Invited Paper: Monotonicity and Opportunistically-Batched Actions in Derecho

Ken Birman[1](✉), Sagar Jha[1], Mae Milano[2], Lorenzo Rosa[1,3], Weijia Song[1], and Edward Tremel[4]

[1] Cornell University, Ithaca, USA
`ken@cs.cornell.edu`
[2] UC Berkeley, Berkeley, USA
[3] University of Bologna, Bologna, Italy
[4] Augusta University, Augusta, USA

Abstract. Our work centers on a programming style in which a system separates data movement from control-data exchange, streaming the former over hardware-implemented reliable channels, while using a new form of distributed shared memory to manage the latter. Protocol decisions and control actions are expressed as *monotonic predicates* over the control data guarding protocol actions. Provable invariants about the protocol are expressed as *effectively-common knowledge*, which can be derived from the monotonic predicates in effect during a particular membership epoch. The methodology enables a natural style of code that is easy to reason about, and it runs efficiently on modern hardware. We used this approach to create Derecho, an optimal Paxos-based data replication library that sets performance records, and we believe it is broadly applicable to the construction of reliable distributed systems on high-bandwidth networks.

1 The Design of RDMA-Friendly Protocols

We are interested in distributed systems in which data transfers are streamed asynchronously by a layer independent of the one used for coordination, and in which peers asynchronously exchange control data. The approach makes it possible for the control layer to be implemented using *monotonic deduction*. We start by sketching the overall approach, after which subsections discuss the framework in greater detail.

In any setting, high performance requires developers to match their protocols to the hardware. The hardware of greatest interest for our work is a type of network that offers remote direct memory access (RDMA), a technology with which a process can reliably read or write the memory of another process asynchronously and without locking (the underlying mechanism involves message-passing between the RDMA network interface cards). RDMA is far faster than TCP/IP, achieving data rates of 100–200 Gbps and latencies as low as 0.75 μs.

Despite its use of RDMA networking hardware, our target environment can still be modeled in a traditional way. It supports concurrently active processes, interconnected by asynchronous networks and experiencing infrequent crash (halting) failures. Network failures do not partition the data center: a severe disruption will either shut down a group of machines or the entire data center.

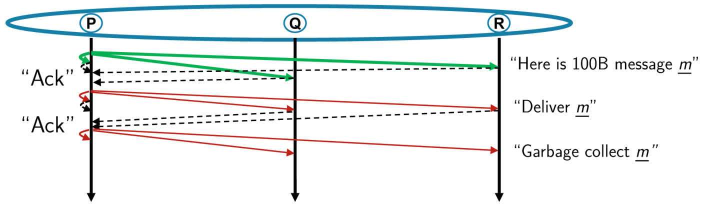


Fig. 1. An event triggers a 2PC that sends data (green) in its first phase, then waits for acknowledgments (dashed) before sending commit messages (red). Efficiency is low: the full run of the protocol does not end until after all the replies are collected and the commit has been successfully sent. On a 100 Gbps RDMA network, each small message takes 0.75 µs to arrive, which means the entire interaction takes 4.5 µs to deliver 100B of data. This uses only 0.02% of the available bandwidth. (Color figure online)

It is natural to wonder whether protocols such as 2-phase commit, atomic multicast, leader election, and replicated logging can take full advantage of RDMA. A first finding is that RDMA is not simply a faster replacement for TCP/IP: Although RDMA can mimic TCP/IP [15], higher-level protocols that treat RDMA as if it was TCP/IP gain little speedup. The central issue is latency: RDMA bandwidth can be 10x-20x better than that of TCP, yet its one-way latencies are not very different, causing a bottleneck (Fig. 1).

What sorts of protocol-engineering steps are needed to fully leverage ultra-fast networking? A system can load-balance updates, allowing them to be initiated by all members of the application. Each member could transmit a sequence or stream of actions, which potentially enables batching multiple operations per message. It may be feasible to have multiple threads per member, and hence transmit multiple data streams per member. Peers can send in a one-to-all manner, enabling decentralized decision-making. As illustrated in Fig. 2, which illustrates a system streaming atomic multicasts, such steps are indeed helpful. Yet (and this is also shown in the figure) data flow is likely to remain bursty. There are limits on how many threads we can have, or how evenly we can spread the workload. Batching is a wonderful idea, but it delays messages, and sooner or later, a partial batch may need to be sent. The network remains lightly used and throughput is limited.

The limiting factor turns out to be the disproportionately high delay noted above: RDMA delay is low compared with a protocol such as IP, and yet can seem high when we consider how much data could have been transmitted during one RDMA round-trip time. Pausing data transmission to exchange control information, even for a single RTT, will substantially reduce throughput. A second issue involves the unpredictable scheduling of endpoint applications: if a new message arrives but the receiver cannot process it immediately, the sender will be left waiting.

Lacking a mechanism to reduce the impact of delay, protocol instances will wait for acknowledgments or commit messages. This will trigger a rapid accumulation of protocol state until resource exhaustion forces the entire system to pause. As a result, applications will be observed to alternate between streaming data and pausing to finalize previously-initiated protocols.

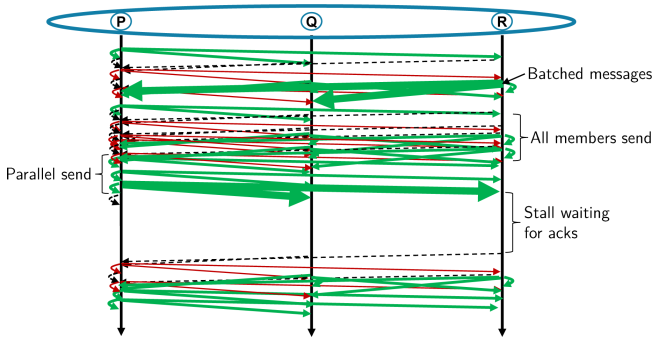


Fig. 2. Techniques such as parallel streaming and batching can improve network utilization, yet there is still idle time in the run due to round-trip delays that stall senders.

Our project first encountered this set of challenges when we created Derecho, a platform intended as a supporting framework for a new generation of cloud computing applications [12]. Derecho centers on data replication supported by a self-managed (virtually synchronous) version of Paxos. We decided to build entirely new protocols from the ground up. Our first step was to separate the data transportation layer ("data plane") of the system from the one handling control data ("control plane"), as seen in Fig. 3, giving each an independent communication channel. Now we can continuously stream, improving concurrency: members send data messages as updates are initiated. On the receiving side, these incoming messages trigger state updates, which are reflected in continuously exchanged streams of control data.

Consider a situation in which a receiver participating in some protocol experiences some form of delay. Data piles up briefly, but then the delay ends and streaming resumes. It will be common for a sequence of pending data messages or control events to become available as a group (for example, a series of data messages may have all become deliverable). We use the term *opportunistically batched* to refer to an action that can be applied to the whole group of messages rather one by one, hopefully enabling the receiver to quickly catch up and amortizing overheads. Opportunistic batching contrasts with sender-side batching, in which senders deliberately delay messages to group them into fixed-sized

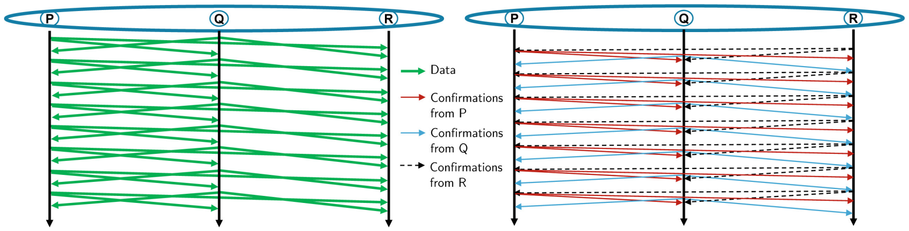


Fig. 3. By separating data plane (left) from control (right), we can design protocols that stream data in a continuous and reliable manner, achieving much higher efficiencies.

batches, delaying messages even when doing so is unnecessary. At RDMA speeds, opportunistic batching is an unqualified improvement: it copes with inevitable delays yet doesn't introduce any of its own. Moreover, the technique turns out to be especially suited to the one-sided RDMA write hardware we use, in which the receiver asynchronously discovers that some section of its memory has changed, rather than being explicitly notified each time a new message arrives.

We now need to address two questions. One centers on the best way to stream data messages reliably while preserving sender order. We describe our work on this problem in [4,13], and will not repeat that material here.

The second question involves the streaming of control data, and is the main focus of this paper. Derecho innovates by reexpressing the concept of a stream of control messages. Rather than viewing such a stream as a series of small messages, we focus on the actual values, and ask whether a stream of control-variable updates can be transmitted using the form of shared memory enabled by RDMA, in which one process is granted permission to asynchronously write into some memory region in a second process.

Our central idea was to focus on *monotonic* control data: given a variable such as a messages-received counter, which only increases, new values can overwrite older ones. For example, suppose that process a sends message x with id 17 and then message y with id 18. Process b receives and acknowledges x by writing 17 into a location in the memory of a using a one-sided RDMA write. Now y arrives, and b overwrites the acknowledgment variable. If a observes 17 first and reacts to this control event, then observes 18 and reacts to it, it is as if we sent two acknowledgment messages from b to a. But if a experiences a small delay and sees the counter jump from 16 to 18, it can *infer* that 17 was previously reported. In effect, the observed value (18) covers the range [1 ... 18]. A single RDMA-shared-memory counter has replaced a stream of acknowledgment messages.

Derecho's entire control plane is monotonic, and this even includes sequences of complex objects, such as proposed membership updates: we guard such an object by a counter and adopt a round-robin model in which we can send some number of objects without delaying. This approach let us eliminate the lock-step dependency on round-trip messages carrying acknowledgments and commits. Jointly, such steps enable the dramatic performance improvements described in [12]: the system is able to replicate data and coordinate distributed actions with strong consistency at speeds orders of magnitude faster than widely used datacenter tools such as the Kafka pub/sub message bus, Kafka-Direct (an RDMA version of Kafka [17,21]), and Zookeeper [14].

1.1 Revisiting an Old Model

Our work builds on a self-managed virtual synchrony membership layer. The idea of building systems that manage their own membership was introduced in the 1985–1987 period [5–7]. Whereas classic protocols struggle to deal with failures, virtual synchrony replaces both liveness and failure-tolerance with a subsuming concept of "dynamic membership."

Processes must *join* the system upon starting and cannot exchange messages with members until they are admitted to the current membership (the current *view*). Upon sensing a possible failure, any member can request exclusion of members *suspected* of having failed from the current view. No process will accept or send messages to a suspected peer, and it will promptly be dropped from the view. This establishes an *epoch* model, in which each epoch starts with a new view, performs protocol actions for a period of time, then ends when the membership management system learns of a new join or failure event. The membership service will pause the active message-sending protocols, assist in cleanup to terminate any that were incomplete when paused, then switch to a new view which initiates a new epoch. Should a full shutdown occur, restart is similar: the membership service forms what will become the first view, repairs any persisted state that was damaged by the failure, and then can initiate the first epoch of the new run. Should any of these steps be impossible (for example, if persisted data is inaccessible due to crashes), the system refuses to restart and a human operator would need to repair the problem, for example by loading the missing data from a backup.

By trusting suspicions and immediately excluding the impacted processes from the view, virtual synchrony systems wall off potentially malfunctioning group members. The policy assumes that the rate of mistaken suspicions will be low, but that assumption is valid in today's cloud data centers. Importantly, protocols such as atomic multicast [6] or Paxos replicated logs [18] are simplified by this model because each instance runs in a single epoch. In effect, we separate functionality: a protocol has a normal failure-free logic component and a distinct view-change component used to clean up when a failure disrupts execution, as we detail immediately below. Finally, the protocol has a component responsible for reissuing requests (while preserving the sender message ordering) in the event that cleanup rolled the partially completed messaging protocol back rather than terminating it by rolling forward and delivering messages. The overall structure must still guarantee atomicity and ordering (linearizability [11]), but these needs do not arise all at once.

Derecho, like other virtual synchrony systems (e.g. [1]), uses a leader-based membership protocol, where age-ranking determines the leader succession sequence in case of failure; the leader initiates a view change when it learns of a new join or failure. However, Derecho's membership protocol is specifically designed to use monotonic logic. The new views proposed by leaders form a monotonic sequence, each building on the prior one. Views must be accepted in order, and each proposal must individually be accepted by every non-suspected member. Additionally, the set of healthy (non-suspected) members that accept a proposal must include a majority of members from both the current view and from each proposed new view up to and including the new proposal. If some new membership event occurs while the protocol is running, the leader extends the list of proposals with follow-on proposals. Similar to the split of data plane from control plane, we can think of the sequence of proposals as a data plane, streaming from leader to participants, and the proposal acceptances as a control plane streaming from participants back to the leader.

We do not wish for this sequence of proposals to ever roll back, or for a proposal to be lost at some members and yet to commit at others. With this in mind, when a new leader suspects the older leaders and prepares to take over, it first queries every non-suspected member it knows of. In doing so, its own suspicions are immediately shared, and it learns any suspicions or proposals known to any of those processes. We can then prove that if the system remains live, any proposal witnessed by a majority of a view will eventually be adopted, and that any proposal that could have been adopted will be learned by the new leader. Conversely, if the system loses majority (i.e. observes that a majority of processes are suspected), it will shut itself down.

Why go to such lengths to make this protocol monotonic? The central issue revolves around the unpredictable sequencing of events that system members can experience and the importance of avoiding logical partitioning, in which one system splits into two separate subsystems that each consider themselves to be in charge. Members might advance at different rates, but due to monotonicity they learn of the same views in the same order. Indeed, protocols in which we think of the peers as reasoning and leveraging monotonicity to take actions based on monotonic deductions are a hallmark of virtual synchrony and arise extensively in the Derecho protocols. In some ways this should not be surprising: one could have made a similar remark about the Isis Toolkit and its protocols in 1987 [5]. Others have reached very similar conclusions. Elaborating the CALM methodology for BLOOM (a distributed computing language based on Datalog [2]), a 2013 paper by Ameloot *et. al.* argues that monotonicity (combined with occasional consensus) is complete for distributed computing in a logically-founded deductive style [3].

1.2 Effectively-Common Knowledge

Any developer of a non-trivial distributed system eventually encounters a protocol that is difficult to prove correct. With fault-tolerant systems one issue is the exponentially large state space that must be considered: distributed runs in which at each instant, any message that could be in flight might be received, but also in which any participant might fail. Focusing on our own Derecho protocols, the most challenging aspects to prove correct are associated with runs in which the failures could include the initial leader or even a series of leaders, each of which could have made proposals and perhaps even received adequate quorums to commit some of them. This forces the developer to formalize the concept of a run, to express the “healthy majority in each view” policy rigorously, and then to demonstrate that all of this yields a single perceived sequence of views that will never partition.

The creation of a proof is ultimately an exercise in logical reasoning, and is increasingly supported by proof checking systems (we have direct experience with Ivy [22] and have experimented with Coq [8]). Highly visible proofs include the Dafny proof for IronFleet [10]; Paxos proofs in TLA+ [19] and recent proofs of a number of protocols using the DistAlgo specification language [20,23]. The question now arises: does our monotonic protocol specification align well with formal reasoning?

Not every style of network is conducive to easy correctness proofs. For example, protocols expressed using exchanges of unreliable point-to-point messaging (UDP) are notoriously hard to reason about. UDP does guarantee that a corrupted message will not be delivered, but messages can be lost, delivered out of order, replayed, or arrive after very long delays. The already large state space becomes daunting.

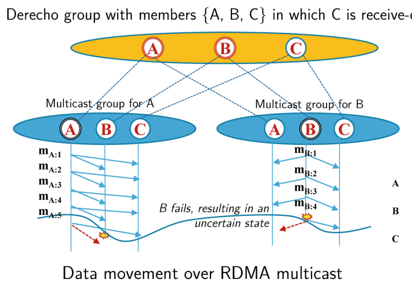
It turns out (and this is one of our main contributions) that a virtually-synchronous monotonic programming style, implemented over RDMA with its hardware-supported reliability features, dramatically simplifies proof tasks. Logicians refer to information that is simply accepted and trusted by all members of a set as *common knowledge* [9], leading us to use the term *effectively-common knowledge* for per-epoch data such as the membership view and other application-specific information that might be piggybacked along with the view.

Common knowledge can be understood as data that all active members of a system possess and trust. Every process knows the information, and also knows that every other process has the identical information. This type of “I know that you know” inference can be iterated to arbitrary degree. Effectively-common knowledge is information tied to the epoch. It introduces facts known to every current member, and that every member can assume that its peers also know, again iterated to arbitrary degree.

An example of effectively-common knowledge employed in Derecho is the message delivery ordering used in an epoch. The ordering rule cannot be anticipated before the epoch begins: it depends on the membership and the anticipated message sending patterns in a group, and neither of these is known *a priori*. By attaching the ordering rule to the view, Derecho can be flexible and adapt this rule as needed, while allowing each member to assume for the duration of the view that every other member knows the same ordering rule. The alternative, used in the classic Paxos protocol, involves a “competition” for each message delivery slot, since members can disagree about delivery ordering. That policy forces the classic Paxos to use just the kind of round-trip message passing we are trying to avoid due to its sensitivity to delay.

Why not just call epoch knowledge “common knowledge?” The issue is that common knowledge cannot be gained while a protocol is running (a main result in [9]). Effectively-common knowledge, in contrast, is easily generated: we simply need to create a new epoch.

Appendix A offers some classic examples of common knowledge, showing how seemingly minor changes to a problem statement can make a protocol "impossible" to implement. Appendix B then goes to highlight the connection to formal verification using automated proof systems.


Derecho group with members {A, B, C} in which C is receive-only
A
B
C
Multicast group for A
A
B
C
Multicast group for B
A
B
C
$m_{A:1}$
$m_{A:2}$
$m_{A:3}$
$m_{A:4}$
$m_{A:5}$
$m_{B:1}$
$m_{B:2}$
$m_{B:3}$
$m_{B:4}$
B fails, resulting in an uncertain state
Data movement over RDMA multicast

SST Contents at A

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

Control is done using knowledge programming on the SST

Fig. 4. The shared state table abstraction offers a convenient representation of monotonic protocol control data. In the example shown, processes a and b are streaming atomic multicasts to a group that also includes c; each has an SST replica and uses it to share control information with its peers (see [12] for details).

2 Deep Dive: Shared State Table

As discussed in Sect. 1, Derecho separates its protocols into a data plane and a control plane. The central abstraction used for representing and exchanging control data is the *shared state table* or SST (Fig. 4). This table is a replicated data structure created afresh for each epoch. Every process in the epoch possesses a copy, within which it owns and can update one row. These updates are then asynchronously written to its peers using one-sided RDMA write operations, which are reliable but lock-free (Fig. 5). The effect is that updates arrive continuously, streaming in an all-to-all pattern.[1] Updates to a row arrive in order, but different peers can see updates to *different* rows in different orders. If we were to pause the updates, all SST replicas would converge, but during normal execution we only do this while switching from epoch to epoch.

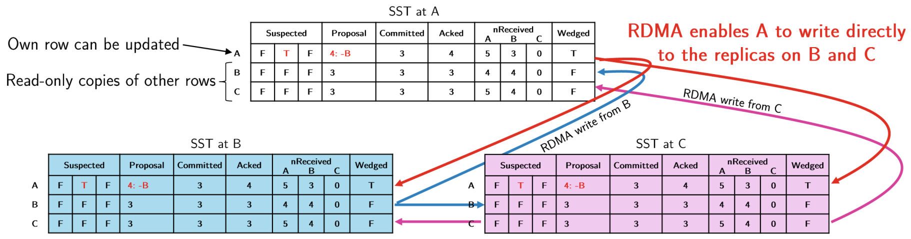
Unnoticed updates are a common phenomenon in Derecho, in part because each process has a single predicate thread. When a process starts up, each protocol registers one or more predicated actions: a tuple consisting of the boolean

[1] All-to-all exchange of control state would scale poorly in many settings, but no issue arises because Derecho is sharded: most activity occurs in tiny subgroups with just 2 or 3 members. We have experimented with far larger subgroups without problems. Future systems deploying Derecho in immense subgroups might need to exchange control data in a different manner, but the underlying principle of asynchronous updates and monotonic deduction of system state would still apply.

function to test and the logic to run if that test evaluates to true. The SST predicate thread then starts up and loops, evaluating predicates one by one like a long list of if statements. By the time a predicate is rechecked, the underlying data may have advanced multiple times, in which case the triggered logic will catch up by processing several events as a batch.

Derecho's use of monotonicity plays into this dynamic. Not only is the underlying data monotonic, but many aggregating operations over monotonic data are as well: obvious examples are *min* and *max*. This leads us to define *predicate monotonicity*: P is a monotonic predicate of some monotonic SST property x if $\forall y > x : \;\; P(y) \Rightarrow P(x)$. It is straightforward to generalize these ideas. Many expressions computed over monotonic inputs have monotonic results, leading to the idea of a monotonic row function: A monotonic expression over monotonic variables in an SST row that can be treated as a higher level abstraction in our protocols. Similarly, we can define monotonic column functions that are evaluated over the rows of an SST. If a set of values must be treated as a unit and updated atomically, but do not fit into a single cache-line (the size at which RDMA writes are atomic), we first update the values, then update some form of guard, such as a "version counter" (which is a monotonic variable). Any participant that sees the updated guard will see the full set of preceding updates, since writes are applied in sender order.

When we set out to create Derecho's virtual synchrony view update protocol, it turned out to remarkably easy to express the algorithm in this manner. Given the epoch mechanism, we then designed simple atomic multicast (very similar to protocol II in vertical Paxos) and durable replicated log update protocols (very similar to classic Paxos in a failure-free run, with the quorum size set to the full current membership of the group, and the read quorum size set to 1). Again, the methodology led us directly to simple, highly efficient solutions.


Own row can be updated
Read-only copies of other rows
RDMA enables A to write directly to the replicas on B and C
RDMA write from B
RDMA write from C

SST at A

	Suspected			Proposal	Committed	Acked	nReceived A	B	C	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

SST at B

	Suspected			Proposal	Committed	Acked	nReceived A	B	C	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

SST at C

	Suspected			Proposal	Committed	Acked	nReceived A	B	C	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

Fig. 5. After updating a row, an SST participant uses RDMA to asynchronously "push" the new data to the remote replicas. These push operations are lock-free and uncoordinated hence the updates are not totally ordered. However, if one process does two push operations with the same target, the updates respect the sender order.

3 Using Logic to Reason About Protocols

A knowledge perspective formalizes a way of describing protocols such as Paxos that most of us use when reasoning about them. For example, consider the first stage of a Paxos protocol [18]. The leader (a) sends a proposed update for a specified slot and ballot number. At the moment of sending it, only a knows the contents. Upon receiving such a message, participant b learns the contents. Because a is the sender, b now also knows that a knows the contents, but would not yet be safe to deliver the message: a and b might both fail, and perhaps no other member has a copy. In the terminology of Paxos, we would say that the update is not yet *stable*. Later, when a process is finally able to deduce that all other processes have a copy (know the contents), it can conclude that the update has stabilized. Depending on the particular Paxos protocol used, some steps can involve all-to-all control-data exchanges. With these versions of Paxos, if process c discovers that message m has become stable, c may also be able to deduce that eventually every healthy process will arrive at this same conclusion.

A knowledge logic introduces operators to represent the idea of reasoning about information directly available to processes in a system (facts), together with indirect knowledge paths: process a may know that process b knows some fact f. For example, if a sends a message to b, initially a has no information about when b will have received that message. Later, b acknowledges the message, and a now is said to *know* that b *knows* any facts carried in the message, etc. This leads to a hierarchy of knowledge: $K_a(f)$ if *a knows f*, $K_a(K_b(f))$, etc. If the set of participants is known, we write $K_a^1(f)$ to denote that *a knows* that every member *knows f*. In a similar sense, K^n denotes the n-fold property that every process knows, that every process knows, ... (n times), that some fact holds. K^* denotes common knowledge: a fact for which K^n holds for all values of n.

To make this concrete, here is an example from Derecho's atomic multicast protocol. Using asynchronous monotonic deduction over the SST, we employ a provably-correct safety deduction to detect the condition that *all messages from m to n can safely be delivered, in a round-robin order that also respects the sender FIFO ordering.* If a member has nothing to send, it sends a *null multicast.* Even under heavy load, this rule is fast: Experiments showed that the delay from sending a multicast to delivery is often as small as $1.5us$: double the one-way RDMA latency on our hardware. Moreover, this same pattern arose in several stages of our protocols, and it lends itself to opportunistic batching.

4 Implications for Other Systems

The success of the effectively-common knowledge model as an enabling tool for Derecho's asynchronous, monotonic control plane surprised even the development team. We were led to adopt this model by the sequence of insights laid out in the paper: first the recognition that asynchronous streaming is the key to high performance on modern networks supporting RDMA, then that this pattern is easy to achieve for data streams but much more challenging with control data. This then led us to the insight that monotonic control data could stream quite efficiently if the applications consuming the data are able to reason using a monotonic deductive style, in which missing an update or two poses no difficulty at all because the next deductive inference simply "catches up" on a batch of events rather than just one. Opportunistic batching doesn't impact protocol complexity but it does change the "constants." Fewer messages are needed, and when one process falls slightly behind it can catch up quickly.

The power of this sequence of steps became clear when we realized that our Paxos protocol achieves theoretical lower bounds proved by Keidar and Schraer [16]. No proof was required: Keidar and Schraer express their bounds in terms of the number of message exchanges required to safely deliver an atomic multicast or a similar update. A colleague of ours at the University of Surrey, Professor Gregory Chockler, offered to review the specification of the Derecho protocol and undertook to count the exchanges of information that occur through the intermediary of the Derecho SST. He pointed out that the number of remote RDMA writes performed by Derecho matches the bound they derived. Interestingly, this is actually a worst case for Derecho: because of batching, Derecho will sometimes omit some writes, performing one write at the end of a batch of message receives. When this happens, we actually do even better than the lower bound! On the other hand, opportunistic batch sizes are limited, so the speedup is at best a constant factor.

Seemingly, simply by setting out to express the control plane of Derecho as an asynchronous stream of monotonic information, we stumbled upon an optimal Paxos implementation. This is particularly striking because, to our knowledge, no prior Paxos is optimal in the Keidar and Schraer sense. Yet we did not set out to achieve this property: it emerged from our methodology. Thus while the idea of effectively-common knowledge may be somewhat esoteric, the pragmatic value of the overall methodology is evident. It forces a new mindset, and this mindset turns out to align closely with the "right" way to think about protocols.

Recalling our 2PC examples from the introduction, it makes sense that monotonic SST-based protocols can achieve optimal behavior. Suppose that a is using a 2PC to stream reliable multicasts to b, c, etc. Clearly, a 2PC can commit as soon as the required set of acknowledgments are received, which we express as an aggregation query over the SST. Process a, looping through a series of SST predicates, will reevaluate this query again and again, reacting as soon as the needed property is observed. Nonetheless, a's discovery of message stability may Snot occur instantly when b performs its RDMA write to acknowledge reception. After all, a also has other work to do and the predicate thread might not get a chance to reevaluate the predicate "instantly" in a real-time sense. But our opportunistic batching approach allows a to sense that the commit property was achieved for this 2PC instance the very next time the aggregation query is performed – and because of monotonicity, a simultaneously detects stability for any other instances that have reached the same knowledge level! Thus, the detection of safety occurs as early as possible and covers all the 2PC instances that are now committable, as a batch.

We have come to believe that distributed protocols are best visualized from an information-theoretic perspective in which the protocol developer asks what knowledge is gained from each deductive inference performed by the system, and what knowledge is communicated in each message or remote RDMA write. We begin to express safety properties as knowledge predicates: K^1 knowledge being the case required for most steps of Paxos (for example, "when process p deduces that all peers have received message m, it learns that m cannot be lost and hence that it is safe to advance to the next protocol stage"). Monotonicity makes the SST compact, while also guiding the developer towards opportunistic batching.

This methodology could usefully be applied in other settings that depend on strong consistency or other forms of strong guarantees. Databases and transaction systems are an obvious candidate to consider, but it is notable that even modern ML training systems provide fault-tolerance and exactly-once semantics (many MapReduce frameworks adopt this approach). Microsoft's Azure storage fabric is strongly consistent, and the AWS S3 infrastructure recently added strong consistency as well. The same sequence of reasoning and development that yielded Derecho would be a promising basis for work that could lead to speedups in all of these cases.

5 Conclusion

Our paper is an outgrowth of work on Derecho, a system created at Cornell to support distributed application development. The paper focuses on *effectively-common knowledge*, defining this concept, discussing its value (illustrated by an unusually efficient message-ordering policy), and describing its implementation. The approach lends itself to a style of monotonic exchange of state information that enables opportunistically batched decision-making, and is particularly efficient in systems supporting RDMA hardware.

Acknowledgements. The authors are very grateful to Luis Rodriguez, who read an earlier draft of this paper and suggested many ways that it could be improved. The SSS 2023 reviewers were incredibly helpful. Our work was funded, in part, by grants from AFRL under its SWEC program, Microsoft Research and Siemens, and the experiments summarized here used hardware generously provided by NVIDA and its Mellanox subsidiary.

Appendices

The two appendices in this section provide additional detail going beyond the material in the body of the paper. Neither is needed to understand our main contributions. Appendix A offers two examples of common knowledge, drawing on examples from [9]. Appendix B discusses the connection between effectively-common knowledge and a tactic used when formally verifying protocols using provers that can fully automate subproofs provided that they are fully expressed in a decidable fragment of first-order logic (often, the subset that the Z3 SMT solver can handle). We considered but decided against including an appendix on RDMA (this kind of hardware has been actively discussed for at least a decade, and there is an excellent Wikipedia article covering the one-sided write feature we used), and on virtual synchrony (well known to the community since 1987).

Appendix A: Common Knowledge

A.1 Impossibility of Outdoor Dining in Seattle

Two friends work in Seattle, a city known for cloud cover and damp weather, but when the sun pops out they would prefer to meet outside. The complication is that both sometimes attend meetings in rooms lacking phone reception. A first idea is that if one of them notices that the weather is fine, they will text the other, who will confirm, and then they can meet outside for lunch.

"But wait", says one to the other. "If I text you, but receive no reply, I will have to assume that my text was not received. In that case I would wait for you here, in the cafeteria." "In fact," replies the other, "I would have the symmetric problem: even if I do receive your text, I wouldn't know that you received my confirmation, and would have no choice but to wait for you in here in the cafeteria. And if you confirm my confirmation, that doesn't help either!"

This is very strange. After all, once the intial text is confirmed, and the confirmation is confirmed, both are aware that it is a sunny day. Yet no matter how many messages they exchange, they do not converge to the identical state. An inductive analysis always leads to the cafeteria: their "default" option.

Both fall silent: the impossibility of meeting outside for lunch now being apparent. "Well," says one, "if the weather is nice I'll just send you a text and will be out here. No need to confirm. If you can't make it, I'll understand!"

This first example illustrates that (1) Posed in this manner, logicians can only base "symmetric" decisions on existing common knowledge. (2) No matter how many messages are exchanged knowledge asymmetry cannot be eliminated. Of course, in real life we don't need common knowledge (and sometimes, things happen, and we can't join the lunch crowd).

Discussion: The insight to take from this first story is that distributed systems in which information must be observed (by some process) and then learned (by other processes) embody an asymmetry. When formalized, their members will never all be in the identical knowledge state, and attempts to achieve symmetry lead to unbounded yet ineffective exchanges of messages.

In what way is this relevant to distributed computing? The main and perhaps only importance relates to specification and proof. It is very easy to write a specification that unintentionally requires common knowledge. However, such a statement must either be implied from the initial conditions (and hence vacuous), or if not, cannot be achieved by any protocol. A proof assistant can check the logic of a given proof, or even find certain kinds of proofs or counterexamples on its own, but will not signal this type of specification error. Thus a seemingly innocent mistake can lead to an impossible-to-prove specification. The person tasked with carrying out the proof would either give up or, more likely, abandon parts of the task. This last scenario should worry us: it suggests that there could be "proved correct" systems for critical tasks that actually ignore parts of the protocols used.

Effectively-common knowledge is in fact not identical to the form of common knowledge of the kind Halpern and Moses considered in [9]. With effectively-common knowledge, we consider a modular system in which one module implements epochs, and the other modules run within epochs and simply trust the view and any annotations as if they were common knowledge. We carry out separate proofs for the two modules, then compose one system from the two modules. Our proof coverage is stronger, and the developer never confronts what would otherwise be an infeasible task.

A.2 The Inscription on the Cake was a Lie!

On Carol's birthday, her friends come to play outside before lunch. It being Seattle, all are quite muddy when they enter the kitchen. "In this house we have a rule!", proclaims her father, Ted. "No dessert for anyone who has a dirty face!". His wording is ill-chosen, because no child likes to wash their face, and every child optimistically believes their own face to be clean until proven otherwise. None moves a hair, although all the children see one-another's dirty faces. Increasingly annoyed, Ted repeats himself a few times. But even after n repetitions (n being the number of children), no child has washed. Ted puts the cake to the side and sends them all to wash up.

Later he relents after Carol explains the inductive proof that justified their action. She first addresses $n = 1$. "Daddy, just the other day this happened. You told me I would need to wash if my face was dirty, but I was hoping it was clean." "Carol, " replies Ted, "all you needed to do was to look in the mirror." "But Daddy, the mirror is too high!". Ted is forced to acknowledge that Carol would have had no way to deduce that her face must have been dirty.

"Now Daddy, consider $n = 2$. Timmy and I come in, both dirty. You remind us of the rule. But neither of us likes to wash our faces, and anyway, Timmy is mean and would love for me to not get cake and have to watch him enjoying it. And I feel the same! So we both look at each other, and I see that Timmy's face is dirty, and he sees that mine is dirty, and neither of us moves." Ted replies, "Yes Carol, but now your logic fails. I repeated myself." "You did, Daddy. But I was hoping my face was clean. Timmy hoped that his was clean. So our decision not to go and wash up was consistent with one of us believing that neither of our faces was dirty, even if it also consistent with one in which both of us had dirty faces. You didn't give us enough information!"

At the next party, when the children come in from playing, Ted first says "Well, I see some very dirty faces here!" and then repeats the household rule n times. On the n^{th} repetition, all the children simultaneously rush to the sink and wash up. Beaming, Ted unveils a cake which is inscribed: "K^* is necessary and sufficient!" The children groan: A typical Seattle "dad joke."

Later, Carol corners her dad. "Daddy, that was embarrassing! What if one of my friends hadn't heard you clearly at the start!" Ted realizes that this is a valid criticism: was his initial statement genuinely common knowledge?

Discussion: Here, we illustrate another peculiarity of common knowledge. Even in classic problems such as muddy children, it is debatable that common knowledge is really being introduced dynamically. To the extent that this does occur, some form of assertion of trust is required: the participants trust that the mechanism that shared the new common knowledge is completely reliable.

An epoch-based virtual synchrony system has an advantage here: to switch from epoch j to epoch j, members definitely must receive and "install" the new view together with any additional data annotating it. Thus for process a to interact with process b as members of epoch j, it genuinely is the case that both have replicas of the new view. By proving that the group membership cannot partition into two logically distinct views, we arrive at guarantee that the annotation can be treated like common knowledge. Ted, for example, waited until all the children were present and then assumed they would understand him.

A.3 Other Forms of Effectively-Common Knowledge

The example we offered in Sect. 1.2 focused on message ordering. What would be other uses for effectively-common knowledge?

A good place to start is with an old, classic, database partitioning scenario. When ATM machines were first introduced, they depended on dialup modems that were not always able to establish a connection (a flurry of ATM use could overload the central modem pool, leading to persistent busy signals). To fix the issue, banks introduced the idea of a "primary ATM". Perhaps, Carol almost always uses the ATM machine at the intersection of Main Street and Old Market Avenue. The bank could give that ATM "ownership" of some of Carol's current balance. For a withdrawal up to this limit, the ATM could authorize that transaction without first phoning the main office. Of course, the bank's other ATMs would not be able to access Carol's full balance: the bank has locked down this portion of her balance. But schemes were then proposed for dynamically adapting the policy.

More broadly, effectively-common knowledge arises in situations where some form of policy will span a dynamically varying set of participants. If the participant set was non-varying, we don't really need effectively-common knowledge: totally ordered multicast would suffice. But if the set of participants changes and simultaneously we need a policy that depends on a nondeterministic decision or attribute of the members, it is hard to avoid an effectively-common knowledge model.

Our insight is that virtual synchrony epochs can be viewed as virtualizing many otherwise intractable behaviors and unachievable guarantees. Within an epoch, failures "do not occur", hence protocols do not need to be fault-tolerant. Instead they can simply trust the view. And then when we realized that it would be faster to preagree on multicast delivery order in Derecho, we simply annotated the view with the ordering policy to use. The fully generalized case simply allows the application itself to provide additional annotations, which it can then treat as effectively-common knowledge once the epoch begins.

Appendix B: Higher-Order Protocol Components

Effective common knowledge in the context of virtually synchronous epochs enables a deductive strategy also seen in protocol verification. This statement may feel like a non-sequitor: any protocol exchanges messages to gain information, and is designed to achieve a state in which it is safe to take whatever action the protocol embodies. Yet we do not normally think of formal reasoning of the kind used in protocol verification as offering ideas that can be directly useful in protocol design.

Developers of complex protocols have always struggled to prove them correct. Today this burden is much reduced: Provers such as Dafny, TLA+ and Ivy are widely used to check the correctness of protocols [10,19,22]. DistAlgo, a specification and proof framework, goes even further, allowing rigorously specified protocols to be proved correct and even generating an executable verified code instance [20]. Less widely appreciated is that they struggle to overcome a significant expressivity limit. Today's most popular provers operate by taking a specification and reducing it to a decidable logic formula expressed entirely in first order logic. The basic tactic is to form a conjunction of protocol invariants, invert it, and then use Z3 (an SMT solver) to search for a counterexample. If Z3 terminates, either it exhibits a counterexample and the protocol is not correct, or it finds none and the protocol is proved. If Z3 fails to terminate, the developer modifies assertions and then tries again. If a protocol is buggy, this yields a concrete example of how the bug can be triggered.

The expressivity issue stems from the inability of first-order logic to capture and hence verify higher order properties, such as conditions that need to be expressed over traces, or progress conditions. However, encountering such an issue is not a dead end. In such systems it is also possible for a developer to *combine hand-created higher order proofs with first order automated checking.*

To see how this is done, we should start by noting that first-order provers normally support modularization of protocol proofs, allowing the user to isolate and reason about a component of the protocol without simultaneously reasoning about the rest of the system. An example of this might involve a "sub-protocol" for forming a collection of processes into a ring: an example relevant to our running example, which used a ring to define the round-robin order used in Derecho message delivery.

It may be surprising to realize that a ring is an example of a system property that cannot be expressed in a first order logic. The central issue is that first-order logics are limited to boolean variables, relations that take boolean inputs and output a boolean result, logical conjunctions and (with significant limitations) existential quantifiers. This model is not strong enough to define the natural numbers, or to talk about the natural order on the natural numbers, and for the same reason, it is not strong enough to express some properties that depend on protocol traces that represent runs. And, to be very specific, first order logic cannot verify a protocol that organizes a set of nodes into a ring.

Yet this is simply a limitation of first-order logic. There are many logics within which we do have access to the natural numbers, can reason about orderings and other properties, and can define a ring. For example, on a ring every process has a predecessor, a successor. Call these $pred(a)$ and $succ(a)$ for process a. Both are unique, and moreover there exists some integer k such that $pred^k(a) = a$ and $succ^k(a) = a$. The issue is that to the extent that Dafny and Ivy proofs are checked by Z3, we accept that it will be infeasible to verify protocol modules that maintain properties such as the ring one. There would be no problem doing this in a higher-order logic such as the one used in Coq, but the task will be much less automated: a human would need to carry out the proof, and perform many steps by hand.

The usual work-around is to provide a second proof framework in which a human developer can express higher order questions and carry out higher order proofs of protocol fragments that rely on higher order logic. To integrate such proofs into the first-order layer, they then need a way to export artifacts from these proofs back into first-order logic (and keep in mind: this cannot involve extending first order logic, which is a fixed and unchangeable aspect of the methodology).

The solution leverages the fact that first order logic can express relations: functions on first-order variables that perform some kind of logical computation and return true or false. We simply treat the higher order protocol as an uninterpreted black box that outputs relations magically populated with the correct content. Our higher order protocol component can be proved to correctly construct these relations. Then, having completed this proof, we can simply declare that "there exists a relation with the following properties", using first-order logic to define those properties. In this way, the higher-order artifact can be reasoned about rigorously, then used as a tool by the first-order relation. This is how first-order systems deal with properties such as the ordering on the natural numbers.

Thus, from the perspective of the first order logic, $succ$, $succ^k$, $pred$, $pred^k$ and k are relations, but uninterpreted ones populated "elsewhere". To reason about how they are constructed we use the higher-order prover. But if we simply need to describe a step in which a protocol takes some action, such as a node a passing a message to its successor, we can use an existential quantifier to assert that there exists a node b such that $succ(a) = b$, and this uses only first-order logic, because the verifier doesn't actually need to compute a value for a or b: it treats the logic statement as a universal property. The same is true for the assertion that in a ring, $\exists k :\ succ^k(a) = a$. This statement is true for all rings, and for all members, and hence the first-order prover can make use of it without needing specific values.

Our realization was that these higher order objects and properties are a bit like effectively-common knowledge: the first-order layer of the protocol simply trusts that they exist and were properly created. By packaging effectively-common knowledge as an annotation to the view, we simplify the use of this idea. The developer writes software to run in the membership leader and able to compute any desired annotations for the next membership view. One would potentially need to prove that module correct, in the higher-order logic. Having done so, the output of the module becomes effectively-common knowledge and can be treated as a well-known fact by processes running during the epoch. In effect, we compartmentalize an otherwise complicated, error-prone task.

We are not claiming that such steps magically make proofs trivial. In the case of Derecho, we are still faced with doing manual higher-order proofs for many properties. As an example, the termination condition for Derecho's virtually synchronous view update protocol is a fixed-point: eventually either the system shuts down, or reaches a point where (1) some process believes itself to be the leader, and (2) it suspects every higher-ranked process, and (3) it gains consent for some sequence of membership updates, (4) that consent is obtained from a majority of the most recently active view, and from a majority of members of each proposed view, and (5) no process in the last of these proposed views suspects the leader. This is clearly not expressible in first-order logic, nor is it a trivial proof goal even when expressed in higher-order logic. Yet it is a *feasible* proof goal, and yields a progress condition for Derecho. We can even express optimality assertions as higher-order statements.

References

1. Agarwal, D.A., Moser, L.E., Melliar-Smith, P.M., Budhia, R.K.: The Totem multiple-ring ordering and topology maintenance protocol. ACM Trans. Comput. Syst. **16**(2), 93–132 (1998). https://doi.org/10.1145/279227.279228
2. Alvaro, P., Conway, N., Hellerstein, J.M., Marczak, W.R.: Consistency analysis in bloom: a CALM and collected approach. In: Conference on Innovative Data Systems Research (2011)
3. Ameloot, T.J., Neven, F., Van Den Bussche, J.: Relational transducers for declarative networking. J. ACM **60**(2) (2013). https://doi.org/10.1145/2450142.2450151
4. Behrens, J., Jha, S., Birman, K., Tremel, E.: RDMC: a reliable RDMA multicast for large objects. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg City, Luxembourg, pp. 71–82. IEEE (2018). https://doi.org/10.1109/DSN.2018.00020
5. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: SOSP 1987, Austin, Texas, USA, pp. 123–138. ACM (1987). https://doi.org/10.1145/41457.37515
6. Birman, K.: Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services. Texts in Computer Science. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2416-0
7. Birman, K.P.: Replication and fault-tolerance in the ISIS system. SIGOPS Oper. Syst. Rev. **19**(5), 79–86 (1985). https://doi.org/10.1145/323627.323636
8. Coquand, T., Huet, G.: Constructions: a higher order proof system for mechanizing mathematics. In: Buchberger, B. (ed.) EUROCAL 1985. LNCS, vol. 203, pp. 151–184. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15983-5_13
9. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37**(3), 549–587 (1990). https://doi.org/10.1145/79147.79161

10. Hawblitzel, C., et al.: IronFleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017). https://doi.org/10.1145/3068608
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
12. Jha, S., et al.: Derecho: fast state machine replication for cloud services. ACM Trans. Comput. Syst. (TOCS) **36**(2), 1–49 (2019)
13. Jha, S., Rosa, L., Birman, K.P.: Spindle: techniques for optimizing atomic multicast on RDMA. In: 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), pp. 1085–1097 (2022). https://doi.org/10.1109/ICDCS54860.2022.00108
14. Junqueira, F., Reed, B.: ZooKeeper: Distributed Process Coordination, 1st edn. O'Reilly Media Inc., Sebastopol (2013)
15. Kashyap, V.: IP over InfiniBand (IPoIB) architecture. Technical report (2006)
16. Keidar, I., Shraer, A.: Timeliness, failure-detectors, and consensus performance. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, Colorado, USA, pp. 169–178. ACM (2006). https://doi.org/10.1145/1146381.1146408
17. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, vol. 11, pp. 1–7 (2011)
18. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998). https://doi.org/10.1145/279227.279229
19. Lamport, L., Matthews, J., Tuttle, M., Yu, Y.: Specifying and verifying systems with TLA+. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, Saint-Emilion, France, pp. 45–48. Association for Computing Machinery (2002). https://doi.org/10.1145/1133373.1133382
20. Liu, Y.A., Stoller, S.D., Lin, B., Gorbovitski, M.: From clarity to efficiency for distributed algorithms. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, Tucson, Arizona, USA, pp. 395–410. ACM (2012). https://doi.org/10.1145/2384616.2384645
21. Network-Based Computing Laboratory at the Ohio State University: RDMA-based Apache Kafka (RDMA-kafka). https://hibd.cse.ohio-state.edu/kafka
22. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. SIGPLAN Not. **51**(6), 614–630 (2016). https://doi.org/10.1145/2980983.2908118
23. Shivam, K., Paladugu, V., Liu, Y.: Specification and runtime checking of Derecho, a protocol for fast replication for cloud services. In: Proceedings of the 2023 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems, ApPLIED 2023, Orlando, Florida. ACM (2023). https://doi.org/10.1145/3584684.3597275