Groups, Subgroups and Auto-Sharding in Derecho: A Customizable RDMA Framework for Highly Available Cloud Services

Ken Birman, Jonathan Behrens[1], Sagar Jha, Matthew Milano, Edward Tremel, Robbert van Renesse
Department of Computer Science, Cornell University, [1]Cornell University and MIT

Abstract

Derecho is an RDMA-based distributed computing framework that unifies group membership management, consistent data replication and persistence. Applications are structured as top-level groups which can be split into subgroups or automatically sharded in a regular manner. To preserve membership invariants, Derecho adjusts subgroup membership as parent-group membership evolves. The resulting mix of features simplifies creation of cloud services.

1 Introduction

Zero-copy RDMA networking has the potential to revolutionize distributed computing, but isn't trivial to use in its native form. We created Derecho to bridge this gap, offering scalable high speed RDMA communication in support of *group-structured distributed applications*.

As a running example, consider a strongly consistent server holding data in a sharded back-end, which is replicated for persistence and availability (Figure 1; the "shards" are simply the blue subgroups shown, each holding a disjoint subset of the data in the store as a whole). Clients access the service via a load balancer. The cache is sharded as well, but more aggressively distributed to soak up a large read load. Because we desire strong consistency, the back-end performs updates as atomic transactions, and uses multicasts to invalidate cached data prior to performing transactions that modify the store. This design would be difficult to implement, especially if there are structural invariants, such as that the load-balancer accurately track the membership and subgroups of the cache and back-end.

Derecho automates the hard parts, offering a way for the developer to specify a desired structure, then customize it by attaching event handlers. A service like the one shown requires as little as a few dozen lines of application-specific logic, yet because data moves out-of-band over zero-copy RDMA, could perform exceptionally well.

Here we focus on datacenter use cases. Our existing implementation supports applications of sizes that range from just a few processes to thousands, and can scale to even larger deployments. In future work, we will introduce an explicitly hierarchical structure that would enable georeplication (hopefully, with similar ease).

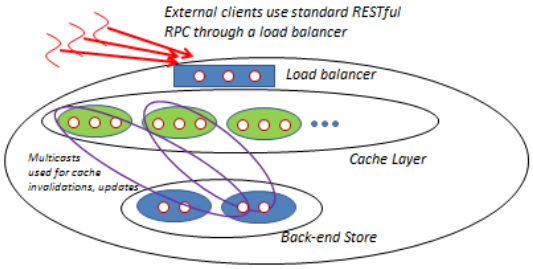

Figure 1: A cached stateful service with three subgroups. Two of the subgroups are additionally *sharded*, as occurs in distributed key-value stores.

2 Derecho Basics

The Derecho protocols and performance are discussed carefully in a companion paper, which has also been submitted to NSDI [8]. In that paper, we show how Derecho uses reliable unicast RDMA to implement both virtually synchronous ("atomic") multicast and Paxos persistence in groups of varied sizes, and include experiments demonstrating that Derecho achieves record-breaking data rates and latencies at exceptionally low overheads. As a multicast, Derecho is hundreds to thousands of times faster than systems like Zookeeper [38], libPaxos [22], and Vsync [37] when similarly configured. Used for persistence in its Paxos mode Derecho offers similar performance to Corfu [7] or the Microsoft Azure Fabric storage [2], but can support larger numbers of replicas with little loss of speed.

Key to this performance is an efficient data-relaying layer that builds an overlay and then uses RMDA unicasts to distribute updates within groups, which we control using an out-of-band framework that asynchronously enforces stronger properties such as totally ordered delivery.

By *group* we mean a set of processes with membership managed by the Derecho system, and reported via *new view* events. We'll use the term *top-level group* to explicitly refer to the full membership of an application. Derecho can support many side-by-side top-level groups. However, and here we depart from prior systems, a Derecho group can also support *subgroups*

with customizable policies to determine membership. Subgroups (and shards, which for us are just a set of subgroups created using a pattern) can overlap in membership. All types of groups offer the same functionality; our terminology (group, parent group, subgroup or shard) is primarily for clarity of exposition except in one sense: membership of a subgroup is strictly determined by the membership of the top-level group to which it belongs, updated atomically when the parent group membership changes.

The Derecho execution model unifies virtually synchronous membership [29] with Paxos [28] state machine replication [18]. This style of integrated model was suggested by Malkhi [36] (for the theory see Chapter 22 in [16]), and was first implemented in the Vsync system [37].

Executions run as a series of *epochs.* During any single epoch, membership of a top-level group (and its subgroups) remains stable. An epoch starts when a new view in the top-level group becomes defined. This induces new views in the subgroups, all reported via upcalls (a single process belonging to multiple groups gets multiple upcalls). Then the epoch becomes active, and new messages will be delivered, in accordance with the relevant level of guarantees. An epoch ends when membership changes due to a join or leave/failure.

When a process is added to a subgroup for the first time, it is initialized either from a constructor (if the group is being created), from the state of some existing member (if joining an active group), or from persisted storage (the Paxos case).

Although any Derecho process can send point-to-point messages or point-to-point RPCs to any other Derecho process, only a group member can multicast to the group, or perform a multicast query. Derecho multicasts are virtually synchronous: they are delivered to all non-failed members, in the same view, in a total ordering that also preserves sender ordering. If an epoch terminates, pending multicasts not delivered in the current epoch will automatically be reissued in the next one, preserving sender ordering.

Each group has an associated protocol that it uses for multicasts. In the Paxos case, a multicast is delivered only after it has been persisted into a set of logs (one per member) and totally ordered. For the atomic multicast case, we deliver totally ordered messages after all members have an in-memory copy, but without logging them. Last is a "raw" multicast called RDMC. Here, latency is minimized but there is no ordering across concurrent sends, no logging, and a failure can disrupt delivery.

It is important to emphasize that even though our platform implements the Paxos *specification*, the protovol it the one from the classic Paxos paper [28]. The classic Paxos runs a specific two-phase protocol implementing state machine replication. Over time, this narrow view of Paxos has been supplanted by a generalized one, in which "Paxos" refers to any protocol that complies with the specification. Today there are dozens of Paxos variants. Derecho's Paxos protocol satisfies durability and total ordering, but (unlike classic Paxos), every delivered message is received, in order, by *every* live group member. Thus, with Derecho, every live group member has its own local copy of the full, consistent state.

This does not mean that members must *behave* identically: each member has its own rank in the membership view, and can play a distinct role, but they do share identical data (if members need just a subset of the updates, we would view that as a case for creating a subgroup or sharding the parent group, and then fully replicating data on a subgroup or per-shard basis). Derecho applications can thus perform consistent coordinated actions.

Derecho tolerates halting failures, which can be detected by hardware, protocol timeouts, or application logic. Membership of the top-level group is updated in a manner that prevents logical partitioning (split brain behavior): an isolated process or set of processes will be blocked from interacting with other healthy processes and rapidly shut down. Importantly, this implies that logical partitions cannot arise in subgroups, but also that subgroups cannot make progress if a majority of the current top-level group members suddenly fail.

3 Challenges Explored Here

Our central innovation relative to prior group computing frameworks are the subgroup features of Derecho, which support *multi-group structures,* as illustrated in Figure 1. Different elements can have different consistency needs. For example, we may wish to use Paxos in the back-end, because it probably needs persistence. In contrast, for invalidations and updates from the back-end to the cache, the best match is an atomic multicast: a cache is volatile, so a protocol designed to persist data to SSD would add unnecessary overhead.

Notice that in our figure, the cache shards are explicitly represented (green), and yet the multicasts to them only occur in the purple groups. Why should Derecho even track these shards? One reason is to maintain consistency between the load balancer and the rest of

the system. A second is that by explicitly creating these subgroups, the application benefits from state transfer: Derecho will automatically bring a joining process up to date by sending a serialized state snapshot from an active member to the joining one, transparently, at a suitable instant during the join protocol.

A cached, stateful, data-management service is just one example among many. We are in dialog with a company exploring the feasibility of creating *provably correct control systems* to run on clusters of computing devices in a self-driving car or other similar settings. Such applications will depend on the correctness of their group structures, membership will need to evolve in response to failure and changing conditions, and safety conditions will thus extend to structural aspects of the solution. While our near-term goal focuses on datacenter uses, our API needs clean semantics that can eventually be used in safety analyses.

Beyond consistency of membership between parent groups and their subgroups, there are also consistency questions when communication occurs internal to Derecho but between members of different groups. For example, suppose that process A (not a member of group G) sends a point-to-point RPC to process B, which it selects on the basis of B's rank in G (recall that all members of the top-level group can see the membership of all groups, subgroups and shards).

Now, suppose that B fails without responding. A, seeing the top-level view change, discovers that C now has B's old role, and resends its request. This triggers a race: view change notifications are concurrent, so perhaps C hasn't yet seen the membership change and is not expecting such requests; it could crash or respond incorrectly. We see this as a likely and common problem, hence to avoid such cases, Derecho delivers a message sent in view K only after the target has been notified of view K. The delay should be minimal (milliseconds), yet potentially tricky application logic is obviated.

But not every such feature belongs in a core system. Over nearly three decades there have been many proposals for cross-group consistency guarantees. The Isis Toolkit guaranteed that multicasts sent in different groups would be totally ordered at any overlapping destinations [29]. The Totem system imposed a *global* total message order [34]. Several systems offer causal ordering between operations that span groups, and Lazy Replication [29] tracks causal paths external to the entire system. In the belief that an API should be minimal, rather than add features like these to the core Derecho platform, we prefer to offer them via optional library APIs (if at all: some proposals never saw wide adoption).

Other challenges arise because of the raw speed of RDMA. We noted that Derecho is faster than previous systems of this kind. Derecho achieves this by separating control-plane from data-plane: its protocols move the data over a fast transport while running the logic that decides when stronger safety and ordering guarantees have been achieved out of band. To fully benefit the application itself must also adhere to such a separation. The question then arises of how these kinds of control and synchronization policies are expressed.

Looking at past work, one can distinguish classes of approaches to distributed application structuring. One approach starts with a single specification and *compiles* a distributed system, as in Fabric [13] or Frenetic [15]. This is powerful because the code is effectively a single distributed program. However, systems like Fabric and Frenetic have not been applied to complex multi-group structures of the kind considered here, and lack tools to assist in separating control from data movement.

A second approach is evident in the MPI HPC library [25]: it assumes a long-running program with cloned on non-virtualized machines. A leader is selected in a topology-aware manner; the clones are gang-scheduled. MPI's startup can be slow (seconds or even minutes), and it performs poorly with scheduling delays, virtualization, or resource contention. These are at best minor issues in large, batch-scheduled HPC clusters, but in our target setting would be cause for concern. On the other hand, MPI has a highly effective integration with RDMA and a number of mechanisms that facilitate the control-plane/data-plane separation we seek in Derecho applications.

A final class of prior work is seen in libraries that offer groups as library-supported abstractions, but focus on one group at a time (for example the Isis Toolkit [29], Totem[34], Horus [17], Ensemble [4], JGroups [20], Vsync [37], LibPaxos [22] and Corfu [7]). Some offered subgroup features [34][23][10], but those often had weak semantics, limited functionality, or high overheads. None could easily leverage zero-copy RDMA: they all touch data at many stages of event processing.

Accordingly, in designing Derecho we set out to create a new option: a new way of promoting the control plane/data plane that would promote separation of roles and clarity of application intent.

4 Montonicity and Monotonic Reasoning

> "You can put your mind in order by focusing on one thing at a time, doing it well, and appreciating the opportunity that this doing offers."
> **— Unknown author**

The Derecho approach to control-plane/data-plane separation leverages *monotonicity*, which can be understood by thinking about the kinds of knowledge that can be acquired about a system as it runs. In any distributed system, whether or not it uses RDMA, knowledge can be viewed as accumulating over time: as processes exchange messages, they learn about one-another's states, reach agreement on such things as event ordering or process health, and agree upon actions that they then carry out.

Our insight was that if the flow of data is very rapid, potentially faster than the control protocol can track on an event-by-event basis, then a monotonic programming model can both encourage batching (needed if control actions lag the data rate) while also ensuring that once a condition is established, it will not promptly be invalidated by a subsequent event. In a monotonic setting, knowledge increases steadily: once a monotonic property is discovered to be true, it remains true, no matter what happens later. Thus, if a system is *monotonically safe*, it suffices to prove correctness on a step by step basis, because no safe action will ever be invalidated by some subsequent event. Derecho's protocols are safe in this monotonic sense.

For example, think about the number of messages some receiver has received from a particular sender. Even if the messages are arriving at an immense rate, the count of received messages only increases. Now suppose that we take a group of known membership (namely, that defined in the current view for the current epoch), and impose a round-robin delivery order on the members. We can make the rule that a message is safe to deliver once (1) every member in a group has received it (and if in Paxos mode, has also logged it); and (2) the member evaluating the rule has delivered every prior message. This is part of Derecho's delivery rule, and is a good example of a property that, once true, remains true. (The full delivery rule contains additional aspects that come into play only during view changes.)

When we set out to implement the Derecho multicast protocols in this manner, we were excited to discover that one can encode 2PC, aggregation protocols, virtually synchronous multicast and Paxos entirely as asynchronous monotonic predicates [8], even covering failure handling. Only membership changes pause the data flow. Our API is intended to help applications benefit from a similar style of asynchrony and monotonic reasoning.

5 API Goals and Approach

> "Everything should be made as simple as possible, but no simpler."
> **— Albert Einstein**

A tension is evident in our goals. On the one hand, we want to support a powerful model and to promote a novel style of monotonic deductive reasoning. However, we also were intent on omitting extra bells and whistles. Desired is the simplest and most flexible API possible:

- The system should have a hierarchical namespace, allowing processes to create/join groups and subgroups without name conflicts.
- The membership of subgroups should be coordinated and consistent with the membership of parent groups.
- A joining process should be given a copy of the current state (state transfer).
- Derecho guarantees should be customizable on a per-group (or per-set-of-shards) basis.

6 Derecho P2P Sends and Group Multicasts

Given these objectives, we start with the basic Derecho API for a single-group application. Consider the MemCacheD class outlined in the code sample above, which could be used to implement members of the "cache" groups in the service shown in Figure 1. Here is an example of a point-to-point send to a single process running the MemCacheD code:

```
auto outcome = g.P2PSend<MemCacheD@put>(who,
               "John Doe", 22.7);
```

Derecho's job is to turn this function call into a message, and then to send it asynchronously to *who*. The target will then perform an upcall to a corresponding handler for `put`. The marshalling scheme for the function arguments can handle any C++17 structure that serializes to a byte vector[1].

[1] We currently require that the *sizes* of all objects be statically fixed with one exception: if the application sends a single argument of type byte-array, we allow it to be of variable size.

```
class MemCacheD : public
DerechoClass<MemCacheD>{
  unordered_map<string,double> underlying_map;
  @P2PEntry-Point void put(string s, double v)
                                    {code...}
  @MCEntry-Point void put(string s, double v)
                                    {code...}
  @P2PEntry-Point const double get(string s)
                                    {code...}
  @MCEntry-Point const double get(string s)
                                    {code...}
  serializes(underlying_map);
  void newView(View& new_view) override {...}
  MemCacheD(group g) : DerechoClass<...>(g)
                                    {code...}
};

{...

  Group<MemCacheD, CacheLayer ,...> g {
  [args_for_first], [args_for_second]};

...}
```

Code Sample: A skeleton of the MemCacheD class and an example of how it is constructed as part of a group.

The MemCacheD class defines two `put` entry points, one for P2P requests and one for multicasts. These can be polymorphic; any invocation of `put` must provide matching argument types[2]. The outcome returned from a P2PSend is an object that the caller can check for errors (e.g. if *who* fails), and can also be used to wait until the P2PSend is known to be stable.

The code sample also shows how the top-level group (containing subgroups such as the cache nodes) is created. Notice that the top-level group has been associated with a list of classes, starting with MemCacheD. CacheLayer is a second class associated with the same group; we'll use it shortly to illustrate the formation of subgroups. The arguments to the Group constructor will be passed to the respective constructors of the listed classes: first a set associated with MemCacheD, then a set for CacheLayer, and so forth.

Finally, we see that each class defines a handler for "new view" events. Within a process, Derecho instantiates each class once for each group or subgroup to which the process belongs. For each such instance, when its group's membership changes, it will receive an upcall listing the members in rank order.

The MemCacheD class explicitly defines its serialization state to be the object called `underlying_map`; it could also have other internal state that isn't part of the serialized state. A joining group member will be initialized from the `underlying_map` of an active member. Derecho also uses the serialization state to compact Paxos logs: a log will have a checkpoint (one of these states) and then a series of updates. The const annotation on the `get` entry point tells Derecho not to log `get` operations and is thus equivalent to a "ReadOnly" annotation.

A P2PQuery is similar to a P2PSend, but additionally returns a result:

```
auto result = g.P2PQuery<MemCacheD@get>(who,
              "Holly Hunter");
```

The result object is a *future*: a C++ class that can be used to track progress of the query (which is sent asynchronously, like the P2PSend) and eventually, to retrieve the result. The result type will be that of the handler that accepted the incoming request, and type checking is done at compile time.

These P2P examples have very similar analogues when using ordered, asynchronous, multicasts. Here is an example of a multicast send to a group of processes running MemCacheD:

```
auto g = Join<Paxos>("myGroup");
auto res = g.OrderedSend<MemCacheD@put>
                      ("John Doe", 22.7);
```

First, the application joins a group called myGroup, selecting the Paxos protocol (other options being AtomicMulticast, and Raw; the latter has minimal latency but weak guarantees). As in the P2P case, OrderedSend returns a future (*res*) that can be used to wait until the message has become stable (that is, is certain to be delivered even if failures disrupt the system).

Had the user specified AtomicMulticast or Raw rather than Paxos, the group consistency semantics would be configured accordingly.

A *multicast query* returns a set of results:

```
for(auto res : g.OrderedQuery<MemCachedD@get>
                      ("John Doe")) { code... }
```

Notice that these two invocations of `get` actually map to different methods: the P2P method in the case of a P2PQuery, and the multicast version in the case of an

[2] The example is for communication *internal* to a Derecho application. Clients *external* to the Derecho application would interact wih it through a standard load-balanced RPC layer such as RESTful RPC.

OrderedQuery. An OrderedQuery is asynchronously enqueued for transmission (just like an OrderedSend). However, now the invoked methods return typed values. These are automatically marshalled and passed to the initiator via P2PSend, where they are collected by the result object. The *for* loop will iterate over the replies as they arrive (should a process fail without replying, all views of all groups will adjust to drop the failed member, and the result aggregator will just skip the failed member). The caller can track progress through *res*, including determining the view in which the Query was done, which member sent each of the results, and which ones failed without replying. This model maximizes asynchrony, yet preserves control.

7 Multi-group Semantics

Our multi-group approach extends a group by attaching a subgroup generator to it, which includes a function that maps the view to the desired subgroup membership. A sharded subgroup generator takes a function from the view to the number of shards, and a second function that, for shard *k*, returns a vector of shard members. Here is an example of a (non-sharded) subgroup generator:

```
g.generateSubgroup<LoadBalancer>
  ([](View, ParentGroupState) ->
   std::bitvec {return {true,true,true,
   false... }; });
```

Here we see the sharded version, with its two lambdas:

```
g.generateShardedSubgroup<CacheLayer>
  ([](View, ParentGroupState){ return 250; },
   [](View, ParentGroupState, shardNumber) ->
     std::bitvec {return {true/false,...};});
```

Thus with reference to Figure 1:

1. Each layer shown is a subgroup, or sharded subgroup, of the parent group.
2. For the load-balancer layer, the membership function shown above returns a boolean vector selecting the first three members of the current view for the load balancer role. In practice, it would be unusual to just take the first three members this way. A more typical selector would pick members based on the view and other criteria (including application-specific data replicated in the top-level group).
3. For the sharded layers, such as the cache, the application decides how many shards to use, 250 in this example. The second lambda will be called with a shardNumber ranging from 0..249, and for each shard, returns a vector designating the membership.

We anticipate providing pre-designed subgroup membership selectors for common cases, as well as prebuilt subgroup generators. For example, the simplest way of sharding a group is to just assign the member with rank k to shard k *mod* s, where s is the number of shards. But if we were to use this naïve approach, failure of a low-ranked member will shift the rank of any higher ranked member down, and in the next view almost every shard would change membership! One can avoid such issues using more sophisticated sharding policies, which can also support shards of varying sizes (e.g. to handle hot spots), etc.

Given such an approach, the entire service of Figure 1 is largely solved. We emphasize the point simply because it is surprising to realize that what would traditionally require tens or hundreds of thousands of lines of code to implement can be described with just tens of lines, and even more so that the solution will embody strong consistency and fault tolerance. Further, because Derecho itself is so fast, it is likely to outperform a typical hand-created version.

Beyond the Derecho wrapper, what customization logic would be required? For "one-shot" transactions (singleton reads or writes), our service might work as follows. A request arrives from the external client, and because the load-balancer is a member of the top-level group, it knows the shard mapping, and can load-balance over the shard members. The request handler in the cache layer need only acquire a read lock, read the data, and then release the lock. Write requests would be directed to the back-end, where the algorithm would first invalidate the cache entries by doing a multiquery within the appropriate group (the purple ovals in the figure), wait until all read locks have been released, and then perform the update itself. The update is just a Paxos OrderedSend in the back-end (perhaps followed by a multicast in a cache-notification group to push the new value up to the cache layer). We could then extend this to the case of multi-operation transactions with a final commit, snapshot isolation, or other more sophisticated behaviors.

8 A Step-By-Step Walk Through

To illustrate how this service would behave, we can walk through a typical run. Some aspects are actually implemented in Derecho itself, but are included for completeness.

1) After a data center shutdown, the service is relaunched on 1000 nodes, including six that

previously held the backing store. One node is started first, and only later are the others started.

2) The first process to restart recreates the top-level application group, which will initially have just one member. It recovers the view sequence used in its last run, needed in step 4a.

   The owner designed the service to run at larger sizes, so no other action is taken: the service isn't yet ready to activate. Subgroup selectors are called via the new view event handler, but return a vector of boolean **false**, of the same length as the top-level view.

3) More processes start. Each discovers the active service instance and sends it a join request (via standard RPC). This causes a series of new epochs, in which batches of as few as 1 and as many as 999 are added to each new view. State transfers in the top-level group send an initial top-level view to each joining process, but again, all the selectors return false and no subgroups are formed.

4) As the top-level count reaches 1000, behavior changes. Now the selector methods return patterns that populate the load-balancer, cache and back-end store layers. (a) The back-end store is restarting from a total shutdown, so for each of its shards (two are shown in Figure 1) Derecho inspects the SSD logs of the last active view of the back-end store, which is persistently tracked as part of the group state of the top-level view (see step 2). (b) Logs are trimmed to include only on intact, completely logged messages, discarding partially writing suffixes. The longest of these logs can safely be used as the initial state for the back-end shard (see the Derecho protocols paper for a detailed discussion of this point, which is tied to the precise way that Derecho terminates an epoch in persistent mode). (c) Some of the restarting shard members may have short logs: ones that crashed in the prior run before receiving those messages. Accordingly, point-to-point sends are used to transfer any missing suffix to a shard member with a shorter log. Now the back-end shard becomes active and is ready for new transactions, or read-requests triggered by cache misses. (d) The load balancer, cache and event notification subgroups are all stateless and hence are initialized using constructors. (e) The load balancer exposes IP endpoints and registers them with the DNS.

5) Incoming requests will be vectored using Derecho point-to-point and RPC messaging, directed to the cache, or back-end, as per Section 2. Because steps 4a-4c were concurrent, a request could be forwarded by the load balancer before the target is finished initializing. However, the rule of Section 3 ensures that if so, the delivery of that request would delay until startup finishes (the restart actions required occur in the new view event handles, hence the new view event doesn't end until recovery is finished).

Notice that very little application logic enters into any of the startup steps, or even into the normal routing of requests once the service is operational. Nonetheless, there is additional application-specific decision making to be done. This relates to the longer term life cycle of the service, and in particular, its behavior when degraded by failures.

The simple case involves faults that leave all shards populated but simply reduce the shard sizes. For the stateless roles, one option is to hold some nodes in reserve as spares. For example, we could use 995 of our thousand nodes for the running service, but hold back 5 as hot standby backups. If one is needed, state transfer would initialize it. In fact for our stateless layers this can work even if the impacted shard became totally depopulated: in that case, it would restart with an empty initial state.

But crashes that depopulate shards in the back-end will require more thought. Here, loss of a whole shard might require temporarily shutting down aspects of the service touching data on that shard: otherwise, we lack required strongly consistent state. Because the top-level view is known to all members of the service, the needed rule is easily implemented in the load balancer.

9 Would Such a Service Perform Well?

We emphasized earlier that Derecho achieves remarkable speedups compared to prior group communication systems, but a hand-coded design might lack the enclosing top-level group seen in Figure 1. Could the top-level group become so large as to result in high overheads?

The answer is that it will not be an issue. First, notice that in our service, the real "work" occurs in small groups of just a few members, via point-to-point sends and RPC operations and subgroup multicast. Derecho at that scale can run at the full line rate: making 4 or 8 replicas costs almost no more than just 1, and the system sustains throughputs as high as 115 Gb/s with single-event latencies as low as 2µs on 100Gb/s RDMA [8]. For a service like the one in a Figure 1, this would be an impressive level of speed, far higher than in any prior system of which we are aware. At the backend,

update rates will be limited by the speed of the persistent storage substrate. On our cluster, the Derecho Paxos is about twice the speed of Corfu and about 4x faster than the Azure fabric storage solution in its Paxos mode. Further, while those prior solutions used protocols that slow down sharply with scale, the Derecho Paxos protocol has flat performance, with no slowdown evident even in groups of 16 or more members.

What about communication loads in the top-level group itself? The Derecho protocols scale to large groups without much loss of performance (in one experiment on the LLNL Sierra cluster with 20Gbps Infiniband (technically, 40Gbps but using a NIC limited to 20Gbps per socket), Derecho ran at 18.5Gbps in 4-node groups. At 64 nodes, only a 25% slowdown was seen, and with 256, multicasts were still running at 6Gbps), hence the top-level group has capacity for a great deal of traffic. But in this design, the top-level group would see little traffic, mostly pertaining to small "metadata" updates: membership tracking, updates to parameters used to carry out the sharding policy, etc. Even configuration changes need not involve the top-level group; for example, updates to load-balancing parameters only require multicasts in the load-balancing group.

Derecho recomputes subgroup and shard membership when the top-level group view is updated (on a join, leave, failure, or if the application requests a new view to trigger a reevaluation of membership). This process completes in 100-150ms. With huge numbers of members, the frequency of such events might become an issue, but Facebook's blob store [1] and its Tao systems [33], including caches, are described as having just a few thousand members per datacenter. When Google first described Chubby [6] the service was purely a back-end one with just 5 members or so, but since then Chubby has added a caching layer similar to the one we've described. Thus at the normal scale of today's important cloud platforms, our server should be highly competitive.

It is interesting to realize that even with this very compact API, Derecho is doing a great deal of work on behalf of the application. In summary:

- Derecho is managing the entire multi-group structure, using just a few lines of definition and a few pre-constructed subgroup membership selection functions.
- Derecho initiates state transfers to new members, automatically serializing the state of some active member and using it to initialize the new member.
- After a complete shutdown of a group, Derecho will reinitialize it, either to the ground state or by loading state automatically from the appropriate Paxos log (this involves loading a checkpoint and then replaying subsequent updates).
- Derecho implements the multicasts needed to replicate state in a consistent manner, moving data with a zero-copy protocol that runs out of band from the Derecho control logic.
- The system manages membership in a consistent manner, enabling safe system-wide use of the top-level membership and subgroup membership in user code.
- Views are consistent and reported to all group members, which can use the data in them and the ranking on members to subdivide work or take on distinct roles.
- Failure handling is automatic, and causes a graceful cleanup and finalization of any multicasts in progress (those that need to be reissued are automatically restarted in the next view, preserving the sender ordering).

10 Insights Gained Developing Derecho

We conclude with a review of insights gained designing the new Derecho API.

10.1 Runtime polymorphism via static analysis

> "The victorious general makes many calculations long before the battle is fought."
> **— Sun Tzu**

The Derecho API looks polymorphic, but this is actually an illusion. Polymorphic APIs are convenient, but polymorphism is expensive at runtime in languages like Java and C#. To offer polymorphism cheaply we use a C++ 17 form of compile-time reflection called variadic templating, creating a form of stub that handles all the polymorphic aspects of the call statically. At runtime, the stub already knows how many bytes will be sent, how much memory will be required, and how it will be laid out; it simply stores arguments into the memory region (a step that involves copying) and transmits. Derecho will then unpack the incoming message on the remote side, passing the handler pointers right into the message object. Marshalling is avoided if a caller specifies a single byte-vector argument. For this case, we transmit data unmarshalled, and deliver it exactly as received, with zero copying.

All type checking occurs at compile time. The Derecho Join protocol "knows" the full set of APIs and can

confirm that the running group is using the same version of the API as the joining member. Static analysis also lets us determine the sizes of marshalled objects: Derecho can preallocate and pin memory before the memory is needed, avoiding last-microsecond delays.

10.2 A control-plane / data-plane separation demands asynchrony

> "E la nave va." ("And the ship sailed on.")
> **— Frederico Fellini**

The key to performance is involves ensuring that the steady transmission of data is never disrupted by control actions such as multicast ordering decisions. Our API treats application code as a control plane, allowing the application to asynchronously stream data and to asynchronously processing incoming messages. An application with a heavy, steady data flow (such as an online Internet of Things system, or a television server) can thus continuously move bytes while tracking the status of the data flow in near synchrony, learning of events a few microseconds after the bytes actually arrive. In this way the RDMA data plane is kept continuously active. Disruptions to the data flow only occur on membership changes, or if the application introduces additional locking.

10.3 Make (but don't rigidly keep) promises

> "Among the fields of gold, I never made promises lightly, and there are some that I've broken."
> **— Sting**

Asynchronous executions confront a peculiar tension. Many actions need information that, in an asynchronous run, won't become available until later. If one dives deep into past implementations of multicast and Paxos libraries, these are common sources of synchronization barriers that can limit performance. For example, in the Vsync library, it is not unusual for a thread to be created just to wait for information such as the actual membership view in which a multicast was sent (for example, this information is needed to construct the reply iterator used in a query).

In Derecho, we represent such information as a *future* or *promise*: a pointer to an object that will be filled in later from a data structure that is known to be correct; "*As soon as it is known, you can find out from whom to expect replies. Iterate to access those replies."* Only failures trigger exceptions in which a promise might not be fulfilled.

A query that will iterate over results is actually just one of many cases where futures arise within Derecho, and we believe this will also be the case for applications layered over the Derecho platform. For example, a multicast sender supplies a future that the Derecho platform calls each time it requires the next message from that source. If the sender is sending continuously, it always has a next message; if not, it sends a *null message*. By delaying the decision until the message is actually needed by the transmission layer, we maximize the likelihood that if data is available to send, we won't needlessly force the sender to wait until its next turn.

10.4 Monotonicity facilitates reasoning about correctness

> "To know what you know and what you do not know, that is true knowledge."
> **— Confucius**

Derecho encourages developers to leverage knowledge by designing algorithms that steadily gain increased knowledge as they execute.

Monotonic safety, and more broadly, the design of systems that employ monotonicity as a design principle, pervades our thinking in Derecho itself, and we believe that applications will similarly rely upon this style of computing. Monotonicity does force the developer to select a manner of expressing code that will lend itself to a monotonic implementation, but the benefit is that once expressed in this manner, reasoning about the correctness of a solution is reduced to step-by-step proofs of safety, as mentioned above.

Earlier we illustrated the idea of monotonicity using examples such as counters that steadily increase within an epoch, resetting only when a new epoch starts. This is just one of many forms of monotonic information. The membership views of the groups used in an application are monotonic and consistent across all members, enabling sophisticated behaviors that use membership as an input, enabling the load balancer of Figure 1 to behave consistently with respect to shard mappings. When data is sent asynchronously, Derecho preserves sender ordering and also ensures that the receiver is in a knowledge state at least as complete as that of the sender. If the sender has reacted to a failure, the receiver will also know about it: monotonicity!

Monotonic designs leverage a powerful theory: the *logic of knowledge*, in which one reasons in terms of information built up incrementally over time, as it runs. There is a natural fit between the intuition used when creating a distributed system and this notion of monotonic knowledge: a system runs, and the processes within it learn more and more about one-another's past states. While it is impossible for asynchronous

processes to have instantaneous shared knowledge of each-other's states, a monotonic perspective in which reasoning and decisions (such as the order in which to deliver a message, or the point at which it is safe to deliver a message) are based on this evolving frontier of information that is stable and will not change offers a natural basis for safe protocol and application design. Indeed, arguably, monotonic knowledge is the only form of knowledge with this property, and hence protocols such as the Paxos protocols, or the virtual synchrony ones, were *always* fundamentally concerned with monotonic safety, whether previously expressed this way or not. By making that linkage explicit, we've ended up with simpler protocols, and by encouraging applications to follow our example, those will be easier to conceive, implement, and prove correct.

10.5 A group is a distributed module

> "I've looked at clouds from both sides now. From up and down, and still somehow its cloud illusions I recall. I really don't know clouds at all."
> **— Joni Mitchell**

The modern cloud treats cloud-hosted services as modules, in the sense that the external client accesses them using load-balanced RESTful RPC (or a similar method) and is oblivious to implementation details such as how many instances are running, where they run, or how they coordinate their actions.

Derecho treat the group as a modularity construct even relative to other components of the same application: each group has a visible membership, but a hidden internal state and algorithm. The broad principle is that the group implementation owns its state and carries out needed coordination. Non-members can talk to members, but cannot directly access or mutate state.

This policy has non-trivial implications:

1. All group members must implement the identical functionality. Derecho checks this, to the extent feasible, by verifying that a joining member implements the same type signatures and software revision level as other members.
2. Only a group member can initiate group multicasts or multicast-queries. For a non-member to talk to a group, it must use a P2P request to some member, which will then need to relay the request on behalf of the real caller.
3. Group *membership* information is shared, hence point to point communication via P2PSend and P2PQuery is straightforward. For consistency, if a message was sent in view K, it will be delivered in view K' ≥ K.

10.6 Reduce complex join/leave/failure scenarios to sequences of safe steps

> "Success consists of going from failure to failure without loss of enthusiasm."
> **— Sir Winston Churchill**

In Derecho, we view failures much like other kinds of state-mutating events, using groups as a barrier within which the module can contain failures, replicate state to ensure that needed information will still be available, compensate, and restore functionality. By treating failures as events, we reduce complex event sequences to a series of new-view upcalls. Each successive event can be handled one by one, and the proof obligation is reduced to that of showing a successive of single-step safety properties. In our experience, distributed applications are easier to design if the program only needs to handle one event at a time. Complex situations do arise, yet the developer isn't forced to reason about multi-event execution cases.

10.7 Trust members to detect failures

> "Ask not for whom the bell tolls. It tolls for thee".
> **— John Donne**

Our failure model is trusting: components of the system sense and report failures, and we immediately act on those reports. In some sense, just as a process can leave voluntarily, it can also be asked to leave by any other component that has reason to suspect an issue. Of course we wouldn't be able to sustain a high rate of mistaken detections, but in a well-tested application, such determinations won't often be wrong. To avoid risk of partitioning (split-brain behavior), the top-level group must retain a majority of its most recent membership to switch to a new epoch, and otherwise quickly shuts down.

10.8 One group can host another

> "Any problem in computer science can be solved with another level of indirection."
> **— David Wheeler**

Derecho seems to be the first system to offer its full semantics through a subgroup API. While prior systems had subgroup features, they offered very reduced subsets of the full-group functionality. Although our choice is aesthetically appealing, and responds to a style of application development needed in cloud environments, Derecho's monotonic design made it easy to offer this layering: all that was needed

was a form of indirection over the basic Derecho logic (since the protocols are presented elsewhere, the reader of this paper should either trust us, or refer to [8] for details). The deeper insight is that monotonic protocols lend themselves to stepwise refinements in which one layer extends a preexisting one by adding additional logic while preserving the safety and liveness properties of the original code.

11 Geo-Distributed Services

> "Don't try to do everything. Do one thing well."
> **— Steve Jobs**

Modern cloud computing systems run in wide area deployments and hence the issue arises of whether direct support for georeplication to Derecho would break the relatively elegant and simple datacenter model we've described here. This is really a topic for future work, but we do want to explain why it seems feasible to add such a capability without harm to the system.

RDMA datacenter systems benefit from uniformly low latency and zero-copy RDMA, whereas enterprise WAN systems have high latencies and run over specially-tuned versions of TCP. Further, rather than allowing casual end-point to end-point communication (which can create a high level of redundant WAN traffic), it is important to deduplicate and funnel WAN traffic through some form of gateways. At WAN scale cryptographic data protection is increasingly necessary, whereas cryptography can be avoided in zero-copy RDMA settings that run over trusted hardware. Thus, we believe that for the foreseeable future wide-area services will need to operate hierarchically: by defining data center service instances and then interconnecting them via wide area network links, with those links used only by the gateway service representatives. Google's Spanner [32] is a state-of-the-art example of the model we have in mind.

One can easily imagine a new form of WAN structure that could be introduced as a geo-scale top-level group construct in Derecho, but differing in its capabilities from the current top-level group. This layer would add WAN-specific functionality, but could also remove some unnecessary or inappropriate existing functionality. For example, there is no reason to track fine-grained membership of a service running on the far side of a WAN link: all that matters is to know the TCP address of the gateway processes.

In our future work, we are motivated by scenarios that might extend Figure 1 to also support geo-scale caching: rather than caching San Francisco data only in a California datacenter, the cache layer in Spain, or Kenya might also cache read-only versions of data which is primarily hosted elsewhere, either holding true read locks or with some form of timed leases (for the latter, Spanner's concept of *true-time* would be an example of functionality that a geoscale top-level group could support). It would also be interesting to explore ways of dynamically migrating the hosting role across geo-distributed datacenters.

12 Other Relevant Prior Work

Theoretical work on monotonicity, notions of knowledge in distributed systems, and the power of reasoning about *stable* properties is a rich area explored by many prior efforts. Well known examples include the knowledge logic of Moses and Halpern[21] and the stable predicate detection research done by Marzullo and Neiger[11] and Marzullo and Sabel [35].

The DHT model emerged when researchers extended the MemCacheD [24] API for use in distributed services. Widely known examples include CAN [31], Chord [5], Pastry [27], Dynamo [12]; all replicate (key,value) tuples within some form of shard to which the key maps. Derecho could standardize the communication infrastructures of such systems.

Particularly relevant is the FaRM DHT [14], which builds a DSM that employs RDMA for high performance and implements what one can think of as a distributed transactional memory model. Our Derecho protocols equal FaRMs performance for unicast, and introduce multicast, which FaRM lacks. Unlike FaRM, however, Derecho also exposes application structure in useful ways: every member knows the membership of every other group and shard, state transfer is automated, and data can be replicated where desired. This enables the developer to build a scalable storage structure specialized to a particular use case, arranging that computational tasks will have needed data directly in local storage. In contrast, FaRM only has a single sharded structure, and runs some risk of unwanted copying because data is stored at the location selected by the DHT hash function, which might potentially select an inconvenient DHT node. For small objects this wouldn't matter, but if the data is very large, the extra copying is a potential issue.

The implementations differ, too. FaRM is optimized for transactional key-value operations, implemented using a novel lock-free approach that guarantees atomicity even if a request updates or reads many keys. Derecho does have totally ordered and persistent updates, but it lacks built-in transactions that would

require multiple multicasts, because we perceive transaction mechanisms as a functionality that only some users would want. The deeper distinction is that we view Derecho as a library for structuring applications into groups and replicating data, while FaRM is intended as a scalable DSM for applications that operate more autonomously, collaborating through FaRM as a shared intermediary.

Publish-subscribe was first proposed in the 1980's, and early implementations mapped publish-subscribe communication patterns to group multicast: V [26], the Isis Toolkit [29], and TIBCO Rendezvous [19]. Unfortunately, those early implementations scaled poorly, and it was ultimately determined that they overloaded IP-multicast routers and create instabilities [9]. Modern pub-sub technologies like Kafka and OpenSplice run primarily on unicast TCP. By implementing our own scalable multicast layer over unicast RDMA, we avoid these pitfalls, and because Derecho has efficient support for subgroups, we offer functionality not seen in prior systems. Pub-sub would be a natural functionality to layer over our API.

13 Conclusions

Derecho is a library for RDMA group communications, offering exceptionally high performance through a control/data separation. The goal is to extend today's cloud computing model by making it much easier to build cooperative distributed services with fault tolerance and strong consistency. Complex, large-scale, high-performance application can be created with a few lines of code.

Derecho complements today's prevailing cloud programming model, which favors stateless applications run on behalf of a single external client (a mobile device, a web browser, etc), which store data in a shared persistent structure such as a key-value store or backend database. Growing availability of RDMA, including the new RDMA-on-Ethernet option (RoCE) should enable Derecho's use in a growing range of settings (a software emulation of RDMA called SoftRoCE would permit use of Derecho in non-RDMA settings, albeit with heavy performance impacts). The project is open-source.

Here, we focused on features that automate generation and management of multi-group structures. Our work is also unusual for mapping all communication to zero-copy RDMA and for promoting monotonic reasoning as a design tool: this allows us to promote asynchrony while also supporting a style of reasoning in which one thinks about behavior and correctness in a rigorous step-by-step manner.

Our open-source code-base is hosted at Derecho.codeplex.com.

Acknowledgements

Our work has benefitted from dialog and comments by many colleagues, but we are particularly grateful to Chris Hobbs, Dave Cohen, Matei Zaharia, Miguel Castro, Ant Rowstron and Kurt Rosenfeld for sharing ideas and suggestions. Our work was supported, in part, by grants from the NSF, DOE/ARPA, DARPA, AFOSR and AFRL, and also by computing resources provided by Mellanox, Amazon, Microsoft, LLNL and the U. Texas Stampede supercomputing center.

References

[1] An Analysis of Facebook Photo Caching. Q Huang, K Birman, R van Renesse, W Lloyd, S Kumar, H C. Li. SOSP 2013, Nemacolin Woodlands Resort, PA, USA. November, 2013

[2] Azure Service Fabric - Reliable Services overview. Microsoft Corporation, 2016

[3] https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-services-introduction/

[4] Building Adaptive Systems Using Ensemble. R van Renesse, K Birman, M Hayden, A Vaysburd, and D Karr. Software--Practice and Experience. Vol28, No. 9, pp, 963—979, August 1999.

[5] Chord: A scalable peer-to-peer lookup service for internet applications. I Stoica, R Morris, D Karger, M.F. Kaashoek, and H Balakrishnan. SIGCOMM '01, 149-160.

[6] The Chubby Lock Service for Loosely-Coupled Distributed Systems. Michael Burrows, 7th Usenix OSDI, 2006.

[7] CORFU: A Distributed Shared Log. M Balakrishnan, D Malkhi, J D. Davis, V Prabhakaran, M Wei, and T Wobber. ACM Trans. Comput. Syst. 31, 4, Article 10 (December 2013),

[8] Derecho: Group Communication over RDMA. Submitted to NSDI 2017.

[9] Dr. Multicast: Rx for Data Center Communication Scalability. Y Vigfusson, H Abu-Libdeh, M Balakrishnan, K Birman, R Burgess, G Chockler, H Li, and Y Tock. Eurosys 2010, 349-362.

[10] A Dynamic Light-Weight Group Service. L Rodrigues, K Guo, P Verissimo, and K Birman. Journal of Parallel and Distributed Computing, 60, 1449-1479 (2000).

[11] Detection of global state predicates. K Marzullo and G Neiger, 1991. Proc. 5th Int'l. Workshop on Distributed Algorithms (WDAG '91), pp.254-272.

[12] Dynamo: Amazon's Highly Available Key-Value Store. G DeCandia, D Hastorun, M Jampani, G Kakulapati, A Lakshman, A Pilchin, S Sivasubramanian, P Vosshall, and W Vogels. 2007.. 21st ACM SOSP, 2007. 205-220.

[13] Fabric: a platform for secure distributed computation and storage. J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, New York, NY, USA, 321-334.

[14] Farm: Fast Remote Memory. A Dragojević, D Narayanan, O Hodson, and M Castro. 2014. NSDI 2014, 401-414.

[15] Frenetic: a network programming language. N. Foster, M. J. Freedman, R Harrison, J Rexford, M L. Meola, and D Walker. ACM SIGPLAN Notices - ICFP '11: Volume 46 Issue 9, September 2011

[16] Guide to Reliable Distributed Systems. Building High-Assurance Applications and Cloud-Hosted Services. (Texts in Comp. Sci.). K. Birman. Springer, 2012.

[17] Horus: A Flexible Group Communications System. R. van Renesse, S. Maffeis, and K. Birman. Communications of the ACM. 39(4):76-83. Apr 1996.

[18] Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. Fred B. Schneider. ACM Comput. Surv. 22, 4 (December 1990), 299-319.

[19] The Information Bus: an architecture for extensible distributed systems. B Oki, M Pfluegl, A Siegel, and D Skeen. 1993. ACM SOSP 1993, 58-69.

[20] JGroups. B. Ban. https://github.com/belaban/JGroups

[21] Knowledge and common knowledge in a distributed environment. J. Halpern and Y. Moses. Journal of the ACM 37(3): 549–587 (Aug. 1990).

[22] LibPaxos. M Primi, D Sciascia, F. Pedone. http://libpaxos.sourceforge.net/

[23] Lightweight process groups in the Isis system. B. Glade, K. Birman, R C.B. Cooper and R van Renesse. Distributed Systems Engineering Journal. July 1993.

[24] MemCacheD. B Fitzpatrick (originally developed for LiveJournal), 2003. https://memcached.org/about

[25] MPI: A Message-Passing Interface Standard. I Foster and the Message P Forum. Standards definition, University of Tennessee, Knoxville, TN, USA., 1994.

[26] One-to-many Interprocess Communication in the V-system. D R. Cheriton and W Zwaenpoel. ACM SIGCOMM 1984.

[27] Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Antony I. T. Rowstron and Peter Druschel. Middleware 2001, 329-350.

[28] The Part-Time Parliament. L. Lamport. ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.

[29] Providing high availability using lazy replication. R Ladin, B Liskov, L Shrira, and S Ghemawat. ACM Trans. Comput. Syst. 10, 4 (November 1992), 360-391.

[30] Reliable communication in the presence of failures. K. Birman, T. Joseph. ACM TOCS 5, 1 (Jan 1987), 47-76.

[31] A Scalable Content-Addressable Network. S Ratnasamy, P Francis, M Handley, R Karp, and S Shenker. SIGCOMM '01, 161-172.

[32] Spanner: Google's Globally-Distributed Database. J C. Corbett, et. al. OSDI 2012.

[33] TAO: Facebook’s Distributed Data Store for the Social Graph. N Bronson, Z Amsden, G Cabrera, P Chakka, P Dimov, H Ding, J Ferris, A Giardullo, S Kulkarni, H Li, M Marchukov, D Petrov, L Puzar, YJ Song, and V Venkataramani, Facebook, Inc. 23rd USENIX, June 2013.

[34] The Totem multiple-ring ordering and topology maintenance protocol. D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. ACM TOCS 16, 2 (May 1998), 93-132.

[35] Using Consistent Subcuts for Detecting Stable Properties. K Marzullo and L Sable. Proc. 5th Int'l. Workshop on Distributed Algorithms (WDAG '91).

[36] Virtually Synchronous Paxos. D. Malkhi and L. Lamport, invited talk, A 30-Year Perspective on Replication, which was held at Monte Verità, Ascona, Switzerland, in November 2007.

[37] The Vsync Cloud Computing Library. vsync.codeplex.com.

[38] ZooKeeper. Benjamin Reed and Flavio Junqueira, O’Reilly Media, 2013.