Derecho: Group Communication at the Speed of Light

Jonathan Behrens[1], Ken Birman, Sagar Jha, Matthew Milano, Edward Tremel
Eugene Bagdasaryan, Theo Gkountouvas, Weijia Song, Robbert van Renesse
Cornell University; [1]*MIT*

Abstract

Today's platforms provide surprisingly little help to developers of high-performance scalable services. To address this, Derecho automates the creation of complex application structures in which each member plays a distinct role. Derecho's data plane runs a novel RDMA-based multicast protocol that scales exceptionally well. The Derecho control plane introduces an *asynchronous monotonic logic* programming model, which is used to implement strong guarantees while providing a substantial performance improvement over today's fastest multicast and Paxos platforms. A novel *subgrouping* feature allows any single group to be structured into a set of subgroups, or *sharded* in a regular pattern. These features yield a highly flexible and expressive new programming option for the cloud.

1 Introduction

We often conceive of the cloud primarily in terms of applications that involve enormous numbers of external endpoints that make requests, triggering individual responses. Typically, some event on a browser or mobile device causes a small stateless task to run on a cloud platform. The task responds, if possible using cached data. Where an update occurs, it will often be initiated asynchronously as well, with the response to the external client sent before the update to persistent state.

But this style of cloud computing runs into limitations for applications that need stronger consistency or high availability, and in any case is feasible only because the cloud incorporates prebuilt and often rather elaborate supporting services to manage caches and support back-end storage, and that automatically repair themselves in the event of failures whiling maintaining needed forms of distributed consistency.

The developer of a *new* infrastructure service, or of an application that needs stronger guarantees than is usual in the cloud, faces a hard problem: the tasks just summarized are poorly supported by development tools, and often must be solved ground-up. With the emergence of real-time applications that require high availability and strong consistency, more and more cloud developers are encountering this challenge. The style of service just described arises in systems that stream Internet videos, infrastructure services that update VMs and containers, IoT systems, and online analytics (to list just a few).

Our system, Derecho, is addressed to this need. We focus on use cases within a single data center equipped with RDMA unicast (future work will tackle georeplication). Derecho's programming model is focused on *process groups* with dynamic membership: members can join, leave, or be ejected because of failure. State transfer is used to initialize a joining process. Beyond these basics, Derecho innovates by automating the creation of subgroups of a group, or patterns of subgroups such as a sharding pattern. A subgroup or a shard is just another form of group, but membership consistency between a group and its subgroups is automatic. Derecho's execution model enables high-speed, scalable state-machine replication [1], with optional persistence, allowing the developer to make a per-group choice between atomic multicast, Paxos, or a raw data transport protocol.

Derecho would often be used side by side with (or integrated directly into) middleware layers such as such as transactional systems [2–4], file systems with strong semantics [5], or publish-subscribe [6, 7]. Even machine-learning systems could benefit: exchange of data often limits the shuffle-exchange step at the core of iterative solvers, and there is growing interest coupling machine learning systems to real-world inputs, where distributing incoming sensory data to the solvers and getting the results back to actuators is a demanding, time-critical need.

For raw data movement, Derecho implements a protocol we call RDMC; it provides a bare-bones reliable RDMA multicast. RDMC constructs a logical overlay out of unicast RDMA connections, sending data in a pat-

tern that can leverage the full bidirectional process-to-node bandwidth of the hardware, yielding an exceptionally fast protocol that also scales well. Derecho can make one copy at the speed of the RDMA hardware, but creating 64 replicas can be as little as 10% slower. Using its small-message protocol, Derecho can move 1 KB messages in an all-to-all pattern at rates exceeding 2M/second, with end-to-end latencies as low as 2µs.

Derecho is by far the fastest technology of this kind yet created. Configured as Paxos, Derecho is twice as fast as Corfu, the prior record holder [8], and ten times as fast as the Windows Azure replication framework [9]. Derecho can scale to huge deployments with no loss of speed, whereas Corfu and Azure are limited to small replication factors. Derecho is nearly 100 times faster than Zookeeper [5] and LibPaxos [10], all running on the identical hardware. Latency is just 20µs for Derecho atomic multicast (and as low as 2µs for small messages in our raw mode), and throughput is roughly 15,000 times that of the Vsync atomic multicast system [11].

At these data rates, the challenge is to to turn the raw multicast protocol, which provides ordering and integrity but lacks strong semantics, into a more sophisticated primitive that can retain the same performance. For this purpose, we designed a novel control framework based on *monotonic logic*. The key insight is this: if data is moving far too fast for the control plane to keep up with it on an event-by-event basis, protocol actions must run out of band. A monotonic protocol is one in which observations are never invalidated by subsequent events. For example, suppose that a protocol determines that every message up to k can safely be delivered in some agreed-upon order. If the protocol is monotonic, no matter what happens next, this delivery rule remains safe.

We discovered that a wide variety of distributed control protocols such as 2-phase commit, K1 knowledge and consensus can be reformulated monotonically. This enabled us to design a new atomic multicast protocol and a new implementation of Paxos. Not only are these protocols surprisingly elegant, they turn out to be easily extended: once we had our basic protocol working for top-level groups, only minor extensions were needed to support subgroups with strong consistency.

Derecho automates many of the hard tasks alluded to earlier: managing a group name space, bootstrapping when a system is first launched, creating group and subgroup structures, coordinating between components when reconfigurations occur, reporting failures and joins, state transfer, and reloading Paxos state from logs. The developer's task is hugely simplified. A companion paper [12] focuses on API design choices. Here, we drill down on the Derecho protocols.

2 Design Considerations

Control plane / Data Plane separation. The overarching trend that motivates our work involves a shifting balance that many recent researchers have highlighted: RDMA networking is so fast that to utilize its full potential developers must separate data from control, programming in a pipelined, asynchronous style [13, 14]. Traditional protocols for virtually synchronous group communication, atomic broadcast or Paxos have stood in the data plane: the logic that decides on ordering and safety also moves bytes. RDMA requires a restructuring in which the control logic runs side by side with data movement, so that if the application has data to send continuously, the data plane will only be delayed if the control plane is doing something unusually costly.

Strong programming models. Classic data replication protocols offer strong consistency guarantees [15, 16], but are structured in a way that makes it hard to leverage RDMA communication. Our insight is that we can adopt these models but implement the protocols in a new way, and in particular, that the way we *reason* about protocols can limit their performance.

To appreciate this point, we should give more context. Paxos [16] is the gold standard for protocols supporting state machine replication [17,18], bringing clarity to protocol correctness goals and opening the door to rigorous proofs [16, 19]. In fact, there are actually many versions of Paxos, each optimized for different target settings. The version most relevent to our is *virtually synchronous Paxos*, originally proposed by Malkhi and Lamport and ultimately formalized in Chapter 22 of [20].

Malkhi and Lamport suggested this Paxos variant to address some issues with the classic Paxos protocol. That 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 and write quorum sizes. 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: a big concern at RDMA rates.

The key insight is that Paxos doesn't necessarily have to be implemented as a multi-phase quorum commit. The Paxos *specification* 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). Any protocol that implements the specification is a Paxos protocol. In particular, one can modify Paxos to use the vir-

tually synchronous group membership model first introduced in the Isis Toolkit [11, 20]. 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 ($Qr = 1$). To make progress after a failure, we reconfigure the group.

Virtually Synchronous Paxos. 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 SSD 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 reissued in the next epoch, or aborted if the sender is not part of the new epoch. In all cases, sender ordering is preserved.

The model requires that any multicast be delivered to all members of the group in the same epoch. Thus, on receiving a multicast an application can make use of the membership (for example, the ranking). For example, the member ranked 0 in the view could take on a leader role. In a group with N members, a task could be divided into N subtasks, each handled by a distinct member.

Derecho's Implementation of Persisted State. In Derecho, we represent a Paxos log as a checkpoint that includes epoch membership information, followed by a series of updates in the proper order and a counter of how many updates have committed. For the top-level group, we also log the view of the top-level group itself and of its subgroups; this is useful when recovering from failure. To compact a log, a member just replaces its log with its current checkpoint. We require that the logic for updating the state associated with a group be deterministic, hence compaction can be done at any time. Notice that this does not preclude members from having different responsibilities: they merely need identical states.

No Split Brain Behavior. Dynamic membership management creates the risk of logical partitioning (split-brain behaviors), hence a 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, and are quickly forced to shut 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. Importantly, because the top-level group cannot experience logical partitioning, no subgroup can become logically partitioned. Additionally, all members see the full membership of the top-level group and of all of its subgroups (including their shards or subgroups). Although a process that does not belong to a group cannot multicast to it, we do permit point-to-point messages; if such a message is sent in view k, will be delivered in view $k' \geq k$.

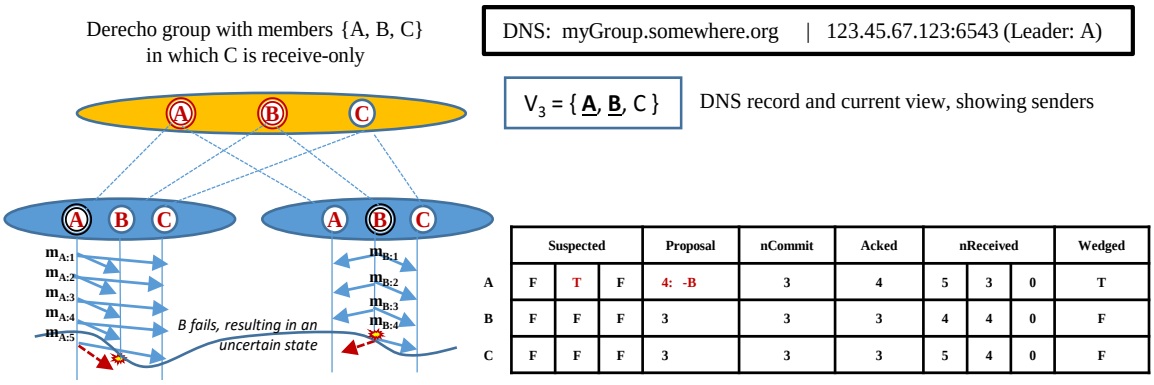
Monotonic Protocols. To create Derecho, we implemented virtually synchronous Paxos as a monotonic protocol. The resulting formulations of virtual synchrony, atomic multicast and Paxos are interesting in and of themselves. Although these monotonic protocols certainly don't look like the classic versions, they turn out to be exceptionally efficient. In fact, the Derecho control plane delivers messages after the minimum number of rounds of distributed communication required for our fault model.

3 Building Blocks

Derecho is comprised of two subsystems: one that we refer to as RDMC, 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 and then focus on how Derecho maps Paxos onto these simpler elements. RDMC details can be found in [21], and SST in [22]. Figure 1 illustrates the structure.

RDMA. RDMC and SST both run over RDMA, which offers reliable zero-copy unicast communication (RDMA also offers several unreliable modes, but we don't use them). For hosts A and B to communicate, they establish RDMA endpoints (*queue pairs*), and then have a few options: (1) The first is like TCP: the endpoints are *bound*, after which point if A wishes to send to B, 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 B, 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. (2) A second option is called "one-sided" RDMA. In this mode, B grants A permission to remotely read or write to pre-agreed upon regions of its memory, without B's active participation; A will see a completion, but B will not be explicitly informed. (3) A third option is new, and we are just looking closely at it: applications using bound queue pairs can enqueue complex sequences of transfers, using a feature that tells the NIC to initiate operation X, and then when X completes, to enable Y.

RDMA is reliable and its 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.



	Suspected			Proposal	nCommit	Acked	nReceived			Wedged
A	F	T	F	4: -B	3	4	5	3	0	T
B	F	F	F	3	3	3	4	4	0	F
C	F	F	F	3	3	3	5	4	0	F

RDMC groups for senders A and B. If B fails, the group wedges along the ragged blue line, disrupting $\mathbf{m_{A:5}}$ and $\mathbf{m_{B:4}}$.

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

Figure 1: Each Derecho group has one RDMC subgroup per sender (in this example, members A and B) 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.

The RDMA standard is supported over both RoCE (Ethernet) and IB (InfiniBand), and there is also a software emulation over standard TCP (SoftRoCE); the latter is slow and interesting mostly for portability. We have both hardware options and in our small testbed, they behave similarly. There is extensive experience with IB at very large scale, but the industry is just starting to experiment with RoCE and we do not yet have access to a genuinely large RoCE deployment. For consistency, the experiments reported here all were done on IB.

To achieve the lowest possible latency, RDMA requires continuously polling for completion events. Unfortunately, this can result in excessive CPU usage even if no RDMA transfers are taking place. As a compromise we chose to dedicate a thread to polling completions while Derecho is active, but after a short period of inactivity we have it switch to sleeping on interupts.

3.1 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 a just one designated sender to transmit messages, so each Derecho group actually coresponds to a collection of RDMC sessions, one for each active in a sender in the current epoch (thus an N member group could have no RDMC sessions, or as many as N).

Small messages. Small messages (1KB or less) are sent using a specialized protocol that employs one-sided writes. When configured to use it, each member allocates a round-robin buffer for each sender. 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 chunks, then routes on an overlay network (the actual network is fully connected, so this purely an abstraction). A number of protocols are supported, all of which are designed so that the receiver will know what chunk to expect so it can ensure that incoming data lands in the proper memory region. Once all chunks are received, the message is delivered via upcall either to a higher-level Derecho protocol (for a group in Paxos or 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 [23]. Individual chunks disseminate down one of several 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. One obviously interesting possibility would be to use the Binomial Pipeline twice: once at the top-of-rack (TOR) level, then again at within each rack. Another option would be to use the *chain pipeline* protocol, which forms a bucket brigade: each message is chunked and then sent down a chain, so that in general each TOR process would be concurrently receiving one chunk while forwarding a prior one. This 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. Because RDMC forwards data asynchronously in chunks over a routing topology, a failure state could involve pending multicasts that have been delivered to some destinations but not to others: a condition we refer to here as a *wedged* RDMC session. The wedge state can also be requested by application logic. Once wedged, RDMC ceases to report newly received messages, and new multicasts cannot be initiated.

Recall that RDMC sessions are associated with a specific sender in a particular epoch of a designated group. A process could have many active RDMC sessions at the same time; one could wedge while others are totally unaffected. Because RDMC has no data retransmission logic, it will be important for Derecho will be to clean up the *ragged edge* left after such a failure, garbage collect the wedged session, and then start the new epoch after the old one has fully terminated.

3.2 Shared State Table

The Derecho SST is a *programming framework for monotonic logic,* used to encode our protocols. The SST is not directly exposed to Derecho users, who see a simple API (the one discussed in the companion paper mentioned earlier [12]) supporting group join, strongly typed and polymorphic point-to-point and multicast operations, and queries. We'll be fairly detailed because our monotonic protocols are expressed in SST's formalism.

SST is built over an all-to-all shared memory table structure. Each group member will have a read-only copy of the SST row for every other member in the group, and a read/write instance of its own SST row. The format of the row (the number of columns and their meaning) is determined by the algorithm using the SST. In Derecho we will instantiate the SST once for each epoch of each active group, so in an application that uses many groups, there may be many SSTs active at the same time.

To share data using the SST, a process updates its local copy of its own row, then *pushes* it to all other group members by enqueuing a set of RDMA requests. A push thus creates a 1-to-N pattern communication pattern, like sending messages, but in fact using RDMA hardware to directly write into remote memories within interrupts or any kind of process-to-process synchronization. While this does impose N^2 load on the RDMA routers, those have full bisection bandwidth and can handle it. One might expect SST to slow down linearly in system size, yet we have never observed such an effect: the time for a process to update its own row is negligible no matter how big the SST since the SST rows are so short (rarely more than 1KB) and the network so fast.

The SST is lock-free, but cache-line atomic: if an object fits within a cache line, applications can safely read it even while writes are occurring. Additionally, when a row is pushed, the RDMA write is done from low to high memory addresses: columns within a row are updated from left to right. For example, if an application had a column X, and then to its right a column Y and updated first X, then Y, then a remote reader that sees Y change will also see the updated value of X: in effect, Y guards X.

The SST framework provides high-level tools for logic programming. The simplest of these is the RowFunction, a wrapper type for associating functions of type $\mathsf{Row} \rightarrow \mathsf{T}$ with a definition-independent name. A RowFunction is performed by some single process, and 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 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, users can construct complex 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 $f : \mathsf{Row} \rightarrow \mathsf{T}$, 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 $\mathsf{RowFunction} \rightarrow \mathsf{RowFunction}$, allowing reducer functions to be arbitrarily combined, somewhat like formulas over a spreadsheet.

Let's look at an example.

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

Here, function `rp` converts the function `row_function` into a RowFunction, calls the reducers `Min` and `Max` on this RowFunction, and then uses the boolean operator reducers to further refine the result. The natural next question is: now that we have defined a RowFunction, how do we take advantage of it? The first step is to assign a name to our new RowFunction so that we may reference it later. We can also register a *trigger* to fire when the RowFunction has attained a specific value. Extending our example:

```
enum 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 the name `Simple` chosen from an `enum class Name`, 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 `volatile SST<T>& -> void` which can make arbitrary modifications to the SST or carry out effectful computation. In practice, triggers will often share one important restriction: they must ensure monotonicity of registered predicates. If the result of a trigger can never cause a previously-true predicate to turn false, reasoning about the correctness of one's SST program becomes easy. Using this combination of RowFunctions, predicates and triggers, one can effectively program against the SST at a nearly-declarative high level, proving an excellent fit for protocols whose descriptions often have the flavor "when everyone has seen event X, then start the next round."

2-Phase Commit in SST. Consider a 2-phase commit protocol in a failure-free run. A leader can initiate such a protocol by placing a request into a designated SST field, indicating that the protocol has been started by incrementing an instance counter and pushing the row. If other members have a predicate that monitors this field, the triggered event would allow each to vote on the request using a value field. When the value is ready (signaled by the value becoming non-zero, or by setting a guard bit), the participant would push its row.

The leader then waits until all the values are ready. Once that condition is achieved, it aggregates the votes and can report the outcome, again with an associated outcome-ready bit. We've accomplished a 2-phase commit, yet expressed the protocol in terms of data shared in the SST and monotonic predicates. The benefit is that SST is very fast: in our experimental cluster, 16 processes can easily share (and witness) 2M updates per second: 125,000/s each. In 4-node groups we reached rates of 275,000/s, with latencies of 1-2us: a factor at least 10,000x faster than could ever be achieved with 2PC over UDP. After every participant has seen the outcome, for example by acknowledging the commit through an SST column dedicated for this purpose, we can start a new protocol instance using the same fields of the SST.

2PC is required to abort in the event of failures. To encode this with the SST, if a failure occurs, the member that senses the event sets the *suspicion* bit in its row, as is seen in Figure 1, where process A suspects process B. This can then be integrated with our protocol: the leader could for example abort if any member fails prior to voting, or could iterate, repeating the protocol until every non-failed member has voted successfully. If the leader itself fails, the next-ranked process can take over the role. Derecho uses this style of commit protocol to implement consensus on changes to the group membership.

3.3 Encoding monotonic protocols in SST

SST predicates have a natural match to the *logic of knowledge* [24], in which we design systems to exchange knowledge in a way that steadily increases the joint "knowledge state.'. If *pred* is true at process A, then A *knows* pred, denoted $K_A(pred)$. Now suppose that all members report when they know *pred* using a bit in the SST. By aggregating this field, process A can discover that *everyone knows pred*. We repeat that pattern to learn $K1(pred)$: every group member *knows* that every other member knows *pred*. Obviously, this form of reasoning would be unstable if the predicates weren't monotonic. However, Derecho uses monotonic values and predicates, hence knowledge programming has a clear fit.

How does a monotonic knowledge protocol differ from a traditional exchange of messages? While the pattern of data exchange is similar, RDMA is very inexpensive. Moreover data might be overwritten while the knowledge-exchange is taking place. The main insight is that in a monotonic protocol, the *deductions* remain valid even when this kind of concurrency is present.

The handling of failures in knowledge protocols now arises. Our approach is motivated by monotonicity. The basic danger introduced by failure is an outcome in which some process discovers that everyone knows *pred*, but then crashes: if it may have acted upon its knowledge before failing, other processes may no longer be able to discover whether or not everyone knows *pred*, and hence would be blocked. But suppose that when a process discovers that the everyone knows *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 K1 was achieved: process B can directly deduce that $K1(pred)$ has been achieved, but could also learn $K1(pred)$ indirectly by noticing that A has done so. This is not merely an optimization: if further failures have occurred, it may be possible to learn $K1(pred)$ indirectly even though $K1(pred)$ is no longer directly discoverable! With monotonic predicates, indirect discovery of $K1(pred)$ is as safe as direct evaluation.

Moreover, because A and B push their updates before acting upon the $K1(pred)$ knowledge, and because SST reports failure suspicions discovered during a push operation before that operation completes, a further guarantee holds: for each group member M, either M *knows* $K1(pred)$, or M has failed (and is already suspected by all other non-failed members). Derecho uses this pattern when stabilizing the ragged edge of an epoch.

3.4 Paxos and Atomic Multicast

Given these tools, we can now turn to our primary objective: a monotonic and asynchronous implementation of atomic multicast and Paxos.

Single sender case. Think first about a single sender in an n member group. By associating an SST with the RDMC session, it is easy to track multicasts through the system without disrupting their flow. Recall that RDMC preserves the sender ordering. So, we can have each process acknowledge receipt in an SST column, using one SST field (one column) for purpose. Now, if our processes track the minimum on this column, they can learn which messages have been acknowledged by all destinations: a monotonically increasing value. Building on this idea, we can support the two basic Derecho modes of receipt for a single-sender group: *volatile* mode, which receives data into a pinned region in DRAM, and *persistent* mode, where we additionally write to NVRAM (SSD) disk. A message is said to be *locally stable* as soon as it is received if in the volatile mode, or when the SSD write finishes if in the persistent mode. We can deliver a message once every group member has reported it as locally stable: first, if in the Paxos mode, we update the log to show that the message committed and force the data to SSD (this is easy because we need just a single monotonically increasing commit counter per log, and can overwrite it each time the value increments). And then we simply do an upcall to the application.

Multiple senders. Given a group with multiple active senders, we can run several single-sender RDMC sessions side by side, one per sender, and dedicate one SST column to each RDMC session. But agreement on the delivery order is needed. Derecho employs a simple round-robin rule. For example, in the figure, A and B are senders; the rule would thus dictate that we deliver A:1, then B:1, then A:2, etc. Each sender will get one slot per round, but can send a null message if there is no actual data to send. If a round consists entirely of senders transmitting nulls, sending pauses briefly to avoid wasting resources.

To implement this rule, we end up with a simple monotonic logic barrier, expressed inductively: the first message in a view is *multi-stable* and *multi-ordered* if it is locally stable at all receivers. It can be delivered, in the same manner as for the single-sender case (e.g. first updating the log, if we are running persistently, and then doing an upcall).

The second message can be delivered when it is multi-stable and the first message has been delivered, and the third message after the second one, and so forth. This already defines the failure-free behavior of Derecho. Notice that we've achieved our goal: the control plane (namely, this delivery rule!) runs parallel to the data plane (namely, the RDMC multicast stream).

Terminating an Epoch. When a group view changes, Derecho must terminate any pending multicasts. Without failures, this is trivial: we wedge the RDMC, so that no new multicasts can be initiated, wait until all pending ones have been delivered, and then we can switch to the new view and start a new epoch, transferring state to whatever member is joining.

With failures, the challenge is to clean up what may be a messy situation, as illustrated in Figure 1 for a crash of process B. RDMC has wedged, and some processes have copies of message A:5, but we don't know whether A:5 reached B before it crashed. A lacks a copy of B:4 and since Derecho does not retransmit missing messages, B:4 is not deliverable.

The epoch termination protocol will start when some member, perhaps A, times out and suspects that B failed. A sets *suspected[B]* and the wedged bit in its SST row, wedges the RDMC and freezes its copy of B's SST row (thus, A will no longer accept new messages from B and B will no longer be able to issue one-sided writes to A's copy of B's SST row). If the group is a top-level group and this causes the number of suspected failures to exceed $\lfloor (N-1)/2 \rfloor$, A throws an exception and terminates, avoiding a split brain scenario if a network failure has isolated A from the primary partition. Otherwise, A

pushes its updated row (that is, A forces an SST update and waits for it to complete).

Other processes (C in our example) will notice the new suspicion. Each mimics A: wedging the RDMC, freezing B's SST row, setting the wedged and suspected bits, and pushing the update. Suspicions thus propagate in a viral manner. As noted above, no member of a group acts on a suspicion until after it has informed every non-failed member in the group.

Next, a leader is selected. Each process computes the lowest-ranked group member that it does not suspect: A in our example. The leader has the role of waiting until all non-failed group members are wedged, then aggregating the minimum for the receive counts in its copy of the SST, *ignoring SST rows for suspected processes*. We will call this a *trim* of the ragged edge.

We assert that it is *safe* to deliver all messages included in the trim, and *necessary* to do so. It is safe to deliver these messages, because every process in the group other than B has received them, and B has failed (or will shut itself down momentarily). It is necessary to deliver all messages included in the trim because B, although inaccessible to us, may actually have delivered these messages itself: after all, all the available evidence suggests that these messages have reached all their destinations. If they did, B might have detected that they were globally stable and could have delivered them. Conversely, any message not included in the trim cannot have been delivered by B or any other process. To be delivered in the normal mode of operation, a message must be globally stable and globally ordered, hence any minimum would have included it. A message not included in A's computation of the minimum must not yet be stable at at least one process in the group, and hence B didn't deliver that message prior to crashing.

However, there is one more case to consider. Suppose that A is the sole group member lacking a copy of message B:4, as in Figure 1. If A is the leader, A's trim computation will obviously omit B:4. But if A were to fail too, and C computed the trim, it would ignore A's SST row, and hence could compute a trim that includes B:4. This situation is precisely analogous to the one we discussed when considering the monotonicity of K1 knowledge predicates, and we can solve the problem by using precisely the same idea as was given earlier: the leader computes the trim, but doesn't act upon it. Instead, it first pushes the trim via the SST. Every other process that sees A's trim echoes the trim in its own SST row, pushing it. Thus A's trim will propagate virally and no process ever acts on the trim until it has echoed it to every non-failed group member.

Once the trim has been pushed, it can be used to terminate delivery for the epoch. This is done by using the trim to extend the standard round-robin order for messages that are multi-ordered: we continue the round-robin delivery order, except that if the trim omits a message, we skip that message (for example, A's trim will omit B:4, hence C, despite having a copy of B:4, will nullify the message in its log and then skip that slot, like the others do, at that point in the round-robin ordering).

Our last concern is that a leader could fail, perhaps even in the act of propagating its trim. So, suppose that C is now the leader in our group (A and B having failed), and further, assume that the group has more members and that C has not lost the primary partition. We need to convince ourselves that either *no* process could have used A's trim to finalize delivery, or that if any process could have done so, then C will use A's trim as its own. But recall that when C took over, it did so because (1) A and B were suspected, and (2) every other non-suspected process has pushed an SST row showing that A is suspected. Further, recall that any process that observes a trim echoes it and pushes it before acting upon it. Thus, if A published a trim and any process could have acted upon it, C finds that trim echoed in at least one row of the SST. Accordingly, our final worry can easily be addressed: When a new leader takes over, it waits until every non-suspected process's row shows that the previous leader is suspected. At this point, it scans the SST. Either it finds a prior leader's trim, in which case it reuses it, or it is safe to carry out a new trim computation. We obtain an iterative consensus protocol that uses the SST to exchange information between processes.

Derecho also uses this protocol to handle agreement on the succession of group views. We dedicate a set of columns in the SST to allow the leader to list a series of one or more proposed changes to the view: "add process P," "remove process Q," etc. So, for example, when A suspected B, the leader would have included "remove B" in this list of changes. Non-leaders echo and acknowledge these changes, and the leader can commit a change once all the non-leaders who are not themselves suspected have acknowledge it. But notice that this pattern is *exactly* the one used to propagate the trim. Thus without extra SST pushes, we accomplish two things at once: we finalize the current epoch and also reach agreement on the initial view of the next one. Just as we did for the trim, a new leader taking over from a prior leader would also take over the prior proposed list of changes, then extend it (presumably, with "remove A", where A is the identity of the prior leader).

Starting the new epoch is now a simple matter: we compute a new view for the group by applying the next change, report it to the members, and to support joins, the leader sends the joining member a state transfer with the new view. All members set up RDMC sessions for the senders in the new view, and a single SST for the new epoch, and progress resumes. Not surprisingly, the

protocol is quite similar to the classic Paxos Synod protocol, or the Isis Toolkit group membership protocol. But notice how this Derecho implementation uses monotonicity: the knowledge-exchange pattern converts individually monotonic data, namely the trim or the list of view changes, into globally monotonic data. Once the protocol has finished, any future run will build on the same data. Further, the protocol waits until that condition holds before taking any action dependent on the basis of that data. Moreover, the protocol is very easy to extend. For example, as sketched here, it would seem that each single change to the membership would trigger a new view. But we can batch changes by having the leader commit a set of changes all at the same time, wait for the members to echo the commit count, and then repeat with the next batch. No extra code is needed, but suddenly our solution will permit a view to change by any number of added or removed members (of course,for a top-level view, the number of removed members can still never drop by more than a minority of the current view).

Total failures. One final task remains to be addressed. In the persistent mode, we need a way to handle failures in which all processes crash, or in which some processes crash while others remained alive. When a process restarts it first checks to see if the group is currently active. If so, it learns whether any of its locally persisted messages became deliverable after it failed, and in what order. Then it joins, and the leader uses an RDMA unicast transfer to send it any other updates that occurred while it was crashed. If the entire group failed but a majority of the last view are ready to restart, recovery involves two steps. First, we need to find the last view of the group (or subgroup) that we are restarting. Recall that in this case the top-level group will be logging each committed view. Accordingly, we can learn the final view of the subgroup that we are restarting. Next, we locate the logs of the subgroup members: they have identical prefixes, but may show differing numbers of committed multicasts, because recording the commit state is a concurrent action. We restart from the log with the highest value for the commit sequence number. Notice that no rollback could ever occur: once a message is committed as deliverable, that status is permanent.

Subgroups. Derecho also automates management of subgroups and shards. Given a top-level group, Derecho offers a simple way to define subgroups of that top-level group, to shard the subgroups in regular ways, and indeed, these mechanisms can be used recursively. Although *membership* of a subgroup is slaved to that of its parent group, in all other ways a subgroup is no different from any other group, and in particular, subgroups and shards offer atomic multicast and Paxos. Because logical partitioning is prevented in the top-level group, subgroups and shards are immune to split brain problems even if membership suddenly drops to a minority (for example, from 3 members to 1). Further, unlike many prior subgroup solutions, our approach sends bytes only between the subgroup members (with no indirect forwarding or filtering). For lack of space, details have been moved to Appendix A.

4 Evaluation

RDMC can scale to very large numbers of replicas: in work that we will report elsewhere, we've used RDMC to create hundreds of replicas, and in its optimal configuration were able to create 256 or 512 replicas in just several times as long as was needed to create just 1 or 2. However, one wouldn't really need Paxos or virtually synchronous multicast at such extreme scale, and our 100Gbps hardware is deployed in a smaller 16-node cluster. Accordingly, we evaluate rack-scale scenarios with 2-16 group members. We ran our experiments on a Dell R720 cluster, each with dual 8-core Intel Xeon E5-2600 processors with 2.5GHz clock speeds, supporting 2 threads per core, with 96GB DRAM memory and two storage options: 4x900GB rotational disk with RAID 0, and an OCZ Technology RevoDrive 3 X2 240G SSD.

These processes are linked by a full-bisection RDMA network. The data reported here is for a 100Gb RoCE Ethernet switch (Mellanox SN2700) and Mellanox MCX456AECAT Connect X-4 VPI dual port NICs. Our cluster is also equipped with a Mellanox 100Gb IB switch; we repeated the experiments with it, but obtained nearly identical results. Although Ethernet is 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 (100Gb/s = 12.5GB/s).

Figures 2 and 3 measure performance of RDMC in our cluster. Figure 2 shows the attained rate for Derecho's small message protocol in groups of various sizes

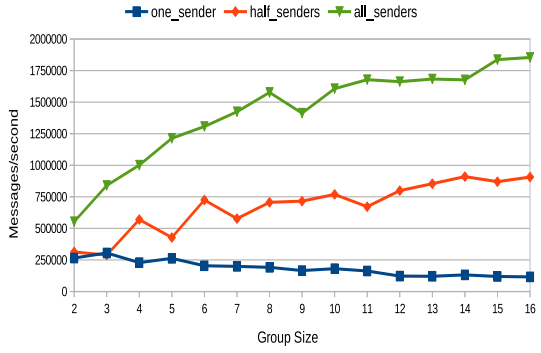


Figure 2: Raw small messages.

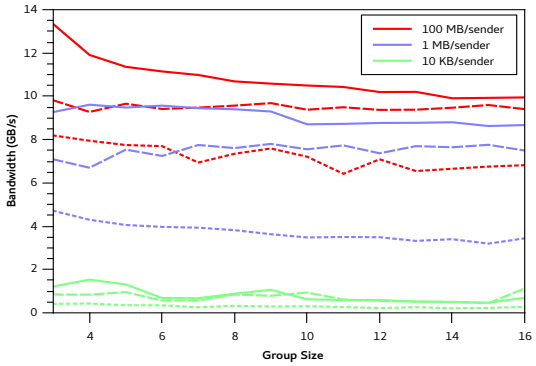


Figure 3: Raw RDMC.

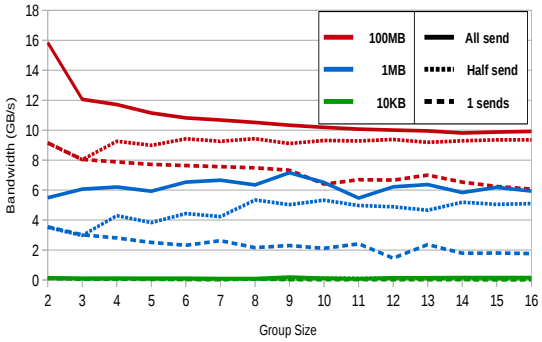


Figure 4: Derecho atomic multicast.

where all members send (green), half send (orange) and one sends (blue). Our peak rate of approximately 2M messages delivered per member, per second, occurs in the 16-member case with all sending. We also measured latency, which varied from 2µs from send to delivery for 4-member groups to 6µs for 16-member groups. In Figure 3, we look at bandwidth for larger messages. At each message size, we again show the case for all sending, half sending and one sending. For this range of group sizes, RDMC performance is fairly steady. We see the highest bandwidth for large messages. Even with just one sender, RDMC substantially exceeds the performence of single threaded memcpy which runs at less than 4 GB/s on our cluster (total memory bandwidth is much higher, but is split over many cores).

Figure 4 shows Derecho's performance in the atomic multicast mode. We observe that Derecho's performance is higher for larger messages; the performance is close to the line rate for 100 MB messages when all nodes in the group are senders. It is clear that Derecho scales well with the group size, as the degradation in performance is negligible with increase in group size.

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 5b.

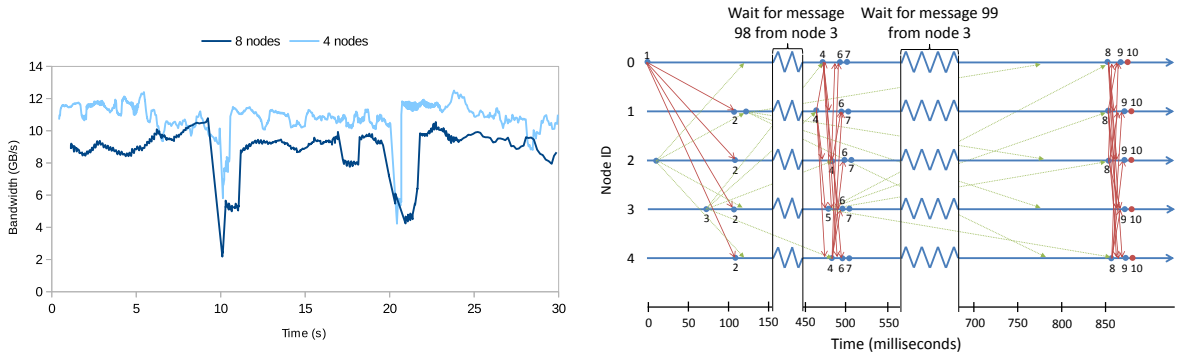
Data type	Local SSD	Derecho
Movie (mp4)	67.7645	63.2405
Random bits	72.1108	67.6758
Facebook page	126.451	112.909
Alphabetic strings	457.55	300.398
Zeros	471.104	292.986

Table 1: Derecho with persistence (MB/s).

Table 1 evaluates Derecho's persistent mode (Paxos). Here we run into the complication that our SSD performance depends on the type of data being written: the device is apparently doing data compression. The table gives the data type, the local write speed for 4MB objects (without Derecho in use at all), and then Derecho in a 4-node configuration where all members send. Note that although Derecho has the same properties as Corfu, direct comparison isn't possible because the SSD technologies differ (our peak rate is about twice that reported for Corfu, on an older SSD technology).

In Figure 5a 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 5b 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.

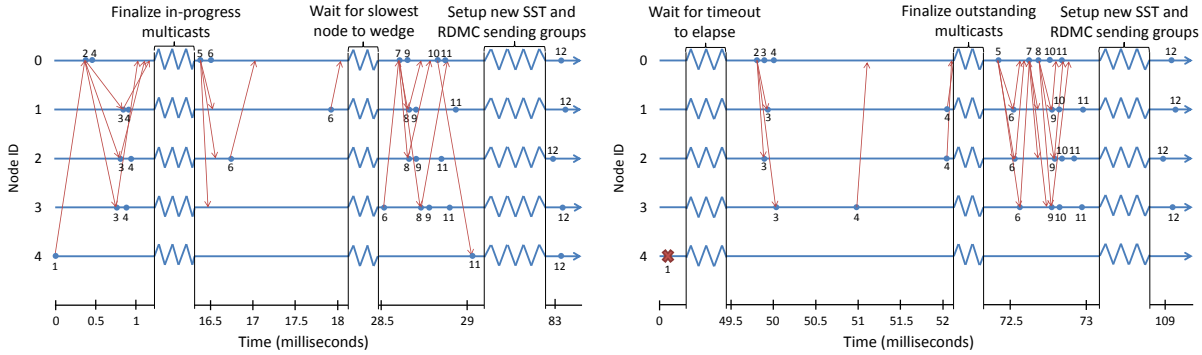
In Figure 6a we see a process joining: ① it requests to join, ②-⑥ are the steps whereby the leader proposes the join, members complete pending multicasts, and finally wedge. In steps ⑦-⑨ the leader computes and shares the trim; all processes trim the ragged edge at ⑨ and the leader sends the client the initial view (⑩). At ⑪ we



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

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

Figure 5: Multicast bandwidth (left), and a detailed event timeline (right).



(a) Timeline for joining an active group.

(b) Crash triggers exclusion from an active group.

Figure 6: Timeline diagrams for Derecho.

can create the new RDMC and SST sessions, and the new view becomes active at (12). Figure 6b 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 5a 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 them is dominated by the amount of data in flight. In experiments with 20MB messages the join and leave disruptions are both much shorter. Compared to Figure 3, we see that Derecho keeps up with RDMC. Its performance for 1 MB and 100 MB messages quite closely matches that of RDMC. However, for smaller sized messages, performance overhead due to SST operations is significant and Derecho performs noticeably worse. Its performance for 10 KB messages is an order of magnitude slower as a result. We conclude that to send high rates of small messages, batching is strongly advisable.

Notice that for 2 nodes at the 100MB message size, Derecho's peak performance exceeds 12.5 GB/s. This is because the network is 12.5 GB/s bidirectional, hence could theoretically reach a data rate of 25GB/s when sending and receiving data simultaneously. In fact, the NIC can't reach this full speed because its attainable bandwidth to the memory unit is slower than the maximum bidirectional RDMA rate.

5 Prior work

Paxos. Prior work on Paxos is obviously relevant to our paper. The most widely cited Paxos paper is the classic Synod protocol [16], but the version closest to ours is the virtually synchronous Paxos described in Chapter 22 of [20] and the Corfu file system [8]. Corfu offers a persistent log, using RDMA for transport and Paxos to man-

age the end-of-log pointer. Microsoft's Azure storage fabric uses a version of Paxos [9]. Round-robin delivery ordering dates to early replication protocols such as [25]. Ring Paxos is implemented in libPaxos [10], and Quema has proved a different ring Paxos protocol optimal [26], but Derecho substantially outperforms both. The key innovation is that by reexpressing Paxos using asynchronously evaluated predicates, we can send all data out-of-band. Appendix B compares performance of libPaxos with Derecho.

Paxos Proofs. The focus of our paper has been practical, and we intend to publish a formal treatment of our protocol and proofs elsewhere. But prior work on Paxos proofs is clearly relevant. The earliest such work of which we are aware was Lamport's own formalization and proofs for Paxos [16], and the first machine-checked proof seems to be Lamport's proof using TLA+ [27] and Lynch's Larch-based simulation result [28]. Keidar did early work on asynchronous protocols for distributed commit and virtually synchronous view management [29,30], Malkhi explored reconfiguring a state machine [1], and the NuPRL system was used to carry out provably correct transformations of Ensemble's virtual synchrony replication protocols [31,32]. Crane [33] and IronFleet [34] are notable for methodology advances, and IronFleet has even been proved live in partially-synchronous networks.

Idit Keidar has proved lower bounds, in message exchanges, for consensus protocols [35]. Through dialog with Gregory Chockler, we've determined that Derecho matches these bounds (Chockler assisted us in counting exchanges of "knowledge" that occur through the SST, and mapping these to Keidar's notion of synchronous rounds of messages). This justifies our claim that Derecho achieves a constructive lower bound in the failure-free case. With failures, FLP applies to all systems strong enough to solve consensus. Chandra and Toueg showed $\Diamond$W [36] to be the weakest failure detector for which progress can occur. Derecho guarantees progress with slightly stronger failure detector, $\Diamond$P. Whether or not it could be modified to use $\Diamond$W, which can "unsuspect" a fault, is an open question.

Atomic multicast. The virtual synchrony model was introduced in the Isis Toolkit in 1985 [37], and its gbcast protocol is similar to Paxos [38]. Modern virtual synchrony multicast systems include JGroups [39] and Vsync [11], but none maps communication to RDMA, and all are far slower than Derecho. At these network rates, batching multiple messages into each send can be highly advantageous, as noted in [40]. The management of a set of subgroups by treating them as properties associated with the current view in an enclosing group was first explored in [41].

DHTs. We noted early in the paper that Derecho is in some sense complementary to key-value stores: Derecho focuses on application structure, replicated state within groups, consistency, coordination and high-availability within application logic. Key-value systems often have this guarantees themselves, but do not offer ways to endow the applications that use them with any special structure or guarantees. On the other hand, any table can be considered as a key-value object by thinking of each row as a value, keyed by the owner, hence the SST has some similarity to RDMA key-value stores such as FaRM [2], HERD [42] and Pilaf [43]. To us, this is a superficial similarity: key-value stores are outstanding tools in support of today's prevailing cloud model, whereas Derecho innovates by creating a new cloud computing model and supporting infrastructure.

A recent trend is for DHTs to 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 [44] is similar in design, using RDMA to build 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. Accordingly we view DHTs, pub-sub APIs, file system APIs, and transactions as higher level services to be offered in the future, using Derecho as a core layer and then extending it with domain-specific logic.

6 Conclusions

The control plane / data plane separation dictated by RDMA motivated us to refactor reliable multicast and Paxos into modules: one providing reliable data transport (RDMC) and the other, asynchronously-evaluated monotonic properties (SST). Derecho sends data via RDMC and encodes protocol properties as sets of SST predicates. The system sets performance records while achieving the full Paxos guarantees when data is persisted to NVRAM, or virtually synchronous multicast when data is simply captured into DRAM. Derecho is coded in C++ 17 and available under 3-clause free BSD licensing.

7 Acknowledgments

Supported, in part, by grants from the NSF, DARPA, DOE/ARPAe, Mellanox, and the UT Stampede computing center. Gregory Chockler, Matei Zaharia, Chris Hobbs, Miguel Castro, David Cohen, Ant Rowstron and Kurt Rosenfeld all provided helpful comments.

Appendix A: Subgroups

As noted earlier, a companion paper discusses the Derecho API [12], which extends the usual notion of a single group to allow any group to be fragmented. A *subgroup* is just a normal group, but with its membership defined as a subset of the view of some parent group, and a sharded subgroup additionally has a rule that computes the correct number of shards and a policy for assigning the subgroup members to shards (various methods are possible; we favor an approach that minimizes churn on membership changes).

It turns out that the Derecho single-group protocol can easily be extended to support subgroups and sharding. This came as a surprise: past group communication systems have struggled to offer subgroups, and without exception have been forced to do so inefficiently (for example multicasting in the full group and filtering out undesired incoming messages) or with weak semantics. Neither limitation applies in our work.

We chose to report this work in an appendix both because we lacked space in the body of the paper, but also because the implementation is still underway. In contrast, all features of Derecho reported in the body of the paper are complete and can be downloaded from Derecho.codeplex.com.

We define a subgroup to be a group (hence with the normal communication options: atomic multicast, Paxos, or raw reliable RDMA multicast), but with membership computed as a subset of the top-level membership using a function that computes on the top-level group view together with data consistently replicated within the top-level group. Subgroups can have subgroups, hence sharding is just another form of subgroup creation, with a very regular pattern.

Thus any top-level group membership change induces atomic changes in the membership of its subgroups, the shards of those subgroups, etc. Our protocol is highly efficient: no additional communication is required, and any multicast sent in a subgroup will be handled purely by the subgroup's members: there is no need for indirection or filtering, as was common in some past subgroup solutions. Further, we allow one member to belong to multiple subgroups, hence any desired pattern of groups and subgroups can be created (see Figure 7).

We start by associating a set of deterministic functions with our main group, which compute the subgroup and shard memberships. These can use the group view or any form of consistently replicated top-level group state as input. Thus when a new view is reported, *every top-level group member will know the full membership of all associated subgroups and shards.* Next, we modify our stability and ordering policies to instantiate them not merely for the top-level group, but also once for each subgroup

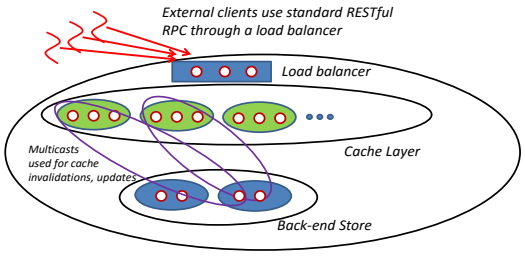


Figure 7: Derecho offers efficient support for creating subgroups of a group, or shards, and these patterns can be applied recursively. This permits developers to create elaborate data center services, such as the one illustrated here. State transfer, atomic multicast or Paxos and membership consistency guarantees extend to the subgroups, greating simplifying the task of the application designer.

and each shard. Thus, if some process is a sender in more than one role, receivers will have separate columns for receive counts from it, one for each of its roles. We create one RDMC session for each role, and one SST for each subgroup (fortunately, RDMC sessions and SSTs are quite inexpensive). For example, with shards of size 3 we would have small RDMC sessions, one per sender, per shard.

One of these functions is called by the Derecho new-view event handler. Given a view in a parent group, it simply selects the members that will belong to the subgroup. For the case of sharding, there is also a shard-generating function that will repeatedly call the shard membership function, supplying the shard number as an extra argument.

Next we consider the Derecho delivery rule. Suppose that we now run this rule once per subgroup. The policy discussed in the main body of the paper generalizes in the obvious way: we simply restrict the rule to the membership of the subgroup in question. For example, in a 3-member shard, an atomic multicast becomes deliverable when it is locally stable at all 3 members and the prior multicasts in the 3-member round robin ordering have all been delivered, and similarly for Paxos.

Tracking the stability information can be highly efficient as well, but for this we need to do something slightly tricky. Recall from the Derecho delivery algorithm that each subgroup will require a set of columns, which its members will use for tracking nReceived within the subgroup. However, since non-members do

not receive subgroup messages, the values would be 0 for non-members. Meanwhile, those non-members are very likely to be members of other shards or subgroups. This leads to the insight that columns can usefully be shared.

For example, in a group sharded into disjoint shards of size 3, we only need 3 columns to represent all the nReceived values. If pairs of shards overlap, we would need 6 columns; if each member belongs to 3 shards, we would need 9 columns, etc. Updates to nReceived need only be pushed to the other shard members, hence in the normal mode of operation would have small and constant cost: Only the top-level view protocol needs to push SST rows to the full set of top-level group members.

To carry out this behavior, we associate a column-mapping function with each shard. Given the shard identifier and sender identifier, the function returns the column to use for the nReceived counters.

In this approach there is no distinct concept of a view change that occurs just in a subgroup or shard: only the top-level view changes, and when that happens, we only need to run the top-level view protocol once, extended to cover subgroups in the "obvious" way: The ragged cleanup only occurs once, run by a single top-level leader on behalf of the top-level group and all its subgroups.

At first glance it may seem that propagating the trim will require one column for each potential sender in each subgroup, which would defeat our column-packing idea, it turns out that we can do better. The "trick" is to compute the trim for each subgroup, one subgroup at a time, and again to reuse columns, but this time we reuse the columns associated with the trim report. Specifically, we have the leader send different trims: when pushing the leader's SST row to a member of subgroup S, the leader uses a version of the row that reports the trim *relevent to that member* on the basis of the column mapping used by that member. Thus, members of shard S will be told about stability in shard S using columns that also report stability in shard Q to members of shard Q, etc.

We thus can build very elaborate structures of groups and subgroups, allow each group or subgroup to select the desired level of persistence (atomic multicast or Paxos). If any subgroup uses Paxos mode, we always log views in the the top-level group even if it does not use Paxos mode, for reasons that will be clear in a moment. Now Derecho will automate construction of the subgroups, state transfer, multicast ordering and epoch termination. Moreover, it does all of this through out of band logic, disrupting the RDMC message flow only when membership changes occur (and only briefly: we saw earlier that switching to a new group view takes just 150ms or so).

Appendix B: Comparisons

Space limitations precluded us from including all our experimental data in the body of the paper. We decided to report the experiments comparing Derecho with prior systems in this appendix.

Using the same cluster on which we evaluated Derecho, over the same networking layer (but now in 100Gbps Ethernet mode), we instrumented libPaxos, Zookeeper and attempted to evaluate Vsync. The patterns of communication we used are intended to match the scenarios evaluated for Derecho. None of these pre-existing packages has been ported to use RDMA explicitly, hence an issue of fairness arises: it was important to us that the comparisons be apples-to-apples, but on the other hand, the prior systems are not generally used in the same manner as we intend for Derecho. As we now explain, this led to a decision to not report data for Vsync.

The Vsync system seems like an obvious one to compare with Derecho: it offers an OrderedSend primitive with semantics similar, although not identical to the OrderedSend in Derecho. However, the details rendered this less reasonable that we had hoped. First, in Vsync, OrderedSend is *optimistic*, meaning that the system delivers messages on arrival, before they are fully stable. Such an action is optimistic because if a failure were to occur precisely as a message is delivered, all copies could be lost, and yet some member might have received that lost message. To prevent such outcomes from having externally visible consequences, Vsync uses a barrier: a separate *flush* primitive should be invoked prior to taking an externally visible action.

In analogous terminology, a Derecho atomic multicast is *pessimistic,* as if Vsync's flush were called on every operation. Our first thought was to try this, but Vsync is not intended to be used this way and it caused a sharp performance degradation.

A further difference is that Vsync runs on IP multicast using its own flow control and acknowledgment scheme,

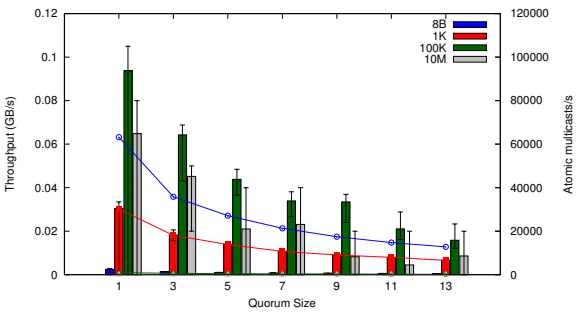


Figure 8: Performance of libPaxos configured to run purely in-memory (an atomic multicast)

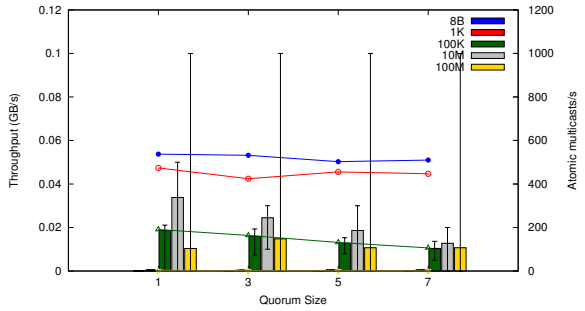


Figure 9: Performance of libPaxos in persistent mode, using SSD disks (true Paxos)

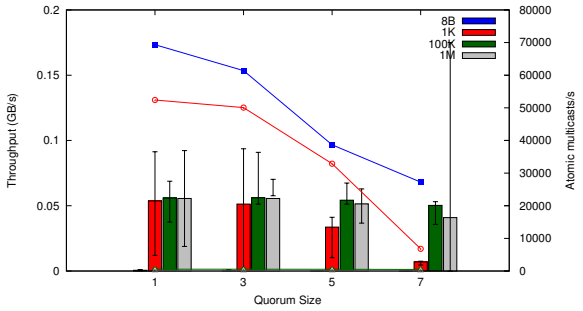


Figure 10: Performance of Zookeeper

and was explicitly designed for small messages. Forcing it to send even moderately large ones involves changing system configuration parameters in untested ways.

Accordingly, we decided not to include graphs comparing Vsync with Derecho. Even so, our limited experiments make it clear that as messages get larger Vsync lags and becomes orders of magnitude slower than Derecho atomic multicast, and for the upper size limits Vsync can handle the performance difference is already a factor of at least 15,000x: not surprising when one considers that we are comparing a UDP-based multicast that sends 8KB packets to a zero-copy RDMA transfer. For small messages, Vsync is roughly 25-100 times slower than Derecho's small-message protocol, depending on the traffic pattern.

A more direct comparison was possible with Zookeeper and libPaxos, both of which are optimized for fast Ethernet. (We have no idea how hard it would be to extend them to directly use RDMA). Derecho's persistent mode has identical semantics to libPaxos or Zookeeper, meaning that one could safely port an application from either system to Derecho. Interestingly, many users run libPaxos directly from memory rather than on files. We evaluated this case too, since it is semantically identical to Derecho atomic multicast.

The results are seen in Figures 8, 9 and 10, and underscore the point made in our abstract and introduction: Derecho is two to four orders of magnitude faster, scales far more efficiently, and has dramatically lower latencies.

References

[1] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a State Machine," *SIGACT News*, vol. 41, pp. 63–73, Mar. 2010.

[2] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "Farm: Fast remote memory," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Berkeley, CA, USA), pp. 401–414, USENIX Association, 2014.

[3] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, (New York, NY, USA), pp. 8–17, ACM, 1988.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," *ACM Trans. Comput. Syst.*, vol. 27, pp. 5:1–5:48, Nov. 2009.

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[6] P. T. Eugster, R. Guerraoui, and C. H. Damm, "On objects and events," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, (New York, NY, USA), pp. 254–269, ACM, 2001.

[7] P. Eugster, "Type-based publish/subscribe: Concepts and experiences," *ACM Trans. Program. Lang. Syst.*, vol. 29, Jan. 2007.

[8] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "CORFU: A Distributed Shared Log," *ACM Trans. Comput. Syst.*, vol. 31, pp. 10:1–10:24, Dec. 2013.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi,

A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 143–157, ACM, 2011.

[10] "LibPaxos: Open-source Paxos." `http://libpaxos.sourceforge.net/`.

[11] "Vsync reliable multicast library." `http://vsync.codeplex.com/`, Nov. 2011.

[12] K. Birman, J. Behrens, S. Jha, M. Milano, E. Tremel, and R. van Renesse, "Groups, Subgroups and Auto-Sharding in Derecho: A Customizable RDMA Framework for Highly Available Cloud Services," Submitted to NSDI '17., 2016.

[13] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 49–65, USENIX Association, Oct. 2014.

[14] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System is the Control Plane," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, (Broomfield, CO), pp. 1–16, USENIX Association, Oct. 2014.

[15] K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, (New York, NY, USA), pp. 123–138, ACM, 1987.

[16] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[17] L. Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 254–280, Apr. 1984.

[18] F. B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, vol. 22, pp. 299–319, Dec. 1990.

[19] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Comput. Surv.*, vol. 47, pp. 42:1–42:36, Feb. 2015.

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

[21] J. Behrens, K. Birman, S. Jha, and E. Tremel, "RDMC: A Reliable RDMA Multicast Protocol," Under review., 2016.

[22] S. Jha, J. Behrens, K. Birman, and E. Tremel, "Distributed State Sharing and Predicate Detection over RDMA," Under review., 2016.

[23] P. Ganesan and M. Seshadri, "On Cooperative Content Distribution and the Price of Barter," in *25th IEEE International Conference on Distributed Computing Systems, 2005. ICDCS 2005. Proceedings*, pp. 81–90, June 2005.

[24] J. Y. Halpern and Y. Moses, "Knowledge and common knowledge in a distributed environment," *J. ACM*, vol. 37, pp. 549–587, July 1990.

[25] J.-M. Chang and N. F. Maxemchuk, "Reliable broadcast protocols," *ACM Trans. Comput. Syst.*, vol. 2, pp. 251–273, Aug. 1984.

[26] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput Optimal Total Order Broadcast for Cluster Environments," *ACM Trans. Comput. Syst.*, vol. 28, pp. 5:1–5:32, July 2010.

[27] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu, "Specifying and Verifying Systems with TLA+," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, (New York, NY, USA), pp. 45–48, ACM, 2002.

[28] T. N. Win, M. D. Ernst, S. J. Garland, D. K. Kaynar, and N. Lynch, "Using simulated execution in verifying distributed algorithms," in *4th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, vol. 2575, pp. 283–297, Springer Verlag Lecture Notes in Computer Science, 2003.

[29] I. Keidar and D. Dolev, "Increasing the resilience of atomic commit, at no additional cost," in *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '95, (New York, NY, USA), pp. 245–254, ACM, 1995.

[30] I. Keidar and R. Khazan, "A Virtually Synchronous Group Multicast Algorithm for WANs: Formal Approach," *SIAM Journal on Computing (SICOMP)*, vol. 32, pp. 78–130, Nov. 2002.

[31] V. Rahli, N. Schiper, M. Bickford, R. Constable, and R. van Renesse, "Developing correctly replicated databases using formal tools," in *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, (Atlanta, GA), June 2014.

[32] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable, "Building reliable, high-performance communication systems from components," *SIGOPS Oper. Syst. Rev.*, vol. 34, pp. 16–17, Apr. 2000.

[33] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos Made Transparent," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 105–120, ACM, 2015.

[34] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: Proving Practical Distributed Systems Correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 1–17, ACM, 2015.

[35] I. Keider and S. Rajsbaum, "Information processing letters," pp. 47–52, December 2002.

[36] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," tech. rep., Ithaca, NY, USA, 1994.

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

[38] "Gbcast protocol." `https://en.wikipedia.org/wiki/Gbcast`, July 2012.

[39] B. Ban, "JGroups reliable multicast library." `http://jgroups.org/`, Nov. 2002.

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

[41] K. Birman, R. Friedman, and M. Hayden, "The maestro group manager: A structuring tool for applications with multiple quality of service requirements," tech. rep., Ithaca, NY, USA, 1997.

[42] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA Efficiently for Key-value Services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, (New York, NY, USA), pp. 295–306, ACM, 2014.

[43] C. Mitchell, Y. Geng, and J. Li, "Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 103–114, USENIX Association, 2013.

[44] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast In-memory Transaction Processing Using RDMA and HTM," in *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (New York, NY, USA), pp. 87–104, ACM, 2015.