Cascade: Leaderless State-Machine Replication with High Throughput

Hao Lu
Zhejiang University
luhao@zju.edu.cn

Jian Liu[✉][*]
Zhejiang University
jian.liu@zju.edu.cn

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

Abstract

State-machine replication (SMR) allows a 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, i.e., 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 Cascade, with the following properties:

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

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

1 Introduction

State-machine replication (SMR) replicates a state machine across a set of *replicas*, and uses a *consensus* protocol to ensure that the service is available and consistent even 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 [22, 23], had been adopted in Chubby [8], Google Spanner [13] and Microsoft Azure Storage [10]; recent systems such as RethinkDB [3], Redis [1] and CockroachDB [2], chose Raft [31] 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 [4].
- *Codebase complexity.* Although the normal case is simple, the fail-over protocol is too complex to implement and debug [12]. For example, the fail-over protocol has introduced bugs to Redis [21] and RethinkDB [21] 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 [5, 16, 28, 30, 34]. However, replicas in such protocols need to keep track of request dependencies, which introduces new overhead for checking dependencies. Furthermore, this paradigm needs a fail-over protocol that is more complex than the single-leader consensus.

Leaderless SMR. In SOSP '21, Haochen et al. presented a leaderless SMR named Rabia [32] 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 three message delays. Haochen et al. claim that case (i) is common in a single datacenter

[*][✉]Jian Liu is the corresponding author

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 three 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 Cascade, which successfully overcomes the two limitations of Rabia: it goes beyond a single datacenter and scales to more replicas. The core idea of Cascade 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 Cascade, each replica proposes a batch of requests as well, but they can agree on upto n batches, and these requests are likely to be different as each replica serves as a proxy for different clients. As a result, Cascade could potentially achieve a throughput that is n times higher than Rabia. Furthermore, we borrow idea from the gossip protocol to make three message delays to be the common case even under a poor network connection. We provide a full-fledged implementation of Cascade and systematically evaluate its performance. Our experimental results show that Cascade achieves a peak throughput around two million TPS, up to 8.7 $\times$ higher than Rabia.

Organization. In the remainder of this paper, we first provide necessary background for this paper in Section 2 and provide a design roadmap for Cascade in Section 3. Then, we describe Cascade in greater details in Section 4 and prove its safety and liveness in Section 5. Next, we extensively evaluate its performance and compare it to the state-of-the-art SMR systems in Section 6. In the end, we survey related work in Section 7 and conclude the paper in Section 8.

2 Preliminaries

2.1 State-machine replication

State-machine replication (SMR) allows a state machine to be replicated across *n replicas* and handle clients' requests as a single machine, even in the presence of f faulty replicas. In more details, (i) each replica locally maintains a log that is divided into slots; (ii) clients submit requests to replicas and wait for responses; (iii) replicas run a *consensus* protocol to agree on the order of requests to be stored in the log; (iv) replicas execute the operations in the requests slot by slot; and (v) replicas respond to the corresponding clients with the execution results.

The core component of a SMR system is its consensus protocol, i.e. step (iii) above, which ensures 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 [18]. As a result, most existing SMR systems [22, 31] ensure both safety and liveness only when the network is stable (i.e., synchronous); they do not ensure liveness when the network becomes asynchronous. Such systems rely on a leader to order the requests and coordinate the protocol. This design places a disproportionately high load on the leader, inevitably impairing the scalability. Furthermore, this design relies on a *fail-over* protocol to tolerate a failed leader, but the fail-over protocol is too complex to implement and debug [12]. For example, the fail-over protocol has introduced bugs to Redis [21] and RethinkDB [3].

2.2 Randomized binary consensus

Another way to circumvent the FLP impossibility result is to allow probabilistic termination. For example, in the *randomized binary consensus* (RBC) proposed by Ben-Or [6], the probability for a replica to terminate approaches 1 as time proceeds. 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). 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 R_i has received (Line 6 in Algorithm 1); otherwise, 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, R_i can safely terminate and output b (Line 14 in Algorithm 1).

- 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), 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 "?", 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*.

Algorithm 1: RBC - R_i

```
Input: A binary value $b_i$
Output: A binary value of consensus
$state \leftarrow b_i$
while $true$ do
    Broadcast(STATE, $r$, $state$)        ▷ $r$ is the round number
    Wait until receiving $\geq n-f$ round-$r$ STATE messages
    if value $b$ appears $\geq \lfloor \frac{n}{2} \rfloor + 1$ times then
        $vote \leftarrow b$
    else
        $vote \leftarrow ?$
    end
    Broadcast(VOTE, $r$, $vote$)        ▷ $vote$ can be 0, 1 or ?
    wait until receiving $\geq n-f$ round-$r$ VOTE messages
    if a non-? value $b$ appears $\geq f+1$ times then
        Broadcast(DECIDE, $b$)
        return $b$        ▷ output $b$ and terminate
    else if a non-? value $b$ appears at least once then
        $state \leftarrow b$
    else
        $state \leftarrow$ CommonCoinFlip($r$)
    end
    $r \leftarrow r+1$        ▷ proceed to next round
end
/* executing in background */
Upon receiving a DECIDE message:
    Broadcast(DECIDE, $b$)
    return $b$
end
```

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.

2.3 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 [32], 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.

We assume that, at any time in the execution, an adversary can dynamically select which replica crashes and which message delivers. However, as mentioned above, messages sent to a non-crash replica will eventually be received and there will be at most f crash replicas. Prior work shows that such an adversary can significantly slow down the fail-over protocol (if any) [4].

3 Design Roadmap

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

3.1 Rabia revisit

In Rabia [32], 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, 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 PQs are non-empty.

During a consensus round of Rabia, each 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, 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 2.2), the consensus in Rabia is guaranteed to terminate in three message delays when either of the following conditions is satisfied:

1. *all* (non-crash) replicas 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 three 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.

3.2 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 (non-crash) replicas propose the same request", which is only true within a single datacenter. In Cascade, we go one step further: we have replicas agree on *whether a certain replica has proposed requests or not*.

In Cascade, each 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 3.3). If 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 three message delays when either of the following conditions is satisfied:

1. *all* (non-crash) replicas 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 input the same request, hence Cascade goes beyond a single datacenter and can be deployed across multiple zones. Furthermore, batching will not introduce any side-effect in Cascade, because we do not require replicas to propose the same batch of requests.

3.3 To be more scalable

During a consensus round of Cascade, each replica R_i locally maintains a vector BI_i of input bits: if R_i has received a proposal from 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 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 Cascade, each replica proposes a batch of requests as well, but they can agree on upto n batches, and these requests are likely to be different as each replica serves as a proxy for different clients. As a result, Cascade could potentially achieve a throughput that is n times higher than Rabia.

3.4 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 Cascade more robust. For simplicity, we say R_i and 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 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 consensus instances for such proposals will output 0s. In either case, the consensus instances are guaranteed to terminate in three 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 consensus is unlikely to terminate in three 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 Cascade under condition (2) can be as worse as under condition (3), because when a replica 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 Cascade under condition (2) will be as good as that under condition (1). To this end, we borrow idea from the gossip protocol [14]: when R_j receives $proposal_i$, it broadcasts it again; to avoid flooding, 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 R_j. Of course, when broadcasting the hash, R_j can exclude the replicas that it has received the same hash from. Suppose the timeout is larger than n times of the message delay between two connected replicas, our optimization is able to make a quorum of strongly connected replicas behave like a quorum of fully 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).

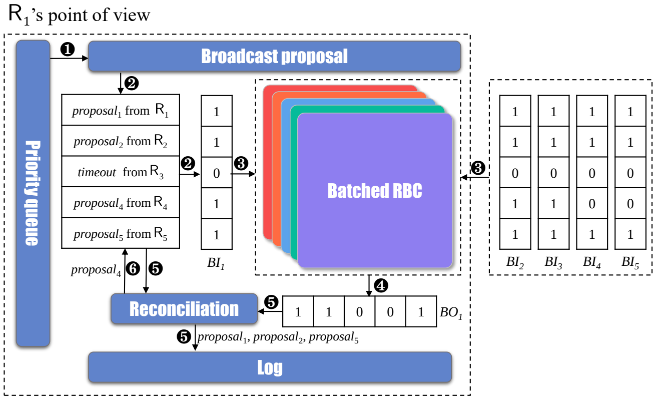


Figure 1: Workflow of Cascade from R$_1$'s point of view.

3.5 Workflow of Cascade

Figure 1 depicts the workflow from R$_1$'s point of view:

❶ R$_1$ retrieves the top-m requests from its priority queue PQ_1 and broadcasts them in a batch $proposal_i$.

❷ Meanwhile, R$_1$ collects proposals from other replicas: when it receives a proposal from R$_j$, it sets $BI_1[j] := 1$.

❸ They run n instances of RBC in batch; R$_1$ uses $BI_1[j]$ as the input for the j-th RBC instance.

❹ 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. Notice that this is crucial for the protocol to be live.

4 Cascade in Detail

In this section, we present Cascade in greater details.

4.1 Message handler

A client sends a request to a designated replica. If it receives no response within timeout, it re-sends it to all replicas. Algorithm 2 shows how a replica R$_i$ in Cascade 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.

4.2 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.

4.3 Batched RBC

Algorithm 4 shows the details of the batched RBC. It takes BI_i (indicators for whether 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).

Algorithm 2: Message handler - R_i

```
Local Variables:
    $Log_i$                                         ▷ local log
    $PQ_i$                                          ▷ priority queue
    $consensus\_on := 0$                            ▷ whether $\mathsf{R}_i$ is in a consensus
    $propose\_on := 1$                              ▷ whether $\mathsf{R}_i$ can make proposals
    $run\_id := 0$                                  ▷ The run id
    $BI_i := [0,...,0]$
    $proposals_i := []$

Upon receiving request from a client:
    $PQ_i$.push($request$)
    if $|PQ_i| \geq m$ AND $propose\_on = 1$ then
        $propose\_on := 0$
        $proposal_i \leftarrow$ (top-$m$ requests in $PQ_i$) || $run\_id$ || $i$
        $BI_i[i] := 1$; $proposals_i[i] := proposal_i$
        Broadcast($proposal_i$)
        if $consensus\_on = 0$ then
            $consensus\_on := 1$
            Cascade_main()
        end
    end
end

Upon receiving proposal_j from $\mathsf{R}_j$:
    if $consensus\_on = 0$ and $proposal_j$ was not committed
     then
        $consensus\_on := 1$
        $BI_i[j] := 1$
        $proposals_i[j] := proposal_j$
        Broadcast($H(proposal_j)$)                  ▷ exclude $\mathsf{R}_j$
        Cascade_main()
    end
end

Upon receiving $H(proposal_k)$ from $\mathsf{R}_j$:
    if $consensus\_on = 0$ and $proposal_k$ was not committed
     then
        $consensus\_on := 1$
        pull $proposal_k$ from $\mathsf{R}_j$
        $BI_i[k] := 1$; $proposals_i[k] := proposal_k$
        Broadcast($H(proposal_k)$)         ▷ exclude $\mathsf{R}_j$ and $\mathsf{R}_k$
        Cascade_main()
    end
end
```

Algorithm 3: Cascade_main - R_i

```
Start timer
repeat
    Upon receiving proposal_j from $\mathsf{R}_j$:
        if $proposal_j$ was not committed then
            $BI_i[j] := 1$
            $proposals_i[j] := proposal_j$
            Broadcast($H(proposal_j)$)              ▷ exclude $\mathsf{R}_j$
        end
    end

    Upon receiving $H(proposal_k)$ from $\mathsf{R}_j$:
        if $BI_i[k] = 0$ and $proposal_k$ was not committed then
            pull $proposal_k$ from $\mathsf{R}_j$
            $BI_i[k] := 1$
            $proposals_i[k] := proposal_k$
            Broadcast($H(proposal_k)$) ▷ exclude $\mathsf{R}_j$ and $\mathsf{R}_k$
        end
    end
until timeout or $BI_i[l] = 1 \; \forall \; 0 \leq l < n$;

$BO_i \leftarrow$ BatchedRBC($BI_i$)

for $k$ := 1 to $n$ do
    if $BO_i[k] = 1$ and $proposals_i[k] = \perp$ then
        pull $proposal_k$ from other replicas
        $proposals_i[k] := proposal_k$
    end
end

Deduplicate($proposals_i$)
commit $proposals_i$: add the requests in $proposals_i$ to $Log_i$
$BI_i := [0,...,0]$
for $k$ := 1 to $n$ do
    if $BO_i[k] = 1$ then
        $proposals_i[k] := \perp$
    end
end
$consensus\_on := 0$
$run\_id := run\_id + 1$
if $BO_i[i] = 1$ then
    $propose\_on := 1$
end
```

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.

In more detail, each replica R_i maintains two arrays: $states_i$ (initialized with BI_i) and $votes_i$ (initially empty). In the first round, 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, R_i forwards it to other replicas, for a similar reason as forwarding the proposals. However, this time, 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], R_i assigns values to $votes_i$

[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.

```
Algorithm 4: BatchedRBC - 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 $\geq(f+1)$ 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 $\geq(f+1)$ 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 $\geq(f+1)$ 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 $\geq(f+1)$ 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
```

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, 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*, R_i assigns b to $votes_i[k]$ (Line 15).

- For other slots, R_i assigns ? to the corresponding slots of $votes_i$ (Line 17).

Next, 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, 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), 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), 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, 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).

4.4 Reconciliation

After terminating the batched RBC, R_i will put the agreed requests into Log_i. More specifically, R_i first goes over all slots in $proposals_i$: if the corresponding RBC instance for a slot returns 1 but 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, 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, 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 [32], Cascade supports a simple mechanism for log compaction. We refer to [32] for more details.

5 Safety and Liveness

In this section, we prove the correctness (i.e., safety and liveness) of Cascade, which is completely due to the correctness of the batched RBC (Algorithm 4). We say "a non-? value b is *r-locked* for the k-th RBC instance" if every replica R_i starts round-r in Algorithm 4 with $states_i[k]$ set to b.

5.1 Safety

We first show that the batched RBC satisfies safety (i.e., agreement and weak validity).

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 R_i and R_j vote for 0 and 1 respectively for the k-th RBC instance in round-r. Then, R_i must have received at least $(f+1)$ *states* with the k-th slot as 0. Similarly, 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 R_i and R_i (quorum intersection argument). This is impossible as we assume replicas can only fail by crashing (cf. Section 2.3) □

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 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 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 R_j is required to receive at least $(f+1)$ *votes* to move on to the next round). Furthermore, by Lemma 1, R_j will not receive any votes for $(1-b)$ for the k-th RBC instance in round-r. As a result, 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), 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 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 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, 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 R_i and 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 R_i and 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 R_i decides b for the k-th instance, then it must have received at least one vote for b. Similarly, 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 R_j decides in round-r', it must have reached Line 28 in rounds $r+1, ..., r'$. If R_i decides b in round-r, by Lemma 4, 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 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, R_i will add the requests in $proposals_i$ to Log_i, and all requests in $proposals_i$ were sent by clients. □

5.2 Liveness

Next, we show that the batched RBC terminates with probability 1.

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). □

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

Proof. Recall that, in Cascade, a client will eventually send requests to a non-crashed replica R_i, which will batch them into a $proposal_i$ and send it to other replicas. If a replica 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. In the later case, they will run the same consensus stage again for $proposal_i$. 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. □

6 Evaluation

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

- Rabia [32]: our closest competitor.
- Multi-Paxos [12] (with pipelining): the most common choice in production systems.
- EPaxos [30] (with pipelining): the state-of-the-art SMR system that achieves fast-path latency of two message delays.

For the above SMR systems, we reproduced the results reported in the paper of Rabia [32], i.e., we run the same code[2] in a similar setting as described in their paper. We also run our Cascade implementation in this setting to provide a fair comparison.

6.1 Implementation

We fully implemented Cascade in Go (version 1.15.8). Our implementation closely follows the pseudo-code in Section 4 and it consists of around 2,100 lines of Go code. We use Sha256 as the hash function and use go-channels for handling connections.

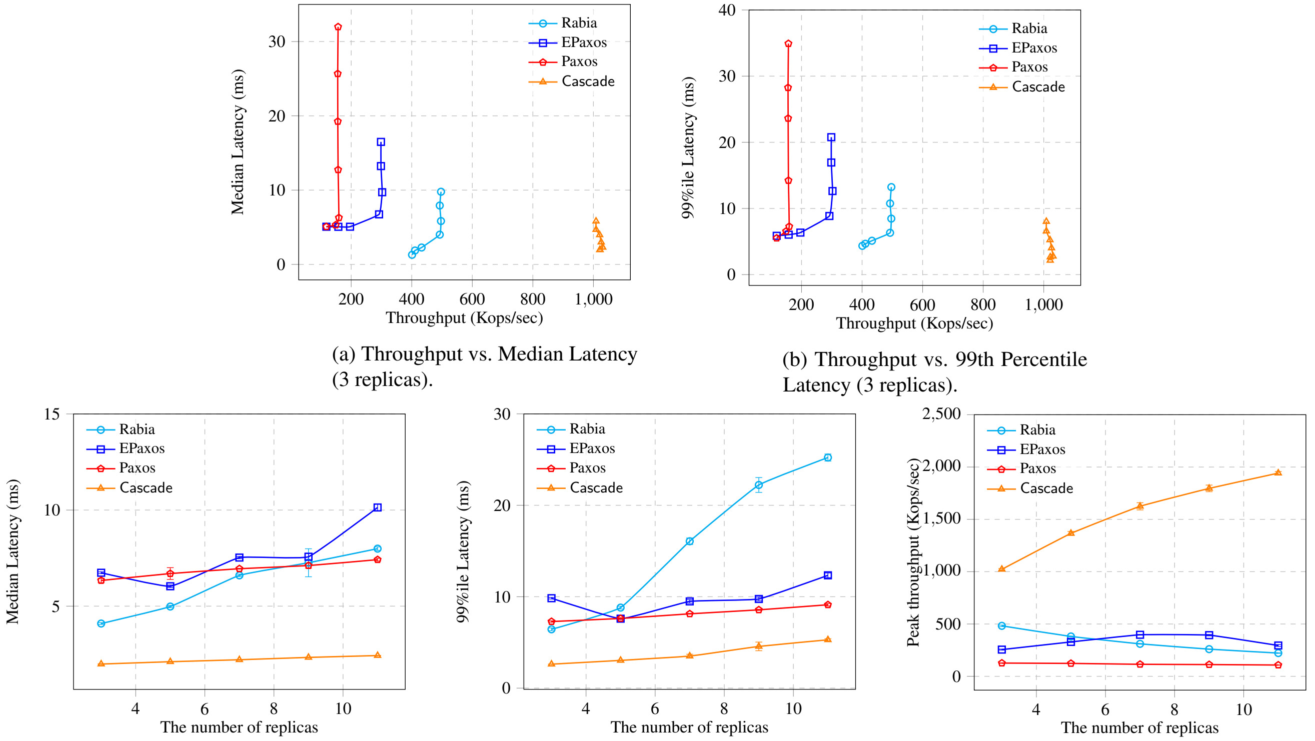
Following prior work [19, 35], we implement the common coin (Line 34 in Algorithm 4) using a random binary generator $G()$ with the same seed s across all replicas. Namely, for each round r, each replica locally runs $b \leftarrow G(s||r)$, which guarantees that all replicas will get the same random number in round-r.

6.2 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 [32], the RTT is around 0.25ms within the same availability zone and 0.4ms between multiple zones in the same region.

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 Cascade's throughput since it executes requests in total order. The size of a single request is 16B [7, 26]. Following [32], 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 Cascade, We use the same proxy batching size as Rabia. That means, in our benchmarks, EPaxos, Paxos, Rabia, Cascade respectively batch at most

[2] The code of Rabia, Multi-Paxos, and EPaxos can be found in https://github.com/haochenpan/rabia.



(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.

Figure 2: Performance in the same zone.

1 000, 5 000, 200, 200 16B-requests in a *proposal*. Following prior work [30, 32], we set a timeout of 5ms for replicas to batch requests if the desired batch size is not reached.

All experiments are repeated 5 times and average values with error bars indicating standard deviations are reported.

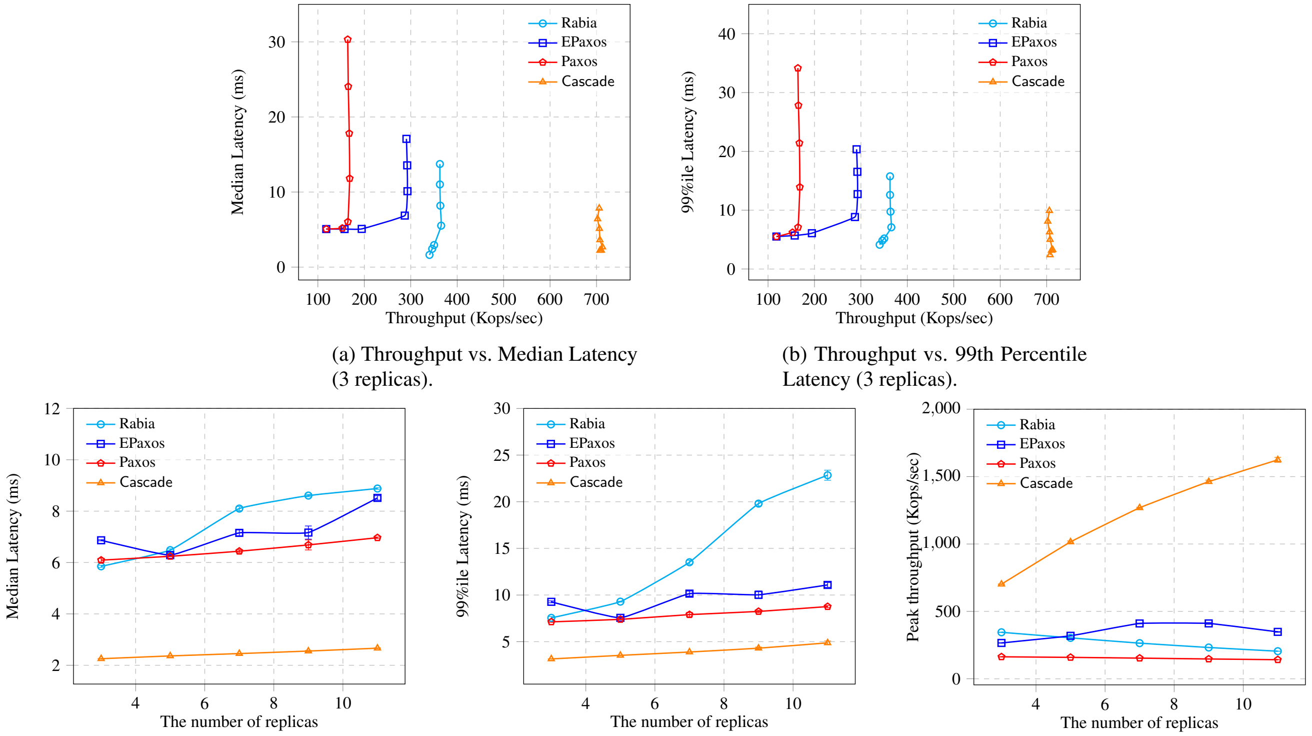
6.3 Performance in the same zone

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 (60-500), and measure throughput vs. latency (both median latency and 99th percentile latency). Figure 2*a* and 2*b* show the results with varying load sizes (the number of clients: 60, 80, 100, 200, 300, 400, 500) on each system. The performance results of Rabia, Paxos and EPaxos are consistent with the results reported in [32] (slightly better due to better CPUs and network connections). As pointed in [32], when the systems are saturated, the 99th percentile latency becomes larger due to the increasing chances of NULL slots. This also happens for Cascade.

Before saturation, Cascade 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 Cascade 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 peak throughput of Cascade 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, Cascade can reach a peak 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 2*c* 2*d* and 2*e* 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. 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, Cascade's latency plots are more flattened, because in Cascade a request can be processed as long as the proposal containing that request can be delivered. Three replicas allow Cascade 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



(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.

Figure 3: Performance in the multiple zones.

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, Cascade can handle more requests without introducing more local computation, hence its throughput keeps increasing.

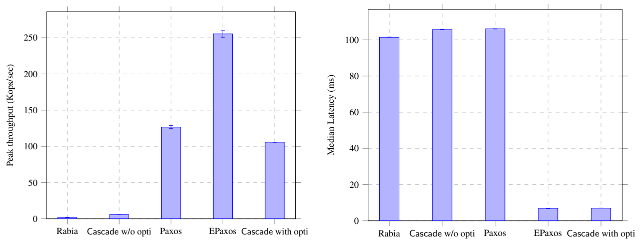
When *n=11*, the stable median latency of Cascade is around 4× lower than EPaxos and 3× lower than Rabia and Paxos; its 99th percentile latency is around 4.7× lower than Rabia, 1.7× lower than Paxos, and 2.3× lower than EPaxos. When considering throughput, the superiority of Cascade becomes more prominent. 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*.

6.4 Performance across multi-zones

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

When there are three replicas, the stable median latency of Cascade 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 peak throughput of Cascade 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, Cascade can reach a peak throughput of 707 280 TPS, with a median latency around 2.2ms.

When adding more replicas, Cascade 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*.



(a) Throughput of three systems.

(b) Median Latency of three systems.

Figure 4: Performance under failures (5 replicas, 2 of which are crash).

6.5 Performance under failures.

The performance of both Cascade and Rabia depend on the underlying network: Rabia requires all non-crash replicas to propose the same batch of requests, and Cascade 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 50ms delay (through NetEm[3]).

Figure 4 shows the performance of Cascade with and without optimization (cf. Section 3.4) compared with Rabia, Paxos and EPaxos when they are saturated. It shows that Cascade (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 Cascade 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, Cascade still has comparable throughput to Paxos and comparable latency to EPaxos.

6.6 Integration with Redis

We integrate Cascade with Redis (dubbed RedisCascade) to perform evaluation for a real-world example. In RedisCascade, we utilize Redis native MGET and MSET commands to process requests (same as RedisRabia). We systematically evaluate the throughput of RedisCascade 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 [32]. 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 Cascade. On the other hand, the throughput of Cascade is 1.7× higher than that of Rabia.

7 Related Work

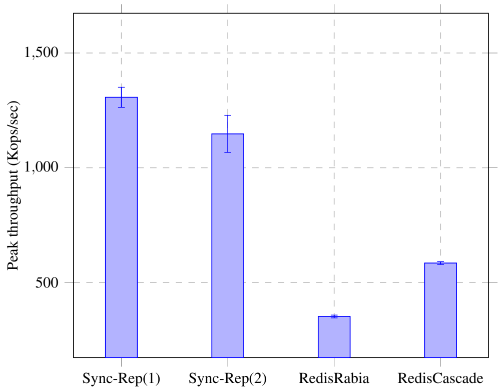


Figure 5: Throughput of different Redis integrations.

Leaderbased SMR. In 1998, Lamport presented Paxos [22, 23], which is the first work in this line of research. Paxos requires three phases with five message delays. Multi-Paxos [12] 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. Fast Paxos [25] 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 [24] 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. Multicoordinated Paxos [11] improves availability by allowing clients to send their requests to a quorum of replicas. Raft [31] 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 [28], every replica acts as a leader for the sequence numbers assigned to it. For example, 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 [30] 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. M^2Paxos [34] gets rid of the dependency graph by assigning different objects to different replicas and enforcing requests accessing the same objects to be ordered by the same replica. However, a request

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

may access objects maintained by different replicas, hence M^{2}Paxos still needs to resolve conflicts. Based on the observation that concurrent failures in geo-distributed systems are rare, Enes et al. presented Atlas [16], which is a multi-leader SMR trading off fault tolerance for scalability. The size of the fast quorum Q depends on f: $Q = \lfloor n/2 \rfloor + f$.

Leaderless SMR. Ben-Or's randomized binary consensus [6] is the main component for constructing a leaderless SMR. Ezhilchelvan et al. [17] use a common coin [35] to reduce the average number of message delays and allow proposers to propose arbitrary values. Pedone et al. [33] exploit the weakly ordering guarantees from the network layer. Cachin et al. [9] 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 [15, 20, 27, 29].

8 Conclusion

In this paper, we propose Cascade, a crash-fault tolerant SMR. It is leaderless, hence completely get rid of the fail-over protocol. It can scale to a large number of replicas and behave well even under a poor network connection. We provide a full-fledged implementation of Cascade and systematically evaluate its performance on a testbed consisting of 11 AWS VMs. Our experimental results show that Cascade achieves a throughput that is up to 8.7× higher than the state-of-the-art leaderless SMR.

References

[1] Redisraft: Strongly-consistent redis deployments, Accessed in May 2022. https://github.com/RedisLabs/redisraft.

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

[3] Rethinkdb 2.1: high availability, Accessed in May 2022. https://rethinkdb.com/blog/2.1-release/.

[4] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, July 2011.

[5] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60. IEEE, 2017.

[6] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30. ACM, 1983.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: facebook's distributed data store for the social graph. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60. USENIX Association, 2013.

[8] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350. USENIX Association, 2006.

[9] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[10] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 143–157. ACM, 2011.

[11] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 316–317, 2007.

[12] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 398–407. ACM, 2007.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[14] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, page 1–12, New York, NY, USA, 1987. Association for Computing Machinery.

[15] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.

[16] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 24:1–24:15. ACM, 2020.

[17] Paul Ezhilchelvan, Achour Mostefaoui, and Michel Raynal. Randomized multivalued consensus. In *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*, pages 195–200. IEEE, 2001.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

[19] Roy Friedman, Achour Mostefaoui, and Michel Raynal. Simple and efficient oracle-based consensus protocols for asynchronous byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, 2005.

[20] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. *Dumbo: Faster Asynchronous BFT Protocols*, page 803–818. Association for Computing Machinery, New York, NY, USA, 2020.

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

[22] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[23] Leslie Lamport. Paxos made simple, fast, and byzantine. In Alain Bui and Hacène Fouchal, editors, *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, volume 3 of *Studia Informatica Universalis*, pages 7–9. Suger, Saint-Denis, rue Catulienne, France, 2002.

[24] Leslie Lamport. Generalized consensus and paxos. 2005.

[25] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[26] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013.

[27] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 129–138, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[29] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[30] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[31] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[32] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 472–487, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Fernando Pedone, André Schiper, Péter Urbán, and David Cavin. Solving agreement problems with weak ordering oracles. In *European Dependable Computing Conference*, pages 44–61. Springer, 2002.

[34] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 156–167. IEEE Computer Society, 2016.

[35] Michael O Rabin. Randomized byzantine generals. In *24th annual symposium on foundations of computer science (sfcs 1983)*, pages 403–409. IEEE, 1983.