest_received[n] = Min(sst[*].latest_received_index[n]);
13        // |= K_me( sst[my_rank].min_latest_received[n] number of messages from n are safe for delivery)
14      }
15      sst[my_rank].ragged_edge_computed = true;
16    }
17
18    deliver_in_order();
19  }
20
21  non_leader_ragged_edge_cleanup() {
22    // copy from the leader
23    for (n = 0; n < |G|; ++n) {
24      sst[my_rank].min_latest_received[n] = sst[leader_rank].min_latest_received[n];
25    }
26    sst[my_rank].ragged_edge_computed = true;
27    deliver_in_order();
28  }
29
30  deliver_in_order() {
31    curr_global_index = sst[my_rank].latest_delivered_index;
32    max_global_index = max over n of (sst[my_rank].min_latest_received[n] * |G| + n);
33    for (global_index = curr_global_index + 1; global_index <= max_global_index; ++
          global_index) {
34      sender_index = global_index / |G|;
35      sender_rank = global_index % |G|;
36      if (sender_index <= sst[my_rank].min_latest_received[sender_rank]) {
37        deliver_upcall(msgs[global_index]);
38      }
39    }
40  }
```

B DETAILED DESIGN OF DERECHO'S PROTOCOLS

Derecho is implemented over the building blocks and programming model described in Section 3. Appendix A 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 Reference [12].

B.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 Reference [10], 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, then 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, then 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, then a notification to the application warns it that Derecho activity has been temporarily suspended.
- If multiple joins and failures occur, including cascades of events, then 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, then 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 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 and then acknowledge it through a special SST column set aside for that purpose. When all 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, then 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, then 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; and (3) no new failures were sensed during the most recent round. This is the well-known group membership protocol of the virtual synchrony model [11, 13, 14] and has been proven correct [10]. 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

[8]Readers familiar with the Paxos Synod protocol [51] 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 two-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 C.

described in Section 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, then 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, then 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 are 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, then it will have k superimposed RDMC sessions: one per sender).

The epoch is now considered to be *active*.

B.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 and 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 their 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 10 and 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 two or three 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.

B.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 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 11. The core idea is as follows: We select a leader in rank order; if the leader fails, then 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 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, and so on. 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 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 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 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 to other members of that same subgroup or shard; 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.

B.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

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

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.

B.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, then it computes the ragged trim on its own. Then it publishes the ragged trim, tagged with its own rank.

B.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 are 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.

B.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 dataflow. These include our use of RDMA itself, if available, for reliable communication, the RDMC (or SMC) for one-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 3.5. 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 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, and so on. The key insight is that if any ragged trim actually reached a majority, then 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 [51] (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 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*) [11]. The protocol is correct because:

- If any message was delivered (in totally ordered (atomic multicast) mode) or committed (in durable totally ordered mode), then 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) [34]. 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.

C 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 [26]. This limitation clearly applies to Derecho and also to any Paxos implementation. This is a well-known observation, dating to the 1990s, 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 [18]. 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, and 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, and hence Paxos requires that a majority of the processes be accessible. Classic Paxos does not 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 articles, but see Reference [34]).

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 and 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: Nine is a minority of 20, and 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.

ACKNOWLEDGMENTS

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 *Transactions on Computer Systems* reviewers for their detailed and constructive suggestions. In addition, a number of our colleagues reviewed early drafts of this article and made helpful comments.

These include the following: 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.

REFERENCES

[1] [n. d.]. LibPaxos: Open-source Paxos. Retrieved from http://libpaxos.sourceforge.net/.

[2] [n. d.]. RDMA-Paxos: Open-source Paxos. Retrieved from https://github.com/wangchenghku/RDMA-PAXOS.

[3] 2011. Vsync Reliable Multicast Library. Retrieved from http://vsync.codeplex.com/.

[4] 2012. Gbcast Protocol. Retrieved from https://en.wikipedia.org/wiki/Gbcast.

[5] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. DOI: https://doi.org/10.1145/2535930

[6] Bela Ban. 2002. JGroups Reliable Multicast Library. Retrieved from http://jgroups.org/.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 29–44. DOI: https://doi.org/10.1145/1629575.1629579

[8] Behrens, Jonathan and Birman, Ken and Jha, Sagar and Tremel, Edward. 2018. 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'18)*. IEEE Computer Society, Los Alamitos, CA, 1–12.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Broomfield, CO, 49–65.

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

[11] Kenneth P. Birman. 1985. Replication and fault-tolerance in the isis system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP'85)*. ACM, New York, NY, 79–86. DOI: https://doi.org/10.1145/323647.323636

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

[13] Kenneth P. Birman and Thomas A. Joseph. 1987. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*. ACM, New York, NY, 123–138. DOI: https://doi.org/10.1145/41457.37515

[14] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.* 5, 1 (Jan. 1987), 47–76. DOI: https://doi.org/10.1145/7351.7478

[15] Eric Brewer. 2010. A certain freedom: Thoughts on the CAP theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'10)*. ACM, New York, NY, 335–335. DOI: https://doi.org/10.1145/1835698.1835701

[16] 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. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 143–157. DOI: https://doi.org/10.1145/2043556.2043571

[17] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*. ACM, New York, NY, 398–407. DOI: https://doi.org/10.1145/1281100.1281103

[18] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (Mar. 1996), 225–267. DOI: https://doi.org/10.1145/226643.226647

[19] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. DOI: https://doi.org/10.1145/214451.214456

[20] Jo-Mei Chang and N. F. Maxemchuk. 1984. Reliable broadcast protocols. *ACM Trans. Comput. Syst.* 2, 3 (Aug. 1984), 251–273. DOI: https://doi.org/10.1145/989.357400

[21] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2391229.2391230

[22] 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. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

[23] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 105–120. DOI: https://doi.org/10.1145/2815400.2815427

[24] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR'15)*. ACM, New York, NY, Article 5, 7 pages. DOI: https://doi.org/10.1145/2774993.2774999

[25] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

[26] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (Apr. 1985), 374–382. DOI: https://doi.org/10.1145/3149.214121

[27] Roy Friedman and Robbert van Renesse. 1997. Packing messages as a tool for boosting the performance of total ordering protocols. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC'97)*.

[28] Prasanna Ganesan and M. Seshadri. 2005. On cooperative content distribution and the price of barter. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. 81–90. DOI: https://doi.org/10.1109/ICDCS.2005.53

[29] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (Jun. 2002), 51–59. DOI: https://doi.org/10.1145/564585.564601

[30] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. 2010. Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.* 28, 2, Article 5 (Jul. 2010), 32 pages. DOI: https://doi.org/10.1145/1813654.1813656

[31] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*. ACM, New York, NY, 202–215. DOI: https://doi.org/10.1145/2934872.2934908

[32] Joseph Y. Halpern and Yoram Moses. 1990. Knowledge and common knowledge in a distributed environment. *J. ACM* 37, 3 (Jul. 1990), 549–587. DOI: https://doi.org/10.1145/79147.79161

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM'14)*. ACM, New York, NY, 295–306. DOI: https://doi.org/10.1145/2619239.2626299

[34] Idit Keidar and Alexander Shraer. 2006. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC'06)*. ACM, New York, NY, 169–178. DOI: https://doi.org/10.1145/1146381.1146408

[35] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. DOI: https://doi.org/10.1145/279227.279229

[36] Butler Lampson. 2001. The ABCD's of Paxos. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. ACM, New York, NY, 13–. DOI: https://doi.org/10.1145/383962.383969

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

[38] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea, and Kenneth P. Birman. 2014. The performance of paxos in the cloud. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS'14)*. IEEE Computer Society, Los Alamitos, CA, 41–50. DOI: https://doi.org/10.1109/SRDS.2014.15

[39] D. Mazieres. 2007. Paxos made practical. *Technical report*. Retrieved from http://www.scs.stanford.edu/dm/home/papers.

[40] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. 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)*. USENIX Association, Berkeley, CA, 103–114.

[41] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, 305–320.

[42] 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. 2011. The case for RAMCloud. *Commun. ACM* 54, 7 (Jul. 2011), 121–130. DOI: https://doi.org/10.1145/1965724.1965751

[43] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 1–16.

[44] Marius Poke and Torsten Hoefler. 2015. DARE: High-performance state machine replication on RDMA networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*. ACM, New York, NY, 107–118. DOI: https://doi.org/10.1145/2749246.2749267

[45] Marius Poke, Torsten Hoefler, and Colin Glass. 2017. AllConcur: Leaderless concurrent atomic broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*. ACM, New York, NY, 18.

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

[47] Dan Pritchett. 2008. BASE: An ACID alternative. *Queue* 6, 3 (May 2008), 48–55. DOI: https://doi.org/10.1145/1394127.1394128

[48] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. DOI: https://doi.org/10.1145/98163.98167

[49] Steve Shankland. 2008. Google's Jeff Dean spotlights data center inner workings. *C|Net Reviews* (May 2008). Retrieved from https://www.cnet.com/news/google-spotlights-data-center-inner-workings/.

[50] Weijia Song, Theo Gkountouvas, Ken Birman, Qi Chen, and Zhen Xiao. 2016. The freeze-frame file system. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. ACM, New York, NY, 307–320. DOI: https://doi.org/10.1145/2987550.2987578

[51] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Comput. Surv.* 47, 3, Article 42 (Feb. 2015), 36 pages. DOI: https://doi.org/10.1145/2673577

[52] Robbert van Renesse and Fred B. Schneider. 2004. 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)*. USENIX Association, Berkeley, CA, USA, 7–7.

[53] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, New York, NY, 40–53. DOI: https://doi.org/10.1145/224056.224061

[54] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and scalable paxos on RDMA. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC'17)*. ACM, Santa Clara, CA, 14.

[55] 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. 2017. vCorfu: A cloud-scale object store on a shared log. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 35–49.

[56] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 87–104. DOI: https://doi.org/10.1145/2815400.2815419

Received September 2017; revised July 2018; accepted December 2018