# Aurora: Leaderless State-Machine Replication with High Throughput

Hao Lu, Jian Liu, *Member, IEEE* and Kui Ren, *Fellow, IEEE*

***Abstract*—State-machine replication (SMR) allows a deterministic state machine to be replicated across a set of replicas and handle clients' requests as a single machine. Most existing SMR protocols are leader-based requiring a leader to order requests and coordinate the protocol. This design places a disproportionately high load on the leader, inevitably impairing the scalability. If the leader fails, a complex and bug-prone fail-over protocol is needed to switch to a new leader. An adversary can also exploit the fail-over protocol to slow down the protocol.**

**In this paper, we propose a crash-fault tolerant SMR named Aurora, with the following properties:**

- **Leaderless: it does not require a leader, hence completely get rid of the fail-over protocol.**
- **Scalable: it can scale up to 11 replicas.**
- **Robust: it behaves well even under a poor network connection.**

**We provide a full-fledged implementation of Aurora and systematically evaluate its performance. Our benchmark results show that Aurora achieves a throughput of around two million Transactions Per Second (TPS), up to 8.7× higher than the state-of-the-art leaderless SMR.**

***Index Terms*—Crash fault-tolerance, state machine replication, distributed systems**

## I. Introduction

STATE machine replication (SMR) is a distributed systems technique where a *deterministic* state machine is replicated across a set of nodes, known as *replicas*, to ensure fault tolerance and consistency. In SMR, using a *consensus* protocol is the most popular method to guarantee consistency in the presence of faulty replicas. More specifically, replicas run the consensus protocol to agree on the order of client requests, store the ordered requests into a local log, and execute the logged requests slot by slot. This strategy has been widely used in real-world systems to provide better reliability. For example, the classical consensus, Paxos and its variants [1], [2], had been adopted in Chubby [3], Google Spanner [4] and Microsoft Azure Storage [5]; recent systems such as RethinkDB [6], Redis [7] and CockroachDB [8], chose Raft [9] over Paxos due to better understandability.

Most consensus protocols like Paxos and Raft are leader-based, i.e., requiring a leading replica to decide the order of requests and coordinate the protocol. The protocol *cannot* make progress if the leader fails and it relies on a *fail-over* protocol to switch to a new leader. This design has negative effects on at least two aspects:

- *Efficiency.* It places a disproportionately high load on the leader, inevitably impairing the scalability. An adversary can also exploit the fail-over protocol to further slow down the protocol [10].
- *Codebase complexity.* Although the normal case is simple, the fail-over protocol is too complex to implement and debug [11]. For example, the fail-over protocol has introduced bugs to Redis [12] and RethinkDB [12] when Raft was integrated into these systems.

To address the first issue, recent consensus protocols adopt a multi-leader paradigm to evenly distribute the leader's responsibility and overhead [13]–[15]. However, this paradigm needs a fail-over protocol that is more complex than the single-leader consensus.

**Leaderless SMR.** In SOSP '21, Pan et al. presented a leaderless SMR named Rabia [16] that targets the setting of a single datacenter. Rabia gets rid of the fail-over protocol by leveraging a *randomized binary consensus* (RBC) to agree on the request for each slot of the log. More specifically, each replica proposes a batch of requests and runs RBC; (i) if *all* (non-crash) replicas propose the same requests, they agree on those requests for the current slots; (ii) if *no* majority of replicas propose the same requests, they forfeit the slots; in either case, RBC terminates in two message delays. Pan et al. claim that case (i) is common in a single datacenter where message delay is small compared to request intervals. Case (ii) reduces the chance for a long tail latency: when RBC seems difficult to terminate fast, the replicas forfeit the slots so that RBC can still terminate in two message delays. Despite these nice properties, Rabia has two limitations: can only work within a single datacenter, and cannot scale to a large number of replicas, which significantly limits its deployability.

**Our contribution.** In this paper, we propose a leaderless SMR named Aurora, which successfully overcomes the two limitations of Rabia: it extends to multiple datacenters deployed in different zones within the same region and scales to more replicas. The core idea of Aurora is to have replicas run RBC to agree on "whether a certain replica has proposed requests", instead of "whether all replicas have proposed the same request"; $n$ instances of RBC can proceed in a batch (where $n$ is the number of replicas). In Rabia, each replica proposes a batch of requests and they can agree on one batch at most. In Aurora, each replica also proposes a batch of requests, but they can agree on up to $n$ batches, and these

This work is supported by National Key Research and Development Program of China (2023YFB2704000).

The authors are with the Zhejiang University, Hangzhou, Zhejiang 310027, China. E-mail: {luhao, liujian2411, kuiren}@zju.edu.cn.

Corresponding author: Jian Liu.

requests are likely to be different as each replica serves as a proxy for different clients. As a result, Aurora could potentially achieve a throughput that is $n$ times higher than Rabia. Furthermore, we borrow idea from the gossip protocol to make two message delays to be the common case even under a poor network connection. We provide a full-fledged implementation of Aurora and systematically evaluate its performance. Our experimental results show that Aurora achieves a throughput around two million TPS, up to 8.7 $\times$ higher than Rabia.

## II. Related Work

**Leader-based SMR.** In 1998, Lamport presented Paxos [1], [2], which is the first work in this line of research. Paxos requires three phases with five message delays. Multi-Paxos [11] eliminates the first phase by designating a stable leader, resulting in three message delays. However, if the leader crashes, a fail-over protocol will be triggered to select a new leader. QuePaxa [17] improves the fail-over component of Multi-Paxos by using randomization. Fast Paxos [18] saves one message delay by allowing clients to bypass the leader and send their requests directly to all replicas. It performs well when no two clients issue conflicting requests concurrently, but if collisions happen too frequent, it may perform worse than classic Paxos. Generalized Paxos [19] improves throughput by leveraging partial ordering, which allows requests to be committed in different orders on different replicas, provided that executing them in any order has the same effect. Multico-ordinated Paxos [20] improves availability by allowing clients to send their requests to a quorum of replicas. Raft [9] was proposed to improve understandability and provide a better codebase for implementing real-world systems. Roughly, it adopts a stronger notion of leader to simplify the conceptual design.

**Multi-leader SMR.** To improve the performance of leader-based SMRs, multi-leader SMRs have been proposed. For example, in Mencius [13], every replica acts as a leader for the sequence numbers assigned to it. For example, $\mathsf{R}_i$ is a leader for all sequence numbers $j$ that satisfies ($j \mod n = i$). Ideally, Mencius can achieve a throughput that is $n$ times larger than leader-based SMRs. However, once a crash occurs, the throughput of Mencius will quickly drop to zero until a revocation starts that makes all correct replicas learn of no-ops for instances coordinated by the faulty replica. EPaxos [14] introduces a dependency graph to track the relationship of different requests. Benefits from this approach, non-conflicting requests can be committed in a fast path of two message delays. However, replicas need to spend more time for checking the dependencies; and in the presence of conflicts, it takes a slow path of four message delays. Some protocols [21], [22] eliminate the dependency graph in EPaxos. However, they still need to resolve conflicts because they suffer the same vulnerability to conflicts as EPaxos. SDPaxos [23] improves the performance by separating replication from ordering. However, it relies on a centralized node for ordering requests, which could potentially become a performance bottleneck. WPaxos [24] is a WAN-optimized multi-leader protocol. It leverages the fast quorum $Q = \lfloor n/2 \rfloor + f$ to cut WAN communication costs. Some work [25], [26] improves performance by running multiple consensus instances, which also increases both the code complexity and the cost of recovery.

**Leaderless SMR.** Ben-Or's randomized binary consensus [27] is the main component for constructing a leaderless SMR. Ezhilchelvan et al. [28] use a common coin [29] to reduce the average number of message delays and allow proposers to propose arbitrary values. Pedone et al. [30] exploit the weakly ordering guarantees from the network layer. Cachin et al. [31] propose a Diffie-Hellman based coin-tossing protocol and construct a practical and theoretically optimal Byzantine agreement protocol. Its efficiency and robustness were further improved in [32]–[35].

## III. Preliminaries

### A. State-machine replication

*State-machine replication* (SMR) allows a *deterministic* state machine to be replicated across $n$ *replicas* and handle clients' requests as a single machine, even in the presence of $f$ faulty replicas. SMR typically employs a *consensus* protocol to achieve consistency and fault tolerance, ensuring the following two properties:

- **Safety:** all non-faulty replicas execute the requests in the same order, a.k.a. **agreement**; each executed request was proposed by a client, a.k.a. **validity**.
- **Liveness:** a request proposed by a client will eventually be executed, a.k.a. **termination**.

The famous FLP result states that it is impossible to achieve deterministic consensus in an asynchronous network where at least one replica may crash [36]. As a result, most existing SMR systems [1], [9] ensure both safety and liveness only when the network is stable (i.e., synchronous); they do not ensure liveness when the network becomes asynchronous.

### B. Randomized binary consensus

Another way to circumvent the FLP impossibility result is to allow probabilistic termination. For example, in the *randomized binary consensus* (RBC) for $n = 2f + 1$ replicas proposed by Ben-Or [27], the probability for a replica $\mathsf{R}_i$ to terminate approaches 1 as time proceeds, even in the presence of $f$ faulty replicas. However, the original RBC has an average latency that is exponential with respect to the number of replicas, as each replica flips a local coin. This can be addressed by replacing the local coins by a common coin whose value is identical across all replicas. The details of RBC with a common coin can be found in Algorithm 1.

RBC proceeds in rounds: in each round, replicas first exchange their states and decide their votes (Line 3-9 in Algorithm 1) and then exchange their votes (Line 10-20 in Algorithm 1). $\mathsf{R}_i$ sets $vote$ to $b$ (which is either 0 or 1) if $b$ appears at least ($\lfloor \frac{n}{2} \rfloor + 1$) times in the STATE messages that $\mathsf{R}_i$ has received (Line 6 in Algorithm 1); otherwise, $\mathsf{R}_i$ sets $vote$ to ? (Line 8 in Algorithm 1).

- If a non-? value $b$ appears at least $(f + 1)$ times in the received VOTE messages, $\mathsf{R}_i$ can safely terminate and output $b$ (Line 14 in Algorithm 1).

**Algorithm 1:** RBC - $\mathsf{R}_i$

```
   Input: A binary value b_i
   Output: A binary value of consensus
1  state := b_i
2  while true do
3      Broadcast(STATE, r, state)        ▷ r is the round number
4      Wait until receiving ≥ n − f round-r STATE messages
5      if value b appears ≥ ⌊n/2⌋ + 1 times then
6          vote := b
7      else
8          vote := ?
9      end
10     Broadcast(VOTE, r, vote)          ▷ vote can be 0, 1 or ?
11     wait until receiving ≥ n − f round-r VOTE messages
12     if a non-? value b appears ≥ f + 1 times then
13         Broadcast(DECIDE, b)
14         return b                      ▷ output b and terminate
15     else if a non-? value b appears at least once then
16         state := b
17     else
18         state := CommonCoinFlip(r)
19     end
20     r := r + 1                        ▷ proceed to next round
21 end
22 /* executing in background */
23 Upon receiving a DECIDE message:
24     Broadcast(DECIDE, b)
25     return b
26 end
```

- If a non-? value $b$ appears at least once (a simple quorum intersection argument guarantees that there is at most one non-? value in all votes), $\mathsf{R}_i$ sets $state$ as $b$ and moves on to the next round (Line 16 in Algorithm 1).
- If all the received VOTE messages are "?", $\mathsf{R}_i$ flips a common coin to determine the $state$ for the next round (Line 18 in Algorithm 1). Notice that replicas executing Line 18 in the same round will obtain the same $state$.

We remark that, when replicas can only crash (not Byzantine), CommonCoinFlip($r$) can be implemented easily by using a random binary number generator with the same seed across all replicas. The number of rounds depends on the outcome of the common coin flip, hence its latency is less predictable compared to the leader-based consensus protocols like Paxos and Raft.

### C. Design goal

We aim to design a leaderless SMR for $n = 2f+1$ replicas where at most $f$ replicas can be faulty (fail by crashing). It achieves both agreement and termination; following Rabia [16], it only achieves *weak validity*: each slot of the log can either store client requests or be empty.

We consider an environment where communication is asynchronous: each message sent to a non-crash replica will eventually be received, but there is no bound on the message delay. Like other leader-based SMR systems, we rely on synchrony to ensure liveness; safety is guaranteed even if the communication is asynchronous.

## IV. Design Roadmap

We explain the design roadmap of Aurora by first revisiting Rabia, and then step-by-step towards the final design.

### A. Rabia revisit

In Rabia [16], a client sends requests to a designated replica (which is this client's proxy) and waits for responses from the same replica. Upon receiving a request, $\mathsf{R}_i$ pushes it to a local priority queue $PQ_i$ and forwards it to all other replicas. Replicas continuously agree on the requests for each slot as long as their $PQ$s are non-empty.

During a consensus round of Rabia, each $\mathsf{R}_i$ sends the top-$m$ requests of $PQ_i$ to other replicas in $proposal_i$ and waits for $(n-f)$ *proposal*s; if all these *proposal*s are with the same requests, $\mathsf{R}_i$ inputs 1 to RBC (Algorithm 1); otherwise, it inputs 0. If RBC outputs 1, replicas store the agreed requests in the current slot of their local logs. If RBC outputs 0, replicas forfeit the current slot and move on to the next slot.

Allowing replicas to forfeit a slot, Rabia violates the validity property of a SMR system (i.e., the output has to be client requests). Instead, it achieves a relaxed version of validity named *weak validity*: the value stored in each slot of the log can either be client requests or a NULL value $\perp$.

Although the latency of RBC is unpredictable (cf. Section III-B), the RBC in Rabia is guaranteed to terminate in two message delays when either of the following conditions is satisfied:

1) *all* replicas, excluding crashed ones, propose the same requests; or
2) *no* majority of replicas propose the same requests.

We remark that condition (1) is the key requirement for Rabia to be efficient (condition (2) also results in two message delays, but replicas will forfeit the slot, hence they still rely on condition (1) to make progress). The authors of Rabia claim that condition (1) is the most common case in a single datacenter where message delay is small compared to request intervals: given that each replica will forward a request (received from a client) to all other replicas, it is highly likely that replicas will have the same oldest pending requests in their local priority queues. However, this is not the case if replicas are deployed across multiple zones, where message delay could be larger than the request intervals. When batching is introduced, the batch of requests proposed by a replica is more likely to be different from others.

### B. Go beyond a single datacenter

Recall that a leader-based SMR relies on a leader to propose requests, and all replicas run consensus to agree on its proposal. Rabia gets rid of the leader by having *all* replicas propose requests and it uses RBC to determine *whether these replicas have proposed the same request*. Consequently, its efficiency highly relies on the condition that "all replicas, excluding crashed ones, propose the same request", which is only true within a single datacenter. In Aurora, we go one step further: we have replicas agree on *whether a certain replica has proposed requests or not*.

In Aurora, each $\mathsf{R}_i$ also sends the top-$m$ requests of $PQ_i$ to other replicas in $proposal_i$ and waits for *proposal*s from other replicas. However, we do not require all replicas to propose the same requests; on the contrary, it will be much better if they propose totally different requests (we will explain in Section IV-C). If $\mathsf{R}_j$ has received $proposal_i$ within a timeout, it inputs 1 to RBC (Algorithm 1), otherwise, it inputs 0. RBC is guaranteed to terminate in two message delays when either of the following conditions is satisfied:

1) *all* replicas, excluding crashed ones, have received $proposal_i$ within a timeout; or
2) *no* replica has received $proposal_i$ within a timeout.

Notice that we do not require all replicas to propose the same request. Hence, Aurora goes beyond a single datacenter and can be deployed across multiple zones within the same region. Furthermore, batching will not have any side-effect on RBC because we do not require replicas to propose the same batch of requests.

### C. To be more scalable

During a consensus round of Aurora, each replica $\mathsf{R}_i$ locally maintains a vector $BI_i$ of input bits: if $\mathsf{R}_i$ has received a proposal from $\mathsf{R}_j$ (i.e., $proposal_j$) within a timeout, it sets $BI_i[j] := 1$; otherwise, it sets $BI_i[j] := 0$; it always sets $BI_i[i] := 1$. Then, they run $n$ instances of RBC (Algorithm 1): each $\mathsf{R}_i$ inputs $BI_i[j]$ to the $j$-th instance; if the $j$-th instance outputs 1, the requests in $proposal_j$ will be stored in the corresponding slot; if it outputs 0, these requests will be left to the next round (forfeit the slot like Rabia). Notice that the $n$ instances of RBC can proceed in a batch, as a single instance. Figure 1 visualizes this process.

In Rabia, each replica proposes a batch of requests and they can agree on one batch at most. In Aurora, each replica proposes a batch of requests as well, but they can agree on up to $n$ batches, and these requests are likely to be different as each replica serves as a proxy for different clients. In addition, replicas do not propose requests forwarded by other replicas because they only push the requests from clients into the $PQ$. As a result, Aurora could potentially achieve a throughput that is $n$ times higher than Rabia.

### D. To be more robust

Next, we discuss different kinds of network connectivities among $n = 2f + 1$ replicas and explain how we optimize the broadcast to make Aurora more robust. For simplicity, we say $\mathsf{R}_i$ and $\mathsf{R}_j$ are "connected" if the message delay between them is *much* smaller than the timeout; in contrast, they are "disconnected" if the message delay is larger than the timeout. A set of replicas are *fully connected* if every two of them are connected; a set of replicas are *strongly connected* if there is a connected path between every two replicas in this set. Roughly, there are following kinds of network connectivities:

1) A quorum $Q$ of at least $f + 1$ non-crash replicas are fully connected, and the rest replicas are either crash or disconnected from $Q$.
2) A quorum $Q$ of at least $f + 1$ non-crash replicas are strongly connected, and the rest replicas are either crash or disconnected from $Q$.
3) No quorum of more than $f$ non-crash replicas are either fully or strongly connected.

Under condition (1), it is clear that every replica inside $Q$ *will* receive $proposal$s sent from the replicas that are also inside $Q$; the binary consensus instances for such proposals will output 1s. On the other hand, replicas inside $Q$ will *not* receive $proposal$s sent from the replicas that are outside $Q$; the binary consensus instances for such proposals will output 0s. In either case, the binary consensus instances are guaranteed to terminate in two message delays. The replicas outside $Q$ might stay in the "while" loop of the binary consensus, but they will eventually receive the DECIDE messages and catch up to the replicas inside $Q$.

Under condition (3), the binary consensus is unlikely to terminate in two message delays, but it will eventually terminate and safety will be guaranteed. Notice that condition (3) is a nightmare for all consensus protocols, so we leave it as it is.

The performance of Aurora under condition (2) can be as worse as under condition (3), because when a replica $\mathsf{R}_i$ broadcasts a $proposal_i$, only its direct neighbors can receive $proposal_i$ within a timeout. We aim to optimize this so that *all* replicas in $Q$ can receive $proposal_i$ within a timeout, then the performance of Aurora under condition (2) will be as good as that under condition (1). To this end, we borrow idea from the gossip protocol [37]: when $\mathsf{R}_j$ receives $proposal_i$, it broadcasts it again; to avoid flooding, $\mathsf{R}_j$ broadcasts the hash of $proposal_i$ instead of $proposal_i$ itself; replica who receives the hash but did not receive $proposal_i$ will pull $proposal_i$ from $\mathsf{R}_j$. Of course, when broadcasting the hash, $\mathsf{R}_j$ can exclude the replicas that it has received the same hash from. When the timeout allows any replica inside the strongly connected $Q$ to receive proposals within the timeout, our optimization can make a quorum of strongly connected replicas behave like a quorum of fully connected replicas. For example, to ensure the optimization is typically effective, we can set the timeout to be $n$ times the message delay (measured under a stable network) between two connected replicas.

This optimization also needs to be applied to the STATE and VOTE messages inside RBC (but no need to broadcast the hash as the STATE and VOTE messages are small); it ensures that replicas inside $Q$ will receive the 1s (sent from replicas inside $Q$) before receiving the potential 0s (sent from the replicas outside $q$).

### E. Workflow of Aurora

Figure 1 depicts the workflow from $\mathsf{R}_1$'s point of view:

❶ $\mathsf{R}_1$ retrieves the top-$m$ requests from its priority queue $PQ_1$ and broadcasts them in a batch $proposal_i$.
❷ Meanwhile, $\mathsf{R}_1$ collects proposals from other replicas: when it receives a proposal from $\mathsf{R}_j$, it sets $BI_1[j] := 1$.
❸ They run $n$ instances of RBC in batch; $\mathsf{R}_1$ uses $BI_1[j]$ as the input for the $j$-th RBC instance.
❹ $\mathsf{R}_1$ uses $BO_1$ to record the output for each RBC instance.

❺ $R_1$ decides which proposals should be committed according to $BO_1$. In this example, $proposal_1$, $proposal_2$, and $proposal_5$ should be committed.

❻ $R_1$ treats $proposal_4$ as the proposal sent by $R_4$ in the next round.

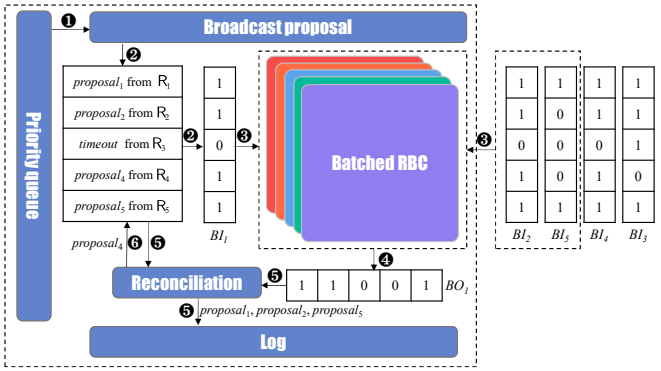



Fig. 1. Workflow of Aurora from $R_1$'s point of view.

## V. Aurora in Detail

In this section, we present Aurora in greater details.

### A. Message handler

A client sends a request to a designated replica. If it receives no response within timeout, it re-sends it to another replica. Algorithm 2 shows how a replica $R_i$ in Aurora handles messages that trigger consensus. It roughly works in the following three cases:

- Upon receiving a request, $R_i$ pushes it to $PQ_i$, which is used to store pending requests, i.e., requests that have not been stored in the log yet (Line 10-11). After gathering at least $m$ requests, $R_i$ picks the top-$m$ requests from $PQ_i$ and broadcasts them in a batch as $proposal_i$ to all replicas; this action brings $R_i$ to the consensus stage (Line 17-20).
- When other replicas receive $proposal_i$, they will also move on to the consensus stage. In addition, they will broadcast the hash of $proposal_i$ to ensure that other replicas that are not connected with $R_i$ can also receive $proposal_i$ (Line 26-30).
- Replicas who received the hash of $proposal_i$ will check whether they have already received the proposal. If so, they simply ignore this message. Otherwise, they will pull $proposal_i$ from the sender of the hash, and move on to the consensus stage. They will also forward the hash to other replicas (Line 36-40).

We introduce the term "run" to represent a run of the entire protocol, starting with every replica making proposals and ending with a decision being made. Each run has a *run_id* (Line 6), which is concatenated to each *proposal* (Line 14), making the proposals different. If a replica does not hear a positive decision on its proposal by the end of the current run, it will make the same proposal in the next run. We achieve this through the "*propose_on*" flag (Line 5): a replica can make a proposal only when this flag is 1. Furthermore, replicas will accept a proposal as long as it was not committed, even if its *run_id* is lower than the current run (Line 25 and 35). In this way, even if a replica has a slow connection to others, its proposals will eventually be received and committed.

**Algorithm 2:** Message handler - $R_i$

```
1  Local Variables:
2      Log_i                                              ▷ local log
3      PQ_i                                               ▷ priority queue
4      consensus_on := 0          ▷ whether R_i is in a consensus
5      propose_on := 1            ▷ whether R_i can make proposals
6      run_id := 0                                        ▷ The run id
7      BI_i := [0, ..., 0]
8      proposals_i := []
9
10 Upon receiving request from a client:
11     PQ_i.push(request)
12     if |PQ_i| ≥ m AND propose_on = 1 then
13         propose_on := 0
14         req := top-m requests in PQ_i
15         proposal_i := (req, run_id, i)
16         BI_i[i] := 1;  proposals_i[i] := proposal_i
17         Broadcast(proposal_i)
18         if consensus_on = 0 then
19             consensus_on := 1
20             Aurora_main()
21         end
22     end
23 end
24
25 Upon receiving proposal_j from R_j:
26     if consensus_on = 0 and proposal_j was not committed
         then
27         consensus_on := 1
28         BI_i[j] := 1
29         proposals_i[j] := proposal_j
30         Broadcast(H(proposal_j))                       ▷ exclude R_j
31         Aurora_main()
32     end
33 end
34
35 Upon receiving H(proposal_k) from R_j:
36     if consensus_on = 0 and proposal_k was not committed
         then
37         consensus_on := 1
38         pull proposal_k from R_j
39         BI_i[k] := 1;  proposals_i[k] := proposal_k
40         Broadcast(H(proposal_k))          ▷ exclude R_j and R_k
41         Aurora_main()
42     end
43 end
```

### B. Proposal exchange

Notice that replicas might send proposals simultaneously. To include all these proposals into consensus, we introduce a "proposal exchange" phase in the beginning of the consensus stage (Line 1-19 in Algorithm 3). Namely, each $R_i$ sets a timer and collects proposals in a similar way as Algorithm 2 until timeout or having received $n$ proposals. After collecting the proposals, they start to run the batched RBC to agree on which requests to be inserted into the log.

**Algorithm 3:** Aurora_main - $\mathsf{R}_i$

```
1  Start timer
2  repeat
3      Upon receiving proposal_j from R_j:
4          if proposal_j was not committed then
5              BI_i[j] := 1
6              proposals_i[j] := proposal_j
7              Broadcast(H(proposal_j))                ▷ exclude R_j
8          end
9      end
10
11     Upon receiving H(proposal_k) from R_j:
12         if BI_i[k] = 0 and proposal_k was not committed
             then
13             pull proposal_k from R_j
14             BI_i[k] := 1
15             proposals_i[k] := proposal_k
16             Broadcast(H(proposal_k)) ▷ exclude R_j and R_k
17         end
18     end
19 until timeout or BI_i[l] = 1 ∀ 1 ≤ l ≤ n;
20
21 BO_i := BatchedRBC(BI_i)
22
23 for k := 1 to n do
24     if BO_i[k] = 1 and proposals_i[k] = ⊥ then
25         pull proposal_k from other replicas
26         proposals_i[k] := proposal_k
27     end
28 end
29
30 Deduplicate(proposals_i)
31 commit proposals_i ▷ add the requests in proposals_i to Log_i
32 for k := 1 to n do
33     if BO_i[k] = 1 then
34         proposals_i[k] := ⊥
35         BI_i[k] := 0
36     end
37 end
38 consensus_on := 0
39 run_id := run_id +1
40 if BO_i[i] = 1 then
41     propose_on := 1
42 end
```

### C. Batched RBC

Algorithm 4 shows the details of the batched RBC. It takes $BI_i$ (indicators for whether $\mathsf{R}_i$ has received a proposal in the current run) as the input states and outputs $BO_i$ (indicators for whether a proposal should be inserted into the log).

Recall that, in RBC, replicas first exchange STATE messages containing a state bit of either 0 or 1, and decide their votes; then, they exchange VOTE messages containing a vote of 0, 1, or ?. The goal of our batched RBC is to run $n$ RBC instances in a single batch. To this end, we batch $n$ state bits into a single STATE message and batch $n$ votes into a single VOTE message. In this way, we can significantly save the bandwidth without affecting the overall latency. This is because the latency introduced by the binary consensus component is determined by the slowest RBC instance even when there is no batching. Specifically, replicas wait for all RBC instances to terminate before proceeding to step 5, as shown in Figure 1.

**Algorithm 4:** BatchedRBC - $\mathsf{R}_i$

```
   Input: BI_i
   Output: BO_i
1  states_i := BI_i; votes_i := []
2  while true do
3      Broadcast(STATE, r, run_id, states_i)        ▷ r-th round
4      repeat
5          Upon receiving a STATE message M from R_j:
6              if this is the first time received M then
7                  Broadcast(M))                    ▷ exclude R_j
8              end
9          end
10     until receiving ≥(f + 1) round-r STATE messages;
11     for k := 1 to n do
12         if (decided, b) appears at the k-th slot of any
             received states_j then
13             votes_i[k] := (decided, b)
14         else if a value b appears ≥(f + 1) times at the k-th
             slot of all received states_j then
15             votes_i[k] := b
16         else
17             votes_i[k] :=?
18         end
19     end
20     Broadcast(VOTE, r, run_id, votes_i)
21     repeat
22         Upon receiving a VOTE message M from R_j:
23             if this is the first time received M then
24                 Broadcast(M))                    ▷ exclude R_j
25             end
26         end
27     until receiving ≥(f + 1) round-r VOTE messages;
28     for k := 1 to n do
29         if (decided, b) appears at the k-th slot of any
             votes_j or a non-? value b appears ≥(f + 1) times
             at the k-th slot of all received votes_j then
30             states_i[k] := (decided, b)
31         else if a non-? value b appears at the k-th slot of
             any received votes_j then
32             states_i[k] := b
33         else
34             states_i[k] := CommonCoinFlip(r)
35         end
36         r := r + 1
37     end
38     if all n slots in states_i have "decided" flags then
39         Broadcast(DECIDE, run_id, states_i)
40         Return states_i
41     end
42 end
43 /* Executing in background */
44 Upon receiving a DECIDE message M from R_j:
45     Broadcast(M)                                 ▷ exclude R_j
46     Return states_j
47 end
```

In more detail, each replica $\mathsf{R}_i$ maintains two arrays: $states_i$ (initialized with $BI_i$) and $votes_i$ (initially empty). In the first round, $\mathsf{R}_i$ broadcasts $states_i$ in a STATE message and waits for the STATE messages from other replicas (Line 3-10). Upon receiving a STATE message, $\mathsf{R}_i$ forwards it to other replicas, for a similar reason as forwarding the proposals. However, this time, $\mathsf{R}_i$ does not need to forward the hash because the STATE message is small (only $3n$ bits). After receiving at least $(f+1)$

STATE messages[1], $\mathsf{R}_i$ assigns values to $votes_i$ based on the received $states$:

- First of all, once the replicas have reached a consensus for any RBC instance, they will add a flag "*decided*" to the corresponding slot in $states$; after seeing this flag, $\mathsf{R}_i$ simply follows the decision and assigns the decided value (together with the flag) to the corresponding slot in $votes_i$ (Line 13).
- If a value $b$ appears at least $(f+1)$ times at the $k$-th slot of all received $states$, $\mathsf{R}_i$ assigns $b$ to $votes_i[k]$ (Line 15).
- For other slots, $\mathsf{R}_i$ assigns ? to the corresponding slots of $votes_i$ (Line 17).

Next, $\mathsf{R}_i$ broadcasts $votes_i$ in a VOTE message, and waits for and forwards the VOTE messages of other replicas (Line 20-27). After receiving at least $(f+1)$ VOTE messages, $\mathsf{R}_i$ decides $states_i$ for the next round based on the received $votes$:

- If any slot has been decided with a non-? value $b$ (has a "$decided$" flag, or there are at least $(f+1)$ replicas vote for $b$), $\mathsf{R}_i$ assigns $(decided, b)$ to the corresponding slot in $states_i$ (Line 30).
- If a non-? value $b$ appears at the $k$-th slot of any received $votes$ (the quorum intersection argument on state messages guarantees that there is at most one non-? value), $\mathsf{R}_i$ assigns $b$ to $states_i[k]$ (Line 32).
- For other slots, replicas flip a common coin to decide the states for the next phase (Line 34).

If all slots have been decided, $\mathsf{R}_i$ broadcasts $states_i$ in a DECIDE message and terminates the batched RBC (Line 38-41). Other replicas who receive the DECIDE message will also terminate the batched RBC (Line 44-47).

### D. Reconciliation

After terminating the batched RBC, $\mathsf{R}_i$ will put the agreed requests into $Log_i$. More specifically, $\mathsf{R}_i$ first goes over all slots in $proposals_i$: if the corresponding RBC instance for a slot returns 1 but $\mathsf{R}_i$ has not received the proposal yet, it needs to pull the proposal from others (Line 23-28 in Algorithm 3). As different replicas may propose duplicate requests, $\mathsf{R}_i$ needs to deduplicate the remaining requests in $proposals_i$ before adding them to $Log_i$ (Line 30-31 in Algorithm 3). In the end of the consensus stage, $\mathsf{R}_i$ resets $BI_i$, $proposals_i$, *consensus_on, propose_on*, and enters the next run (Line 32-42 in Algorithm 3).

We remark in the end that, similar to Rabia [16], Aurora supports a simple mechanism for log compaction. We refer to [16] for more details.

## VI. Safety and Liveness

In this section, we prove the correctness (i.e., safety and liveness) of Aurora. Recall that replicas commit proposals according to the output $BO$ of the batched RBC, which implies that the safety of Aurora is completely due to the safety of the batched RBC (Algorithm 4).

[1]Recall that RBC (Algorithm 1) requires receiving $(n-f)$ STATE messages. As we assume $n = 2f+1$, it is equal to receiving $(f+1)$ STATE messages.

### A. Safety

Next, we prove the safety of Aurora by showing that the batched RBC satisfies safety (i.e., agreement and weak validity). We say "a non-? value $b$ is *r-locked* for the $k$-th RBC instance" if every replica $\mathsf{R}_i$ starts round-$r$ in Algorithm 4 with $states_i[k]$ set to $b$.

**Lemma 1.** *In the same round, it is impossible for two replicas to vote differently for the same RBC instance.*

*Proof.* The proof is by contradiction. Suppose $\mathsf{R}_i$ and $\mathsf{R}_j$ vote for 0 and 1 respectively for the $k$-th RBC instance in round-$r$. Then, $\mathsf{R}_i$ must have received at least $(f+1)$ $states$ with the $k$-th slot as 0. Similarly, $\mathsf{R}_j$ must have received at least $(f+1)$ $states$ with the $k$-th slot as 1. As there are $2f+1$ replicas in total, there must be at least one replica that has sent different $states$ to $\mathsf{R}_i$ and $\mathsf{R}_i$ (quorum intersection argument). This is impossible as we assume replicas can only fail by crashing (cf. Section III-C) □

**Lemma 2.** *If a replica decides $b$ for the $k$-th RBC instance in round-$r$, then $b$ is (r+1)-locked for the $k$-th RBC instance.*

*Proof.* Suppose $\mathsf{R}_i$ decides $b$ for the $k$-th RBC instance in round-$r$ (Line 30 in Algorithm 4). There are following cases:

1) "$b$ appears $\geq(f+1)$ times at the $k$-th slot of all received $votes$".
2) "$(decided, b)$ appears at the $k$-th slot of any received $votes_j$".

In case (1), there must be at least $(f+1)$ replicas vote for $b$ for the $k$-th RBC instance. Then, any replica $\mathsf{R}_j$ that starts round-$(r+1)$ must have received at least one vote for $b$ for the $k$-th RBC instance due to the quorum intersection argument (recall that $\mathsf{R}_j$ is required to receive at least $(f+1)$ $votes$ to move on to the next round). Furthermore, by Lemma 1, $\mathsf{R}_j$ will not receive any votes for $(1-b)$ for the $k$-th RBC instance in round-$r$. As a result, $\mathsf{R}_j$ will set $states_i[k]$ to $b$ in Line 32 (Algorithm 4) in round-$r$ and start round-$(r+1)$ with $states_i[k] = b$.

In case (2), $\mathsf{R}_j$ must have decided $b$ for the $k$-th RBC instance in round-$r'$ with $r' < r$. Based on the discussion above for case (1), any replica $\mathsf{R}_l$ entering round-$(r'+1)$ will have $states_l[k] = b$, and the same for round-$(r+1)$. □

**Lemma 3.** *If a value $b$ is $r$-locked for the $k$-th RBC instance, then any replica reaching Line 28 (Algorithm 4) in round-$r$ will decide $b$ in round-$r$.*

*Proof.* Suppose $b$ is $r$-locked for the $k$-th RBC instance. Then, the $k$-th slots of all $states$ received in Line 5 (Algorithm 4) of round-$r$ are with $b$. As a result, all replicas reaching Line 20 (Algorithm 4) will vote for $b$ for the $k$-th RBC instance. Any replica $\mathsf{R}_i$ reaching Line 28 (Algorithm 4) in round-$r$ must have received at least $(f+1)$ $votes$ and all the $k$-th slots are with $b$. Then, $\mathsf{R}_i$ will decide $b$ for the $k$-th instance in round-$r$. □

**Lemma 4.** *If a replica decides $b$ for the $k$-th instance in round-$r$, then any replica reaching Line 28 (Algorithm 4) in round-$(r+1)$ will decide $b$ for the $k$-th instance in round-$(r+1)$.*

*Proof.* By Lemmas 2 and 3. □

**Theorem 1** (Agreement)**.** *If two replicas* $\mathsf{R}_i$ *and* $\mathsf{R}_j$ *decide* $b$ *and* $b'$ *for the* $k$*-th RBC instance in rounds* $r$ *and* $r'$ *respectively, then* $b = b'$*.*

*Proof.* Suppose two replicas $\mathsf{R}_i$ and $\mathsf{R}_j$ decide $b$ and $b'$ for the $k$-th RBC instance in rounds $r$ and $r'$ respectively. There are following two cases:

1) $r = r'$. If $\mathsf{R}_i$ decides $b$ for the $k$-th instance, then it must have received at least one vote for $b$. Similarly, $\mathsf{R}_j$ must have received at least one vote for $b'$. As this is in the same round, by Lemma 1, $b = b'$.
2) $r < r'$. Given that $\mathsf{R}_j$ decides in round-$r'$, it must have reached Line 28 in rounds $r+1, ..., r'$. If $\mathsf{R}_i$ decides $b$ in round-$r$, by Lemma 4, $\mathsf{R}_j$ will decide $b$ in round-$(r+1)$. That means $b' = b$.

□

Theorem 1 implies that replicas that have terminated the batched RBC will have the same log: if a RBC instance outputs 1, all replicas will store the corresponding requests in the current slot; otherwise, all of them will forfeit the slot.

**Theorem 2** (Weak validity)**.** *If a replica stores a request other than* $\perp$ *in its log, this request must be sent by a client.*

*Proof.* If $\mathsf{R}_i$ stores a request other than $\perp$ in its log, the corresponding RBC instance must have output 1. In Line 31 of Algorithm 3, $\mathsf{R}_i$ will add the requests in $proposals_i$ to $Log_i$, and all requests in $proposals_i$ were sent by clients. □

### *B. Liveness*

Next, we first show that the batched RBC terminates with probability 1. Then, we show how Aurora guarantees the liveness.

**Lemma 5.** *With probability 1, there is a round-*$r$ *where a non-? value* $b$ *is r-locked for the k-th RBC instance.*

*Proof.* There are following three cases:

1) In round-$(r-1)$, if all replicas execute Line 34 in Algorithm 4, they will start round-$r$ with $states[k]$ being set to the output of $\mathsf{CommonCoinFlip}(r-1)$, and the claim directly follows.
2) Similarly, if all replicas execute Line 32 in Algorithm 4, they will start round-$r$ with $states[k]$ being set to the same value $b$, and the claim follows.
3) It becomes complex when some replicas execute Line 32 and adopt $b$, while others execute Line 34 and adopt $b'$. Due to the properties of the common coin, the value it computes at a given round is independent from the values it computes at the other rounds. Thus, $b$ is equal to $b'$ with probability $p = 1/2$. Let $P(r)$ be the following probability:

$$P(r) = Pr[\exists r' : r' \leq r : b_{r'} = b'_{r'}],$$

where $b_{r'}$ denotes the value of $b$ in round-$r'$. We have

$$\begin{aligned} P(r) &= p + (1-p)p + ... + (1-p)^{r-1}p \\ &= 1 - (1-p)^r. \end{aligned}$$

As $\lim_{r \to +\infty} P(r) = 1$, the claim follows.

□

**Lemma 6.** *The batched RBC terminates with probability 1.*

*Proof.* By Lemmas 3 and 5, with probability 1, there is a round-$r$ where all correct replicas decide $b$ as long as they reach Line 28 (in Algorithm 4). Based on the assumption of asynchronous communication, they will eventually reach Line 28. That means the $k$-th RBC instance terminates in round-$r$.

As the above argument is for any RBC instance, then, with probability 1, there is a round-$r'$ where all RBC instances terminate (i.e., the batched RBC terminates). □

Next, we prove the liveness of Aurora.

**Theorem 3** (Liveness)**.** *A request sent by a client will eventually be executed.*

*Proof.* Recall that, in Aurora, a client re-sends a request to another replica when it receives no response within a timeout. Therefore, the client will eventually send this request to a non-crashed replica $\mathsf{R}_i$, which will batch them into a $proposal_i$ and send it to other replicas (Line 17 in Algorithm 2). Notice that the "re-send" mechanism also maintains the linearizability when clients switch between replicas. Specifically, it ensures that a replica always sends the pending requests before sending a new one. If a replica $\mathsf{R}_j$ receives $proposal_i$, it inputs 1 to the corresponding RBC instance; otherwise, it inputs 0. By Lemma 6, the batched RBC terminates with probability 1. If all correct replicas input 1, the batched RBC will output 1; otherwise, it may output 0 or 1. In the former case, the requests will be executed (Line 31 in Algorithm 3). In the later case, they will run the same consensus stage again for $proposal_i$ Line 32 to 42 in Algorithm 3. Based on the assumption of asynchronous communication, $proposal_i$ will eventually be received by all non-crash replicas. Recall that replicas will process a proposal as long as it was not committed, even if its *run_id* is lower than the current run. Then, they will input 1 to the corresponding RBC instance, which will output 1 and the requests will be executed. That is to say, a request sent by a client will eventually be executed. □

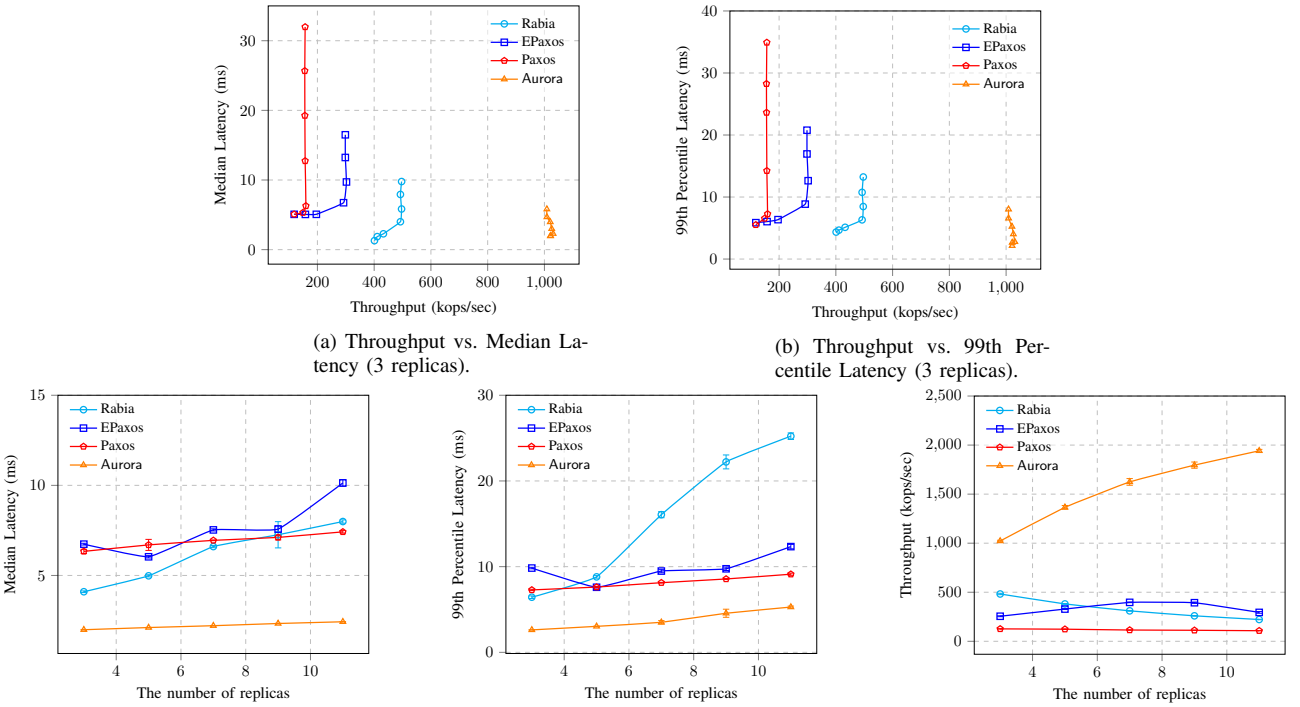
## VII. Evaluation

In this section, we systematically evaluate the performance of Aurora and compare it against three SMR systems:

- Rabia[2]: our closest competitor.
- Multi-Paxos[3] (with pipelining): the most common choice in production systems.
- EPaxos[3] (with pipelining): the state-of-the-art SMR system that achieves fast-path latency of two message delays.

We implement Aurora using the same programming language (Go 1.15.8) as these three implementations. We use SHA256 as the hash function. Additionally, we use the same replicated key-value store implementation as Rabia, which follows closely to the one from Multi-Paxos and EPaxos. Therefore, we can perform a fair comparison.

[2]https://github.com/haochenpan/rabia
[3]https://github.com/efficient/epaxos




(a) Throughput vs. Median Latency (3 replicas).

(b) Throughput vs. 99th Percentile Latency (3 replicas).

(c) Median Latency with the number of replicas.

(d) 99th Percentile Latency with the number of replicas.

(e) Throughput with the number of replicas.

Fig. 2. Performance in the same zone.

### A. Setting

Each replica runs on a separate AWS VM with four 3.1GHz vCPUs and 8 GB RAM, running Ubuntu Server 18.04.1 LTS 64. Following the setting of Rabia [16], the RTT is around 0.25ms within the same availability zone (us-east-1-b) and 0.4ms between multiple zones within the same region (us-east-1-b, us-east-1-c, us-east-1-d).

We generate closed-loop clients on a separate VM with 32 3.1GHz vCPUs and 128 GB RAM. In the EPaxos evaluations, we generate non-conflicting requests to achieve the maximal throughput for a fair comparison. The ratio of conflicting requests does not affect Aurora's throughput since it executes requests in total order. The size of a single request is 16B [38], [39]. Following the best practice in [14], [16], all systems use client batching size 10, i.e., each client batches 10 requests into a single request; EPaxos, Paxos, Rabia use proxy batching size 100, 500 and 20 respectively. In Aurora, We use the same proxy batching size as Rabia for a fair comparison. That means, in our benchmarks, EPaxos, Paxos, Rabia, Aurora respectively batch at most 1 000, 5 000, 200, 200 16B-requests in a *proposal*. Following prior work [14], [16], we set a timeout of 5ms for replicas to batch requests if the desired batch size is not reached.

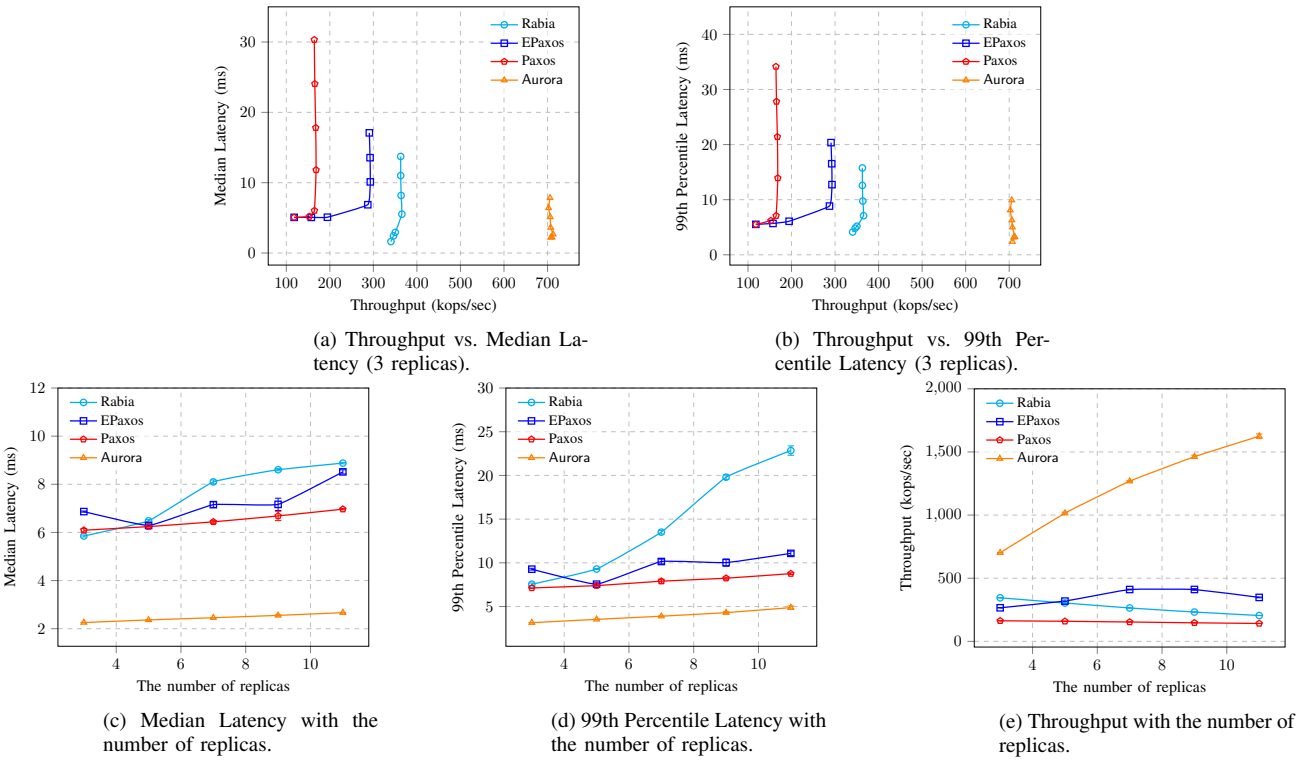
Throughput is measured in agreed requests per second, while latency quantifies the time taken for a single request to complete. The null slots do not contribute to the throughput as they do not contain any requests. All experiments are repeated 5 times and average values with error bars indicating standard deviations are reported.

### B. Performance in the same zone

We first measure their performance when all replicas are deployed in the same availability zone. We fix the number of replicas $n$ as three, increase the number of concurrent closed-loop clients, and measure throughput vs. latency (both median latency and 99th percentile latency). Figure $2a$ and $2b$ show the results with varying load sizes (the number of clients: 60, 80, 100, 200, 300, 400, 500) on each system.

Before saturation, Aurora and Rabia have smaller stable latency than Paxos and EPaxos, despite of their three-message delay. This is due to their higher throughput, making the request queuing time small. The stable median latency of Aurora is around 2× better than Rabia and 3× better than Paxos and EPaxos; its 99th percentile latency is around 3× better than Rabia and Paxos, and 4× better than EPaxos. The throughput of Aurora is around 2× better than Rabia, 3.5× better than EPaxos, and 6.4× better than Paxos. In particular, in the same availability zone with 3 replicas, Aurora can reach a throughput of 1 021 320 TPS, with a median latency around 2ms.

Next, we increase the number of replicas $n$ and measure the scalability of these SMR systems. Figure $2c$ $2d$ and $2e$ show the results. The 99th percentile latency of Rabia increases significantly with more replicas being added. Recall that Rabia requires as many replicas as possible to propose the same proposal; otherwise, it will not take the fast path or forfeit a slot, increasing the possibility of long tail latency and null slots. Specifically, three replicas allow Rabia to use the fast path 99.626% of the time and have only 0.3% null slots, compared to 90.498% and 5.9% with 11 replicas. In contrast,




(a) Throughput vs. Median Latency (3 replicas).

(b) Throughput vs. 99th Percentile Latency (3 replicas).

(c) Median Latency with the number of replicas.

(d) 99th Percentile Latency with the number of replicas.

(e) Throughput with the number of replicas.

Fig. 3. Performance in the multiple zones.

Aurora's latency plots are more flattened, because in Aurora a request can be processed as long as the proposal containing that request can be delivered. Three replicas allow Aurora to use the fast path 99.996% of the time and have only 0.012% null slots, compared to 99.962% and 0.033% with 11 replicas.

With the number of replicas increasing, EPaxos can handle more requests and its throughput increases. On the other hand, EPaxos needs to check request dependencies and the local computation becomes its bottleneck at some point, after which its throughput starts decreasing. In contrast, with the number of replicas increasing, Aurora can handle more requests without introducing more local computation, hence its throughput keeps increasing.

When *n=11*, the stable median latency of Aurora is around 4× lower than EPaxos and 3× lower than Rabia and Paxos; its 99th percentile latency is around 4.7× lower than Rabia, 1.7× lower than Paxos, and 2.3× lower than EPaxos. When considering throughput, the superiority of Aurora becomes more prominent. It achieves a throughput (1 941 720 TPS) that is around 6.6× higher than EPaxos, 18× higher than Paxos, and 8.7× higher than Rabia when *n=11*.

### C. Performance across multi-zones

Figure 3 shows the performance of all systems when the replicas are deployed across multiple availability zones within the same region. Paxos and EPaxos are barely impacted, while Aurora and Rabia are moderately impacted (around a 30% drop and a 26.4% drop).

When there are three replicas, the stable median latency of Aurora is around 2.5× better than Rabia and 3× better than Paxos and EPaxos; its 99th percentile latency is around 3× better than Rabia and Paxos, and 3.7× better than EPaxos. The throughput of Aurora is around 2× better than Rabia, 2.5× better than EPaxos, and 4.3× better than Paxos. In particular, in multi-zones with 3 replicas, Aurora can reach a throughput of 707 280 TPS, with a median latency around 2.2ms.

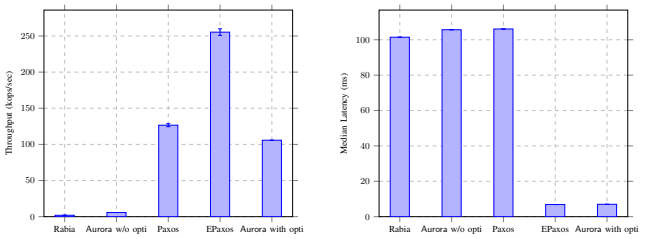
When adding more replicas, Aurora still performs the best among all four SMR systems. It achieves a throughput (1 624 480 TPS) that is around 5× higher than EPaxos, 11× higher than Paxos, and 8× higher than Rabia when *n=11*. For the same reason mentioned in Section VII-B, as the number of replicas increases, Rabia has a higher probability of forfeiting a slot and fewer opportunities to take a fast path compared to Aurora. These results can lead to an increasing waiting time for all the proposals in the local priority queue, resulting in a longer tail latency.

### D. Performance under poor networks with crash replicas.

The performance of both Aurora and Rabia depend on the underlying network: Rabia requires all non-crash replicas to propose the same batch of requests, and Aurora requires all non-crash replicas to receive the proposal. Next, we measure their performance under a poor network connection with crash replicas. We set up a cluster consisting of 5 replicas, 2 of which are crash; the rest 3 replicas are connected with each other by 3 bidirectional links, one of which was added with a 50ms delay (through NetEm[4]).

Figure 4 shows the performance of Aurora with and without optimization (cf. Section IV-D) compared with Rabia, Paxos and EPaxos when they are saturated. It shows that Aurora

[4]https://man7.org/linux/man-pages/man8/tc-netem.8.html




(a) Throughput of three systems. (b) Median Latency of three systems.

Fig. 4. Performance under poor networks with crash replicas.

(with optimization) has a massive performance drop under this setting, because the correct replicas need to always wait for the proposals from the crash replicas until a timeout (5ms). Nevertheless, it is still around 20× better than that w/o optimization and 50× better than Rabia in throughput, and it is around 20× better than both that w/o optimization and Rabia in latency. This is because both Aurora w/o optimization and Rabia need to wait 50ms to get the message from the slow link. Although Paxos and EPaxos achieve good performance due to their pipelining implementation, Aurora still has comparable throughput to Paxos and comparable latency to EPaxos.

### E. Integration with Redis

We integrate Aurora with Redis (dubbed RedisAurora) to perform evaluation for a real-world example. In RedisAurora, we utilize Redis native MGET and MSET commands to process requests (same as RedisRabia). We systematically evaluate the throughput of RedisAurora and compare it against three Redis-based systems: (i) synchronous-replication with one master and one replica (Sync-Rep(1)); (ii) synchronous-replication with one master and two replicas (Sync-Rep(2)); and (iii) RedisRabia [16]. All systems use a batching size of 200. We evaluate these systems by deploying them across three replicas (except for Sync-Rep(1), two replicas) in the same zone, and test their performance when they are saturated. Figure 5 shows the evaluation results.

In Sync-Rep(1) and Sync-Rep(2), the master makes choices and dictates them to the backup; the system's throughput is limited by the rate of state machine execution. As a result, the throughput of Sync-Rep(2) is 2× that of Aurora. The delay in completing slots introduced by the storage engine significantly reduces the throughput of Aurora and Rabia because they have not implemented pipelining. However, the throughput of Aurora is still 1.7 times higher than that of Rabia.

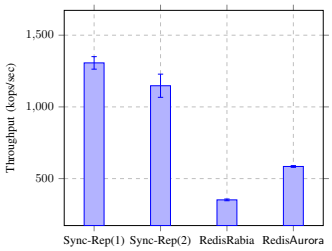



Fig. 5. Throughput of different Redis integrations.

### F. Performance Analysis

We analyze the complexity of Aurora and three other protocols (Multi-Paxos, EPaxos, and Rabia) as shown in Table I to better understand our protocol. From the table, Multi-Paxos and EPaxos have better results. However, both Rabia and Aurora outperform them in practice, which may seem counterintuitive. This is because the communication overhead is not a bottleneck with a small number of replicas[5] in a stable network.

The bottleneck of Multi-Paxos lies with the leader due to the unbalanced workload stemming from its leader-based design. As for EPaxos, the bottleneck is local computation because it needs to check all dependencies to ensure safety. Rabia achieves a higher performance by evenly distributing the workload to all replicas and avoiding costly local computations. Aurora shares the same advantages as Rabia but can agree on up to $n$ decisions per consensus round, compared to only 1 decision per consensus round in Rabia. Therefore, Aurora achieves even higher performance.

TABLE I
COMPLEXITIES.

| | **Multi-Paxos** | **EPaxos** | **Rabia** | **Aurora** |
|---|---|---|---|---|
| Message complexity | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(n^3)$ |
| Message delays | 2 | 2 | 3 | 3 |
| Decisions per round | 1 | $n$ | 1 | $n$ |

## VIII. CONCLUSION

In this paper, we propose a leaderless crash-fault tolerant SMR, which eliminates the need for a fail-over protocol. To demonstrate its scalability and robustness, we systematically evaluate its performance on a testbed consisting of 11 AWS VMs across multiple zones within the same region.

In future work, we plan to explore applying this idea in more challenging settings. For example, we will investigate how to make it tolerant to Byzantine faults and how to adapt it for WAN environments.

## REFERENCES

[1] L. Leslie, "The part-time parliament," *ACM Trans. on Computer Systems*, vol. 16, pp. 133–169, May 1998.
[2] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.
[3] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.
[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.
[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 143–157.
[6] "Rethinkdb 2.1: high availability," Accessed in May 2022, https://rethinkdb.com/blog/2.1-release/.
[7] "Redisraft: Strongly-consistent redis deployments," Accessed in May 2022, https://github.com/RedisLabs/redisraft.

[5]In crash-tolerant SMR protocols, the number of replicas beyond 7 is rarely seen in practice (maximum 9 or 11).

[8] "Replication layer — cockroachdb docs," Accessed in May 2022, https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html.

[9] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[10] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 564–577, July 2011.

[11] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 398–407.

[12] KyleKingsbury(Aphyr), "Redis-raft1b3fbf6," Accessed in May 2022, https://jepsen.io/analyses/redis-raft-1b3fbf6.

[13] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 369–384.

[14] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.

[15] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, "Speeding up consensus by chasing fast decisions," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 49–60.

[16] H. Pan, J. Tuglu, N. Zhou, T. Wang, Y. Shen, X. Zheng, J. Tassarotti, L. Tseng, and R. Palmieri, "Rabia: Simplifying state-machine replication through randomization," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 472–487.

[17] P. Tennage, C. Basescu, L. Kokoris-Kogias, E. Syta, P. Jovanovic, V. Estrada-Galinanes, and B. Ford, "Quepaxa: Escaping the tyranny of timeouts in consensus," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 281–297.

[18] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[19] L. Lamport, "Generalized consensus and paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005.

[20] L. J. Camargos, R. M. Schmidt, and F. Pedone, "Multicoordinated paxos," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 316–317.

[21] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, "Be general and don't give up consistency in geo-replicated transactional systems," in *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Cham: Springer, 2014, pp. 33–48.

[22] V. Enes, C. Baquero, A. Gotsman, and P. Sutra, "Efficient replication via timestamp stability," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 178–193.

[23] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai, "Sdpaxos: Building efficient semi-decentralized geo-replicated state machines," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 68–81.

[24] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar, "Wpaxos: Wide area network flexible consensus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 211–223, 2019.

[25] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, "State machine replication scalability made simple," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 17–33.

[26] M. Whittaker, N. Giridharan, A. Szekeres, J. M. Hellerstein, and I. Stoica, "Bipartisan paxos: A modular state machine replication protocol," *arXiv preprint arXiv:2003.00331*, 2020.

[27] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 27–30.

[28] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, "Randomized multivalued consensus," in *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*. IEEE, 2001, pp. 195–200.

[29] M. O. Rabin, "Randomized byzantine generals," in *24th annual symposium on foundations of computer science (sfcs 1983)*. IEEE, 1983, pp. 403–409.

[30] F. Pedone, A. Schiper, P. Urbán, and D. Cavin, "Solving agreement problems with weak ordering oracles," in *European Dependable Computing Conference*. Berlin, Heidelberg: Springer, 2002, pp. 44–61.

[31] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[32] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[33] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[34] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[35] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th symposium on principles of distributed computing*, 2020, pp. 129–138.

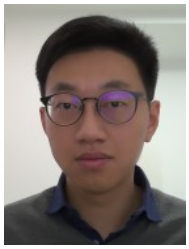
[36] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[37] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.

[38] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "Tao:facebook's distributed data store for the social graph," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 49–60.

[39] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 313–328.


**Hao Lu** is currently working toward the PhD degree with Zhejiang University, Hangzhou China. His research interests include blockchain, fault-tolerance consensus, and distributed systems.



**Jian Liu** is a ZJU100 Young Professor at Zhejiang University. Before that, he was a postdoctoral researcher at UC Berkeley. He got his PhD in July 2018 from Aalto University. His research is on Applied Cryptography, Distributed Systems, Blockchains and Machine Learning. He is interested in building real-world systems that are provably secure, easy to use and inexpensive to deploy.

**Kui Ren** is the director of Institute of Cyberspace Research of Zhejiang University, ACM/IEEE Fellow. He chaired and participated in projects from NSFC, NSF, US Department of Energy and Air Force Research Laboratory which cost more than $10 million. He was awarded Achievement Award of IEEE ComSoc, Sustained Achievement Award of exceptional scholar program of Suny Buffalo, and NSF youth Achievement Award. Prof. Kui Ren is internationally recognized for accomplishments in the areas of cloud security and wireless security. He has made many fundamental innovations in both theory and practice from the discipline to the society. His works have significant impact on emerging cloud systems and wireless network technologies. Prof. Kui Ren has published more than 200 papers, and cited more than 20 thousand times. His research results have been widely reported by Xinhua news agency, Scientific American, NSF news, ACM news and other media.