Building Smart Memories and Cloud Services with Derecho

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

The coming generation of Internet-of-Things (IoT) applications will process massive amounts of incoming data while supporting data mining and online learning. In cases with demanding real-time requirements, such systems behave as smart memories: high-bandwidth services that capture sensor input, processes it using machine-learning tools, replicate and store “interesting” data (discarding uninteresting content), update knowledge models, and trigger urgently-needed responses.

Derecho is a high-throughput library for building smart memories and similar services. At its core Derecho implements atomic multicast and state machine replication. Derecho’s replicated template defines a replicated type; the corresponding objects are associated with subgroups, which can be sharded into key-value structures. The persistent and volatile storage templates implement version vectors with optional NVM persistence. These support time-indexed access, offering lock-free snapshot isolation that blends temporal precision and causal consistency.

Derecho automates application management, supporting multigroup structures and providing consistent knowledge of the current membership mapping. A query can access data from many shards or subgroups, and consistency is guaranteed without any form of distributed locking. Whereas many systems run consensus on the critical path, Derecho requires consensus only when updating membership.

By leveraging an RDMA data plane and NVM storage, and adopting a novel receiver-side batching technique, Derecho can saturate a 12.5GB RDMA network, sending millions of events per second in each subgroup or shard. In a single subgroup with 2-16 members, throughput peaks at 16 GB/s for large (100MB or more) objects. When using version-vector storage, Derecho is limited by the speed of the SSD or RamDisk, showing no loss of performance as group sizes grow. While key-value subgroups would typically use 2 or 3-member shards, unsharded subgroups could be large. In tests with a 128-member group, Derecho’s multicast and Paxos protocols were just 2-3x slower than for a small group, depending on the traffic pattern. With network contention, slow members, or overlapping groups that generate concurrent traffic, Derecho’s protocols remain stable and adapt to the available bandwidth.

Authors’ addresses: Sagar Jha, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA; Jonathan Behrens, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, behrensj@mit.edu; Theo Gkountouvas, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, tedgoud@cs.cornell.edu; Matthew Milano, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, milano@cs.cornell.edu; Weija Song, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, ws393@cornell.edu; Edward Tremel, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, edward@cs.cornell.edu; Kenneth P. Birman, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, ken@cs.cornell.edu; Robbert van Renesse, Cornell University, Department of Computer Science, Gates Hall, Ithaca, NY, 14850, USA, rvr@cs.cornell.edu.

0. 0734-2071/0/0-ART0 $15.00
https://doi.org/0.0

ACM Transactions on Computer Systems, Vol. 0, No. 0, Article 0. Publication date: 0.
INTRODUCTION

There is a pervasive need for cloud-hosted services that can guarantee rapid response based on fresh data, that scale well, and that support high event rates. The challenge of building them limits the creation of IoT services for applications that generate high rates of incoming data that must be filtered, transformed, analyzed, reliably archived and made available for queries from online machine learning applications, real-time analytics, or incident-response forensics.

A key issue is that in many such settings, the volume of “uninteresting” data can be huge, and unnecessary forwarding or storage is costly, forcing such systems to intelligently filter data and trigger urgent actions at the edge. Today’s cloud is a poor fit with the resulting coding style. Edge nodes are typically required to be lightweight and stateless. For read-only queries, the cloud serves requests using stale data \[19, 77\], despite the risk of inconsistencies. Update processing entails storing incoming data in the global file system; a back-end system will later reload it and process off the critical path. The resulting pipelines consume significant storage and introduce long delays.

Classic approaches to real-time data warehousing work well in smaller settings \[47\] but do not address these needs. Scaling is also a problem for databases that use two-phase update protocols with locking \[38\]. A recent generation of data warehouse products targets real-time applications, supporting streaming data acquisition and time-indexed queries (see, for example, Spark/DataBricks \[94\], Timescale DB \[49\], Open TSDB \[68\], RealTime FB \[24\], Influx DB \[30\], and Beringei \[71\]). However, none offers ways to customize the data acquisition or query pathways.

For developers who decide to build a solution from scratch, the underlying O/S building-blocks offer limited assistance. For example, although modern operating systems incorporate basic mechanisms for data replication (such as the Linux Distributed Replicated Block Device), overheads are significant and the functionality is not integrated with higher level programming tools or consistency mechanisms. Moreover, there is little standard support for managing distributed application structure, or ensuring coordinated reaction to failures.

Our response was to create Derecho: a thin software layer over the hardware, designed to assist the developer in creating a new kind of service that would run on more powerful machines, connect directly to large numbers of external sensors, and carry out such tasks as efficiently as possible. The system name refers to a form of intense storm characterized by powerful straight-line winds, and evokes one of our goals: to keep the hardware running at peak speed.

A Derecho application consists of processes on some potentially large set of computing nodes, one per node, structured as a set of subsystems that play distinct roles. In Figure 1 we see an application with four subsystems: the load-balancer, the cache layer, the back-end and the cache-invalidation mechanism (these specific examples are purely illustrative: the developer determines the number of subsystems, their roles, and provides the associated logic). Each corresponds to a subgroup (as in the case of the load balancer), or a set of shards (the three other subsystems). Subgroups and shards are typically stateful, updating data in accordance with the state machine replication
Building Smart Memories with Derecho

Derecho applications are normally structured into subsystems. Here we see 16 processes (small circles) organized as 4 subsystems, 3 of which are sharded: the cache, its notification layer, and the back-end. A process can send 1-to-1 messages or RPC requests to any other process, and can multicast to subgroups or shards to which it belongs, with consistent ordering and durability.

As Figure 1 suggests, we anticipate that Derecho applications will make heavy use of sharding and key-value load-balancing, in which requests arrive with some form of key that can be hashed to select a shard that will do the bulk of the work for that request. Shards would generally just have 2 or 3 members, replicating data for fault-tolerance and perhaps varying the level of replication slightly to handle hot-spots. Yet Derecho can also support other patterns. We might see pipelines in which a series of subgroups each is responsible for a different computational task, and tasks traverse from subgroup to subgroup. Another interesting case involves iterative machine learning, in which a leader multicasts machine-learned models or parameter updates to sets of workers. Here, the group could be quite large. Workers compute independently, then either propose model improvements that the leader combines, or exchange data in an all-to-all pattern. The algorithm repeats until the model converges. Since applications will have multiple subsystems, a single service might combine more than one such style of computing.

The challenge is to enable smart memory applications to employ whatever structure makes the most sense, but also to make efficient use of networking and storage. Services of this kind process data using machine-learning tools, replicate and store “interesting” data, discard uninteresting content, update knowledge models, and trigger urgently-needed responses. Modern networking and storage devices run at such high speeds that only a pipelined, asynchronous style of communication can keep them busy, moving data without software intervention. The machine learning community is well aware of this issue, and has created a new generation of scalable distributed machine learning algorithms that run asynchronously and minimize cross-node synchronization [36, 37, 66, 80]. Derecho seeks to be a good partner for such solutions, using protocols that continuously accept new data, continuously append data to storage structures, and that block only in the event of congestion or to reconfigure after a failure (which will be infrequent [83]).

Locks and other forms of blocking are common in today’s scalable systems. Derecho greatly reduces the need for locking by handling read-only queries on a lock-free path disjoint from the one used for updates. By holding multiple versions of data and indexing to select the best version...
for each query, Derecho achieves temporal precision and logical consistency even for data spread
over multiple shards: a novel form of temporal snapshot isolation [84].

Derecho also innovates in other ways. Derecho supports complex multigroup patterns and offers
consistent handling of their memberships, a requirement that apparently has not arisen in prior
work. Derecho’s RDMA performance represents a significant speedup over prior data-replication
systems. For example, Derecho can make small numbers of replicas faster than a single-core process
can make in-memory copies (memory-to-memory copying involves a single core interacting with
a single memory unit, whereas RDMA moves data from machine to machine, creating a highly
efficient hardware pipeline between two memory units). When making larger numbers of replicas,
Derecho sends data asynchronously on highly efficient overlays, yielding an exponential fan-out.
The transfers occur in blocks and are heavily pipelined, so that no matter how large the object,
the delay to make 4 replicas is just one block-transfer-time more than for 2 replicas, and so forth.
Making 128 replicas requires just 6 block-transfer delays more than to make 2 (a 1MB block can
be transferred in about 83ms). Batching is determined by the speed of receivers, offering a way for
lagging nodes to quickly catch up.

Derecho offers several forms of consistency. One guarantee enables a service to be sure it is
responding to requests from the freshest available data. A second ensures that when data is
replicated, the replicas track the identical sequence of updates. A third relates to roles: if component
P believes that component Q is playing a key role, then Q knows itself to have that role. A fourth
form of consistency is temporal: if an action is based on data as of time $t$, the data retrieved will
correspond to time $t$, up to clock synchronization limits. A fifth concerns causality: if event X was
followed by event Y (and hence may have caused or influenced Y), then if an application reads data
at time $t$ and observes event Y, it also sees X. More formally, this is the property that concurrent
reads occur along what Chandy and Lamport term consistent cuts [22]. Individual subsystems can
selectively disable forms of consistency that they do not require.

At its core, Derecho obtains consistency using atomic multicast and Paxos, but using protocols
that have been reformulated to maximize asynchronous pipelining. In particular, Derecho avoids
the leader-based pattern seen in traditional protocols, and by so doing removes a second significant
source of blocking or delay, this time in the data acquisition pipeline. Our new approach centers on
using asynchronous event flows to build up distributed knowledge [41]. Derecho’s protocols are
optimized to ensure that actions occur as as soon as it is safe to do so. Indeed, they can be proved
to achieve constructive lower bounds; we believe the system to be the first Paxos-based technology
for which such a claim is justified.

1.1 IoT Examples

One setting that has the needs just outlined involves smart highways [62, 70]. Future systems of
this kind will enable a real-time dialog between cars and a shared highway service, much as an
air traffic control (ATC) system coordinates airplane takeoffs, landings, and flight plans [93]. The
challenge is that an ATC system handles a few planes at a time [79], whereas smart highways
would need to control enormous numbers of tightly-packed vehicles.

Figures 3 and 2 illustrates a system capturing data from a diversity of sources: cameras deployed
on the highway and in vehicles, radar and lidar units, and so forth. Notice that even if the sensing
devices themselves help (for example by segmenting images at the time of capture), many tasks
require access to accurate, frequently-updated machine-learned models (such as known vehicles,
their trajectories, etc). In today’s cloud, we lack a way to accurately replicate and update machine-
learned models in the outermost layer of application-customized compute nodes: the “edge” of
the service. Furthermore, today’s edge lacks ways for nodes to communicate with one-another, or
to collaborate. This limits edge-computing to tasks a single node can perform on its own, using
A movie can be generated by combining sub-results

Massive inflow of real-time data

A smart memory interprets data using machine learning and analytics, then stores findings in temporal version-vectors. Processes communicate to compare data from distinct video streams. Within a shard, Paxos state-machine replication guarantees consistency, durability, fault-tolerance.

Data is acquired by a non-blocking pipeline, then saved in time-stamped version-vectors, which are replicated for high availability.

A shard can have more than one version-vector, but with a large or unpredictable number of streams, it can be easier to combine data from multiple streams.

Using standard C++ libraries for querying collections, a version vector can be treated like a database or filtered.

An intelligent edge-computing service could maintain and draw upon accurate, rapidly-updated knowledge models, and its components could cooperate, for example to correlate data from distinct streams or even to control distinct vehicles in consistent ways. Such a service could rapidly discard uninteresting data after the shortest-possible delay and with the minimum eventually do reach the edge, but after unpredictable delays.

Thus, today’s cloud forces the developer to relay most of the captured data to the back-end systems. This entails moving that data over the data center network, storing it into the global file system, and then requesting that it be analyzed. When the processing task is scheduled, the data would be reloaded for analysis by rereading it from the file system and again copying it over the data center network. When one considers the very large data volumes that arrive at the edge, such as video streams, a huge level of resources would need to be dedicated just to those activities even for uninteresting video such as a known vehicle following its expected trajectory. The long delays preclude urgent reactions.

In contrast, an intelligent edge-computing service could maintain and draw upon accurate, rapidly-updated knowledge models, and its components could cooperate, for example to correlate data from distinct streams or even to control distinct vehicles in consistent ways. Such a service could rapidly discard uninteresting data after the shortest-possible delay and with the minimum
waste of compute and storage resources. It could perform edge-tagging of interesting data, apply real-time updates to vehicle trajectory models, and react instantly if urgent actions are needed. By the time data is finally stored, the content would be reduced to the high-value input, which can then be sharded in a (key-value) manner and hosted directly in the same parallel architecture that captures it. The back-end would work with smaller amounts of high-value content, while being relieved from time-critical roles.

A smart memory would also function as a resource for the back-end nodes. Today’s cloud storage systems treat the edge purely as a load-reduction layer, not as a coherent service that might have an understanding of real-time phenomena, and be capable of responding intelligently to queries. A smart memory service could be used as the primary storage system for many kinds of data, such as the image data in our example.

The Derecho-based solution would thus represent a new kind of machine intelligence service: one that understands what it observes, decides what to ignore and what to remember, reacts immediately to urgent events, and later can draw on its accumulated knowledge to answer questions. By leveraging consistency, it enables rapid data-driven decisions that make logical sense. All of this centers on the integration of domain-specific machine learning techniques directly into the data acquisition and data query paths of a flexible, customizable scaffolding.

1.2 Cloud Service Examples

Related challenges are seen in many cloud-computing services. Facebook’s binary large object (blob) store [43] does not incorporate machine intelligence, yet has a structure much like that shown in Figure 1. Within each point of presence a load-balancer routes incoming requests to the proper machines. These run a sharded caching layer that tries to be smart about what it retains and is integrated with a subsystem that carries out data transformations such as image and video resizing. A back-end system stores data that needs to be persistent, plays offline roles such as tagging individuals in photographs, and includes an event-reporting infrastructure that notifies the cache of any updates or invalidations.

Consistency at the load-balancer would allow a load-balancing component of the blob store to have confidence that when a request is directed to a particular cache shard, the target node is genuinely playing the expected role. When a posting is edited or deleted, consistency would offer certainty that the old version will be replaced and will not linger for extended periods. Consistency is also important for system-wide configuration parameters and meta-data that evolve at runtime. On the other hand, the blob store is a read-only cache of self-certifying write-once objects, and Facebook would not want to pay extra cost for unnecessary guarantees, such as totally ordered updates or extra logging [9, 18, 43, 44, 86].

Microsoft’s Cosmos computational pipeline and storage service [32, 33] is an existing storage structure that was originally used for data compression, but is evolving into a smart memory in response to increasingly sophisticated opportunities seen in Microsoft’s Bing social networking infrastructure. Cosmos manages storage and computing systems (hundreds of thousands of nodes
organized into rack-scale storage cells). On arrival, each object is replicated for fault tolerance and to spread load. Next, Cosmos carries out compression, data mining, data analytics, and so forth. The output of each computational step is a set of additional objects, which are also replicated for reliability and in some cases, additionally processed. This yields a robust processing pipeline, similar to a multi-stage Hadoop or MapReduce task.

Many Cosmos objects are self-certifying: if an intact copy can be found, it is guaranteed to be correct. Furthermore, there is often a deterministic recipe to generate any desired object from inputs, so that if an object cannot be found, it can often be created. This justifies an optimization reminiscent of the end-to-end argument [81]: when handing off objects from one stage to the next, perfect reliability often is not required and might be wasteful, particularly if failures are rare. Cosmos therefore rejects a fancy fault-tolerant handoff, accepting some risk of data loss.

This example highlights a design principle. Derecho is positioned as a flexible software library that avoids imposing policies. Whereas many contemporary software systems offer black-box primitives with complex runtime behaviors, Derecho offers building blocks that permit a wide range of application choices, but have stable, predictable performance.

1.3 The Derecho Library

The Derecho library addresses these needs through a lean API. The core functionality is built from C++ objects of type `replicated<T>`, where T is some application-defined data type. A simple annotation scheme permits developers to designate methods that receive upcalls when an application’s membership changes, and to declare methods that will be used as message delivery handlers. Methods that take a byte vector as their sole argument are implemented with zero copy protocols, whereas method calls that involve more complex arguments are automatically marshalled in a way that minimizes copying both on the sending and receive side.

Data managed in a Derecho subgroup or shard is declared as fields of the `replicated<T>` class, and consists of versioned objects having type `volatile<T’>` and `persistent<T’>`, where T’ can be any serializable data type. The stored data could be large (video snippets, for example). Applications update version vectors by creating a new version (perhaps from the prior one), and then appending it to the vector. Old versions are read-only and are typically accessed by time.

Whereas a Derecho update occurs within a single subgroup or shard, a read-only data access may touch multiple shards or subgroups. For such situations, the application issues a set of RPC requests (perhaps concurrently), each using the same temporal index. The replies can then be combined to compute the result of the overall query. Derecho deterministically executes the read operations within a lock-free temporal snapshot with optimal temporal precision and causal consistency [22, 84]. Derecho’s file API maps standard read-only file operations into consistent data queries. Thus, data stored into version-vectors can be accessed directly from data-mining tools such as Spark/DataBricks [94].

We believe that consistency will often matter, hence Derecho employs a model that has been studied formally, and protocols long-known to implement that model. The main Paxos protocol we use is a classic one, originally proposed for a non-RDMA network (see Chapter 22 in [12]). Although we restructured it extensively to match an asynchronous RDMA setting, the underlying protocol logic is preserved, hence the existing proofs still hold. Derecho supports totally ordered message delivery with Paxos-style guarantees [53], which can be configured both with either non-durable or durable (NVM) state. In its raw mode, Derecho offers virtually synchronous membership management [16] but data movement is via our lowest-overhead protocol: this minimizes latency, but is weaker than a full-fledged atomic multicast. Despite offering a more complete programming model, Derecho equals or exceeds performance of prior systems with related functionality such as Corfu, Kafka, DARE, NetPaxos, APUS, NoPaxos and AllConcur [6, 29, 48, 59, 73, 75, 90].
Fig. 4. When launched, a process linked to the Derecho library configures its Derecho instance and then starts the system. On the top, process P discovers that it must restart the service from scratch; below, process S joins a service that was already running with members \{P,Q,R\}.

The system is open source, hosted at https://derecho-project.github.io. The system can run on any RDMA network via the Verbs library, and on non-RDMA hardware using the SoftRoCE emulation library.

1.4 Structure of this paper

The remainder of the paper is organized into three parts. Part I focuses on how Derecho is used. Within it, Section 2 presents the Derecho API and Section 3 describes the version-vector storage abstraction. The P2P and multicast API is described in Section 4, while Sections 5 and 6 look at the way that multicast delivery is integrated with a versioned storage abstraction. Part II shifts emphasis and looks at implementation. Sections 7 and 8 present RDMC, which provides a reliable RDMA-based multicast, and the SST, a replicated table that uses RDMA to report updates, then show how the Derecho functionality was implemented over these core elements. Section 9 presents our protocols in a more detailed way, discussing correctness and information theoretic efficiency. Part III is experimental: Section 10 focuses on Derecho’s performance, while Section 11 runs similar experiments using libPaxos, Zookeeper and APUS.

PART I. FUNCTIONALITY AND API

2 THE DERECHO PROGRAMMING MODEL

2.1 The Top-Level Group

Derecho’s overall functionality is defined with respect to a single top-level group that encloses the full membership of the application, as depicted by the outermost black oval in Figure 1. Each top-level group can be understood as a distinct instance of the application (multiple instances can coexist without interference, but Derecho treats each separately). A top-level group has a unique name: typically, an ASCII name concatenated with a unique instance identifier.

In what follows, we first discuss dynamic membership for this single top-level group. Then, we show how a top-level group can be used to automatically create subgroups and shards. Figure 4 illustrates two initialization cases. On the top a group restarts after a complete shutdown. Process P initiated the restart by calling the group constructor, which executed a rendezvous protocol that determined that the group was not currently active. Derecho will locate, repair, and reload the correct persisted state (green version vector). The top-level group then restarts with an initial membership view containing P. For a large-scale application, restart by a single process might not be adequate to enable normal operations; Section 2.4 explains how this is handled.
Building Smart Memories with Derecho

Fig. 5. Multicasts occur within subgroups or shards and can only be initiated by members. External clients interact with members via P2P requests.

Fig. 6. If a failure occurs, cleanup occurs before the new view is installed. Derecho supports several delivery modes; each has its own cleanup policy.

On the bottom we see the case of a joining process that discovered an active instance of the application. Here, process S is added to the existing membership, and this defines a new view, which will start a new membership epoch: a period of execution that starts with installation of a view, ending when Derecho terminates activity in the epoch and switches to the next view, and the next epoch. Because S is joining an active group, it needs to synchronize with the other members. For this purposes, state is transferred from R to S. The state can be understood as a checkpoint and could contain a full copy of the data the group manages. However, if S was previously active and posses old state, it is often more efficient to finalize any updates that were pending when S crashed, and then to append a suffix of more recent log entries that S missed while it was faulty. Derecho automates the decision of which mode to use, obtaining the required state by calling a method that serializes the contents of the relevant objects within R, then deserializing them on arrival in S. After the view is installed, and state is transferred, a period of normal execution ensues during which the group members send and receive messages.

In Figure 5 we see one point-to-point request (an asynchronous send: there is no response\(^1\)) from an external client, C, to a group member that acts as its proxy, P. P relays this request into the group, where method handlers are invoked when the messages are received. The application issues requests through polymorphic method stubs.

In Figure 6 group members are sending totally ordered multicasts when a crash occurs. Although issued concurrently, multicasts within any single group will be delivered in the same order to all members, including the sender. Notice that the failure disrupts one of the multicasts. Derecho itself never initiates a retransmission in such cases. Instead, the sender is informed that the multicast failed (it can chose to reissue it), and a new view of the top-level group is defined, containing only \{P, R, S\}. Any disrupted multicasts are cleaned up, as described in Section 9.3. In large services spanning multiple racks in a data center, it is sometimes necessary for an entire rack to be taken offline or brought online in one step, hence Derecho joins and leaves can add or remove multiple processes per epoch.

Methods can also return results. Figure 7 illustrates a state-machine style query followed by an incast of replies, and Figure 8 shows a case in which a query is issued to members of several distinct shards. The former style of query actually runs within the Derecho data acquisition path; it occurs within a subgroup or shard and is really a totally ordered multicast that invokes a non-void function. This affords an opportunity for parallelism (the members can subdivide a task, such as searching for a particular data item). On the other hand, this approach to querying state can only access data within a single subgroup or shard.

The second style of query is the one best matched to our primary use case (asynchronous pipelined data acquisition with out-of-band read-only queries). Here the query can access data

\(^1\)Not shown is a low-level acknowledgement generated by the network layer to confirm successful receipt.
Fig. 7. A multicast initiated within a single subgroup or shard can return results. Here, the handler would run on the most current version.

spread over many shards. Moreover, because the temporal access model itself provides consistency, there is no need for the query to be ordered relative to updates.

If a query only accesses past data, it will never block. In contrast, a query that tries to access the future will wait until the desired time is reached. If a query accesses very fresh data, Derecho may be able to perform it immediately, but the query will be forced to wait if there is any risk that an in-flight update could fall into the temporal window covered by the query. This guarantees that the result of the query is complete and deterministic. Such delays are small: tens of microseconds.

Derecho assumes that multicast handlers that modify subgroup or shard state are deterministic and non-blocking. They need not be thread-safe (Derecho issues upcalls to the application from a single thread). If the user employs additional application-specific threads, the associated locking would also require close scrutiny both to ensure that determinism is preserved, and because extended delays could impact the performance of the entire system. The core problem is that if the Derecho upcall thread blocks, Derecho will eventually become congested, resulting in backpressure that would ultimately delay incoming sensor data.

2.2 DerechoGroup<...> and Replicated<T>
The top-level group in Derecho functions as a distributed container that hosts a set of subgroups. Developers extend the top-level group by registering classes of type replicated<T>, where T is user-defined. The full set of types that a group will use is defined when the application first creates a handle on the top-level group, which will be of type DerechoGroup<T0, T1, ...>. The constructor takes parameters used in the initialization sequence we discussed in Section 2.1 and a list of factory functions for the replicated<T> objects, for use when Derecho creates new subgroup or shard instances, as discussed below.

2.3 Subgroup and Shard Membership
The first question is how to map the top-level group membership to the subgroup and shard membership. For this purpose, the user employs Derecho’s built-in membership function, which can be parameterized to control the mapping of each replicated<T> type registered in the group.

For a given type, one of two cases applies. (1) For a non-sharded subgroup, the mapping function simply computes the subgroup membership from the top-level view. (2) For a sharded subgroup, the mapper also breaks down the subgroup into a series of shards. Most commonly, no process is mapped into more than one subgroup or shard at a time, but there are two exceptions, discussed below. If the top-level membership includes spare processes, those are simply held in reserve: because they are not mapped to any role, they will not send or receive any Derecho messages, belong to any subgroups or shards, or even have a way to access the current group membership...
Building Smart Memories with Derecho

Our mapper dynamically optimizes process layout, taking into consideration such aspects as machines that may have special hardware or particular files on them, information provided by the owner about failure domains, etc. The approach allows the number of members to vary shard by shard (for example, to accommodate differing loads), but requires that the total number of shards within any given subgroup be fixed. It also seeks to minimize churn. Consider the cache layer in Figure 1 and assume that the lowest-ranked member of the top-level group fails. A trivial layout function might assign three members to the load-balancer subgroup, then count off by threes to define the cache-layer shards. A failure that shifts the rank in the top-level view to the left, however, would then impact sharding of the cache layer in a drastic way: every member takes on a new role! Our presupplied membership function therefore keeps processes in the same roles even as membership changes. That is, if some shard had membership \{P,Q,R\}, but then P fails and process S is assigned to replace it in the next view, the new shard membership would be \{S,Q,R\}, with S thereby “taking over” for P. Q and R remain in the same shard and retain their prior rankings. While S does need to be initialized via a state transfer, this is much cheaper than shuffling the entire layout. In work still underway, we are extending the layout function to attempt to place restarting processes into their prior roles, so that persisted state can be recovered and reused.

Although we use the capability sparingly, Derecho can support overlapping shards, as with the purple event-notification shards in Figure 1 used to invalidate or proactively update cache contents. We provide a “cross-product” membership function to cover such communication patterns. Overlap is also seen when doing key-value sharding. Many key-value subsystems make \(k\) copies of each (key, value) tuple by placing data at the node to which some key maps and then replicating it on the subsequent \(k-1\) nodes. Each node thus is primary for 1 set of keys, and a backup for \(k-1\) others. The resulting shards overlap, together covering the entire key-value store.

In many systems, the primary goal is to replicate data to nodes in distinct failure domains, yet today’s operating systems lack standard ways to expose network topology, failure domains, or other common-mode dependencies. As a consequence our Derecho membership functions lack a way to determine which node is in which failure domain (an example of a system that does have this data is Google’s Slicer, which leverages it to offer very sophisticated sharding behaviors [5]). In future work, we plan to extend our current solution to support a plug-in whereby the developer could supply such information.

### 2.4 Inadequate Provisioning

A membership layout might not always be feasible. For example, suppose that a layout requires at least \(k\) processes, but the pool of processes available for the next top-level view currently has fewer than \(k\). In such situations, we consider the top-level group to be inadequately provisioned. Other examples include a situation in which some shard is depopulated, or where a system is trying to restart, but needs access to files persisted on nodes that have not yet recovered. The layout function signals inadequacy by returning an error.

Derecho’s main approach to handling an inadequacy is to simply wait for additional processes to join. For example, upon restart after a total shutdown, Derecho designates the first process that restarts to play a special role: it collects a list of the recovering processes and information about the nodes on which they are running. If the corresponding membership view would be inadequate (discovered by attempting to run layout on the view before proposing it), the entire restart pauses until enough processes have joined.

---

2If the number of shards varied in a view-dependent manner, a subgroup that had \(N\) shards prior to failure might restart with \(N' \neq N\) shards, and restart would require Derecho to shuffle data to fit the new configuration: feasible, but costly.
wait until additional processes recover and request to join, so that the view would be adequate. Similarly, if a failure leaves an inadequate set of operational processes, Derecho finalizes the current view but then waits until an adequate set becomes available before switching to the next view.

During periods of inadequacy, the service will be wedged. The current view terminates, but the next view is not yet known, and hence cannot be installed immediately. To ensure that this will not cause the application to malfunction, we report inadequacy via an upcall. Although Derecho is wedged during periods of inadequacy, notice that read-only access to replicated data could actually continue, if desired, since the Paxos semantics ensure that any data still available will be consistent.

The important thing is that the application must not initiate operations that might attempt to access a failed process or a depopulated shard (such an error is analogous to dereferencing a null pointer).

## 2.5 New-view Upcalls

Earlier, we indicated that Derecho notifies system members each time a new view becomes defined. Membership views are numbered sequentially from 0, and from the moment a process joins Derecho until the moment it leaves the system, it receives every view for the top-level group. Within a view is a vector of the members, ranked in an order that reflects when they joined the system. Thus if the view is \{P, Q, R\}, then P has rank 0, Q rank 1, and R has rank 2. The notification is identical in all members, hence the logic implementing a subgroup or shard can make use of the current view, and can safely assume that all the members of that subgroup or shard will do so in a consistent manner. A view also includes a list of members that joined in the current view, a list of members that departed, and a function that returns the current rank of a process given its Derecho process-id (the process-id is just a small integer, but it can be used to obtain a member’s IP address if desired).

In Derecho, every member of the application learns the subgroup and sharding structure each time the top-level view is updated. This makes it easy to load-balance over the members or to ensure that a request will be sent to a specific member (when leader-based algorithms are implemented in Derecho, the member with rank 0 often plays the leader role).

The first new-view upcall event occurs on all of the top-level group members, and reports the new top-level view, which contains the new subgroup and sharding layout. The upcall is delivered to a method that each process registers (as a constructor argument) when it joins the top-level group. Because the subgroup membership function is deterministic, every member sees the same mapping. Next, each subgroup and each shard to which a process belongs will receive its own new-view upcall, with a view specific to that subgroup or shard.

When a member first joins, the first view it receives shows itself joining. If a member is ever removed from a subgroup or shard (other than because of a failure), the last view it receives will show itself departing, and it will not be listed in the final ranked list of members.

## 3 VOLATILE<T> AND PERSISTENT<T>

The application designer is responsible for defining the replicated state for each `replicated<T>` class. Such state must be stored in serializable fields of the `replicated<T>` object. To update replicated state, the application calls `ordered_send` to multicast the update request. Members perform the update operation when multicast delivery occurs. Read operations that access replicated state require no remote interactions and are carried out purely using local data.

An application can maintain its own representation of state, as variables within a `replicated<T>` object. However, although their use is optional, Derecho additionally offers two forms of version-vector for storage of temporal data. Both are templates that the developer customizes by providing a

---

[Derecho provides a built-in serializer optimized for RDMA applications. If objects have a direct byte-vector representation, Derecho detects this at compile time, and initiates RDMA operations directly from the object into the target.](#)
serializable base type. (1) volatile<T> declares a version-vector of objects of base type T, indicating to the system that they will reside in memory (in DRAM). Such data is replicated within the subgroup or shard where it resides but would be lost in the event that all members crash simultaneously. (2) persistent<T> declares a durable version-vector: the objects will automatically be written to NVM files after updates and automatically be reloaded when the system restarts.

In Figure 9 we see a single persistent version vector; in practice, each member of the subgroup or shard would have a complete local copy with identical content. This invites comparison with the Paxos protocol [53, 87], which maintains message logs at processes playing what Lamport characterizes as an acceptor role. The analogy is valid: if we had objects of type command (Paxos often uses this term), a version vector of type persistent<command> would be a Paxos log.

Paxos is typically defined so that each write needs to occur on just a majority subset (a quorum) of the acceptors, which means that any single Paxos log might have just a subset of the data (the learner protocol, used to read Paxos data, operates by merging a quorum of logs). In contrast, every Derecho version-vector is fully replicated, corresponding to a Paxos write-quorum that includes the entire subgroup or shard. Derecho can run in this mode because it reconfigures membership after a failure: the virtual synchrony membership model [12, 16].

All Derecho version vectors implement two sort orders: one by version number and the other by time. Nonetheless, for any given vector, only one mode of use would be employed. Version ordering is a sequential ordering in which version \((i + 1)\) of an object is created by mutating version \(i\). The version number will be an integer and can be used to index into the vector. Temporal ordering arises from timestamps specified by the application, or captured from the clocks on the computers running Derecho. Here, the temporal sort order would be used when indexing into a version vector to perform a read-only query. If the user-specified type includes a timestamp, Derecho will use it. If not, the timestamp will be the clock time when the object was created.

Each version vector supports a set of internal methods that are used by Derecho but not normally called from application logic. These include vec.new_version(), which creates a new version that will initially have the same value as the prior version, and vec.persist(), which initiates an asynchronous task that will persist the tail of the vector (the red objects in the figure), by doing an NVM disk write. This operation is a no-op in the case of volatile<T>. The method vec.last_persisted() returns an integer \(n\) such that every version from \(0...n\), inclusive, has been persistently committed.

The system does not throw exceptions, but mixed-mode access, in which a single version vector is sometimes accessed by version number and sometimes by time, can exhibit unexpected behavior because the two orderings will often differ.
persisted. Finally, the method vec.reset_version(k) is used by the platform when cleaning up after a failure: it trims a set of pending versions from the end of the vector.

Version vectors also support functions that the developer can access directly. With “version indexing” the application treats the vector as a sequence of versions that will arise from the round-robin delivery order. The application can read the most current version by simply accessing vec as if it were a variable of the given type, and it can similarly modify the current version. vec[k] will access version k, and as a convenience to the developer, vec[-1] will access the previous version. The function vec.length() returns the version number of the last object in the vector, and vec.truncate(int n) truncates versions 0...(n – 1). This allows an application to reclaim space for versions it will never revisit.

With temporal indexing, the application accesses data (only) using time as an indexer: vec[t], for a time t. Here, Derecho automatically maintains versions in a compact form, as a sequence of values that held for periods of time. The first access to a version searches this index (in time \(O(\log(n))\) but the results are memoized, and subsequent accesses run in \(O(1)\) time.\(^5\)

The issue of when persistent version vectors need to be flushed to disk poses interesting tradeoffs. Throughout this paper, we assume that no update is durable until flushed, and that no replicated write can commit until it is guaranteed to be durable across failures. This involves costs: flushed writes are slow. For example, with the models of SSD we use in our experiments a flushed write costs 900us. However, newer models of SSD offer sharply higher speeds: 10x or better improvements, with 100x improvement in the case of Intel’s new Optane hardware. Unflushed I/O, such as with Zookeeper’s popular FlushSync=no configuration, runs some risk of data loss: if a crash occurs while I/O is still buffered, those writes can be lost. Zookeeper’s documentation recommends that the user who desires higher speed use this feature, but compensate by periodically invoking a file system write barrier (the Linux FSync operation), or by storing data in a RAID SSD with battery-backed DRAM cache. In future work, we plan to explore the pros and cons of departing from the Paxos model in these ways, but it is interesting to realize that the new hardware may reduce the performance gap sufficiently to make the question moot.

As noted earlier, Derecho’s version vectors are accessible from a file system API. The system currently supports the HDFS API, which is fully POSIX compliant, as well as the Ceph object-oriented storage API. Both HDFS and Ceph have snapshot mechanisms, which we overload to obtain the temporal index.

4 P2P AND MULTICAST FUNCTIONS

There are several ways to communicate with a Derecho-based application. The first was seen in Figure 1 where, on the top left, a set of external clients interact with group members using REST or WCF remote method invocations. Derecho plays no role on these communication paths, which are supported by standard, widely used packages. On the other hand, RDMA would only be used for such a path if the RPC package employs it.

The second involves messages exchanged when members of a Derecho application interact with other members of the same application. Figure 5 illustrates one such use: a client process sends a request to a group that causes the method RequestFoo to be invoked in member P. P then forwards the request as a multicast to the full membership. All execute method Foo in parallel. By using Derecho’s built-in APIs the developer benefits from the strongly typed compile-time checking offered by the system, is able to leverage RDMA, and obtains semantics integrated with the Derecho

---

5Because this second mode of operation sorts data by time, it is important that the event handler that creates such a version simply store new data into the version vector, and not try to create a new version by mutating a prior one. The reason is that while capturing real-time data, there can be a delay until all updates from a period of time are stored, and during that brief period, the “temporally prior” version may not yet be available.
consistency model. Within a Derecho application, two TCP-based unicast communication options are supported. A p2p_send is an asynchronous message that will be delivered in the same order used by the sender with no data loss, duplication, or corruption. A p2p_query is similar, but includes a reply sent back to the sender, carrying the value returned by the invoked method returns a value.

In both cases the application must identify the subgroup to which the request is being sent as well as the specific member within that subgroup that will be the target of the operation. To do this, a handle on the subgroup is created. For example, if the cache layer in our example is associated with the class replicated<MemCacheD> and is the \( k^{th} \) subgroup of type MemCacheD, the code to obtain a handle for P2P requests to the subgroup is as follows:

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

Note that \( g \) is the Group handle representing the top-level group, and ExternalCaller<T> is an interface for replicated<T>. This interface is restricted to P2P messages, and is available to any process (in contrast, the multicast interface is available only to members). Next, we can send an asynchronous message by invoking

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

The outcome variable (which has a type inferred by C++) can be used to track the progress of the send until the delivery acknowledgment, which is actually obtained from the RDMA hardware itself (an RDMA completion will report success only if the data was successfully stored in the memory of the target process, identified by \( \text{who} \) in this example).

To perform a remote procedure call, the application issues a similar request, but now the method responds with some sort of object:

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

Here, the return type of the operation will match the return type of the target method (get() with a string argument, in this example).

We also support list versions of both P2P operations; these concurrently send to a set of targets, or concurrently query a set and return a list of responses. A list is actually less efficient than a subgroup for reasons that will become clear in Part II. Hence, for repeated patterns of communication, we strongly recommend creating a subgroup or shard and performing the operation using group multicast. However, some IoT systems have unpredictable query patterns. This was seen in our smart-highway example: the application creating the film strip in our running example is probably the first to have ever seen a motorcycle driving at high speed in this particular pattern. Thus, when it computes the list of shards that may have video for that vehicle, the list is basically a one-time set. This would therefore be a situation in which use of the target-list API would be appropriate.

Notice that both of the examples use an RPC tag to match the P2P or multicast query with the desired target method. By resolving method invocation at compile time, we obtain a nearly direct code-path from Derecho’s caller to the RDMA hardware and then into the receiver method, and can also signal type-mismatches as compile-time errors. This is similar to the code-path in Von Eicken’s U-Net system [89], the first paper to propose RDMA functionality.

In future work we will explore mapping such operations onto the framework Derecho implements for RDMA communication. Issues that arise include managing buffers and avoiding delays on Derecho’s update and query paths.

More precisely, the result object from a query can be used to track progress of the request, just as for a P2P send. However, it has an associated getter method, and this method will be called if result is referenced in an expression. The method returns whatever type the target method returns. Thus, result behaves like a value type if used by value, but like an outcome object when tracking an asynchronously-issued query.
Process crashes or network failures result in run-time errors. The outcome object can be queried to track progress of the request and will signal that it failed, and in the case of a query, an attempt to get the value of the result will throw an exception.

Multicast messages sent within a subgroup or shard similarly break down into two cases. A multicast is performed by issuing a call to ordered_send from within a subgroup or shard and will be delivered in order to all the members (the sender itself included). The syntax is very similar to that of the P2P send:

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

Here, a full replicated<T> handle is used, and we no longer need to specify a specific process as the target of the request since all members of the subgroup will receive it (including the sender, which must be a member). If the subgroup is sharded, the request will be delivered to all members of the sender’s shard of the subgroup. Again, the transmission is asynchronous and delivery will be reliable, but it will also be ordered relative to messages from other senders: Derecho imposes a strict round-robin ordering within the senders actively sending to a subgroup.

A call to ordered_query starts in the same way as ordered_send, but here the invoked methods return values, and the set of result values is returned to the sender, which can iterate through them as they are received. Much like the P2P operations, these functions are asynchronous and can be tracked via the outcome or results object. A P2P message is initially in state accepted, then transmitted and finally acknowledged, meaning that the RDMA hardware layer has confirmed successful delivery. A query has a final state capturing replies, and the application can use the results future as an iterator to read the replies out (they will be of whatever type the remote method returns, in the order they were received). However, P2P messages should never modify replicated group state, whereas both ordered_send and ordered_query can safely do so.

Derecho allows the developer to specify which group members will be active senders in a given membership epoch. At runtime, processes authorized to send in the epoch can dynamically signal that they will cease to send or that they wish to resume sending (a process not initially in the sending set cannot, however, join the sending set until the start of the next epoch). Given the list of senders (which all members track in a deterministic way), the delivery order is strictly round-robin among the authorized senders. This is unlike many previous protocols, in which the ordering is non-deterministic, and permits us to simplify our protocols, as will be seen below.

5 TEMPORALLY PRECISE, CAUSALLY CONSISTENT INDEXING

Derecho’s version vectors support two modes of use: version indexing (version 0, 1, ...), which will respect the round-robin ordering on updates, and temporal indexing, in which each version is considered to be the value for some span in time. A single application could use multiple version vectors in a single replicated<T> class, treating each one differently, but any single versioned object should only be accessed in one mode.

In Derecho, temporal data is easily generated and tracked, but by the time a multicast is delivered to replicate the data into a version vector, a delay may have arisen between when the data was captured and when it is stored, particularly in heavily pipelined systems capturing large objects such as video snippets. As a result, the interaction between multicast and temporal indexing poses some challenges. There are two cases:

**Single-writer objects.** Often an update is generated by a sensor or some other source that has access to an accurate real-time clock, and is the sole writer to the version vectors in which the update will be stored. Here, we record the originator’s real-time clock value, so the only overhead is the very small cost of the timestamp field itself.
**Multi-writer objects.** A slightly more complex situation arises when an update originates in some system component that has causal (read-write) dependency upon data that was read from Derecho. Here, three additional issues enter the picture. First, we need a way to track the causal state of such a component, which is done using a library layer that implements a logical clock using a method proposed by Kulkarni [50] and based on Lamport’s logical clock methods [51]. The Kulkarni method tracks both real-time and logical time, yielding what he calls a hybrid logical clock: an HLC clock value includes the real-time, but it also includes a logical clock that represents the causal dependency state of the initiating process. Song et al., show how to take advantage of this to tag updates so that subsequent queries will be causally consistent [84].

The main challenge that arises when employing this method within Derecho is associated with multicast. If a multicast will carry HLC values, the Kulkarni method for generating a consistent cut requires that data replicas all have the same HLC value (which is easy, since the value originates at a single sender), but also that the stored data be sorted in increasing order of HLC value. This, however, might not be the ordering employed by our multicast protocol. Thus, there is a brief period during which new updates might actually “jump backwards” in HLC time. The temporal read algorithm therefore should delay any read that attempts to access a very recent portion of the version vector, since it may not yet be stable for the time period being accessed.

**Read-only temporal queries.** The power of the temporally indexed vectors becomes apparent for read-only queries, which can access any set of processes (even a set that does not belong to some single subgroup or shard), and does not require ordering relative to multicasts that deliver updates.

The key insight is this: because a temporal query does not modify data, so long as it accesses the stable portion of the version-vector, it will not matter when such a query is performed; all that matters is that Derecho’s implementation of temporal indexing is deterministic, temporally precise, and causally consistent.

For example, suppose that some system is updating version vectors foo and bar at a very high rate, faster than the real-time clock can track. Now suppose that some process P reads from foo[t] and sees value x. First, Derecho guarantees that the value read will be a value that held at time t. Next, Derecho guarantees that if this is the first read P has done, x is the “earliest” value that foo had during the time period that included t. Finally, provided that P is linked to the Derecho library, a subsequent read or write to bar will capture the causal sequence.

For example, suppose that after the read from foo that retrieved x, P reads bar[t]. Derecho guarantees that the value returned, call it y, will be at least as current at x: the reads occur along a consistent cut [22]. Similarly, if P writes y into bar after reading x from foo[t], Derecho will capture the causal ordering. If Q later reads bar[t’] and sees y, then a read of foo[t’] by Q will return x (or a subsequent value).

To obtain these guarantees, Derecho sometimes introduces a slight delay on queries that try to access the current time. The issue is as follows: during a multicast that will update version vectors, a message might contain data with an HLC timestamp smaller than prior, previously saved versions. This can occur because senders in a group will have slightly different clock values, and because the round-robin delivery order might not be in order of increasing HLC timestamp. Accordingly, Derecho implements a protocol that asynchronously tracks the temporally stable portion of each replicated version vector, namely that portion that (1) has been persisted by all group members, and (2) for which no earlier write could still occur. Then the system forces a query to delay if it tries to access the unstable portion of a version vector.

The logic used to track temporal stability is as follows. Each process continuously maintains a local variable that tracks the smallest timestamp that could originate from it, and hence that might still be applied to the version vector. If multicasts are underway, this will be the minimum over...
pending multicasts that are not yet stable. If no multicasts are occurring, the value will be based on
the local real-time clock value, updated at a frequency that the user can control (by default, every
1ms). Each process reports the resulting value to other members of the subgroups and shards to
which it belongs using Derecho’s shared state table, which we describe fully in Part II. Prior to
performing a temporal query, a process computes the stability frontier: the minimum time reported
by members of the subgroup or shard in which the version-vector resides. Temporal queries are
delayed, if needed, until the frontier of stability has grown to include the time index of the query.
In our experimental section, Figure fig:e2e will measure the temporal stability delay, showing that
for small objects it would rarely exceed 10ms (obviously, for large objects, the network transfer
time can increase this value).

Earlier, we noted that Derecho’s version vectors can also be accessed via the file system API. This
case is discussed more fully in [84]. The paper also provides detailed examples of HLC timestamps
arising under various temporal and causal ordering scenarios, illustrating the manner in which
HLC timestamps are used to satisfy reads in a way that guarantees consistency.

6 STATE-MACHINE REPLICATION

The delivery of a state-machine multicast to a version-vector of data poses further problems if the
data is persisted. Here we start by laying out all the cases, then drill down on the case of durable
versioned data, showing why this requires an additional mechanism, and how it works.

6.1 Three modes of operation

When a subgroup or shard is defined, its mode of message delivery must be specified.

(1) In raw mode, overheads are minimized: Derecho respects the sender’s message ordering but
does not order concurrent multicasts and does not clean up after a crash.

(2) In Derecho’s fast totally ordered (atomic multicast) mode messages are totally ordered, but
not logged to NVM prior to delivery. This mode is recommended for groups that have no
state, or that only use volatile<T>.

(3) In durable totally ordered mode, persistent<T> updates are saved to NVM on initial delivery,
and are automatically recovered after restart from a total failure.

6.2 Message Delivery

In each epoch, the layout function plays a secondary role of identifying the active senders
for each subgroup and shard. Any such member is expected to send a stream of multicasts. The key
difference between the modes involves the way that incoming messages from our data transport
layer (which respects the sender’s message order) are handled:

Raw mode. For this case, Derecho delivers messages as soon as they are received.

Although updates carry timestamps provided by sensors, a Derecho process with local clock time T only accepts updates
with timestamps in the range [T-\Delta,T+\Delta] where \Delta is a configuration parameter. This prevents attempts to overwrite the past.
The actual value of \Delta would normally be the sum of the worst-case network delay, measured from when a sensor value is
reported to when the multicast to replicate that value would occur, and the clock-skew. The smallest possible value of \Delta, 0,
would be seen in a system where the sensors are directly attached to the nodes that replicate sensed values and share a
single hardware clock.

If a sender has no message to contribute in a given round, one option is to send a null message. If the situation will persist,
it can instead send a temporarily not sending message. Upon delivery, this drops the sender from the round-robin delivery
order. To resume sending the member would use P2P send to request that the rank-0 member of the group multicast a
member k is resuming as a sender message on its behalf. Upon deliver, this adds process k back into the list of active senders.
The rank-0 member has a slightly different behavior: if it has no messages to send, and no other sender is active, it simply
pauses sending (rather than sending nulls) until new outgoing requests are initiated.
Fast totally ordered (atomic multicast) mode. For this case, Derecho buffers incoming messages if necessary. When messages $m_0, m_1, ..., m_k$ can safely be delivered, an upcall delivers the entire batch. Safety has two aspects. First, the sequence must be gap-free and in the message ordering defined by the round-robin delivery order among the currently active senders. Additionally, we require that before delivery occurs for $m_i$, every subgroup or shard member has a buffered copy of $m_i$. Part II explains the protocol that implements this rule.

Durable totally ordered mode. The durable totally ordered mode is a bit more complex: here, Derecho employs a two-step message delivery model, very much like a transaction that first performs updates, then commits them. The difference is that a transactional database treats “abort” as a program-callable API, whereas Derecho has no such API, and aborts an update only if a failure forces a cleanup. Here we focus on the behavior as seen by the user, deferring protocol details to Part II:

[i] Update, step one. In the first stage, delivery occurs as soon as a message can be delivered in the round-robin delivery order among the currently active senders, without buffering messages until every sender has a copy. A new version is added to the version vector for any $\text{volatile<T>}$ or $\text{persistent<T>}$ variables in the target subgroup or shard, and the upcall to the user-specified handler occurs. Each update builds on the prior one: $m_k$ creates the version that will be used as an initial value by the handler for $m_{k+1}$. Because the delivery order is the same in all members and the update is required to be deterministic, all replicas will be identical.

The Paxos model requires that (1) all members perform the same updates in the same order, and (2) if any member performs an update, then all will perform the same update, even if failures occur, and (3) should the system crash, any restart will recover the $\text{persistent<T>}$ versions that were committed. Clearly, we cannot yet deem the versions created by the first update state to have been “committed” in this sense, because if P were to crash after the first phase update, the failure could prevent the update from reaching Q and R, and Derecho might finalize the view and move to a new epoch in which that update had effectively been erased. Thus, P’s version must be viewed as provisional, like an uncommitted transaction.

[ii] Update, step two. In the second stage, Derecho automatically calls the $\text{persist()}$ methods for the new versions. For $\text{persistent<T>}$ objects, the call starts an asynchronous process of writing versions to NVM storage. Our method of doing these writes is ordered (version $k$ cannot become persistent unless all versions numbered lower than $k$ do as well) and batched for efficiency (to minimize the number of distinct NVM I/O operations).

Recall that by calling the $\text{vec.npersisted()}$ method, Derecho is able to track the completion status of persistent writes. It does so, sharing this data with other subgroup or shard members through an all-to-all information exchange protocol discussed in Part II. By taking the minimum over the reported values, we obtain a version number such that every member has performed the update and persisted the resulting data.

As will become clear in Part II, once all members have persisted versions up to $k$ (meaning that $k$ is the minimum among the reported persisted version numbers), all processes can concurrently deduce that version $k$ has committed, with no further exchange of information required: the system no longer could lose or erase the pending versions.

---

It should be noted that although the Derecho delivery rule as currently implemented does not order messages sent to different subgroups or different shards, the same protocol can be embellished to use more elaborate pre-agreed delivery orderings, which could allow variable sending rates, enforce ordering even between distinct shards, introduce hierarchical ordering policies (for multicasts sent to an entire subgroup relative to ones sent to its shards), etc.
6.3 Integration of Persistent Memory with Queries

Earlier, we noted that Derecho offers two forms of queries: multicasts that return a value, and time-indexed ad-hoc queries.

The first form is useful in applications that need to know when certain updates have committed (and hence are no longer at risk of loss in the event of a crash). Because \texttt{vec.ncommitted()} cannot advance to include \texttt{vec.current_version()} until after the handler has returned, an update handler cannot simply block until the commit occurs. Accordingly, for this purpose the update handler instead should return the value of \texttt{vec.current_version()} to its caller. The caller can then wait until \texttt{vec.ncommitted()} == \texttt{k}. Either this will occur within a few milliseconds, or a new view will be delivered, in which case the caller learns that the update failed. Derecho offers an API, \texttt{vec.delay_until_safe(k)} that packages this barrier functionality in a standardized form.

The second form of query interacts with \texttt{vec.ncommitted()} through a mechanism internal to Derecho: when a persistent object is indexed by a temporal query, Derecho waits until the temporal window of the query is entirely committed. This ensures that any query that accesses temporal data obtains a result that is deterministic, temporally precise, and consistent. Moreover, such a result is certain to persist in the event of a crash.

6.4 Masking Failure During Interactions With a Derecho Service

Although Derecho facilitates implementation of services that can self-repair and recover from failures, this does not prevent users from experiencing disruption.

The primary way that external processes and systems interact with a Derecho application is through a client-facing subsystem. Commonly, these use web-services event handlers to accept data from sensors, to interact with actuators, and to accept queries from external systems. Here, each request arrives at a single process, which must then forward it to an appropriate subgroup or shard (if the request mutates state), or query some set of subgroups or shards (if the request is a read-only operation). In standard implementations, web services employ a single TCP session for each connection from a client to a cloud-hosted service. If this TCP session breaks, ongoing interactions report loss of connectivity even if the requested operation actually completed successfully.

There are several ways to mask TCP connection breakage when an end-point server crashes. One option is for the application to explicitly create redundant connections and send requests in duplicate, then have the server end-points deduplicate them for at-most-once execution. A second option is for the service itself to employ a transparent TCP fail-over solution [85].

The question then arises of whether or not to build such mechanisms directly into Derecho itself. We considered doing so, but could not identify any approach that would not impose additional costs even on applications not using the mechanism. With this in mind, we leave it to the application developer to implement failure masking directly in the application, if desired.

A similar trade-off arises internal to Derecho, when subsystems within a single Derecho application interact with one-another. For example, consider a service that employs a pipelined processing model in which each computational stage is handed by a distinct subgroup. Because Derecho focuses on state machine replication, the challenge is to build a series of state machines that can talk to one-another robustly: a problem studied by Cooper, who proposed the Circus methodology [27]. Circus is a software-design pattern, hence the question arises of whether to standardize Circus as a core part of Derecho. However, Circus is costly, and many IoT services that need high availability would not actually require fault-tolerant task handoffs [13]. Accordingly, we decided again to leave both the choice to use such a mechanism, and its implementation, to the developer.
PART II. IMPLEMENTATION

7 OVERALL DESIGN CONSIDERATIONS

In this section we turn to the question of how we implemented Derecho.

Control Plane / Data Plane Separation. 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 [11, 72]. Any protocol that will keep the RDMA engine continuously busy must emit long sequences of tasks.

Strong Programming Models. Derecho’s consistency model centers on Paxos. Historically, protocols in this class are leader-based [16, 53]. The leader is given a request, or perhaps a batch of requests. It now engages in a multi-step interaction with other protocol participants, first sending a round of messages, then waiting for a set of responses, while attempting to secure a quorum majority vote on a proposed ballot, and then attempting to notify participants of the successful outcome. This pattern of delay is at odds with the RDMA mode, because at RDMA speeds, immense amounts of protocol state may accumulate on the processes playing leader roles, introducing overheads associated with data buffering, protocol state management, thread scheduling, and potential retransmission of data (if two or more leaders contend for the same Paxos log slot). If resource exhaustion occurs on the leader, the entire update pipeline will become backlogged.

The question that arises is whether the same outcome can be achieved using a protocol that reaches agreement on the messages to deliver, their ordering, and that they have been durably logged, but is implemented in a new style that avoids a pileup of protocol state and data in the processes that initiate new multicasts. The Derecho design seeks to be as asynchronous as possible. As we will see below, the resulting “refactoring” of Paxos yielded a protocol that is fully asynchronous and heavily pipelined, with no two-phase interactions, up to the point when a crash occurs or some other membership reconfiguration is required. When reconfiguring, Derecho does block, but only briefly (long enough to finalize the current epoch, apply the full set of membership changes to the epoch and to start a new epoch).

By shifting the blocking steps of Derecho into the epoch termination logic, we implicitly assume that membership changes will be relatively rare events. Support for this assumption can be found in talks and essays published by Google’s Jeff Dean. For example, discussing the challenges of deploying one of Google’s largest services, which runs on pools of 1500 machines per data center, Dean comments that the system experiences no more than one to three unanticipated crashes per hour in each data center, with the peak rate seen immediately after a new version is rolled out [83]. For Google, these are the difficult case, because when a node suddenly crashes, the service must automatically discover the fault and repair itself. Dean views planned maintenance events as a different challenge: these involve taking racks of machines offline or bringing them back online, but are relatively rare and because they are management actions, the service can be manually instructed to reconfigure. Later, we will see that Derecho can use RDMA hardware to sense failures within a few hundred milliseconds, and then can terminate an epoch and switch to a new one with a disruption that lasts for about 150ms: clearly far below the threshold at which this would be viewed as a problematic delay.

Virtually Synchronous Paxos. To appreciate the implications of the points just made, we will need to dive deeper into the issue of consistent replication. Paxos [53] is the gold standard for protocols supporting state machine replication [52, 82], bringing clarity to protocol correctness goals and opening the door to rigorous proofs [53, 87]. However, there are actually many versions of Paxos, each optimized for different target settings. The version most relevant to our work is
virtually synchronous Paxos, originally proposed by Malkhi and Lamport and ultimately formalized in Chapter 22 of [15].

Malkhi and Lamport suggested this Paxos variant to address some issues with the classic Paxos protocol. The classic Paxos protocol performs reads and writes using a multi-stage protocol that requires a quorum within a set of $N$ acceptors; subsequent work extended the model to distinguish read ($Q_r$) and write ($Q_w$) quorum sizes [55, 56]. For example, to tolerate 1 failure in a group of 3 acceptors, Paxos could commit updates on any 2 acceptors, but a learner (the “client”) would then have to read and merge 2 logs: $Q_w = N - 1, Q_r = 2$. This was reasonable in 1990, but at modern data rates the extra reads and log-merge slash peak performance. In particular, the need to read two or more logs could halve the achievable throughput, which is a big concern at RDMA rates.

Similar concerns can be expressed about modern versions of Paxos that run on RDMA, such as DARE and APUS [73, 90], as these have leader-based structures with the potential for pauses on the critical path (an exception is AllConcur, which uses a leaderless approach [75]).

The key insight is that Paxos does not necessarily have to be implemented as a multi-phase quorum commit. The Paxos specification says nothing about quorums. Correct behavior of a Paxos solution requires non-triviality, total ordering and persistence for data logged by a set of acceptors (to this one adds further requirements covering log compaction and reconfiguration of the acceptor set). Classic Paxos is better viewed as a particular way of implementing the specification and of proving the correctness of the resulting logic.

In particular, one can modify Paxos to use the virtually synchronous group membership model first introduced in the Isis Toolkit [12, 16]. By doing so, we preserve the Paxos guarantees, yet every group member can fully replicate the group state ($Q_w = N$) and can safely read from any single replica ($Q_r = 1$). To make progress after a failure, we reconfigure the group. In the new model, Paxos runs in a dynamically defined group of processes. Each group evolves through a series of epochs. An epoch starts when a new membership for the group is reported (a new view event). The multicast protocol is available while the epoch is active, sending and delivering totally ordered, persistent multicasts that are also persisted into NVM logs. An epoch ends when a member joins, leaves, or fails. Because a failure may end an epoch while a multicast is underway, at the end of the epoch all active multicasts are either finalized and delivered or must be reissued (by the sender) in the next epoch. In all cases, sender ordering is preserved.

**No Split Brain Behavior.** Dynamic membership management creates the risk of logical partitioning (split-brain behaviors), hence the protocol must ensure that any group has a single sequence of membership views, which we refer to as the primary partition. Processes that have dropped from the primary partition cannot form new views or send messages (multicast or point-to-point) to processes that remain in the primary partition. Such an isolated process will quickly shut itself down. Thus, once a view reports that some process has failed, it will not be heard from again, and it is safe to take remedial actions on its behalf.

If a Derecho group has subgroups, the same guarantees extend to the subgroups themselves. Interestingly, because the top-level group cannot experience logical partitioning, no subgroup of the top-level group can become logically partitioned. This permits us to improve system-wide recoverability. As will be seen below, Derecho makes progress given a majority of the top-level group and representatives of all subgroups and shards that hold persistent data, even if the system does not have access to a majority of members of some of those subgroups or shards. Further, the protocol used to recover after a total failure inspects the state of the top-level group, and from this can determine which among the various logs that exist for subgroup members are complete. Thus, even if a subgroup or shard were to abruptly drop from having $n$ members to having $n/2$ or fewer members, no risk of logical partitioning arises.
Additionally, all members see the full membership of the top-level group and of all of its subgroups (including their shards) and can use membership information in interesting ways: the load-balancer in Figure 1 knows precisely which processes belong to each shard and therefore will not erroneously forward a request to a process not in the appropriate shard. Although a process that does not belong to a group cannot multicast to it, we do permit point-to-point messages. Suppose that P sends a message to Q because Q has some role that P deduces from the view. In Derecho, Q will be in the same\(^{11}\) membership epoch as P, and will have instantiated the role that P believes it to hold.

**Monotonic Protocols.** To create Derecho, we implemented virtually synchronous Paxos in a novel way: as a protocol that continuously and asynchronously learns about the evolving state of an underlying data stream that never stops unless the entire system needs to be reconfigured (a relatively rare event). Key to re-expressing Paxos in this manner was the idea that we can track the state of the system through all-to-all information exchanges that occur through a table composed of monotonic variables: typically single-writer, multiple-reader counters that advance in a single direction. Next, we express the Paxos goals as predicates over these monotonic variables: predicates that are stable in the sense that once they hold, they continue to hold (even as new data is exchanged). Additionally, these stable predicates are themselves monotonic, in that they often cover a batch of events rather than just a single event: if such a predicate holds for the \(k^{th}\) message, this implies that it also holds for all messages from \(0...(k-1)\). Monotonic predicates are particularly valuable because when a protocol discovers such properties as ordered deliverability or safety, the discovery event covers a set of messages which can then be delivered as a batch. In contrast, many traditional Paxos protocols form a batch at the leader, but then handle those batches one by one, repeatedly pausing the underlying data streams. The Derecho approach is opportunistically batched, on the receiver side, and with no pauses.

Although these monotonic protocols certainly do not look like the classic versions, they turn out to be exceptionally efficient, and surprisingly easy to reason about. With respect to efficiency, we will show that the Derecho control plane delivers messages after the minimum number of rounds of distributed communication required by our fault model, which is the weakest fault model of interest in practical systems for the cloud. In effect, Derecho is a constructive lower bound for fault tolerant consensus and state machine replication.

### 8 BUILDING BLOCKS

Derecho is comprised of two subsystems: one that we refer to as RDMC \([10]\), which provides our reliable RDMA multicast, and a second that we call SST, which supports the monotonic logic framework within which our new protocols are coded. In this section, we provide overviews of each subsystem, then focus on how Derecho maps Paxos onto these simpler elements.

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

1. “Two-sided” RDMA. This offers a TCP-like behavior in which the RDMA endpoints are bound, after which point if P wishes to send to Q, it enqueues a send request with a pointer into a pinned memory region. The RDMA hardware will perform a zero-copy transfer into pinned memory designated by Q, along with an optional 32-bit immediate value which, for instance, can be used 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.

\(^{11}\)Of course, if further failures trigger additional epoch changes while P’s message is in flight, P might need to resend its request to Q’s successor in the role.
Fig. 10. Our RDMA multicast protocol, RDMC, breaks large messages into chunks, then forwards them chunk by chunk over a network overlay of reliable FIFO channels. Here the left diagram illustrates a binomial copying protocol, with processes represented by circles and data transfer steps represented by numbered arrows; the same number can appear on multiple arrows if transfers are concurrent. The middle and right diagrams illustrate the idea of running \(d\) binomial protocols simultaneously on a \(d\)-dimensional hypercube, created as a network overlay within our RDMA network (which allows all-to-all connectivity). For the binomial pipeline, transfers occur in sets of \(d\) chunks: 3 blocks colored red, green and blue in this example. When the pipeline is fully active, every node has one block to send and one block to receive at each logical timestep, somewhat like in BitTorrent, but with a fully deterministic pattern.

(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, or in the extremely unlikely event of data corruption, the hardware will sense and report this. Each request also has an associated retry limit, hence failures can also be triggered by fabric-level disruptions or switch failures; in such cases the connection will break, and a failure will be reported, despite the fact that both endpoints may still be healthy.

The RDMA standard is supported over both RoCE (Ethernet) and IB (InfiniBand), and there are also two software emulations over standard IP or IP tunneled over TCP (Soft-iWarp and SoftRoCE; both are quite robust, but the latter is slow and interesting mostly for portability). Our small testbed includes both hardware options and in our experience, they behave similarly. There is extensive experience with IB at very large scale, but the industry is just starting to experiment with RoCE (Microsoft reports very positive findings [40]). We do not yet have access to a genuinely large RoCE deployment, hence the experiments reported here all were done on IB.

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 initiate of the RDMA transfer sees that the target is sleeping, and uses an RDMA feature that triggers an interrupt to wake up the thread.

8.1 RDMA Multicast: RDMC

RDMC implements a zero-copy reliable multicast abstraction which guarantees that messages will be delivered in sender order without corruption, gaps or duplication. A single RDMC session allows
one designated sender to transmit messages, so each Derecho group actually corresponds to a
collection of RDMC sessions, one for each potential sender in the current epoch (thus an N member
group could have no RDMC sessions, or as many as N).

**Small messages.** Small messages (the limit is under user control, but by default, 16KB or less) are
sent using a specialized protocol that employs one-sided writes. For clarity we refer to it as SSTMC.
When using SSTMC, each member allocates a ring buffer for each sender: an approach that dates
(at least) to Unix pipes, but first applied to RDMA by systems such as U-Net [89], BarrelFish [8] and
Arrakis [72]. A sender waits until there is a free slot, then writes the message and increments a
count. A receiver waits for an incoming message, consumes it, then increments a free-slots counter.

**Large messages.** RDMC supports arbitrarily large messages which it breaks into 1MB chunks,
then routes on an overlay network (the actual network is fully connected, so this overlay is purely an
abstraction). A number of 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 fast
totally ordered (atomic multicast) mode) or directly to the application (for a group in raw mode).

**Binomial pipeline.** For most cases evaluated here, RDMC uses a binomial pipeline, which we
adapted from a method originally proposed for synchronous settings [35] and illustrate in Figure 10.
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 optimal latency. If the number of processes is a power of 2, all receivers deliver simultaneously;
if not, they deliver in adjacent protocol steps.

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

**Failures.** When RDMC senses a failure, it informs the higher level using upcalls and then wedges,
accepting no further multicasts and ceasing to deliver any still in the pipeline. This can leave some
protocols incomplete (multicasts may have been delivered to some destinations but not to others).
RDMC does not do any retransmissions, but Derecho includes a mechanism (described later) that
will detect such a problem, cleaning up by discarding the incomplete multicasts. Notice also that
because a process could have many active RDMC sessions at the same time; one could fail, while
others are unaffected. Derecho also handles this case.

8.2 Shared State Table

RDMC is a powerful data movement tool, but is not useful without a substantial surrounding
infrastructure. One issue is that RDMC does not manage group membership: it assumes that the
application using it will track membership and arrange for the RDMC endpoints to simultaneously
set up each needed RDMC session, select a single sender, and coordinate to send and receive data
on it. With multiple senders to the same group of receivers, one can set up multiple overlapping
RDMC groups, but nothing will be done to order concurrent messages from distinct failures. A
second issue is that when a failure occurs, RDMC simply wedges and reports the problem; it does
no cleanup or retransmissions. A third is that if there are several sessions with the same members,
one might wedge while the others remain active. 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 use of RDMA introduces some technical complications. RDMA has a minimum wire-level write size determined by hardware factors such as the number of data lanes in the RDMA optical links. Each transfer also has a non-trivial startup delay (on our hardware, 1.73 $\mu$s). This is large compared to the transfer speed, hence a one-byte transfer and a 4KB transfer have very similar total transfer times. Transfer size only emerges as a consideration above 8K bytes (a 4 $\mu$s transfer time). This limits bandwidth for small writes, which peak at a throughput comparable to that of the bus connecting a processor to its memory unit. For larger transfers, RDMA bandwidth can saturate the memory unit, outperforming a single-core version of memset on our test cluster. In contrast to the bandwidth ratio, RDMA latencies are high when compared to processor-to-memory latencies (about 4 $\mu$s, compared to about 80ns for random DRAM reads).

**Derecho’s SST implementation.** As outlined earlier, the SST offers a form of replicated table, with one replica hosted at each member of the top-level group. Within this table, there is one identically-formatted row per member. Each member has full read/write access to its own row, but is limited to read-only copies of the rows associated with other members. To share data using the SST, a process updates its local copy of its own row, then pushes the row to other group members by enqueuing a set of asynchronous one-sided RDMA write requests. We also support pushing just a portion of the row, or pushing to just a subset of other processes.

Notice that a sequence of updates to a single SST cell will overwrite one another. Even if a reader were continuously polling its read-only copy of that cell, it might see the values jump forward, skipping some intermediary values. In our algorithms that run on the SST, this behavior is treated as another opportunity for asynchronous pipelining. Indeed, we perceive this aspect of the SST as a new opportunity: our goal is to design protocols that are opportunistically batched and that have a batch size determined by the speed of the receiver.

To appreciate this point, think about batched multicast protocols: a sender decides on a batch size, then constructs a single message that includes a set of smaller messages. Doing so avoids the inefficiency of a high rate of I/O operations that each handle a tiny number of bytes. With the SST, the batching is a side-effect of reading less quickly than updates are occurring. For example, if a sender is incrementing a counter in an SST row, a receiver might see the counter jump from 0 to 15, then to 17, then to 35. Here we see that form of batching occurs because the receiver misses an unpredictable number of intervening values, as a function of how often updates occur and how often it polls. The receiver infers that the value passed through every value from 17 to 35 because it knows that the sender updates the field monotonically, by 1’s. We generalize this insight below.

The SST is lock-free, but guarantees that writes are atomic at the granularity of cache lines, typically 64 bytes in size. SST writes also respect the sender’s FIFO ordering, and occur from lower to higher memory addresses (useful for guarding a set of columns with a counter or a ready bit residing above the guarded data).

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, the SST would impose an $N^2$ load on the 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. 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. For example, in an application like the one shown in Figure 1, a typical SST update would write 12 bytes from one process to two or three peers. As a result, we have never seen a situation in which the SST was a bottleneck.
8.3 A simple SST example

One example of a protocol implemented over the SST is the SSTMC small messages protocol mentioned in Section 8.1. Each process maintains a set of ring buffers for each RDMC session and, for each, a count of how many messages it contains (Figure 11 shows the SST fields, but not the ring buffers or counters). As a sender, P first uses RDMA to write the next small message to the next free buffer slot (with N receivers, this involves $N - 1$ writes, one per receiver: we cannot use hardware RDMA multicast because it is unreliable). Then, P increments a sent-messages counter in its SST row and copies the resulting value to Q. Q eventually learns that additional messages are available from the RDMC session, consumes them from the ring buffer, then increments a counter of how many multicasts have been received from P, pushing this value back to P. With a second SSTMC protocol running from Q to P, we obtain the SST table shown. Notice that the SSTMC protocol will run in batches: if P happens to send several messages before Q rechecks its copy of the SST, Q discovers and delivers the whole group as a single batch.

8.4 Programming with the SST

The SST is a flexible abstraction. Even our simple example passes blocks of data and counters, and illustrates how these can be combined to obtain a simple sender-ordered multicast. The confirmation of receipt illustrates a way of using the SST for many-to-one incast: if P in a group with membership {P,Q,R} computes the minimum over this column from the SST rows corresponding to P, Q and R, it can determine how many multicasts have been confirmed by all receivers, hence learns which buffer slots are free for sending new messages. The SST can even implement barrier synchronization with the Filter version of Peterson’s Algorithm, or with Lamport’s Bakery Algorithm.

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

The SST framework layers high-level tools for logic programming over the basic functionality. The simplest of these is the RowFunction, a wrapper type for associating functions of type $\text{Row} \rightarrow T$ with a definition-independent name. A RowFunction is performed by some single process and

![Fig. 11. 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.](image-url)
typically just retrieves some field within some row, although doing so can involve more complex
reasoning (for example, a RowFunction could index into a vector).

On first impression, it may seem as though RowFunctions will do little to ease the programming
burden of reasoning about consistency, beyond offering a convenient place to support variable-
length fields and to implement any needed memory barriers. The true power of RowFunctions
becomes clear when combined with reducer functions, SST’s primary mechanism for resolving
shared state. A reducer function’s purpose is to produce a summary of a certain RowFunction’s
view of the entire SST, not just a single Row. 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 RowFunctions, our Derecho protocols can employ com-
plex predicates over the state of the entire SST without reasoning directly about the underlying
consistency. The functionality of a reducer function is simple; it takes a RowFunction \( f : \text{Row} \to T \),
allocates a new cell in the SST’s row to store a \( T \), produces a summary of \( f \) over the SST, and
stores the result of that summary in the cell. In this way, the reducer function actually has type
RowFunction \( \to \) RowFunction, allowing reducer functions to be arbitrarily combined, somewhat
like formulas over a spreadsheet.

Let’s look at an example.

```cpp
struct SimpleRow { int i; }

int iget(const volatile SimpleRow& s){
    return s.i;
}

bool rp(){
    return (Min(as_rf(iget)) > 7 ) ||
           (Max(as_rf(iget)) < 2);
}
```

Here, function \( rp \) converts the function \( \text{iget} \) into a RowFunction, calls the reducers \( \text{Min} \) and
\( \text{Max} \) on this RowFunction, then uses the boolean operator reducers to further refine the result.

We can do far more with our new RowFunction, \( rp \). The first step is to associate \( rp \) with a constant
designating the SST cell to which the output of \( rp \) will be written by a process that has evaluated it.
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 RowFunction, will use
the value. Additionally we can also register a trigger to fire whenever \( rp \) becomes true. Extending our example:

```cpp
eenum class Names { Simple};

SST<T> build_sst(){
    auto predicate = associate_name(Names::Simple, rp());
    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 \( rp \) with the name \( \text{Simple} \) chosen from an \texttt{enum class Names}, allowing
us to register the trigger \( act \) to fire whenever \( rp \) becomes true. As we see here, a trigger is simply a
function of type \texttt{volatile SST<T>&} \rightarrow \texttt{void} which can make arbitrary modifications to the SST or carry out effectful computation. In practice, triggers will often share one important restriction: they must ensure monotonicity of registered predicates. If the result of a trigger can never cause a previously-true predicate to turn false, reasoning about the correctness of one’s SST program becomes easy. Using this combination of RowFunctions, 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 [41], 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 \( K_P(\text{pred}) \). Now suppose that all members publish the truth value 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 \( K_1(\text{pred}) \): every group member knows that every other member knows pred. 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, \( \text{pred}' \). 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, then 27, 31. If the values are used in monotonic predicates and some deduction was possible when the value reached 30, both will make that deduction even though they saw distinct values and neither was actually looking at the counter precisely when 30 was reached. If events might occur millions of times per second, this style of reasoning enables a highly pipelined protocol design.
**Fault tolerance.** Crash faults introduce a number of non-trivial issues specific to our use of the SST in Derecho. We start by adopting a very basic approach motivated by monotonicity. When a failure is sensed by any process, it will:

1. Freeze its copy of the SST row associated with the failed group member (this breaks its RDMA connection to the failed node);
2. Update its own row to report the new suspicion (via the "Suspected" boolean fields seen in Figure 12);
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\(^\text{12}\) as its source of failure suspicions, although we also support a user-callable interface for reporting failures discovered by the software. In many applications the SST itself can be used to share heartbeat information by simply having a field that reports the current clock time and pushing the row a few times per second; if such a value stops advancing, whichever process first notices the problem can treat it as a fault detection.

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

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

**Knowledge and failure.** 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 \( \text{pred} \) 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 \( \text{pred} \) held at P, leaving uncertainty about whether or not P might have acted on \( \text{pred} \) prior to crashing.

Fortunately, there is a way to eliminate this uncertainty: before acting on \( \text{pred} \), P can share its discovery that \( \text{pred} \) holds. In particular, suppose that when P discovers \( \text{pred} \), it first reports this via its SST row, pushing its row to all other members before acting on the information. With this approach, there are two ways of learning that \( \text{pred} \) holds: process Q can directly deduce that \( \text{pred} \) has been achieved, but it could also learn \( \text{pred} \) indirectly by noticing that P has done so. If P possessed knowledge no other process can deduce without information obtained from P, it thus becomes possible to learn that information either directly (as P itself did, via local deduction) or indirectly (by obtaining it from P, or from some process that obtained it from P). If we take

\(^{12}\)If RDMA is properly configured, these should be extremely accurate. The classic quorum-based Paxos protocol was designed for settings where failure sensing would be very inaccurate, and hence would not be an obvious choice for use with RDMA. Derecho uses a different state machine replication protocol, with equivalent guarantees to the classic Paxos, but optimized to match the RDMA model.
care to ensure that information reaches a quorum, we can be certain that it will survive even if, after a crash, the property itself is no longer directly discoverable! With stable predicates, indirect discovery that a predicate holds is as safe as direct evaluation. By combining this behavior with monotonic predicates, the power of this form of indirection is even greater.

Notice also that when Derecho’s majority rule is combined with this fault tolerant learning approach, P either pushes its row to a majority of processes in the epoch, then can act upon the knowledge it gained from \( \text{pred} \), 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 \( \text{pred} \) holds. This will turn out to be a powerful tool in what follows.

9 THE DERECHO PROTOCOLS

We now have the background to discuss the way that Derecho layers its virtual synchrony and Paxos protocols over the building blocks just described. In this section, we first review the overall approach, then provide details, and finally justify our claims of optimality.

9.1 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 [12], modified to use the SST for information passing. The key elements of the membership subsystem are:

- When a new process is launched and tries to join an application instance, if Derecho was already active, active group members use the SST associated with epoch \( k \) to agree on the membership of the top-level group during epoch \( k + 1 \).
- Conversely, if Derecho was not active, Derecho forms an initial membership view that contains just the set of recovering processes.
- When a process fails Derecho terminates the current epoch and runs a “new view” protocol to form a membership view for the next epoch. These steps are normally combined, but if Derecho lacks adequate resources to create the new view (Section 2.4), a notification to the application warns it that Derecho activity has been temporarily suspended.
- If multiple joins and failures occur, including cascades of events, these protocol steps can pipeline. Moreover, multiple joins and failures can be applied in batches. However, in no situation will the system transition from a view \( k \) to a view \( k + 1 \) unless the majority of members of \( k \) are still present in view \( k + 1 \), a constraint sufficient to prevent logical partitioning (“split-brain” behavior).

The corresponding protocols center on a pattern of two-phase commit exchanging information through the SST. We designate as the leader the lowest-ranked member of the SST that is not suspected of having failed. Notice that there can be brief periods with multiple leaders: if P is currently the lowest-ranked member, but Q suspects P of having failed, then R might still believe P to be the leader, while Q perhaps believes itself to be the leader. Such a situation would quickly converge, because Derecho aggressively propagates fault suspicions. In what follows protocol-state shared through the SST is always tagged with the rank of the current leader\(^{13}\). If a succession of leaders were to arise, each new leader can identify the most recently disseminated prior proposal by scanning the SST and selecting the membership proposal with the maximum rank.

\(^{13}\)Readers familiar with the Paxos Synod protocol [87] may find it helpful to think of this rank number as the equivalent of a Paxos ballot number. We revisit this analogy later in the discussion.
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 13). In general, the change proposal is a list of changes: the next view will be the current view, minus process Q, plus process S, and so forth. Non-leaders that see a proposal copy it into their own proposal columns, then acknowledge it through a special SST column set aside for that purpose. When all processes have acknowledged the proposal, it commits.

If a new failure is sensed while running the two-phase commit to agree upon a new view, the leader extends the proposal with additional requested changes and runs another round. Recall that if any process ever suspects a majority of members of the current view of having failed, it shuts itself down; this is also true for the leader. It follows that progress only occurs if at most a minority of members of the prior view have failed. Agreement on the next view is reached when (1) every process has acknowledged the proposal, or has been detected as faulty; (2) the set of responsive members, plus the leader, comprise a majority of the current membership; (3) no new failures were sensed during the most recent round. This is the well-known group membership protocol of the virtual synchrony model [14, 16, 17] and has been proven correct [12]. We will not repeat the proof; passing data through the SST rather than in messages does not change the fundamental behavior.

Next, Derecho uses the new top-level view to compute the membership of subgroups and shards. Specifically, as each new top-level view becomes defined, Derecho runs the mapping function described in Section 2.3 to generate a list of subviews for the subgroups and shards. The new membership can now trigger object instantiation and initialization, as detailed momentarily. Given a new top-level view, for each process P, Derecho examines the subgroup and shard views. If P will be a member of some subgroup or shard that it is not currently a member of, a new instance of the corresponding object type is created and initialized using constructor arguments that were supplied via the constructor call that created the top-level group.

Initialization of newly created objects is carried out as follows:

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

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

Fig. 13. Each Derecho group has one RDMC subgroup per sender (in this example, members P and Q) and an associated SST. In normal operation, the SST is used to detect multi-ordering, a property defined to also include data persistence. During membership changes, the SST is used to select a leader. It then uses the SST to decide which of the disrupted RDMC messages should be delivered and in what order; if the leader fails, the procedure repeats.

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

The epoch is now considered to be active.

9.2 Atomic Multicast and Paxos

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

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

Although Derecho has several modes of operation, raw mode simply passes RDMC or SSTMC messages through when received. Fast totally ordered (atomic multicast) mode and durable totally ordered mode employ the same protocol, except that message delivery occurs at different stages. For the fast totally ordered (atomic multicast) mode Derecho uses the SST fields shown as \( n\text{Received} \) in Figures 12 and 13. Upon receipt, a message is buffered and the receiver increments the \( n\text{Received} \) 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 \( n\text{Received} \), the SST push only needs to write the changed field, and only needs to push it to other members of the same subgroup or shard. This is potentially a very inexpensive operation since we might be writing as few as 8 bytes, and a typical subgroup or shard might have just 2 or 3 members.
For durable totally ordered mode, recall from Section 6 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. In the introduction, we argued that smart memories for IoT and similar scenarios will often require this sort of steady streaming behavior.

### 9.3 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 6. 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 mQ:4 and mP:5 in Figure 13. The core idea is as follows: we select a leader in rank order; if the leader fails, the process of next-highest rank takes over (the same leader as is used for membership changes, and as noted previously, a single pattern of information exchange will carry out both protocols if the next proposed view is already known at the time the current epoch is terminated). As leader, a process determines which multicasts should be delivered (or committed, in the two-stage Paxos case) and updates its nReceived fields to report this. Then it sets the “Final” column to true and replaces the bottom symbol shown in Figure 13 with its own leader rank. We refer to this as a ragged trim. Thus one might now see a ragged trim in the nReceived columns and “T|r” in the column titled “Final”, where r would be a rank value such as 0, 1, etc. Other processes echo the data: they copy the nReceived values and the Final column, including the leader’s rank.

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

For example, in Figure 6 the round-robin delivery order is mP:1, mQ:1, mP:2, mQ:2, mP:3, mQ:3, mP:4, mQ:4, mP:5, mQ: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
Building Smart Memories with Derecho

The leader would thus change its own nReceived columns to show P:4, Q:3. Then it would set T\n0 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\n0 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 13 shows a case with just a single subgroup or shard, the general case will be an
SST with a block structure: for each subgroup or shard, there will be a set of corresponding columns,
one per sender. During normal execution with no failures, only the members of the subgroup or
shard will actively use these columns and they push data only one one-another; other top-level
members will have zeros in them. When the epoch ends and the row wedges, however, every
process pushes its row to every other process. Thus, the leader is certain to see the values for every
subgroup and shard. At this point, when it computes the ragged trim, it will include termination
data for all subgroups and shards. Every top-level member will learn the entire ragged trim. Thus,
a single top-level termination protocol will constitute a ragged trim for active multicasts within
every subgroup and shard, and the single agreement action covers the full set.

9.4 Recovery From Full Failure

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

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

9.5 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,

---

14Reuse 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.
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, it takes over as leader and scans its SST instance to see if any ragged trim was proposed by a prior leader. Among these, it selects and reused the ragged trim proposed by the leader with the highest rank. If it finds no prior proposal, it computes the ragged trim on its own. Then it publishes the ragged trim, tagged with its own rank.

9.6 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 fast 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 (Section 2.4) if the persisted data for a shard is not accessible. Suppose that the state is adequate, the leader is able to inspect the state of subgroup member Q, and that it discovers a persisted vector containing versions 0..k. Clearly, this vector must include any committed data for the subgroup because (prior to the failure) any commit is not accessible. Suppose that the state is adequate, the leader is able to inspect the state of subgroup member Q, and that it discovers a persisted vector containing versions 0..k. Clearly, this vector must include any committed data for the subgroup because (prior to the failure) any commit would have had to first be persisted by every member, including Q. Furthermore, we know that subsequent to the crash, the leader did not compute a ragged trim for this subgroup.

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

9.7 Discussion

Our scheme can be understood as a new implementation of older protocols, modified in ways that preserve their information structure but leverage the SST by adopting an asynchronous, pipelined (receiver-batched) style of execution. Where consensus is required, 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 8.4. 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 6 as a column "Final" where values are either F, with no leader (bottom), or T, in which case the nReceived values report the ragged trim, and the rank of the leader that computed the ragged trim would replace the bottom symbol: 0, 1, etc. The key insight is that if any ragged trim actually reached a majority, it will be learned by any leader taking over because the new leader has access to a majority of SST rows, and at least one of those rows would include the ragged trim in question. Thus, in the general case, the ragged trim used to clean up for the next epoch will either be learned from the prior leader, if that trim could have reached a majority, or it will be computed afresh, if and only if no prior value reached a majority.

Readers familiar with Paxos will realize that this is a familiar pattern. Derecho’s protocol for terminating an epoch is, in fact, isomorphic to the famous Paxos Synod protocol [87] (the slot-and-ballot protocol at the core of Paxos). The main difference is that classic Paxos uses this protocol for every message, whereas Derecho uses the Synod protocol only when a membership change has occurred. Nonetheless, this pattern is really the standard one, with the leader rank playing the role of ballot number.

Although the Derecho protocol is expressed very differently in order to take full advantage of the SST, it can be recognized as a variant of a protocol introduced in the early Isis Toolkit, where it was called Gbcast (the group multicast) [14]. The protocol is correct because:

- If any message was delivered (in fast totally ordered (atomic multicast) mode) or committed (in durable totally ordered mode), the message was next in round-robin order and was also received (persisted) by all group members, and every prior message must also have been received (delivered) by all group members. Thus, the message is included into the ragged trim.
- If the ragged trim includes a message (or version), then before the ragged trim is applied, it is echoed to at least a majority of other members of the top-level group. Therefore, the ragged trim will be discovered by any leader taking over from a failed leader, and that new leader will employ the same ragged trim, too.
- Upon recovery from a total failure, the inspector runs as a single non-concurrent task and is able to mimic the cleanup that the leader would have used. Moreover, the inspector can be rerun as many times as needed (if a further failure disrupts recovery) until the cleanup is accomplished.

9.8 Control/Data Separation

The authors of the Arrakis library suggested that RDMA protocols should employ a control plane/data plane separation [72]. Derecho can be seen as an application of this idea to data replication with strong consistency. First, data is transported via RDMC while control occurs in the Derecho
code running on the SST. But beyond this superficial observation, notice the substantial benefits that derive from receiver-side event batching.

Classic Paxos protocols generally assign a leader to run each Paxos ballot, which involves state management on the node where the leader runs, and a loss of time while waiting for exchanges of messages, rescheduling the paused thread, and so forth. Derecho uses a static ordering and reports receipt and persistence via the SST, through counters that can increment rapidly, and avoids any need to assign a leader or maintain that form of state. Receiver-side batching occurs because the SST rules that allow Derecho to deliver atomic multicasts (or the ones that report commit events) operate through aggregation: they are monotonic predicates. As such, each time a delivery predicate triggers, it covers a batch of events (all messages from 0...k). A receiver that slightly lags the incoming message stream might sometimes deliver one message, sometimes a batch of twenty, but is able to keep up because the delivery batch size arises directly from a safe, fault-tolerant, deductive inference.

During the epoch termination protocol, Derecho necessarily pauses the delivery of new messages until the current epoch is terminated, but then terminates all pending multicasts simultaneously. This occurs through the execution of a single protocol that generates a single ragged trim and a single completion event, but covers the entire history of messages across all subgroups and shards in the entire epoch. Joins and leaves are similarly processed in batches rather than one-by-one.

We find it interesting that this heavily asynchronous style of programming does not complicate the protocols. Early leader-based protocols such as the Isis Toolkit implementation of virtual synchrony, or classic implementations of Paxos, often required multiple rounds of 2-phase or 3-phase commit-style interactions between a leader and a set of participants. Further complexity was introduced by the logic to cover leader and participant crashes. The Derecho protocols are certainly not more complex than these prior ones, and one could argue that in some ways, they are much simpler. Further, prior protocols placed control logic on the critical path, disrupting the data plane while reaching decisions, and by doing so were prone to pausing while waiting for acknowledgements or votes. For example, Jallali has shown that classic quorum-based Paxos protocols can be highly bursty [63]. The issue is that with a quorum protocol, responsive acceptors will jump ahead of unresponsive or overloaded ones. Eventually the fast nodes become loaded, but then the whole service must pause until the laggards catch up. Indeed, Jallali sees such issues even in lightly loaded systems with no congestion. Derecho’s highly asynchronous protocols avoid this phenomenon, as we will see in the experimental section.

9.9 Feasibility of Model-Checking

Although the Derecho protocols are based on ones proved correct some time ago, by re-expressing them in terms of the SST we’ve arrived at versions that have non-standard descriptions, and also behave differently from the original versions. For example, whereas the original protocols involved steps at which a leader would wait for acknowledgements, in Derecho we sought a maximally asynchronous behavior. In the approach we adopted, during failure-free runs the only delays and acknowledgements arise at the level of the RDMA hardware. On the other hand, higher-level wait states do arise during the ragged trim used to terminate an epoch, and in the protocol for agreeing on the next view.

In our view, these differences argue not just for paper-and-pencil proofs of the kinds sketched above, but for fully formal proofs checked using a mechanical prover. In past work, we undertook a machine-checked proof of Paxos as part of work on a recovery subsystem for a database [78] and used formal methods to create a suite of versions of the Ensemble system that were optimized under various runtime conditions [61]. Other work that resulted in formal proofs of Paxos include Lamport’s TLA+ proofs [54] and DePrisco’s use of Larch to prove an IOA version of the Paxos
protocol [76]. A recent example is the IronFleet protocol and system [42], which were proved correct using a Java-based language called Dafny [57, 58].

Accordingly, we are collaborating with a team that has considerable experience using the Ivy theorem prover (a proof-checking tool that has itself been verified using ACL2), with the goal of formally specifying and verifying Derecho’s Paxos protocols. Because the SST is expressed in C++, Ivy is not (yet) able to handle the full syntax used in the system. With the help of this group, we are working to produce a fully formal specification of the Derecho protocols, expressed in a formalism based on Lynch’s I/O Automata approach. Work to formalize our hand-written proofs and then to verify them will be underway shortly. We made a decision to omit the pseudo-code from this paper because (1) the IOA formalism would require an extensive discussion, (2) the work is not yet finished, and (3) the topic itself digresses into issues outside of the engineering focus of the present paper.

9.10 Efficiency

A further remark relates to the efficiency of Derecho. In Section 10 we consider this question from an experimental perspective, showing that Derecho is achieving the highest performance of any known protocol suite with equivalent properties. However, one can also pose theoretical questions about efficiency. Keidar has developed lower bounds on the information exchanges required to achieve Uniform Agreement [46]; by counting exchanges of data as information flows through the SST, one can show that Derecho achieves those bounds. Finally, because RDMC has logarithmic fan-out, once the binomial pipeline is “primed” with data, all processes make full use of their incoming and outgoing NIC bandwidth. Thus RDMC has an optimal data dissemination pattern among protocols that use block-by-block unicast data movement. It follows that Derecho achieves a constructive lower bound for Paxos.

9.11 Prior work

Popular data warehouse and temporal data query solutions include Time Scale DB [49], Open TSDB [68], RealTime FB [24], Influx DB [30] and Beringei [71]. As noted, these are complete solutions, and as such are obvious first choices for applications that fit their models. They typically operate as black boxes, programmed using versions of SQL that are augmented to support a temporal query notation. Similar remarks apply to the Spark/Databricks framework for deep learning and data mining [25, 94]. Our premise is that IoT applications will give rise to a new developer need: IoT systems that run on the cloud at scale will have a mix of application structure, data replication, and persistence requirements. Data warehouse solutions would not always leave the developer with adequate control over application structure, or adequate freedom to decide on a case by case basis which data will be stored, how it will be queried, or what form of consistency should be available. Derecho aims at the class of developers requiring control from the lowest layers up.

Paxos. While we do not perceive Derecho as a direct competitor with existing Paxos libraries, 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 invariably 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 [63].

With respect to guarantees offered, the prior work on Paxos is obviously relevant to our paper. The most widely cited Paxos paper is the classic Synod protocol [53], but the version closest to ours is the virtually synchronous Paxos described by Birman, Malkhi, and van Renesse [15]. A number of papers how suggested ways to derive the classic Paxos protocol with the goal of simplifying understanding of its structure [21, 28, 56, 64, 76, 87]
Among software libraries that offer high speed Paxos, APUS [90] has the best performance. APUS implements a state-machine replication protocol proposed by Mazieres [64]. 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 [6] offers a persistent log, using Paxos to manage the end-of-log pointer, and chain-replication as a data-replication protocol [88]. In the Corfu model, a single log is shared by many applications. An application-layer library interacts with the Corfu service to obtain a slot in the log, then replicates data into that slot. Corfu layers a variety of higher level functionalities over the resulting abstraction. Our version-vectors are like Corfu logs in some ways, but a single application might have many version vectors, and each vector holds a single kind of replicated data. Our RDMC protocol is more efficient than the Corfu replication protocol at large scale, but Corfu is not normally used to make large numbers of replicas. More interesting is the observation that with one original copy and two replicas, we outperform the chain replication scheme used in Corfu: RDMC never sends the same data twice over any RDMA link. With two replicas, the leader (P) sends half the blocks to replica Q, which then forwards them to R, and the other half to S, which forwards them to R, and the resulting transfer completes simultaneously for Q and R almost immediately (one block-transfer) after the leader has finished sending; potentially, twice as fast as any protocol that either uses a chain of participants, or where P sends separately to both Q and to R.

Round-robin delivery ordering for atomic multicast or Paxos dates to early replication protocols [23]. 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 [39], 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 11 compares performance of libPaxos with Derecho.

RAFT [67] is a popular modern Paxos-like protocol; it was created as a replication solution for RamCloud [69]. Microsoft’s Azure storage fabric uses a version of Paxos [20], 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 [29]. DARE [74] looks at state machine replication on an RDMA network. RDMA-Paxos is an open-source Paxos implementation running on RDMA [2]. NOPaxos [59] 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 [14], and its gcast protocol is similar to Paxos [4]. Modern virtual synchrony multicast systems include JGroups [7] 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 [34]. 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 [26]. The core result is the CALM theorem, which establishes

---

15Corfu has evolved over time. Early versions offered a single log and leveraged RDMA [6], but the current open-source platform, vCorfu [91] virtualizes the log and focuses on support for complex data structures.
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 [31], HERD [45] and Pilaf [65]. Only some of these DHTs support transactions. FaRM offers a key-value API, but one in which multiple fields can be updated atomically; if a reader glimpses data while an update is underway, it reissues the request. DrTM [92] 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.

**PART III. DERECHO PERFORMANCE**

**10 EVALUATION**

We evaluated Derecho on our local cluster, Fractus, for small scale experiments and on Stampede 1, a supercomputing cluster in Texas, for large scale experiments. Fractus consists of 16 machines running Ubuntu 16.04 with a 100Gbps (12.5 GB/s) RDMA InfiniBand switch (Mellanox SB7700). The machines are equipped with Mellanox MCX456AECAT Connect X-4 VPI dual port NICs. Fractus is also equipped with a 100Gb 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 56 Gb/s (8 GB/s) FDR Mellanox NIC, housed in 160 racks (40 nodes/rack). The interconnect is an FDR InfiniBand network of Mellanox switches, with a fat tree topology of eight core-switches and over 320 leaf switches (2 per rack) with a 5/4 bandwidth oversubscription. Nodes on Stampede are batch scheduled with no control over node placement. Node setup for our experiments consists of about 4 nodes per rack. Although network speeds are typically measured in bits per second, our bandwidth graphs use units of GB/s simply because one typically thinks of objects such as web pages, photos and video streams in terms of bytes. Figure 14 shows how performance of our RDMA hardware and SSD hardware varies for such objects, showing RDMA up to 1MB (the RDMC chunk size), and SSD for arbitrary sizes.

Figure 15 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 performance for both raw mode and fast totally ordered (atomic multicast) mode.

We see that Derecho performs close to network speeds for large message sizes of 1 and 100 MB, with a peak rate of 16 GB/s. In raw mode, Derecho’s protocol for sending small messages, SSTMC, ensures that we get high performance (close to 8 GB/s) for the 10 KB message size; we lose about half the peak rate in our totally-ordered atomic multicast. As expected, increasing the number of senders leads to a better utilization of the network, resulting in better bandwidth. For the large message sizes, the time to send the message dominates the time to coordinate between the nodes for delivery, and thus raw mode and fast totally ordered (atomic multicast) mode achieve
similar performance. For small message sizes (10 KB), those two times are comparable. Here, raw 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 22b.

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

Earlier, we noted that while most communication is expected to occur in small groups, there will surely also be some communication in larger ones. To explore this case, we ran the same experiment on up to 128 nodes on Stampede. The resulting graph, shown in Figure 16, shows that Derecho scales well. For example we obtain performance of about 5GB/s for 1MB-all-senders on 2 nodes, 2.5 GB/s on 32 nodes, and 2 GB/s on 128 nodes: a slowdown of less than 3x. Limitations on experiment

16The drop in performance for the 100 MB all senders case (as group size grows from 8 to 9 nodes) turns out to be caused by a firmware bug in the Mellanox NICs, which causes a form of resource exhaustion if a high data rate is maintained over a long period of time. Mellanox is currently testing a fix for this problem.
duration and memory prevented us from carrying out the same experiment for the 100 MB case on 64 and 128 nodes. This also explains the absence of error bars: each data point shown corresponds to a single run of the experiment. Note, however, that the performance obtained is similar to that seen on Fractus for small groups.

Next, we evaluate performance in an experiment with a large sharded group. Here, the interesting case involves multiple (typically small) subgroups sending messages simultaneously, as might arise in a sharded application or a staged computational pipeline. We formed two patterns of subgroups of fixed size: disjoint and overlapping. For a given set of $n$ nodes, assume unique node ids from 0 to $n - 1$. Disjoint subgroups partition the nodes into subgroups of the given size. Thus, disjoint subgroups of size $s$ consist of $n/s$ subgroups where the $i^{th}$ subgroup is composed of nodes with ids $s \ast i, s \ast i + 1, \ldots, s \ast (i + 1) - 1$. Overlapping subgroups of size $s$, on the other hand, place every node in multiple ($s$) subgroups. They consist of $n$ subgroups where the $i^{th}$ 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 17 shows that for large messages (100 MB), the aggregated performance increases linearly with the number of nodes for all subgroup types and sizes. This is as expected; the subgroup size is constant, and each node has a constant rate of sending, leading to a linear increase in aggregated performance with the number of nodes.

We do not include data for small messages (10 KB) because this particular case triggered a hardware problem: the “slow receiver” issue that others have noted in the most current Mellanox hardware [40]. 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 SSTMC, creating one SSTMC 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 SSTMC subgroups will share SST columns. This should significantly reduce the size of our SST rows.

Our next experiments focus on the costs of versioned storage. Whereas the data shown above was for applications that invoke a null message handler, figures 18a and 18b look at a full code path that includes marshalling on the sender side (required only if the data being sent is not suitable for direct RDMA transmission), demarshalling for delivery, and then storage into volatile or persistent version-vectors, respectively. The latter incur a further marshalling cost, followed by a DMA write, while the former entails allocating memory and copying (this occurs within the Linux “ramdisk” subsystem). Not surprisingly, Derecho’s multicast is far faster than the copying and DMA write costs, hence we see flat scaling, limited by the bandwidth of the relevant data path. For the SSD case, our experiment pegs our SSD storage devices at 100% of their rated performance.

We were initially surprised by the relatively low bandwidth for volatile storage of large objects: we had expected the 100MB speed to be higher than the 1MB numbers, yet as seen in Figure 18a, 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 8GBps on our machines if the source object is in the L2 cache, bandwidth drops to just 3.75GBps for objects too large to cache. In our experiment, this slowdown effect is amplified because several copying operations occur.

In Figure 19 we see Derecho’s end-to-end performance, measured in two ways. The graphs on the left are for an experiment that initiates all-to-all multicasts in a subgroup of 2 nodes (blue) or 3 nodes (orange), which store the received objects into version vectors, showing 10KB objects (top two graphs) and 1MB objects (bottom two). We measured time from when the update was initiated until the new version is temporally stable and the ordered_send has completed.

To obtain the histograms on the right, we launched an additional thread within the same processes; it loops. During each iteration, our test samples the local clock, then indexes into the local instance of the versioned object at the corresponding time, measuring the delay until the data is available. This yields a pure measurement of delay on the query path used to retrieve information from a
Building Smart Memories with Derecho

Fig. 19. 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".

Fig. 20. 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.

temporal snapshot (recall that in Derecho, updates and temporal queries use different code paths, as illustrated in Figures 2 and 8 and discussed in Section 5).

ACM Transactions on Computer Systems, Vol. 0, No. 0, Article 0. Publication date: 0.
In Figure 20, 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, then collects replies in parallel. The temporal indices for these requests were picked to be far enough in the past so that the data would be temporally stable, yet not so old that data would no longer be cached in memory. We graph the time from when the request was made until the full set of replies is available.

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

The experiments shown above were all performed on lightly loaded machines. Our next investigation explores the robustness of control plane/data plane separation and batching techniques for Derecho in a situation where there might be other activity on the same nodes. Recall that traditional Paxos protocols degrade significantly if some nodes run slowly [63]. 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

![Number of Nodes Slowed](a) 10 KB  ![Number of Nodes Slowed](b) 100 MB

Fig. 21. Derecho performance for various values for efficiency and number of slow nodes, as a fraction of the no-slowdown case.
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 21.

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

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

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

Next, we considered the costs associated with membership changes. In Figure 22a 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 22b 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 23a we see a process joining: 1 it requests to join, 2–6 are the steps whereby the leader proposes the join, members complete pending multicasts, and finally wedge. In steps 7–9 the leader computes and shares the trim; all processes trim the ragged edge at 9 and the leader sends the client the initial view (10). At 11 we can create the new RDMC and SST sessions, and
Derecho multicast bandwidth with 200MB messages. A new member joins at 10s, then leaves at 20s.

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

Timeline for joining an active group.

Crash triggers exclusion from an active group.

the new view becomes active at 12. Figure 23b 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 22a the disruptive impact of a join or leave grows as a function of the number of active senders: the number of active sends and hence the number of bytes of data in flight depends on the number of active senders. Since we are running at the peak data rate RDMA can sustain, the time to terminate these sends is dominated by the amount of data in flight. In experiments with 20MB messages the join and leave disruptions are both much shorter.

**Optimizations**

The Derecho performance-critical path is sufficiently unusual to require some novel optimizations.

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

**SSTMC for small messages.** In our initial work on Derecho, RDMC was used to send all messages. However, RDMC was designed for large messages and large number of receivers, and its performance for small messages was quite poor. Referring back to Figure 14a, we see that for the smallest message sizes, RDMA has a minimal delay of 1.73\(\mu s\), reflecting a low-level handshake between the RDMA NIC and the target to ensure that the receiver has adequate space. Because RDMC always uses the same overlay in any single group, small messages are very likely to stall while waiting for these handshakes to occur. We didn’t notice the issue when using RDMC for large chunks of data: in such cases, the NIC is able to overlap data transfer for the current transfer with the the credit handshake for the next transfer. Thus it is only with small objects, where the transfer delay is short, that the handshake becomes a dominating factor.

Two further concerns arise with small messages: 1) RDMC’s binomial pipeline protocol relays chunks of data in multiple steps (logarithmic in the group size). With large transfers, this works well because after the first few chunks have been transmitted, the hypercube overlay becomes busy with a high rate of concurrent bidirectional data transfers. However, if an entire message fits into a single chunk, the pipeline never fills. 2) RDMC runs asynchronously, yet its pattern of data movement is really intended for a nearly-synchronous sequence of operations. With large messages, where data transfer times dominate, the near-synchrony assumption holds even if receivers delay: transfer times look long relative to scheduling variability. However, with small messages, data transfer delays are minimal and any receiver delays snowball.

These observations led to our decision to implement a separate small message multicast protocol: the SSTMC. In a shard or subgroup of size \(s\), SSTMC uses \(s - 1\) direct of unicast connections, and initiates \(s - 1\) one-sided RDMA writes concurrently. The handshakes all occur simultaneously at once transfers begin, the NIC is able to run steadily. In practice, all SSTMC uses in a single top-level group of size \(n\) share the same set of \(n \times (n - 1)\) connections. This will be well within the connection capacity of a single NIC, particularly because for any given sender, only a small subset are likely to be in use, and NICs are designed to cache connection state for active connections.

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

**Receiver-side 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.

### 11 Comparisons

Using the same cluster on which we evaluated Derecho, we instrumented libPaxos, Zookeeper, and APUS. Within this set, APUS should be compared with Derecho atomic multicast, while libPaxos and Zookeeper should be compared with Derecho’s persistent version-vector storage. In each case, we configured the protocol to tolerate 1 fault by creating groups of size \(N\) and then setting the
update quorum size to $N - 1$, then used a single sender to issue as high a rate of requests as possible (limited only by flow control). The patterns of communication match the scenarios evaluated for Derecho earlier, except that the largest Derecho object size is 100M, which exceeded the maximum size limit for Zookeeper and APUS. As a result, our Zookeeper data only goes to 1MB objects, and our APUS data, to 10MB objects. Zookeeper and libPaxos exhibited high variance in bandwidth for large objects; we did not investigate closely to understand the root causes.
The results are seen in Figures 24, 25, 26 and 27. For comparison, we show the analogous Derecho results in Figure 28. Notice that object sizes are not always identical. The Derecho data is actually just a reformatted version of the more complete evaluations seen earlier in Figures 18b and 15b.

Notice that in all of these graphs, we superimpose two forms of data: bandwidth, shown as heavy bars and using the left Y axis to indicate the corresponding data rates, and events per second, shown as lines with the right Y axis giving the corresponding event rates. Broadly, Derecho is faster (often by orders of magnitude) and scales far more efficiently, but two cases need discussion.

Comparisons with Zookeeper lead to a very interesting insight about the Paxos specification itself. Notice that Zookeeper has remarkably good performance for small updates (8B and 1KB). As it happens, these numbers would be "impossibly" good for our SSD using a true Paxos: the SSD we used requires 900us to finalize a new update, and hence is limited to about 1000 synchronized write operations per second. One could speculate that Zookeeper simply has a fantastic batching scheme, but in fact the explanation is simpler: the Zookeeper protocol (ZAB) is slightly different from Paxos, and in particular offers an asynchronous mode, in which requests can be sent without waiting for results and in which the underlying Zookeeper storage system does not use "forced write synchronization". We used this feature in our experiment because Zookeeper is nearly 50x faster in this mode than with its strongest level of synchronization.

Used asynchronously, Zookeeper has a behavior that would be familiar to any user of the Linux file system: after doing a series of writes, the application should call \texttt{fsync} (a so-called \texttt{write barrier}). Alternatively, the Zookeeper log can be stored on a RAID-style SSD unit with battery-backed DRAM cache. But it is unsafe to use Zookeeper asynchronously without compensating in one of these ways. The core risk is that without synchronization, Zookeeper could accept a request, yet a subsequent could then cause the associated update to be lost.

We measured Zookeeper in this configuration because otherwise our results would have seemed unfairly slow. Moreover, even with its synchronous configuration, ZAB is not a full Paxos implementation. Thus, a purely apples to apples comparison is simply not possible. In fact, although not shown, the Derecho durable totally ordered mode has similar performance to the synchronous configuration of Zookeeper for small writes. On the other hand, Zookeeper slows down as the size of the replication group increases, whereas Derecho maintains its performance.

We are considering adding an asynchronous mode to Derecho, designed to behave just like the Zookeeper option and with the same limitations. Individuals considering a switch from Zookeeper to Derecho might find this appealing. Arguing against such a step is the observation that the next generation of NVM hardware is likely to erase the performance gap.

APUS implements an RDMA version of atomic multicast, using a protocol similar to vertical Paxos developed by Mazieres [64]. This protocol is quite fast, yet still lags Derecho except at the largest object sizes. To better understand why this occurs, we contacted the developers. The APUS protocol has a multi-phase structure: for each action, a leader requests that acceptor processes vote on a proposal to place proposed data into a designated slot in the Paxos log; each request is numbered with a ballot number. Participants vote, and if the outcome lacks a quorum, the protocol either retries using a larger ballot number or switches to the next free slot. If a quorum accept the proposal, the protocol uses a 2-phase commit to finalize the data in the requested slot. This leads to a pattern similar to a 3-phase commit (4 or more phases may be needed if multiple leaders contend for a slot, or after a failure).

Relative to Derecho in its fast totally ordered (atomic multicast) mode, APUS can stall in two ways: (1) while waiting for votes on a current protocol instance, the leader will not initiate the next protocol instance, resulting in extended idle periods; (2) the leader must retain the current message in order to re-propose it if the vote fails, consuming space, which will eventually trigger congestion backlogs. We conjecture that the system is unable to keep the data plane busy. APUS comes close
to Derecho only if the objects being sent are large enough so large that the time spent running the
protocol is incidental compared to the time spent moving bytes.

12 CONCLUSIONS

Derecho is a new programming environment for creating Internet of Things applications that need
to apply machine intelligence directly on the data acquisition path, must move large amounts of
data at RDMA speeds, and yet need strong consistency when accessing that data soon after it is
captured. Examples include smart memory systems (data warehouses), high-speed image analysis
tools, and other forms of scalable smart infrastructures. Our focus is on developers who seek tight
control over application structure, data movement and overheads associated with consistency, and
hence prefer a C++ library programming tool rather than a SQL database or data warehouse.

Derecho solves these problems by introducing a new way of structuring applications into
subgroups and shards, which are managed automatically. The system offers a new way of hosting
potentially large amounts of data, which are stored in version vectors. These can reside in memory
or be persisted to NVM, with update semantics that can vary from a raw model with basic reliability
guarantees and minimal latency to atomic multicast, even with NVM durability. The option of
using a file API to access to the versioned data facilitates integration with existing data warehouse
and data mining tools.

To decouple reads from updates, Derecho is designed so that updates can stream into a system
asynchronously and be continuously appended to version logs. Applications then query the data
using a separate asynchronous path in which the query is sent to a set of nodes that hold data. The
data access scheme ensures that although executed concurrently and without locks, such a query
runs a temporally-indexed snapshot that offers a mix of temporal precision and causal consistency.
If the data rate does not trigger congestion, Derecho’s data acquisition path and query path can be
continuously active, without blocking.

Performance of Derecho sets records when compared with many prior systems, and there are
many uses for which Derecho is also faster than APUS, which is a contemporary RDMA Paxos
software library. The system also scales exceptionally well. The resulting data rates are in many
ways disruptive: Derecho can create large numbers of replicas of a large object faster than most
machines can do single-core memory-to-memory copying. Moreover, Derecho brings a powerful
new form of structuring to the cloud service developer. It enables developers to create a new
kind of smart communications and memory fabric, customized with machine-learning or other
application-specific logic wherever desired, and that should be able to support a completely new
kind of intelligent online behavior.

ACKNOWLEDGMENTS

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


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


Received Aug. 2017; revised --; accepted --

ACM Transactions on Computer Systems, Vol. 0, No. 0, Article 0. Publication date: 0.