Acuerdo: Fast Atomic Broadcast over RDMA

Joseph Izraelevitz
University of Colorado, Boulder
Boulder, CO, USA

Gaukas Wang
University of Colorado, Boulder
Boulder, CO, USA

Rhett Hanscom
University of Colorado, Boulder
Boulder, CO, USA

Kayli Silvers
University of Colorado, Boulder
Boulder, CO, USA

Tamara Silbergleit Lehman
University of Colorado, Boulder
Boulder, CO, USA

Gregory Chockler
University of Surrey
Guildford, Surrey, UK

Alexey Gotsman
IMDEA Software Institute
Madrid, Spain

ABSTRACT

Atomic broadcast protocols ensure that messages are delivered to a group of machines in some total order, even when some of these machines can fail. These protocols are key to making distributed services fault-tolerant, as their total order guarantee allows keeping multiple service replicas in sync. But, unfortunately, atomic broadcast protocols are also notoriously expensive.

We present a new protocol, called Acuerdo, that improves atomic broadcast performance by using remote direct memory addressing (RDMA). Acuerdo is built from the ground up to perform communication using one-side RDMA writes, which do not use the CPU of the remote machine, and is explicitly designed to minimize waiting on the critical path. Our experimental results demonstrate that Acuerdo provides raw throughput comparable to or exceeding other RDMA atomic broadcast protocols, while improving latency by almost $2x$.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Networks → Network algorithms**.

KEYWORDS

atomic broadcast, RDMA, consensus

ACM Reference Format:
Joseph Izraelevitz, Gaukas Wang, Rhett Hanscom, Kayli Silvers, Tamara Silbergleit Lehman, Gregory Chockler, and Alexey Gotsman. 2022. Acuerdo: Fast Atomic Broadcast over RDMA. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545041

1 INTRODUCTION

Remote direct memory access (RDMA) is a network protocol that allows one machine to access the memory of another one. In contrast to TCP/IP, RDMA accesses can bypass the CPU (and the kernel), thus giving an opportunity for significant performance gains, especially in instances where the kernel is the limiting bottleneck [38]. In the past several years, RDMA has moved from being a tool used primarily for high performance computing into a commodity capability accessible to a wider variety of computing needs. This migration was caused by the decrease in RDMA hardware costs to the point where RDMA can be integrated into relatively small data centers and racks. While early research in RDMA focused on a reasonably narrow set of topics related specifically to high performance computing, there is now an opportunity for more general software tools to benefit from RDMA (e.g. [1, 5, 8, 12, 20, 34]).

In this work, we explore how RDMA can be used to scale up *atomic broadcast protocols* that ensure that messages are delivered to a group of nodes in some total order despite some of the nodes failing. Atomic broadcast protocols are key to making distributed services fault-tolerant [31]; they allow a service to run on several replicas and propagate client operations to them in a common order. Since atomic broadcast guarantees that all replicas receive the same set of operations in the same order, the individual replicas remain synchronized, and the system can continue serving client requests correctly even if some of the replicas become unavailable. Atomic broadcast protocols are often used to implement *coordination kernels*, such as ZooKeeper [14], which are used by larger systems to manage critical data.

The design of an atomic broadcast protocol can be viewed as consisting of two main parts: normal mode operation and failure recovery. In the normal mode, the message ordering is typically driven by a single *leader* process, which is responsible for assigning every message a consecutive entry in a *message log*, and propagating it to a set of *followers* (or *backups*). For fault-tolerance, the message ordering needs to be finalized through *consensus*, which in the normal mode, stipulates a round of acknowledgments from the followers to the leader. Failure recovery is initiated by the failure of the leader, after which the remaining replicas choose a new leader in an *election* — once the new leader is elected, we again enter the normal broadcast mode.

In this paper we present *Acuerdo*, an atomic broadcast protocol which enables incredible speedup by running at the speed of the fastest quorum, instead of the slowest active node as in prior work. Acuerdo leverages RDMA properties and is specifically designed to *minimize waiting* within the algorithm: a node almost never needs to wait for a message from another node to make progress, and nodes are almost always doing useful work.

For most atomic broadcast algorithms, waiting is a common occurrence, either because some nodes become out of date (and the rest of the instance must wait for them to catch up), messages go missing (and nodes must wait for them to arrive before continuing useful work), nodes are slow to respond (and therefore the rest of the instance stalls since stale values cannot be garbage collected), or simply due to transient network hiccups (triggering one or more of the above issues). We argue that minimizing waiting is the first step towards optimizing an atomic broadcast algorithm: algorithms with no waiting can support effectively infinite "pending" messages, and can process messages as fast as clients can send them.

For minimizing waiting, RDMA is an exceptional opportunity. When using a reliable RDMA connection, messages will be delivered in order at the receiver, avoiding unnecessary waiting where a later message arrives first and cannot be processed. RDMA writes are also handled at the remote NIC, as opposed to the remote CPU, so messages can be delivered even if the remote process is descheduled. Finally, the RDMA protocol design, and many of the RDMA-specific optimizations that exist in the literature, support efficient catch-up through a batching mechanism. Thus, even if a node falls behind, it can catch up to the remainder of the instance because RDMA catch-up mechanisms run faster than the overall system.

Unfortunately, prior art in atomic broadcast algorithms over RDMA fails to take advantage of these benefits. Some work fails to take advantage of the FIFO delivery benefits of RDMA [29, 38]. Most work adds some waiting on the critical path with dramatic performance consequences: a single delayed message in these systems can halt the entire system's progress [16, 17]. Our work is specifically designed to take advantage of the benefits that RDMA can offer, resulting in latencies nearly $2x$ faster than prior art while providing competitive bandwidth.

To further minimize service interruption and waiting, the failure recovery in Acuerdo is based on a novel leader election protocol that aims to restore the system back to operational state while minimizing communication overheads and service interruption. To achieve this extremely low downtime after a failure, our election protocol is guaranteed to elect the most up-to-date node in a quorum, thereby avoiding additional waiting wherein the leader must acquire (from other nodes) any missing messages before beginning to accept messages from clients (cf. [18, 21]).

In summary, Acuerdo is a novel atomic broadcast protocol designed to leverage the benefits of RDMA atop the following building blocks which are the contributions of this work:

(1) A novel and efficient catch-up mechanism, optimized for RDMA (Section 3), which achieves superior latency and throughput by systematically avoiding waiting;
(2) A new election protocol that is guaranteed to elect an "up-to-date" leader, thus preventing long-latency state transfers.
(3) A new failure recovery procedure; and
(4) Case studies demonstrating that Acuerdo maintains a throughput which is comparable to other existing RDMA-based atomic broadcast protocols while improving message latency by $2x$ (Section 4).

2 BACKGROUND

2.1 RDMA

As the RDMA protocol [30, 33] is a significant departure from traditional transport methods, its design takes a whole stack approach. RDMA is for the most part managed at the network interface card (NIC), which sits on the PCIe bus. The card has access to memory through this bus and on most machines resides within the cache coherency domain [15, 23].

To establish an RDMA connection between two NICs, applications must exchange device information and register the connection (usually this exchange is done via a different protocol, such as TCP). Next, applications exchange memory permissions to allow the remote node access to their own memory. This exchange is done by first registering memory with the NIC for remote access, which pins the memory to prevent it from being swapped out and returns a remote access key. This key is sent to the remote node to give it access to the registered local memory region.

Once the connection is established at both NICs and remote keys have been exchanged, the application can begin to use RDMA messages (called *verbs*), such as read and write. It is important to note that verbs are sent to the NIC via user space — the kernel is only involved when we initially establish the connection.

In this work, we use RDMA's *reliable connection* type, which guarantees the lossless delivery of messages in FIFO order [30, 33]. In order to guarantee the delivery of writes at the remote node, the local NIC must receive some acknowledgment (or *completion* in RDMA terminology). Until the completion for a given write is received, the NIC will maintain space in its memory in case the write has been dropped. If the completion is not received within a preset timeout, the NIC reissues the remote write. In an important optimization to the reliable connection protocol (and one we use extensively), not every write must be acknowledged with a completion. Rather, because the reliable connection ensures FIFO delivery of messages, the completion of a later write serves to acknowledge the receipt of *all* earlier writes, effectively allowing us to batch acknowledgments. Consequently, for connections which issue large numbers of writes, we only need to request a completion periodically in order to clear our send queue — this optimization is known as *selective signaling* [3]. Our implementation issues a write with a completion request every thousand messages, which works well on our network. Our system is tuned specifically for RDMA over Converged Ethernet (RoCE), but would presumably also work over an Infiniband network with appropriate tuning.

2.2 Atomic Broadcast

Acuerdo is a distributed implementation of an *atomic broadcast* protocol supporting reliable totally ordered broadcast of messages to a set of replica nodes. Messages are received from a client and `broadcast` to the nodes. At each replica, messages are `delivered` to the application running on the same node.

A system implements atomic broadcast if every one of its executions satisfies the following properties:

Integrity. Every message delivered was previously broadcast (i.e., no out-of-thin air values).

No Duplication. No message is delivered more than once to the same node.

Total Order. All nodes are delivered a prefix of a common message order (i.e., messages are delivered in the same order to all nodes, without gaps).

Atomic broadcast is a key building block for implementing fault-tolerant services by means of the *state-machine replication* approach [31]. This approach runs a service on several replicas and propagates client operations to the replicas using the `broadcast` primitive; when a replica `delivers` an operation, it applies the operation to the local service state. Since atomic broadcast guarantees that all replicas deliver the same set of operations in the same order, the individual replicas remain synchronized, and can continue serving client requests correctly even if some of them become unavailable.

As is standard for atomic broadcast protocols, Acuerdo runs over a group of $n = 2f+1$ processes, at most f of which can fail by crashing. The processes are connected by reliable FIFO channels, provided by the reliable RDMA connections that we use. We call any majority of $f+1$ processes a *quorum.*

3 THE ACUERDO PROTOCOL

Acuerdo is a novel distributed atomic broadcast protocol designed to leverage all the benefits of RDMA to improve performance. Acuerdo is designed around the principle of running as fast as the fastest quorum. This goal is accomplished through four design strategies: (1) leveraging FIFO delivery, (2) efficient catch-up, (3) tolerance of transient delays, and (4) novel leader election.

Leveraging FIFO Delivery RDMA's FIFO delivery ensures that messages are received in the order that they were sent, and Acuerdo leverages this ordering to minimize waiting in two distinct ways. First, the use of FIFO delivery avoids a critical source of waiting in many atomic broadcast algorithms—waiting for missing messages before processing subsequent ones that have already arrived [18]. Secondly, RDMA's FIFO guarantees allow Acuerdo to efficiently represent and transmit replica state. Since a replica's state is equivalent to the messages it has received and processed, we can, with some care, represent the replica's state using message identifiers. In Acuerdo, the replicas leverage these identifiers to efficiently notify each other of received and committed messages, which, thanks to the RDMA FIFO guarantees, implicitly acknowledge their previously received and committed messages. As an added advantage, since subsequent RDMA writes to the same location overwrite earlier values, these replica state updates can always be directed to the same address, simplifying update handling [17].

Efficient Catch-Up A node that falls behind needs to have the ability to catch up to the remainder of the instance, instead of falling further and further behind and eventually inhibiting progress of the entire system. For efficient catch-up, we use a *receiver-side batching* model [11, 19]. Since RDMA messages do not wake up the CPU, they will instead be handled in batches determined by the receiver's event loop and scheduler. Since the CPU can process messages faster than they can be sent over the network, the CPU will drain the batch faster than it accumulates, even in the presence of long scheduler induced delays. The message identifier-based notification scheme similarly accelerates this catch-up mechanism. The receiver only notifies its peers once it drains its batch, and this late notification avoids network traffic when catching up. Consequently, the cost of consensus is amortized over the size of the (receiver-determined) batch.

Tolerating Transient Delays In real networks, intermittent disruptions of service are likely. While efficient catch-up can be seen as a way of handling these delays, Acuerdo incorporates this principle into other parts of its design. In particular, Acuerdo is a quorum-based protocol—messages are committed once accepted at a majority of nodes. As with most quorum based systems, the quorum is "flexible"; the system as a whole does not manage which nodes are in or out but instead simply gathers a minimal number of responses. This design decision stands in contrast to a virtual synchrony model [6, 7, 9], as used in the Derecho algorithm [16, 17], in which the quorum is explicitly managed in response to detected failures or slowdowns. A virtual synchrony scheme will take action if the delay is "bad enough" to merit the cost of reconfiguration. In contrast, Acuerdo does not need to take explicit action when a node is slow; it simply expects that most nodes will be fast, and the slow node will catch up efficiently.

Novel Leader Election Acuerdo uses a novel election mechanism that is guaranteed to elect an "up-to-date" leader that is aware of any messages that could have been previously committed. This property means that the new leader, upon winning the election, need not confer with its followers to determine what earlier messages to truncate or include into the log (cf. [18, 21]), nor do they need to wait before beginning to accept client messages. Instead, the new leader can unilaterally decide how to proceed, avoiding an additional round trip communication to all its followers and reducing the downtime caused by the fail-over.

3.1 Acuerdo Overview

Like many atomic broadcast protocols, Acuerdo uses a *leader-based* approach. It is most similar to the Zab protocol [18] from the popular ZooKeeper coordination kernel, with the difference that Acuerdo leverages RDMA for minimizing waiting [14] (see Section 5 for a detailed comparison). On initialization, nodes in Acuerdo attempt to elect a *leader* in the *election* mode; the leader wins an election with a majority of votes, thereby winning the right to *propose* new messages in Acuerdo's *broadcast* mode. Once elected, the leader has begun a period of sovereignty (called an *epoch*) in which it is the sole node that can propose messages to its *followers.* The followers will store them in an ordered message *log* and *accept* the message, by publishing this fact to the leader. Once a message has been accepted by a majority, it is *committed*, and a follower can *deliver* its contents to the local application. If the leader at any point becomes unresponsive, the remaining nodes abandon it and start a new leader election, taking care to ensure that all messages already committed are preserved into the following epoch and within the new leader's log.

An epoch is identified by a tuple composed of an increasing round number and the identification number of its leader, that is, $epoch = (round_{nbr}, leader_{id})$ (see Figure 1). Epochs are totally ordered by the round number and then by leader ID and increase over time: it is always the case that a larger epoch comes after

```
typedef pid uint32;  // process ID type
// all tuples are ordered by values left to right
typedef epoch tuple<uint32 round, pid ldr>;
typedef msghdr tuple<epoch e, uint32 cnt>;
typedef message tuple<msghdr hdr, payload pyld>;
typedef vote tuple<epoch e_new, msghdr acpt>;
typedef diff map<msghdr, message*>;

vector<pid> NodeIDs[NUM_NODES]; //node IDs
pid Self; //my process ID
epoch E_cur; // our current epoch
epoch E_new; // new epoch we plan on joining
msghdr Accepted; //last message from leader
msghdr Committed; //last committed message
msghdr Next; //next message to commit
uint32 Count; //num msgs sent this epoch as ldr
enum Role = {ELECTING, LEADER, FOLLOWER};

SST<msghdr> Accept_SST;
// used for msg acceptance
// Accept_SST[j] : last message j accepted

SST<vote> Vote_SST;
// used for leader election
// Vote_SST[j] is j's vote with
// Vote_SST[j].e_new.ldr as pending ldr with
// ldr's last accepted msg Vote_SST[j].acpt

SST<msghdr> Commit_SST;
// Used for diffs.
// Commit_SST[j] : j's last committed msg

map<msghdr,message*> Log;

ringbuffer outgoing_buff;
ringbuffer[NUM_NODES] incoming_buff;
```

Figure 1: Types and Local Process Variables

a smaller one. Each node tracks which epoch they are currently in (called `E_cur`), and, if in transition to a new leader, the epoch (called `E_new`) they wish to join (if not in transition, the values are the same). They also track what their `Role` is in the current epoch: "leader", "follower", or "electing".

3.2 Normal Broadcast

Broadcast is Acuerdo's normal mode, in which a single leader proposes a sequence of messages to its followers in order to ensure the messages are failure-safe. Acuerdo's broadcast mode is inspired by Zab [18], an atomic broadcast algorithm that leverages FIFO delivery on TCP/IP. Like Zab, Acuerdo's broadcast mode has two different tasks for all nodes: accepting and committing. The leader of an epoch has a third task in addition to these two, broadcasting. The main difference between Zab's broadcast mode and Acuerdo's broadcast mode is that thanks to the FIFO delivery of RDMA and *a shared state table (SST)* data structure [17] described below, Acuerdo's followers do not need to acknowledge every message. Instead they can acknowledge only the most up-to-date message, minimizing traffic and latency. This difference is important, as it also removes the need for the leader to receive acknowledgments for all messages.

During the broadcast phase, nodes are accomplishing three tasks. The leader *broadcasts* new messages to its followers, and the followers *accept* these messages by acknowledging receipt. Once a

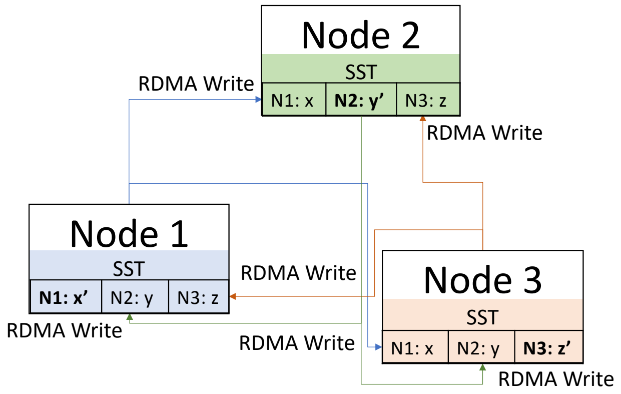


Figure 2: The Shared State Table (SST) allows nodes to communicate efficiently when only the last write matters by allowing a node to write into the array element that corresponds to it.

quorum of nodes has accepted the message, it is guaranteed to survive tolerated failures and can be *committed* at all nodes.

Broadcasting Within the broadcast mode, the leader broadcasts messages on behalf of clients to the other replicas (Figure 4). When the leader of an epoch receives a message from a client to broadcast, it first computes the message header. The *message header* consists of both the epoch tuple in which the message was proposed, *i.e.*, $(round_{nbr}, leader_{id})$, along with a unique, monotonically increasing *message ID* number within the new epoch (generated at the leader). The total order of the messages is dictated first by the epoch and then by the message ID. Having computed the header, the leader broadcasts the message with its header to all processes (including itself).

```
// broadcasting
On: Broadcast payload pyld per client request
Pre: Role == LEADER
msghdr hdr = <E_new, ++Count>;
outgoing_buff.send_to_all(new message<hdr,pyld>);
```

Figure 4: Broadcasting

Acuerdo uses RDMA ring buffers to broadcast. The RDMA *ring buffer* is a common communication primitive that supports a single sender and multiple receivers. The ring buffer allows a node to send messages via broadcast or unicast [11, 25]. To send a message, the message contents are written to the sender's local buffer and then mirrored to a receiver's remote buffer using RDMA writes. Receivers poll for new messages at their current incoming tail until it changes to indicate a new incoming message and its size. The RDMA ring buffer allows the leader to continually pipeline messages to followers without waiting for any acknowledgment or interrupting the remote CPU; received messages will be handled in a batch at the follower once it discovers them. This mechanism follows the receiver-side batching style.

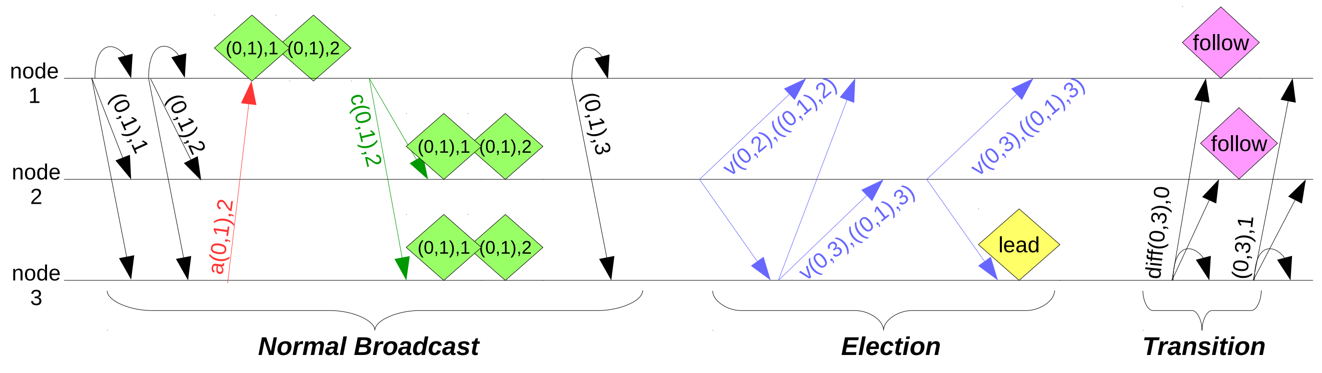


Figure 3: Acuerdo Example

The RDMA ring buffer has several performance benefits important to optimize communication between the leader and the followers. First, polling on incoming RDMA write locations is extremely efficient for the receiver. So long as the location is unchanged and cache pressure is moderate, the location will reside in the receiver's L1 cache. A cache miss will generally only occur upon actually receiving a message from the remote sender. Second, ring buffers support aggressive pipelining: as long as the buffer is not completely full, the sender can continue sending and eventually the receiver can catch up by reading and acknowledging all available messages at once. This pattern fits into the receiver-side batching model, where the size of a batch is determined by the receiver's event loop, and it avoids any waiting at the sender's side through RDMA writes. In order to support all-to-all communication, each node has access to a single `outgoing` buffer, used to broadcast to remote nodes, and multiple `incoming` buffers (one per remote node), used to receive messages.

We illustrate our protocol using the running example in Figure 3. The example begins when the leader (in this case node 1) is in broadcast and shows the leader broadcasting messages to its followers within epoch (0,1), where 0 is the round number and 1 is the leader ID. Within epoch (0,1), node 1 first sends two messages with successive headers ((0,1),1) and ((0,1),2), where the last number is the message ID number within the epoch.

Accepting All nodes in broadcast mode have two tasks — accepting and committing messages, shown in Figure 5. A node accepts a message from a leader over a ring buffer only if the message belongs to the epoch the node is in (line 47). The node adds the message to the `Log`, an ordered map which stores messages in their header order. The node also stores the most recently accepted header in the `Accepted` variable, and, finally, responds to the leader to acknowledge the message acceptance using the `Accept_SST`, which is a *shared state table (SST)* data structure.

The shared state table (SST), introduced in [17] and leveraged in Acuerdo for the both the broadcast and election modes, is an abstraction specifically designed for RDMA messages that overwrite each other, i.e., where the receiver only cares about the last write received. For instance, this is the case when the writes carry the value of a monotonically increasing counter. The SST is represented by a shared array indexed by node ID, which is replicated on all nodes (as shown in Figure 2). A node has write access to its own slot in the array, and read access to all others. On writing to its own slot, a node can *push* its update to some or all other nodes using RDMA writes, thereby overwriting its most recent value on remote arrays. By traversing its local copy of the SST, a node can get a type of "snapshot" of the shared state. In the case of acceptance, the leader can traverse its own, in-memory, `Accept_SST` to determine the most recently accepted messages at each remote node.

```
42 // accepting
43 On: Receive
44  message<msghdr<epoch e, uint32 cnt> hdr,
45   payload pd> M via incoming_buff[i]
46
47 if(e == E_new && e == E_cur){
48   // normal message acceptance
49   Log[hdr] = M;
50   Accepted = hdr;
51   Accept_SST[Self] = hdr;
52   Accept_SST.push_mine_to(E_cur.ldr);
53 }
54 elif(E_new ≤ e){
55   // diff acceptance
56   // and transition into broadcast
57   assert(hdr.cnt == 0);
58   E_new = e;
59   E_cur = e;
60   if(e.ldr ≠ Self){Role = FOLLOWER;}
61   diff d = ((diff)M.pyld);
62   ∀N∉Log ∧ N.hdr≥d.minKey() Log.remove(N);
63   ∀(h,n)∈d Log[h] = n;
64   Accepted = hdr; Accept_SST[Self] = hdr;
65   Accept_SST.push_mine_to(E_cur.ldr);
66   Next = <E_cur,0>;
67 }
```

Figure 5: Accepting

The acceptance mechanism in Acuerdo is also used outside of normal broadcast to safely transition to a newly elected leader; the remainder of the acceptance code (line 54 and forward) is used for this task. We will return to this topic in Section 3.4.

We illustrate the acceptance step using our example execution in Figure 3. In particular, this example demonstrates how nodes receive messages and report their message acceptance to the leader by writing to the remote SST (denoted by red arrows). As nodes receive messages via one-sided RDMA writes, they may discover two messages at once, and will simply accept the later message, implicitly accepting the earlier one due to RDMA's FIFO guarantee—this is the key difference with Zab that allows Acuerdo to have significant performance improvements. In the running example in Figure 3, node 3 discovers message ((0,1),2) together with message ((0,1),1), leading to an acceptance notification of only the later message. This acceptance implicitly acknowledges the first message and removes a source of latency associated with the process of catching up.

Committing The other role of nodes in the broadcast mode is to commit messages once they have been accepted by a quorum, shown in Figure 6. Nodes commit messages in the order they are stored in their logs. Nodes keep track of the next message to be committed (with the `Next` variable) based on this log order. The leader determines when to commit a message using its `Accept_SST`, which contains information as to what messages have been accepted at which nodes. Namely, the leader checks if the header stored in the next message to be committed (the `Next` variable) is below or equal to the one posted to its local `Accept_SST` by a majority of nodes (a quorum) from the same epoch (line 74). If so, it commits the message; it will later forward this commit to its followers off the critical path, pushing a commit SST message out using a `Commit_SST` (line 93). Followers determine when to commit a message based on the leader's `Commit_SST` entry at the current epoch. A follower simply needs to check if the next message header is below or equal to the header stored in the leader's entry of the `Commit_SST` (line 79).

Upon committing a message, a node caches its header in the `Committed` variable as the most recently committed message, and delivers the message to the local application. As with acceptance, Acuerdo's commit sequence is also used outside of normal broadcast to confirm a newly elected leader (lines 83 and on). We will return to this in Section 3.4.

The running example in Figure 3 shows messages ((0,1),1) and ((0,1),2) committed at the leading node 1 (shown in green) upon receiving the acceptance notification from node 3 (the message has been accepted at node 3 and locally). After node 1 propagates its commit to the other nodes, they can commit the messages as well.

3.3 Election

The Acuerdo election mechanism gains performance advantages because it guarantees to elect a node that is aware of any messages that could have been previously committed. This is a novel property in Acuerdo: other systems either need to transfer state to the leader [21, 24], or use a secondary step to verify the leader is up to date and retrigger an election if it is not (e.g., RAFT [28], Zab [18], and DARE [29]), possibly hanging the instance in a livelock.

The Acuerdo election mechanism is initiated at system start-up or after a node suspects that the current leader has failed (e.g., due to a heartbeat timeout). The goal of the election protocol is to choose a leader who (1) has convinced a quorum of followers to join their epoch and (2) has a log which is at least as up to date as any of their voters. This second requirement, which we call the "up-to-date" property, guarantees that the new leader knows of any message

```
68 // committing
69 On: Periodically when
70  (Role == LEADER || Role == FOLLOWER)
71
72 if(  (Role == LEADER &&
73  majority$_{k∈N}$(Accept_SST[k] ≥ Next
74  && Accept_SST[k].e == E_cur))
75   ||
76  (Role == FOLLOWER && Commit_SST[E_cur.ldr] ≥ Next
77  && Commit_SST[E_cur.ldr].e == E_cur) ){
78   // normal message commit
79   if(∃$_{M∈Log}$ M.hdr == Next && Next.cnt ≠ 0){
80     deliver(M); Committed = Next;
81   }
82   // diff commit
83   elif(Next.cnt == 0){
84     d = {(h, m) ∈ Log | Committed < h < Next})
85     while(d ≠ NULL){
86       n = d[d.minKey()]; deliver(n);
87       Committed = n.hdr; d.remove(d.minKey());
88     }
89   }
90   Next.cnt++;
91 }
92
93 On: Periodically
94 Commit_SST[Self] = Committed;
95 Commit_SST.push_mine();
```

Figure 6: Committing

which has been committed in the previous epoch. This avoids a transfer of messages to the leader before it can begin broadcasting.

The Acuerdo election uses a designated SST for the election itself, the `Vote_SST`. This replicated array coordinates the election by giving every node a view of every other node's vote. A vote consists of two pieces of information: (1) the proposed epoch of the new leader and (2) the last accepted message of the new leader. A node's vote is ordered first by the epoch and then by the accepted message and is always increasing. The election mechanism is a type of "fixed point" algorithm. At every step, some nodes will increase their votes in order to converge to a situation where a quorum votes for a single leader.

While in election mode or after a timeout, nodes repeatedly run the election mechanism (Figure 7). Nodes have two vote rules that dictate when they can vote for a candidate. Nodes vote either for (1) the largest vote they are aware of *if* the vote's candidate has accepted a message with an equal or larger header than the local node (line 106), or (2) themselves, taking care to ensure that their vote is increasing (line 100) by creating a larger epoch number with themselves as the leader (line 102). Nodes will propose themselves as candidates only if no suitable other vote exists or the best candidate has timed out. Regardless of who they vote for, nodes update their new epoch variable (`E_new`) to indicate their intention of following their preferred candidate, update their entry in the `Vote_SST`, and use the SST to propagate their vote to all other nodes. This algorithm terminates provided all non-failed nodes continue to respond within the timeout period. This is in contrast to the election process in RAFT and DARE which could result in a livelock [28, 29]. Once termination happens, the mechanism converges on a candidate who has convinced a majority of nodes to vote for its new epoch and which is in possession of a log at least as up-to-date as any of its voters. This property obviates the need for a leader to spend

```
96  On: Timeout or Periodically when Role == ELECTING
97  Role = ELECTING;
98  votes_cpy = Vote_SST;
99  mx = max_vote(votes_cpy);
100 if(timed_out || Accepted > mx.acpt){
101   // vote for self
102   E_new = new_bigger_epoch(E_new,mx.e_new,Self);
103   Vote_SST[Self] = <E_new,Accepted>;
104   Vote_SST.push_mine();
105 }
106 elif(mx > Vote_SST[Self] && Accepted ≤ mx.acpt){
107   // join max vote and vote for someone else
108   E_new = mx.e_new;
109   Vote_SST[Self] = <E_new,mx.acpt>;
110   Vote_SST.push_mine();
111 }
112
113 comm_cpy = Commit_SST;
114 if(majority_{v∈Vote_SST}(v == Vote_SST[Self])
115  && Vote_SST[Self].e_new.ldr == Self){
116   // transition to leader
117   Role = LEADER;
118   Count = 0;
119   foreach(j in NodeIDs){
120     msghdr hdr = <E_new, 0>;
121     payload pd = new diff();
122     for(h in {h'| ∃_{m'} Log[h'] = m' ∧
123      comm_cpy[j] ≤ h' ≤ Accepted})
124       pd[h] = Log[h];
125     outgoing_buff.send_to(j, new message<hdr,pd>);
126   }
127 }
```

Figure 7: Leader Election

time to bring its state up to date based on the messages committed at the followers. Instead, Acuerdo's election guarantees that the leader is the most up-to-date node of its quorum. This guarantee is leveraged during the transition period as explained in Section 3.4.

Our running example (Figure 3) shows an election (shown in blue) after the original leader (node 1) times out, having failed to send message ((0,1),3) to node 2. Node 2 falls to election, proposing itself as leader of epoch (0,2). However, once node 3 joins the election, it rejects node 2's candidacy, as node 3 has a more up-to-date log that includes message ((0,1),3). Subsequently, node 2 votes for node 3 as its proposed vote is higher, increasing its vote to vote for a better candidate. Having collected the votes of nodes 2 and 3, node 3 wins the election and can begin epoch (0,3) by sending a `diff` message, described in the next section.

3.4 Transition into Broadcast

Acuerdo's novel election mechanism guarantees that the elected leader is up to date, and consequently there is no need to verify the election in the transition period. Instead the transition period leverages ring buffers to quickly arrive at consensus regarding the last message committed prior to the current leader's reign.

In order to enter into broadcast, the leader must ensure that all of its followers have equivalent commit logs, that is, they all agree on what messages were included in the previous epoch. In order to ensure this invariant, the first message that the leader sends, and that a follower will receive, is a *diff message* through the ring buffer. As the first message sent by the leader of the new epoch, a diff always has a message header of count zero. The diff message not only provides notification that a new leader has been elected, but also includes a list of whatever messages the follower may be missing. Diffs are handled in similar ways to regular messages, in that they are both accepted and committed using the same actions.

The construction of the `diff` message is shown in Figure 7 and starts at line 119. Upon receiving and accepting this diff (message number zero in the epoch) a node can transition out of the election state and into the broadcast state as a follower. Since the header of the latest committed message implicitly acknowledges the headers of all previously committed messages, the diff only needs to include messages that have not yet been committed at the receiving node.

When a follower receives a diff, it accepts this diff *on the condition* that it has not voted for a larger epoch; that is, so long as its proposed new epoch, `E_new`, is less than or equal to the epoch in the diff's header (Figure 5, line 54). In the case when the proposed new epoch, `E_new`, matches the incoming diff's epoch, this indicates the local node voted for the new leader. If the proposed new epoch, `E_new`, is smaller, this indicates the local node voted for some other candidate with a smaller vote, or entirely missed the election. Accepting this incoming diff means that the follower has officially joined the new epoch and is ready to accept additional messages from this leader. Therefore, the follower sets both the current epoch and the proposed new epoch to the diff's epoch and transitions into the follower role. The node then applies the diff to its local state in order to synchronize its own `Log` with that of its new leader, first by removing any uncommitted values newer than the diff's first message, and then replacing them by the contents of the diff. It then stores the diff's message header as the last accepted message and pushes this fact via the `Accept_SST`.

To commit a diff, a follower uses the same criteria as a normal message (upon notification from the leader). Committing a diff (Figure 6, line 83) means committing all the included messages that have not been committed earlier and delivering them to the local application.

The final section of our running example shows the transition back into broadcast (Figure 3 in pink). Having won the election, node 3 becomes the leader of epoch (0,3). It then sends a diff message to all nodes with header ((0,3),0), which includes any messages they might be missing. In particular, the diff includes message ((0,1),3), which has not yet been received by node 2. Nodes that receive the diff then transition into the new epoch (0,3) as followers; they subsequently accept node 3's diff and future messages.

4 EXPERIMENTAL RESULTS

We ran our experiments on Cloudlab, an open platform for running network experiments that gives exclusive access to the nodes [36]. In particular, our experiments used a cluster with nodes that have an Intel E5-2640v4 processor each running Ubuntu 18.04 (for reference, Cloudlab calls this cluster **xl170**). Each node has 64GB of DRAM and a dual-port Mellanox ConnectX-4 25 GB NIC. The experiment network is confined to a single chassis hosting a Mellanox 2410 switch that connects each core with 25Gb ethernet links that support RDMA over Converged Ethernet (RoCE). Code was compiled using g++ 7.5.0 at the `-O3` optimization level.

We ran our experiments on a number of available atomic broadcast algorithms, designed for both TCP and RDMA (we review the latter in more detail in Section 5):

derecho Derecho is an atomic broadcast protocol built for RDMA [16, 17]. Unlike Acuerdo, Derecho uses a virtual synchrony model — that is, all nodes are assumed to be up and failed nodes are configured out of the system. This model facilitates the transfer of large messages, since the protocol can use a peer-to-peer style transfer for these [4]. Derecho can operate in two modes. In **derecho-leader**, only the leader sends messages. In **derecho-all**, all acceptors send messages in a round-robin pattern.

apus Apus is another atomic broadcast protocol designed for RDMA; like Acuerdo, it uses a leader-based approach [38]. Leaders have exclusive access to remote logs; this property is controlled at the RDMA connection. Apus uses a more traditional Paxos-based commit protocol than Acuerdo: it executes a separate consensus instance on every message, which has been identified as a major performance and scalability bottleneck in practical systems (e.g., [22]).

libpaxos Libpaxos is an open-source implementation of the standard Paxos algorithm which operates over TCP [32].

zookeeper Apache ZooKeeper [14] is a distributed coordination service implemented over the Zab protocol [18, 37]. ZooKeeper includes a distributed key value store on top of a Zab instance, which runs over TCP; we use the most recent version (3.4.14).

etcd Etcd [13] is an open source implementation of the RAFT [28] algorithm operating over TCP. Like ZooKeeper, etcd includes a key value store on top of the underlying atomic broadcast engine. Our experiments used the most recent version (3.4.7).

4.1 Broadcast Performance

We evaluated raw broadcast speed by running experiments on a stable network where no nodes failed. Both throughput and latency were monitored for the same experiment across varying loads. The client regulates the system load, ensuring that at most a fixed number of messages (the "window") are outstanding and unacknowledged. At low loads, the system responds quickly with very low latency. As load increases, bandwidth tends to increase with slight impact on latency until the system reaches maximum bandwidth and latency spikes, giving a "knee" in the graph.

We increased the window size by powers of two, starting at one, until reaching the saturation of the system. Message sizes are fixed for the entire experiment at either 10 or 1000 bytes, and we provide results for small (three) and large (seven) replica counts (the leader is included in this count). Leaders broadcast messages immediately, without batching, upon receiving messages from a client.

Figure 8 shows the results of these raw performance experiments. For small node counts and message sizes, Acuerdo outperforms all competitors in both latency and throughput. The comparison with the next closest competitor, Derecho-leader, is illustrative of the design decisions made within Acuerdo.

Acuerdo has far better latency than Derecho-leader, and improves over all other competitors by at least an order of magnitude. Messages can be committed to the Acuerdo instance within around 10 usec for small groups and messages. This compares to at least 19 usec for small messages in Derecho.

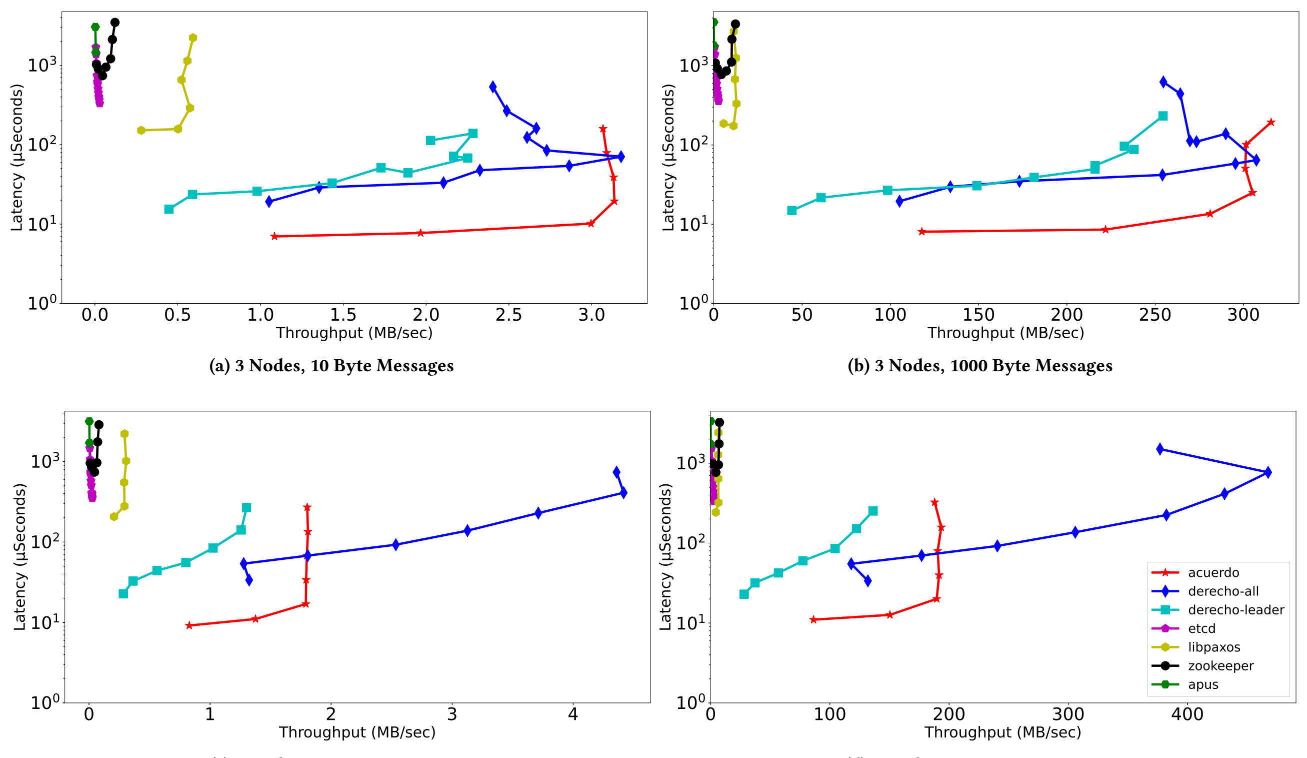
The performance difference is a consequence of two design decisions made within Acuerdo, both chosen to avoid waiting (and consequently reduce latency). First, Derecho's design, like Acuerdo's, uses a ring buffer for communication between nodes. However, they differ in when a message slot can be reused. Acuerdo can reuse a slot once the receiver has simply accepted the message. Long buffers are sufficient to cover any transient interruptions, and with receiver-side batching, it is easy for the receiver to catch up. In contrast, Derecho can only reuse a slot once the message has been committed across all active nodes, thereby magnifying the impact of a single slow node. This design decision makes sense in virtual synchrony, where slow nodes already impact throughput, and is consequently tightly integrated into Derecho's failure recovery algorithm. Second, Derecho uses a virtual synchrony design — messages are committed once all active nodes have acknowledged them. As a consequence, a single slow node will force the entire cluster to commit operations at its speed, whereas Acuerdo will simply leave the node behind to catch up later.

Similar waiting-based issues impact the other RDMA-based system, APUS. APUS constructs small batches consisting of at most a single message from each client (it assumes messages from disparate clients are unordered). However, APUS is only able to have a single pending batch at any one time — its underlying Paxos-based implementation [26] was designed for networks that reorder messages and can only process a single, complete batch at any one time. Any delay on any message in the batch will cause a total system stall. In contrast, Acuerdo and Derecho can process partial batches by taking advantage of FIFO delivery guarantees.

Another point to note is that Acuerdo significantly outperforms Derecho's single leader results across all replicas for both latency and throughput. For throughput, this difference has to do with the ring buffer implementation: Acuerdo uses a single RDMA write and Derecho uses two. As the minimum size of an RDMA message is 80 bytes, for small messages, this design decision means that Acuerdo is twice as bandwidth-efficient (6 MB/s vs. 3 MB/s for Derecho with 10 byte messages on 3 nodes).

Importantly, this difference is not a simple implementation detail, but rather a consequence of the algorithm's expected use case: Derecho is designed for large message transfers, while Acuerdo is designed for smaller ones. To send a Derecho message, the message is sent in a single RDMA write, and then a separate counter is incremented with a second RDMA write. The counters, which track the number of messages sent between each pair of nodes, are subsequently forwarded across the instance to determine message acceptance information, and can be "wedged" upon entering a leader change due to failure. The use of separate paths for message metadata (the counter) and the data within allows Derecho to vary the data delivery method depending on the size of the message: for very large messages, Derecho can use a peer-to-peer delivery system to reduce the load on the leader bandwidth [4]. In contrast, Acuerdo is optimized for small messages and tightly couples message metadata with data, thereby improving performance for messages small enough to transfer efficiently via RDMA writes from the leader.

Derecho's all-to-all variant takes the optimization argument further, prioritizing bandwidth over latency. In all-to-all mode, Derecho does not elect a leader, but instead allows each node to propose



(a) 3 Nodes, 10 Byte Messages

(b) 3 Nodes, 1000 Byte Messages

(c) 7 Nodes, 10 Byte Messages

(d) 7 Nodes, 1000 Byte Messages

Figure 8: Broadcast performance. This experiment shows latency (usec) and throughput (MB/sec) under varying load. The “knee” in the graph occurs when the system begins to suffer from queuing effects and maximum bandwidth is reached; ideal performance lies in the bottom right corner. The load varies the outstanding messages from the client in range $(2^0, 2^1, \ldots, 2^N)$ to saturation. Note log scale on y-axis.

a message according to a round-robin policy. This strategy allows the cluster to avoid throttling on a single leader’s send bandwidth (see Acuerdo’s latency spikes when saturated) when transferring large messages across large clusters, but leaves the system even more vulnerable to slow nodes when transferring small messages: it fails to provide good latency in this scenario. This result can be seen in Figure 8 by the vertical difference between derecho-all (blue line) and acuerdo (red line) before Acuerdo’s saturation point.

4.2 Election Performance

Acuerdo elections result in very little downtime. Table 1 shows the election duration as the number of replicas grows. Our experiment runs for several minutes and sets the leader to propose 10-byte messages in an open loop. We then repeatedly cause the leader to sleep five seconds after winning its election. The election is timed from the point at which the new leader detects the old leader as down until the new leader can begin sending its first new message. That is, the time includes the election protocol and the diff transfer, but does not include the time it takes to detect a failure. We report the average duration for all elections across the run. Interestingly, we found election times were far more sensitive to the proportion of “long-latency” nodes (nodes that take longer, in general, to respond) than to the overall number of replicas. For instance, a cluster of seven nodes is more affected by two long-latency nodes than a cluster of nine as the long-latency nodes are less likely to enter the active quorum or election. The displayed data shows averages across runs, where we migrate the active set of nodes between machines to eliminate this effect.

3 nodes	5 nodes	7 nodes	9 nodes
.3 ms	6.8 ms	12.1 ms	12.6ms

Table 1: Average Acuerdo election duration as a function of replica count (includes the diff transfer).

4.3 Application Use Case

In order to experiment with Acuerdo in an application use case, we integrated an Acuerdo instance with an RDMA hash table (similar to [11]). The replicated hash table sits at all Acuerdo replicas, each of which has a complete copy. This scenario is an expected use-case for Acuerdo; it replicates the data structure across multiple nodes for fault tolerance.

The hash table is accessed by a separate, external, client machine that can send requests via RDMA to the Acuerdo instance.

Using Acuerdo, we replicate update commands (create, set, delete) received from the client for crash resilience; once commands are committed by Acuerdo they are delivered to the replica's local hash-table copy and acknowledged to the client. Hash-table gets can be done directly via RDMA from the client to any replica, thereby bypassing the Acuerdo instance.

To benchmark this configuration, we used the YCSB benchmark suite, commonly used for testing key-value stores. We specifically use the YCSB-load test [10], which continually applies writes in a .99 skewed zipfian distribution, to show the effects of using the Acuerdo protocol. This hash-table configuration is effectively equivalent to an Apache ZooKeeper [14] or etcd [13] deployment with in-memory storage. Figure 9 shows the result of this comparison. As expected, the Acuerdo deployment significantly outperforms both systems, generally by around 10x for ZooKeeper and 50x for etcd.

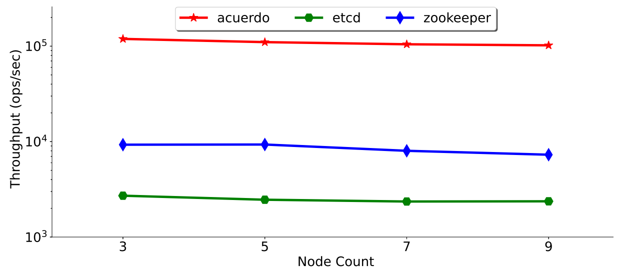


Figure 9: Throughput (ops/sec) on YCSB-load as a function of node count. Note log scale on y-axis.

5 RELATED WORK

Acuerdo is one of a long series of atomic broadcast protocols, the most well-known of which are Paxos [21], Zab [18, 37], View-stamped Replication [24] and Raft [28]. The design of Acuerdo is closest to that of Zab, used in the popular ZooKeeper coordination kernel [14, 35]. Like Acuerdo, Zab is leader-based and relies on FIFO communication links. The contribution of Acuerdo lies in a broadcast protocol that avoids waiting and in exploiting RDMA to do so.

The election phase of Acuerdo is one of the main differences from that of Zab. The published description of Zab [18] allows an arbitrary process to become a leader. This makes recovery expensive, since a prospective leader has to transfer the state from a majority of other processes to determine the state in which the broadcast needs to be restarted. Zab's actual implementation in ZooKeeper tried to avoid the need for this transfer by electing a leader that is already up to date, but this implementation was later shown to be incorrect [27]. The optimized leader election has never been properly fixed: currently ZooKeeper has to perform an additional message exchange (and wait) after the leader is elected to check if its state is up to date. If this check fails, the election process is restarted. The election in Acuerdo yields a leader that dominates the state of a majority of acceptors; no expensive state transfer from acceptors to the leader is necessary.

Other atomic algorithms have also been designed for RDMA. The earliest, DARE [29], uses a novel broadcast method that leverages the ability of nodes to temporarily close RDMA connections. Acceptors following a leader close their (data) connections to all other nodes, giving the leader exclusive access to the acceptor's memory and allowing the acceptor's CPU to remain effectively passive. Leaders use this exclusive access during broadcast to manipulate the remote logs stored in the acceptors' memory. While a novel idea, DARE's usage of this exclusive access by the leader is unfortunately relatively slow as it relies on fine-grained RDMA completions: in order to send a message to a remote acceptor, leaders must first write to the log, ensure the write is completed, then mark the entry as valid. For leader election, DARE uses an algorithm which combines ideas from both ZooKeeper and Raft. Similar to Acuerdo, its leader election guarantees that the leader of a new epoch will know all committed messages from the prior epoch. Unlike Acuerdo, however, DARE's leader election requires every acceptor to vote at most once per election round. Consequently, DARE can deadlock when several acceptors fall into an election but split their vote among several valid contenders; this split vote deadlock will result in another expensive timeout and election round. To deal with this deadlock issue, DARE uses randomized timeouts to avoid large numbers of acceptors falling to an election at once, necessitating somewhat slack timeout values.

APUS [38] is a more recent work that significantly improves on DARE's performance. APUS accelerates broadcast by a simple batching strategy that includes in each batch a single message from each active client. In APUS, the remote acceptor periodically acknowledges to the leader batches of messages that have been delivered. The leader, upon receiving a quorum of acknowledgments, can commit the message in some later packet. The use of batching here, along with a more effective acknowledgment implementation that avoids the use of RDMA completion queues, significantly improves APUS's performance over DARE. For leader election, APUS uses a Paxos-based protocol similar to Raft [26].

Derecho [16, 17] is another consensus algorithm explicitly designed for RDMA, but it takes a significantly different approach from APUS, DARE, and Acuerdo. Unlike the other algorithms, which use a quorum of acceptors to ensure a message is committed, Derecho uses a virtual synchrony model [6, 7, 9]. In virtual synchrony, in order for a message to be committed, it must be replicated to all active nodes. Failures, which in a quorum system can be transient, are treated as hard outages and the node is configured out from the active group (but can later be reconfigured back in). Derecho runs as fast as the slowest node in the system. In contrast, Acuerdo gets its speedup by being able to run as fast as the fastest quorum.

The most recent work, Mu [2], introduces a novel mechanism of using the RDMA completion for acceptance acknowledgment. Mu does not require follower CPUs to wake up to acknowledge messages, instead using RDMA completions from the NIC as the ack. As such, Mu requires closing and opening the RDMA connection in the case of an election since followers must ensure the only open connection is to the leader. Unfortunately, Mu's software is both tuned and specialized for an Infiniband network and was incapable of running on our RoCE cluster.

6 CONCLUSION

In this paper we have introduced Acuerdo, a novel atomic broadcast protocol designed from the ground up for RDMA. Acuerdo takes advantage of several RDMA design points in order to maximize its performance and avoid waiting. Acuerdo maintains excellent bandwidth performance when compared to other RDMA atomic broadcast protocols, but improves latency by almost $2x$ and experiences minimal downtime upon leader failure.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (Starting Grant RACCOON) and Genuen.

REFERENCES

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conf. (USENIX ATC 18)*. USENIX Association, 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symp. on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 599–616. https://www.usenix.org/conference/osdi20/presentation/aguilera

[3] Dotan Barak. 2019. Tips and tricks to optimize your RDMA code. (2019). http://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/.

[4] Jonathan Behrens, Sagar Jha, Ken Birman, and Edward Tremel. 2018. RDMC: A Reliable RDMA Multicast for Large Objects. In *2018 48th Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*. 71–82. https://doi.org/10.1109/DSN.2018.00020

[5] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (2016), 528–539.

[6] Ken Birman and Thomas A. Joseph. 1987. Exploiting Virtual Synchrony in Distributed Systems. In *Eleventh ACM Symp. on Operating Systems Principles (SOSP '87)*. ACM, Austin, Texas, USA, 123–138. https://doi.org/10.1145/41457.37515

[7] Kenneth P. Birman and Thomas A. Joseph. 1987. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems* 5, 1 (Jan. 1987), 47–76. https://doi.org/10.1145/7351.7478

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Eleventh European Conf. on Computer Systems (EuroSys '16)*. London, United Kingdom, 26:1–26:17.

[9] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. 2001. Group Communication Specifications: A Comprehensive Study. *Comput. Surveys* 33, 4 (Dec. 2001), 427–469. https://doi.org/10.1145/503112.503113

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symp. on Cloud Computing (SoCC '10)*. ACM, Indianapolis, Indiana, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[11] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *11th USENIX Conf. on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Seattle, WA, 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

[12] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *25th Symp. on Operating Systems Principles (SOSP '15)*. ACM, Monterey, California, 54–70. https://doi.org/10.1145/2815400.2815425

[13] The Linux Foundation. 2020. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[14] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Conf. on USENIX Annual Technical Conf. (USENIX ATC'10)*. USENIX Association, Boston, MA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[15] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Technical Report 325462-060US. Intel Corporation.

[16] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. on Computer Systems* 36, 2 (April 2019). https://doi.org/10.1145/3302258

[17] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Sydney Zink, Ken Birman, and Robbert Van Renesse. 2017. Building Smart Memories and High-speed Cloud Services for the Internet of Things with Derecho. In *2017 Symp. on Cloud Computing (SoCC '17)*. ACM, Santa Clara, California, 632–632. https://doi.org/10.1145/3127479.3134597

[18] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance Broadcast for Primary-backup Systems. In *2011 IEEE/IFIP 41st Intl. Conf. on Dependable Systems&Networks (DSN '11)*. IEEE Computer Society, 245–256. https://doi.org/10.1109/DSN.2011.5958223

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *2014 ACM Conf. on SIGCOMM (SIGCOMM '14)*. ACM, Chicago, IL, 295–306. https://doi.org/10.1145/2619239.2626299

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[21] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. on Computer Systems* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[22] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Conf. on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Savannah, GA, USA, 467–483. http://dl.acm.org/citation.cfm?id=3026877.3026914

[23] ARM Limited. 2009. *Implementing DMA on ARM SMP Systems.* Technical Report Application Note 228 (ARM DAI0228A).

[24] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited.* Technical Report MIT-CSAIL-TR-2012-021. Massachusetts Institute of Technology.

[25] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High Performance RDMA-based MPI Implementation over InfiniBand. In *17th Annual Intl. Conf. on Supercomputing (ICS '03)*. ACM, San Francisco, CA, USA, 295–304. https://doi.org/10.1145/782814.782855

[26] David Mazieres. 2007. Paxos made practical. *Unpublished manuscript* (Jan. 2007).

[27] André Medeiros. 2012. ZooKeeper's atomic broadcast protocol: Theory and practice. (2012). Available from www.tcs.hut.fi/Studies/T-79.5001/reports/2012-deSouzaMedeiros.pdf.

[28] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Conf. on USENIX Annual Technical Conf. (USENIX ATC'14)*. USENIX Association, Philadelphia, PA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[29] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *24th Intl. Symp. on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, Portland, Oregon, USA, 107–118. https://doi.org/10.1145/2749246.2749267

[30] Renato J. Recio, Bernard Metzler, Paul R. Culley, Jeff Hilland, and Dave Garcia. 2007. *A Remote Direct Memory Access Protocol Specification.* Technical Report RFC 5040. Internet Engineering Task Force.

[31] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[32] Daniele Sciascia and Marco Primi. 2013. LibPaxos. (2013). http://libpaxos.sourceforge.net/.

[33] Hemal Shah, Felix Marti, Wael Noureddine, Asgeir Eiriksson, and Robert Sharp. 2014. *Remote Direct Memory Access RDMA Protocol Extensions.* Technical Report RFC 7306. ISSN: 2070-1721. Internet Engineering Task Force.

[34] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *2019 Intl. Conf. on Management of Data (SIGMOD '19)*. ACM, Amsterdam, Netherlands, 433–448. https://doi.org/10.1145/3299869.3300069

[35] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. 2012. Dynamic Reconfiguration of Primary/Backup Clusters. In *2012 USENIX Conf. on Annual Technical Conf. (USENIX ATC'12)*. USENIX Association, Boston, MA, 39–39. http://dl.acm.org/citation.cfm?id=2342821.2342860

[36] University of Utah. 2022. CloudLab. (2022). https://www.cloudlab.us/.

[37] Robbert van Renesse, Nicholas Schiper, and Fred B. Schneider. 2015. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Trans. on Dependable and Secure Computing* 12, 4 (July 2015), 472–484. https://doi.org/10.1109/TDSC.2014.2355848

[38] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *2017 Symp. on Cloud Computing (SoCC '17)*. ACM, Santa Clara, California, 94–107. https://doi.org/10.1145/3127479.3128609