RMWPaxos: Fault-Tolerant In-Place Consensus Sequences

Jan Skrzypczak, Florian Schintke, and Thorsten Schütt

Abstract—Building consensus sequences based on distributed, fault-tolerant consensus, as used for replicated state machines, typically requires a separate distributed state for every new consensus instance. Allocating and maintaining this state causes significant overhead. In particular, freeing the distributed, outdated states in a fault-tolerant way is not trivial and adds further complexity and cost to the system. In this article, we propose an extension to the single-decree Paxos protocol that can learn a *sequence of consensus decisions* 'in-place', i.e., with a single set of distributed states. Our protocol does not require dynamic log structures and hence has no need for distributed log pruning, snapshotting, compaction, or dynamic resource allocation. The protocol builds a fault-tolerant atomic register that supports arbitrary read-modify-write operations. We use the concept of *consistent quorums* to detect whether the previous consensus still needs to be consolidated or is already finished so that the next consensus value can be safely proposed. Reading a consolidated consensus is done without state modifications and is thereby free of concurrency control and demand for serialisation. A proposer that is not interrupted reaches agreement on consecutive consensus decisions within a single message round-trip per decision by preparing the acceptors eagerly with the previous request.

Index Terms—Consensus, Paxos, atomic register, consistent quorum, fault-tolerance, data management

1 Introduction

State machine replication [1] is a common technique for implementing distributed, fault-tolerant services. Commonly, replicated state machine (RSM) implementations are centred around the use of a consensus protocol, as replicas must sequentially apply the same commands in the same order to prevent divergence.

Existing consensus protocols such as Paxos [2], [3], Raft [4], or variations thereof [5], [6], [7] that can be used to build an RSM are based on the idea of a command log. Once a replica learns one or multiple commands by consensus, it appends them to its persistent local command log. Several practical systems [8], [9], [10] follow this general approach.

However, the implementation of such a command log incurs additional challenges such as log truncation, snapshotting, and log recovery. In case of Paxos, these problems have to be addressed separately on top of the consensus algorithm. This is a challenging and error-prone task, as noted by Chandra *et al.* [11]. Other consensus solutions, e.g., Raft, consider some of these issues as part of the core protocol while sacrificing the ability to make consensus decisions without an elected leader. In either case, implementing consensus sequences requires extensive state management.

A command log is worth its overhead when the commands are small compared to the managed state. However, aggregating largely independent data into a bigger managed state, such as multiple key-value pairs in a key-value store, to compensate for the log overhead is counterproductive because the log would then unnecessarily order commands targeting different keys. Managing each key-value pair separately would be ideal, but this is unpractical when using a log due to the implied overhead and challenges.

In this paper, we present a novel approach called *Read-Modify-Write Paxos* (RMWPaxos) where the state of an RSM is managed 'in-place'. Instead of replicating a command log as an intermediate step, RMWPaxos replicates the latest state directly. A new command is processed by applying it to the current state and proposing the result as the next value in a sequence of consensus decisions. Thereby, it is possible to use a fixed set of state variables for all decisions, which avoids the state management issues. At the same time, distributed consensus can be used on a finer granularity than before and it becomes trivial to use an arbitrary number of parallel consensus instances. This allows the fault-tolerant implementation of ubiquitous primitives like counter, locks, or sets. In addition to existing use cases like key-value stores, we believe that such fault-tolerant, fine-granular RSM usage might become more and more relevant with the rise of byte-addressable non-volatile memory and RDMA-capable low latency interconnects.

Before presenting RMWPaxos, we introduce the notion of a *consensus sequence register*, an obstruction-free multi-writer, multi-reader register that performs any submitted write operation at-least-once. Writes are expressed in the form of update commands applied on an opaque object. Instead of explicitly agreeing on a sequence of commands, such register agrees on the sequence of object states that result from the submitted update commands. By adhering to the safety properties of consensus, reads are guaranteed to observe the latest consistent state. Strengthening the register to apply writes exactly-once

- *The authors are with the Zuse Institute Berlin, the Department of Distributed Algorithms, 14195 Berlin, Germany. E-mail: {skrzypczak, schintke, schuett}@zib.de.*

Manuscript received 6 Sept. 2019; revised 6 Mar. 2020; accepted 10 Mar. 2020. Date of publication 30 Mar. 2020; date of current version 15 May 2020. (Corresponding author: Jan Skrzypczak.)
Recommended for acceptance by P. Balaji.
Digital Object Identifier no. 10.1109/TPDS.2020.2981891

results in RMWPaxos—a fault-tolerant general atomic read-modify-write (RMW) register.

The main contributions of this paper are:

- We introduce the abstractions of a *consensus sequence register* and strengthen it to provide an *atomic RMW register*(Section 4). These abstractions can be used to implement RSMs. If updates are idempotent the *consensus sequence register* suffices to build an RSM. Otherwise, the atomic RMW register is required (Section 5.6).
- We provide a new implementation of a fault-tolerant atomic *write-once* register by modifying the Paxos algorithm. In particular, we enhance Paxos by using the concept of *consistent quorums*—a set of replies containing identical answers—to reduce contention in read-heavy workloads. Once a consistent quorum is detected, the consensus decision is known. This allows learning the register's value in two message delays and prevents concurrent reads from blocking each other (Section 5.4).
- By further exploiting consistent quorums, we extend the atomic write-once register to a multi-write register that is a *consensus sequence register*. Here, a consistent quorum indicates the most recent consensus decision. This makes it possible to propose a follow-up value in-place, i.e., without a command log or multiple independent consensus instances (Section 5.4). If there is only a single writer, follow-up decisions can be made in two message delays (Section 5.8) without electing a leader.
- The *consensus sequence register* applies submitted updates from multiple writers *at-least* once, which is sufficient when updates manipulate the opaque object (or parts of it) in an idempotent way (like adding a member to a set). We show that by using ordered links, *exactly-once* semantics can be achieved to build an *atomic RMW register*, called RMWPaxos (Section 5.5).

2 System Model

We consider an asynchronous distributed system with processes that communicate by message passing. Processes work at arbitrary speed, may crash, omit messages and may recover with their internal state intact (a recovering process is indistinguishable from one experiencing omission failures). We do not consider Byzantine failures. A process is *correct* if it does not crash or recovers from crashes in finite time with its (possibly outdated) state intact. We assume that every process can be identified by its process ID (PID).

In the first part of this paper, processes send messages to each other via direct unreliable communication links. Links may lose or delay messages indefinitely or deliver them out-of-order. While a fair-loss property [12] is desirable to support progress, it is not formally necessary. In Section 5.5, we strengthen this and require reliable in-order message delivery. Such reliable links can easily be constructed on top of unreliable fair-loss links [12]. In practice, TCP is often used as reliable communication protocol [6].

3 The Consensus Problem

The consensus problem describes the agreement of several processes on a common value in a distributed system. We differentiate between ***proposer*** processes that propose values and ***learner*** processes that must agree on a single proposed value. In practice, a process can also implement both roles. A correct solution to the consensus problem must satisfy the following ***safety*** properties [13]:

C-Nontriviality. Any learned value must have been proposed.

C-Stability. A learner can learn at most one value.

C-Consistency. Two different learners cannot learn different values.

In addition to safety, the ***liveness*** property requires that some value is eventually learned if a sufficient number of processes are correct. However, guaranteeing liveness while satisfying the safety properties of consensus is impossible in an asynchronous system with one faulty process [14].

4 Problem Statement

We define a fault-tolerant register that is replicated on N processes and tolerates the crashes of a minority of replicas. The register holds a value v. Its initial value is $v = \perp$. Any number of clients can read or modify v by submitting *commands* to any replica. The primary motivation of our work is to provide a register abstraction that allows the implementation of a replicated state machine. For that, we start with a simpler abstraction, which we then extend.

Write-Once Atomic Register. Commands submitted to the register either write a value or read its current value. Read commands return either $\perp$ or a value v_w that has been submitted by some write. The register is linearisable [15], i.e., all commands appear to take effect instantaneously at some time between their submission and the corresponding completion response from the register. Thus, once a read returns v_w, then all subsequent reads must return v_w as well. However, an arbitrary number of reads is allowed to return $\perp$ beforehand if no value was written yet. This is achieved by satisfying the safety properties stated in Section 3.

Consensus Sequence Register. We extend the write-once atomic register by allowing multiple clients to submit *update* commands that change the register's value. We say that a value v is the result of *update sequence* $s(v) = u_1, \ldots, u_n$, iff v equals $u_n \circ \ldots \circ u_1$ applied on $\perp$ ($\circ$ being function composition). The register ensures that reads return values with growing update sequences. For that, we extend the safety properties of consensus for consensus sequences.

CS-Nontriviality. Any read value is the result of applying a sequence of submitted updates.

CS-Stability. For any two subsequent reads returning values v_1 and v_2, $s(v_1)$ is a prefix of $s(v_2)$.

CS-Consistency. For any two reads (including concurrent ones) returning values v_1 and v_2, $s(v_1)$ is a prefix of $s(v_2)$ or vice versa.

The prefix relation on update sequences is reflexive. Every update sequence is its own prefix. Update sequences are merely a tool to argue about the register's properties. The actual register implementation does not explicitly store them. It simply keeps the value resulting from the latest update.

For updates, we also require the following properties:

CS-Update-Visibility. Any completed update is included at least once in the update sequence of all values returned by subsequent reads.

CS-Update-Stability. For any two subsequent updates u_1 and u_2, u_1 appears before u_2 in the update sequence of any returned value that includes both u_1 and u_2.

Atomic Read-Modify-Write Register. To satisfy linearisability, we strengthen CS-Update-Visibility by requiring that every completed update is included *exactly-once* in the update sequence of all values returned by subsequent reads. This results in a general atomic read-modify-write (RMW) register [16]. Unlike specialised RMW registers that can perform a single type of RMW operation like test-and-set or fetch-and-add, this register can atomically execute arbitrary computations on its previous value.

As liveness is impossible in our system model, wait-freedom [17] cannot be provided. However, we require obstruction-freedomness [18] for a valid implementation of the registers. If wait-freedom is still required, an obstruction-free implementation can be extended by a leader oracle assuming a $\mathcal{W}$ failure detector [19].

5 In-Place Consensus Sequence

In this section, we present our protocols that satisfy the properties of the register abstractions introduced in Section 4. The write-once atomic register makes use of the principles of Paxos consensus [2], [3] and adopts the concept of *consistent quorums* [20]. These concepts are then extended for the more powerful abstractions to allow a sequence of multiple consensus decisions 'in-place', i.e., on the same set of state variables by overwriting the previous consensus. A more detailed, albeit more informal description of a previous version is given by Skrzypczak [21]. We discuss how to build an RSM with our register in Section 5.6.

5.1 Paxos Overview

Our approach is derived from the Paxos protocol. In addition to proposers and learners, Paxos introduces the role of ***acceptor*** processes that coordinate concurrent proposals by ***voting*** on them. If a sufficient number of acceptors have voted for the same proposal, the proposal's value can be learned by a learner. Such a set of acceptors is called a ***quorum***. A proposal is ***chosen*** if it has acquired a quorum of votes. The value of a chosen proposal is a chosen value. The size of quorums depends on the application and Paxos variant in use [7], [22], [23]. However, it is generally required that any two quorums have a non-empty intersection to prevent two disjoint quorums that voted for different values (as this would allow two learners to learn different values).

For Paxos to learn a value, a quorum of acceptors, a learner, and the proposer that has proposed the value, must be correct during the execution of the protocol. For simplicity, we consider any majority of acceptors to be a quorum. Thus, a system with $2F+1$ acceptors can tolerate at most F acceptor failures.

If enough processes are correct, then Paxos is obstruction-free [18], i.e., an isolated proposer without concurrent access succeeds in a finite number of steps. However, concurrent proposals can invalidate each other repeatedly, thereby preventing learners from learning any value. This scenario is known as ***duelling proposers***.

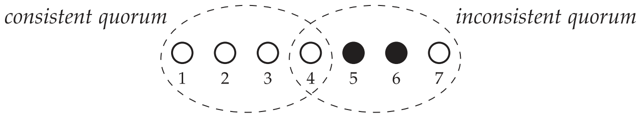


Fig. 1. Consistent/inconsistent quorum with 7 acceptors. A quorum view Q for a system using n acceptors consists of $|Q| = \lfloor \frac{n}{2} \rfloor + 1$ elements (here 4).

5.2 Consistent Quorums

Similar to Paxos, our approach structures the communication between proposers and acceptors into phases. In each phase, a proposer sends a message to all acceptors and waits for a minimal quorum of replies. The seen quorum is ***consistent*** if the indicated state by the acceptors in the quorum is identical, otherwise, it is ***inconsistent*** (see Fig. 1). Not waiting for more replies than necessary ensures tolerating a minority of failed acceptors without delaying progress.

If a proposer p observes an inconsistent quorum, it cannot infer which of the seen values is or will be chosen and learned. For example, if p receives the quorum depicted in the right part of Fig. 1, it cannot decide if $\bullet$ or $\circ$ exists in a majority since it has no information about the state of acceptors 1–3. In contrast, it is trivial for p to deduce the chosen value with a consistent quorum (Fig. 1 left). Existing Paxos variants do not distinguish consistent or inconsistent quorums. As we will see, detecting a consistent quorum allows the proposer to terminate the protocol early in the single-decree case. Furthermore, the consistent state can be used as the basis for follow-up proposals if multiple consensus decisions are needed in sequence.

5.3 Paxos Write-Once Atomic Register

In the following, we present our modifications to the original single-decree Paxos protocol for implementing a write-once atomic register. Its pseudocode is depicted in Algorithm 1. We note that no separate learner role exists, as each proposer also implements the functionality of a learner in our implementation. To make the algorithm easier to understand, we provide an execution example in Fig. 2. We discuss differences to Paxos in Section 5.3.2. Before proceeding to the algorithm description, we first cover some general concepts and conventions.

Rounds. Concurrent proposals are ordered by so-called ***rounds*** (analogue to 'proposals numbered n' in [2] and 'ballot numbers' in [3]). A round is a tuple (n, id), where n is a non-negative integer and id some globally unique identifier. Rounds are partially ordered. $r_1 < r_2$ iff $r_1.n < r_2.n$. Furthermore, $r_1 = r_2$ iff $r_1.n = r_2.n \land r_1.id = r_2.id$. Newer proposals are indicated by higher rounds. Rounds with the same n but different id cannot be ordered.

Acceptor State. Acceptors act as the distributed, fault-tolerant storage. Each acceptor manages three values (cf. Algorithm 1, line 24): (1) the highest round r_{ack} it has acknowledged, (2) the last value val it has voted for, and (3) round r_{voted} in which the proposal including the value was proposed in. By acknowledging a round, acceptors promise not to vote for lower-numbered proposals in the future.

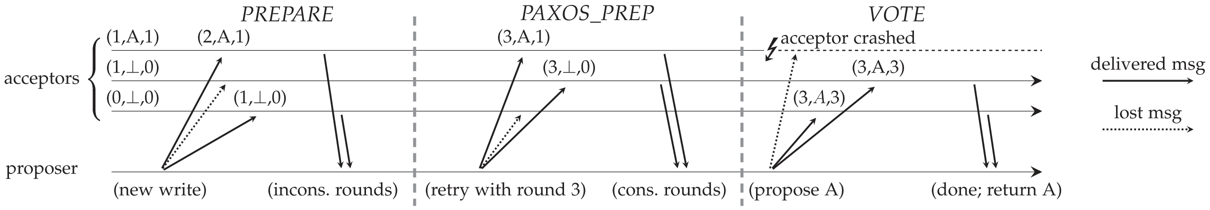


Fig. 2. Example message exchange of a write, starting with inconsistent acceptor states. Time moves from left to right. Acceptor states are shown as (r_{ack}, val, r_{voted}). Round IDs are omitted for simplicity.

Algorithm 1 Paxos-based write-once atomic register

```
--------------- Proposer: Phase 1 ---------------
1:  on receive $\langle REQ, op = write|read, val\rangle$ from client:
2:      Store $op$ and $val$
3:      $r_{id} \leftarrow$ globally unique ID
4:      send $\langle PREPARE, op, r_{id}\rangle$ to all acceptors
5:  on receive $\langle ACK, inc, r_{ack}, r_{voted}, v\rangle$ from quorum $Q$:
--------------- Phase 2 ---------------
6:      if $t \leftarrow cons_Q(r_{voted}) \wedge (t.n > 0 \vee op = read)$ then
7:          ▷ return existing consensus
8:          send $\langle DONE, v\rangle$ to client
9:      else if $r \leftarrow cons_Q(r_{ack}) \wedge inc$ then
10:         ▷ r consistently prepared
11:         $v_{prop} \leftarrow max_Q(r_{voted}, v)$
12:         if $v_{prop} = \bot$ then
13:             ▷ no consensus yet; propose own value
14:             $v_{prop} \leftarrow val$
15:         ▷ propose value
16:         send $\langle VOTE, r, v_{prop}\rangle$ to all acceptors
17:     else    ▷ inconsistent quorum; retry with higher round
18:         $r_{new} \leftarrow max_Q(r_{ack})$
19:         $r_{new}.n \leftarrow r_{new}.n + 1$
20:         send $\langle PAXOS\_PREP, r_{new}\rangle$ to all acceptors
21: on receive $\langle VOTED, v\rangle$ from quorum $Q$:
22:     send $\langle DONE, v\rangle$ to client

--------------- Acceptor ---------------
23: initialise:
24:     $r_{ack} \leftarrow (0, \bot), val \leftarrow \bot, r_{voted} \leftarrow (0, \bot)$
--------------- Phase 1 ---------------
25: on receive $\langle PREPARE, op, r_{id}\rangle$ from proposer $p$:
26:     ▷ incremental phase 1
27:     if $op = write$ then
28:         $r_{ack} \leftarrow (r_{ack}.n + 1, r_{id})$
29:         send $\langle ACK, true, r_{ack}, r_{voted}, val\rangle$ to $p$
30:     else
31:         ▷ no state change for read
32:         send $\langle ACK, false, (r_{ack}.n, r_{id}), r_{voted}, val\rangle$ to $p$
33: on receive $\langle PAXOS\_PREP, r\rangle$ from proposer $p$:
34:     ▷ canonical Paxos phase 1
35:     if $r > r_{ack}$ then
36:         $r_{ack} \leftarrow r$
37:         send $\langle ACK, true, r_{ack}, r_{voted}, val\rangle$ to $p$
--------------- Phase 2 ---------------
38: on receive $\langle VOTE, r, val_{new}\rangle$ from proposer $p$:
39:     if $r \geq r_{ack}$ then
40:         $r_{ack} \leftarrow r$        ▷ improve $r_{ack}$ consistency
41:         $r_{voted} \leftarrow r$
42:         $val \leftarrow val_{new}$
43:         send $\langle VOTED, val\rangle$ to $p$
```

Pseudocode Conventions. For brevity's sake, we use the following conventions when handling sets of reply messages: Let a process receive a set of reply messages S. Each message in S is an n-tuple denoted as $\langle t, e_1, \ldots, e_{n-1}\rangle$. We make use of pattern matching techniques commonly found in functional programming. The type t of the message is matched to ensure it has the correct format. Its payload is stored in tuple elements e_1 to e_{n-1}. Since messages in S may hold different values in the same tuple element, we define the following functions: $cons_S(e_i)$ returns the value of e_i if it is equal for all messages in S, or $false$ otherwise; $max_S(e_i)$ returns the largest value of e_i; $max_S(e_i, e_j)$ returns the value of e_j from the message with the largest value of e_i.

We furthermore assume that processes can keep track of multiple concurrent requests and know to which outstanding request a received reply belongs.

5.3.1 Protocol Description

The protocol has two phases. In the first phase, a proposer checks for concurrently proposed values and prepares acceptors to deny outdated proposals. In the second phase, a proposer proposes either its own or a value seen in the first phase. To eventually learn a value, both phases must be passed without interruption by other proposers.

The protocol begins with proposer p receiving a request from a client (line 1). The request is either a $write$ that tries to set the register to a value val, or a $read$ that returns the register's current value (here, $val = \bot$). The request of the client is handled asynchronously. The client will be notified by a $DONE$ message once the request has been processed.

Proposer p starts the first phase by choosing a round ID and sending it in a $PREPARE$ message along with the request type to all acceptors (lines 2–4). Any acceptor $\mathcal{A}$ that receives a $write$ request from p acknowledges this by incrementing r_{ack} and updating its ID. Thereby, $\mathcal{A}$ promises p to not vote for any lower-numbered proposals in the future (lines 28–29). If $\mathcal{A}$ received a $read$ request, then it does not increment r_{ack} as p does not intend to modify the register's value by submitting a proposal. Letting the state untouched when processing reads reduces their interference with other ongoing requests and is not part of canonical Paxos.

After processing the request, $\mathcal{A}$ replies with its current state and indicates if its r_{ack} round was incremented (lines 29, 32). The second phase begins as soon as p has received replies from a quorum Q of acceptors. Depending on the replies, p proceeds in one of the following ways:

1) If all acceptors in Q have voted for the same proposal (same r_{voted}), then p knows that consensus was already reached and that the proposal's value is chosen. Thereby, p has learned the register's value and returns it to the client. Similarly, p can be certain that consensus was not reached if no acceptor in Q has voted for any proposal yet. Thus, it can return an empty value if it is processing a $read$ (lines 6–8).
2) If all acceptors in Q incremented their rounds and responded with a consistent r_{ack} round, then p can propose a value. If at least one of the acceptors has

voted for a past proposal, p receives an inconsistent quorum as shown in Fig. 1. It cannot decide if the proposal's value is already established or not. In order to not violate safety, p must propose the value seen in the highest round. If no acceptor has voted for any proposal yet, p can choose its own value. The proposal is sent in a *VOTE* message to all acceptors using the acknowledged round r_{ack} (lines 9–16).

3) In all other cases, p has to retry the first phase. This happens if acceptors are currently in an inconsistent state, e.g., because of an ongoing proposal, lost messages, or a crashed proposer. As p has already knowledge about the current state of the acceptors, it can choose an explicit round number that is higher than all rounds observed so far, which is then included in *PAXOS_PREP* messages (lines 17–20). An example of this is depicted in Fig. 2.

Each acceptor that has received a proposal by p (case (2)), votes for the proposal if they have not given a promise for a higher round during a (concurrent) phase 1 and notifies p of its vote (lines 40–43). Otherwise, the acceptor ignores the proposal or may optionally notify p that its proposal is outdated (not shown). Once p has received a quorum of positive replies, it knows that its proposed value is chosen and notifies the client on the established consensus (line 22). This concludes the protocol.

5.3.2 Comparison to Paxos

Our write-once atomic register is based on the same mechanism for safety as Paxos, but differs from the canonical single-decree Paxos [2] in several aspects:

Consistent Quorums. In canonical Paxos, all proposals must complete both phases of the algorithm even if a value was already chosen. This effectively serialises concurrent reads and causes unnecessary state changes in acceptors (their round numbers). Our protocol, instead, terminates early and returns the result after the first phase, when a proposer observes a consistent quorum. This prevents (1) state modifications by reads, (2) allows termination in two message delays and (3) prevents live-locks caused by duelling proposers once all correct acceptors have agreed on a proposal. This is possible because once a proposal with value v is chosen, any proposal made in a higher round will contain v (see Section 5.3.3). As the value of the register cannot change any more, it is needless to execute the second phase.

Distinguishing Between Reads and Writes. In canonical Paxos, to read the state of a consensus it is necessary to propose a value for consensus when no proposal was seen yet, i.e., actually performing a write, which is unintended. For a read, a client can either (1) initiate the protocol as a proposer and—in accordance to the protocol—has to propose a (dummy) value itself when no value was chosen yet or (2) it can ask a learner. However, a learner that has not learned a value also has to propose a (dummy) value to ensure its answer is up-to-date. As this dummy value might be written, the read semantic is violated. Drawing from the concept of consistent quorums, we support reads without the risk to change the register's value and are also able to reliably recognise an empty register. A read acts like a write only when an ongoing, partially accepted proposal is seen that may need help to fully establish. However, no value will be proposed that was not already proposed by a write.

Incremental Round Number Negotiation. Proposers have to choose a high enough round number for their proposal to succeed. Canonically, a proposer chooses the round number itself. If it is too low, the proposer's attempt fails and it has to try again with a higher round. This works well when a leader makes the proposals, as it knows the previous used round number. Without a leader, however, the first guess of a proposer is likely to fail, costing a round trip even without concurrent access. Instead, we let the acceptors increment their round on an initial round-less attempt and retrieve the 'assigned' round from the replies when they form a consistent quorum. Otherwise, we calculate a higher round number from the replies and retry like in Paxos.

Using incremental rounds is optional. If a proposer can determine a round number that likely succeeds, it can also start with that without violating the protocol's safety.

Single Learner Per Request. In canonical Paxos, acceptors send their *VOTED* messages to a set of learner processes, which learn the value once they have received a quorum of votes for a proposal. Therefore, the number of messages sent is the product of the number of acceptors and the number of learners. In our approach, the proposer that has received a request acts as its sole learner. Thus, every acceptor sends only a single *VOTED* message.

5.3.3 Sketched Proof of Safety

In this section, we provide a proof sketch for our Paxos atomic write-once register. We show that the safety requirements of Section 3, as well as linearisability are satisfied. Since our protocol has a close resemblance to canonical Paxos, we can use analogue arguments and invariants as described by Lamport [2] to prove safety.

Proposition 1. *If a proposal p was learned in round r, then there exists a quorum of acceptors Q such that any acceptor in Q has given a vote for p (i.e., the proposal must have been chosen).*

Proof Sketch. For any two acceptors a_1, a_2, which have voted for proposal p_1 and p_2 respectively in the same round r, it holds that $p_1 = p_2$ because rounds are uniquely identified by their ID. To learn a value, a proposer must either (a) receive a consistent quorum of r_{voted} rounds from acceptors at the beginning of phase 2, or (b) receive a quorum of *VOTED* messages. For (a) to be possible, a quorum with $r = r_{voted}$ must exist. For (b), a quorum of acceptors must have voted for the proposer's proposal. Since a proposal is issued for a specific round, all replying acceptors have voted for a proposal in the same round. □

C-Nontriviality is trivial to proof using proposition 1 since acceptors can only vote for any value that was previously proposed by a proposer. C-Stability and C-Consistency hold by satisfying the following invariant:

Proposition 2. *If a proposal with value v_c and round r_c is chosen, then every proposal issued with round $r > r_c$ by any proposer has also value v_c.*

Proof Sketch. By Proposition 1, there is a quorum Q that has voted for v_c in r_c. Since any two quorums have a non-empty intersection, any proposer p will receive at least one ACK reply of an acceptor included in Q. Furthermore, no acceptor has voted for a proposal valued v' with $v' \neq v$ in round r' with $r' > r_c$. This would imply the existence of a quorum Q' for which every acceptor has acknowledged round r' before voting for the proposal in round r_c. This contradicts the existence of Q since acceptors cannot vote for a lower round than they have previously acknowledged. Therefore, the proposal with the highest round that p receives has value v_c. Thus, p issues a proposal with v_c. □

Proposition 2 assumes that rounds can be totally ordered. However, they are only partially ordered due to our modified negotiation mechanism. Thus, we must show:

Proposition 3. *For any round number n, at most one proposal is issued.*

Proof Sketch. A proposer can only issue a proposal in a round with round number n once it has received an acknowledgement from a quorum of acceptors with consistent and increased r_{ack} with round number n. Any acceptor can send at most one ACK message in which it has also increased its r_{ack} to have round number n. Thus, at most one proposer can receive such a quorum to make a proposal. If incremental rounds are not used, proposers have to choose their own unique round numbers (cf. canonical Paxos). □

Proposition 4. *The Paxos-based write-once atomic register is linearisable.*

Proof Sketch. Proposition 1 and 2 show that all writes return value v_c of the first chosen proposal as their result. Reads differ from writes in that they can return the initial value $\perp$, but only if no value is chosen since a consistent quorum is required. Since a proposer must have learned a value before any write (or read not returning $\perp$) can complete, any subsequent read will return v_c. □

5.4 Consensus Sequence Register

The typical approach to learn a sequence of consensus values is to chain multiple consensus instances on separate resources [2], [11]. In contrast, we aim to operate on the same set of resources. For that, we extend our fault-tolerant write-once atomic register to support a sequence of updates.

The interface of our extended register changes slightly. Instead of including a specific value *val* (see Algorithm 1, line 1) in a write request, clients include an update command *cmd*, which transforms the current value of the register to the next value. The required changes of the proposer's second phase are highlighted in Algorithm 2. The behaviour of the acceptors remains unchanged.

We introduced the concept of consistent quorums in our write-once register to detect if the current value is chosen or not (see Section 5.3). We can use this information to handle a sequence of updates: A proposer is allowed to propose a new value if the current value is chosen. Otherwise, it must complete the unfinished consensus by proposing an existing value. We refer to the former as a ***successor proposal*** and to the latter as a ***write-through proposal***.

Algorithm 2 Proposer's modified phase 2 supporting a sequence of writes

```
 1: on receive ⟨$ACK, inc, r_{ack}, r_{voted}, v$⟩ from quorum $Q$:
 2:     if $cons_Q(r_{voted}) \wedge$ $op = read$ then
 3:         ▷ read: return current consensus
 4:         send ⟨$DONE, v$⟩ to client
 5:     else if $cons_Q(r_{voted}) \wedge r \leftarrow cons_Q(r_{ack})$ then        ▷ modifications to
    apply next command
 6:         ▷ consensus established, r prepared
 7:         $v_{new} \leftarrow cmd(v)$
 8:         if $v_{new} \neq NOOP$ then
 9:             ▷ propose successor value
10:             send ⟨$VOTE, r, v_{new}$⟩ to all acceptors
11:         else
12:             ▷ cmd, e.g. test-and-set, not applicable to latest value v
13:             send ⟨$DENIED$⟩ to client
14:     else if $r \leftarrow cons_Q(r_{ack}) \wedge inc$ then
15:         ▷ r consistently prepared
16:         $v_{prop} \leftarrow max_Q(r_{voted}, v)$
17:         ▷  write-through unfinished consensus
18:         send ⟨$VOTE, r, v_{prop}$⟩ to all acceptors
19:     else
20:         ▷ inconsistent quorum; retry with higher round
21:         $r_{new} \leftarrow max_Q(r_{ack})$
22:         $r_{new}.n \leftarrow r_{new}.n + 1$
23:         send ⟨$PAXOS\_PREP, r_{new}$⟩ to all acceptors
```

Lines 5–13 shows how a proposer submits a successor proposal. It first applies the update command *cmd* it has received from the client on the current established value. If the update is a valid operation, the proposer can send the result to all acceptors. Sometimes, the update reduces to a no-op as it cannot be applied to the current value, for instance, if it includes compare-and-swap semantics or requires a write lock that is missing. The proposer does not have to complete the second phase as the update has no effect and can therefore immediately return to the client.

The submission of a write-through proposal (line 14–23) is equivalent to proposing a value using our write-once register. The proposer proposes the value seen in the highest round. Afterwards, it must re-execute the protocol to process the received write request as a successor proposal.

Safety. Intuitively, the register behaves as if executing multiple single-decree Paxos instances in sequence, with each instance using the previously chosen proposal and its round as initial state. Updates are applied on top of a chosen value, which is ensured by observing a consistent quorum. Thus, for any two values v_1 and v_2 that are chosen in this order, $s(v_1)$ is the prefix of $s(v_2)$. By an argument analogous to Proposition 4, reads always return the latest chosen value. Thus, CS-Stability and CS-Consistency are satisfied.

An update u can only complete if a value that includes u in its update sequence is chosen, as a quorum of *VOTED* messages is required. As only chosen values are returned, CS-Update-Visibility is guaranteed. No proposer applies u on any chosen value after u is completed. Thus, every subsequent update appears after the last occurrence of u in $s(v)$ of a subsequently chosen value v (CS-Update-Stability).

5.5 RMWPaxos: Atomic Read-Modify-Write Register

The consensus sequence register presented in the previous section is not atomic, as it is possible that an update

Algorithm 3 RMWPaxos: A fault-tolerant atomic read-modify-write register

```
--------------- Proposer: Phase 1 ---------------
1:  on receive $\langle REQ, op = write|read, cmd\rangle$ from client:
2:      Store $op$, $cmd$, generated ReqID as $req_{cur}$
3:      $r_{id} \leftarrow$ globally unique ID
4:      send $\langle PREPARE, op, r_{id}\rangle$ to all acceptors
5:  on receive $\langle ACK, inc, r_{ack}, r_{voted}, v, req_{prev}\rangle$ from quorum $Q$:
--------------- Phase 2 ---------------
6:      if $cons_Q(r_{voted}) \wedge op = read$ then  ▷deliver read value
7:          send $\langle DONE, v\rangle$ to client
8:      else if $cons_Q(r_{voted}) \wedge req_{cur} = cons_Q(req_{prev})$ then
9:          ▷ Proposer's previous attempt failed but was established by write-through
10:         send $\langle DONE, v\rangle$ to client
11:     else if $cons_Q(r_{voted}) \wedge r \leftarrow cons_Q(r_{ack})$ then  ▷can apply next cmd
12:         $v_{new} \leftarrow cmd(v)$
13:         if $v_{new} \neq NOOP$ then  ▷propose successor value
14:             send $\langle VOTE, r, v_{new}, req_{cur}, v, req_{prev}\rangle$ to all acceptors
15:         else ▷cmd, e.g. test-and-set, not applicable to latest value
16:             send $\langle DENIED\rangle$ to client
17:     else if $r \leftarrow cons_Q(r_{ack}) \wedge inc$ then  ▷execute write-through
18:         $v_{prev} \leftarrow max_Q(r_{voted}, v)$
19:         $req_{tmp} \leftarrow max_Q(r_{voted}, req_{prev})$
20:         send $\langle VOTE, r, v_{prev}, req_{tmp}, \bot, \bot\rangle$ to all acceptors
21:     else ▷retry with higher round
22:         $r_{new} \leftarrow max_Q(r_{ack})$
23:         $r_{new}.n \leftarrow r_{new}.n + 1$
24:         send $\langle PAXOS\_PREP, r_{new}\rangle$ to all acceptors
25: on receive $\langle VOTED, v\rangle$ from quorum $Q$:
26:     if proposer executed a write-through then
27:         restart protocol from phase 1 with same ReqID
28:     else
29:         send $\langle DONE, v\rangle$ to client
30: on receive $\langle LEARNED, v, req\rangle$ from any acceptor:
31:     if proposer has not completed request $req$ yet then
32:         send $\langle DONE, v\rangle$ to client

--------------- Acceptor ---------------
33: initialise:
34:     $r_{ack} \leftarrow (0, \bot), val \leftarrow \bot, r_{voted} \leftarrow (0, \bot), req \leftarrow \bot$
--------------- Phase 1 ---------------
35: on receive $\langle PREPARE, op, r_{id}\rangle$ from proposer $p$:
36:     ▷ incremental phase 1
37:     if $op = write$ then
38:         $r_{ack} \leftarrow (r_{ack}.n + 1, r_{id})$
39:         send $\langle ACK, true, r_{ack}, r_{voted}, val, req\rangle$ to $p$
40:     else
41:         ▷ no state change for read
42:         send $\langle ACK, false, (r_{ack}.n, r_{id}), r_{voted}, val, req\rangle$ to $p$
43: on receive $\langle PAXOS\_PREP, r\rangle$ from proposer $p$:
44:     ▷ canonical Paxos phase 1
45:     if $r > r_{ack}$ then
46:         $r_{ack} \leftarrow r$
47:         send $\langle ACK, true, r_{ack}, r_{voted}, val, req\rangle$ to $p$
--------------- Phase 2 ---------------
48: on receive $\langle VOTE, r, val_{new}, req_{cur}, val_{prev}, req_{prev}\rangle$ from proposer $p$:
49:     if $r \geq r_{ack}$ then
50:         if $req_{prev} \neq \bot$ then
51:             ▷ notify previous proposer that its proposal was learned
52:             send $\langle LEARNED, val_{prev}, req_{prev}\rangle$ to proposer in $req_{prev}$
53:         $req \leftarrow req_{cur}$
54:         $r_{ack} \leftarrow r$        ▷improve $r_{ack}$ consistency
55:         $r_{voted} \leftarrow r$
56:         $val \leftarrow val_{new}$
57:         send $\langle VOTED, v\rangle$ to $p$
```

command submitted by a client is proposed and applied multiple times by the same proposer. For example, consider the following scenario: Proposer p_1 completes phase 1 and submits a successor proposal. However, it only gets a minority of acceptor votes, as some concurrent proposer p_2 already increased the r_{ack} rounds of a quorum of acceptors. In this case, p_2 may observe an inconsistent quorum and therefore executes a write-through of p_1's proposal. If it succeeds, then p_1's proposal was effectively accepted because the value proposed by p_1 is chosen. However, p_1 does not know this and retries, potentially executing the command twice.

For atomicity, we must ensure that a proposer does not re-submit a successor proposal once the proposed value of a previous attempt is chosen. For that, we assume reliable in-order message delivery (see Section 2). This can be provided by reliable communication protocols such as TCP. Note, that messages can be lost if a TCP connection fails and is later re-established during the processing of a request. To solve this issue, processes can be treated as crashed until the request is completed. Now, the protocol can be modified as follows (cf. Algorithm 3):

For every write request that proposer p receives, it generates a request ID (ReqID) consisting of its PID and some locally unique value (line 2). Every acceptor holds the ReqID of the last proposal it voted for and includes it in all phase 1 ACK messages it sends.

If a proposer submits a successor proposal, it includes its own ReqID as req_{cur} and the ReqID received in phase 1 as req_{prev} in its $VOTE$ messages (line 14). Here, req_{prev} indicates the last successor proposal that was chosen by the register. If the proposer submits a write-through, it includes the ReqID received in phase 1 as req_{cur}. Since the last chosen proposal is now known, req_{prev} remains empty (line 20).

Each time an acceptor votes for a new proposal, it updates req to req_{cur} (line 53). If req_{prev} is non-empty, it sends a $LEARNED$ message to the respective proposer (line 52). Receiving a $LEARNED$ message guarantees that the corresponding proposal was chosen. A proposer that retries a request with some ReqID can stop the protocol if (1) it observes a consistent quorum with this ReqID (line 8), or (2) it receives a $LEARNED$ message with it (line 30). In both cases, it notifies the client that its write request succeeded.

We note that it is easy to avoid sending values in $LEARNED$ and $VOTED$ messages back to the proposer if the proposer keeps track of its proposed values locally. By extension, it is not necessary to include val_{prev} in $VOTE$ messages. For simplicity, this is not shown in Algorithm 3.

Safety. Assume a write request with ReqID r and update command u is processed by proposer p. Assume that p's attempt failed, but its proposed value is chosen (e.g., due to a write-through). Proposer p does not propose u as the direct successor of its own proposed value because it would observe a consistent quorum with ReqID r beforehand. Thus, assume that some successor value proposed by a different proposer is chosen. This means that $LEARNED$ messages with ReqID r are sent to p by some quorum Q. Let p retry its request. In order to apply u and propose a new value, p must observe a consistent quorum Q'. As $Q \cap Q' \neq \varnothing$ and reliable ordered links are used, p receives a $LEARNED$ message before receiving a consistent quorum. Thus, p does not apply u on a value whose update sequence already includes u.

5.6 State Machine Replication

By using RMWPaxos, we can build a fault-tolerant replicated state machine using a fixed set of storage resources. The state is stored in the register and state changes are done by the corresponding update commands. If updates are idempotent, the consensus sequence register suffices. One way to achieve this is by using transactional semantics such as compare-and-swap.

In log-based approaches like Multi-Paxos [2, Sect. 3], acceptors accept commands, i.e., state transitions of the state machine. In our approach, in contrast, the acceptors accept the complete state. This has several implications. First, a dedicated set of learner processes is no longer required. Any process that wishes to learn the current state of the RSM can do so by executing a read. This process then acts as the sole learner in the context of this command. In contrast, Multi-Paxos requires multiple learners in order to have access to the state in a fault-tolerant manner. Since every learner must also learn every command to make progress, $n * m$ *VOTED* messages are required in a setup with n acceptors and m learners. Our approach requires only n *VOTED* messages.

Second, by keeping the full state in acceptors, a sequence of commands can now be applied to the RSM in-place using the same set of acceptors. Thus, it is not necessary to allocate and free storage resources. This simplifies the protocol's complexity and its implementation. Due to the absence of any state management overhead, it is trivial to use arbitrary many RMWPaxos instances in parallel, allowing a more fine-granular use of the RSM paradigm. This is especially useful if the state can be split into many independent partitions, as it is often the case in key-value structured data.

5.7 Liveness

Reads and writes are obstruction-free [18] as long as a quorum of acceptors and the proposer receiving the requests are correct. Wait- or lock-freedom [17] cannot be guaranteed without further assumptions, as postulated by the FLP result [14]. A common assumption is the existence of a stable leader to which all requests are forwarded. The leader then acts as the sole proposer of the system. To handle leader failures, a $\mathcal{W}$ failure detector [19] is necessary.

5.8 Optimisations

There are several ways to optimise the basic protocol.

Fast Writes. Handling writes requires a proposer to complete both phases of the protocol. That means that at least four message delays are needed. By using a mechanism similar to Multi-Paxos [2], the first phase can be skipped by a proposer that processes multiple writes uninterrupted by other proposers. We refer to such writes as ***fast writes***.

The modification is simple. Whenever an acceptor votes for a proposal made in round r, it sets r_{ack} to $(r.n+1, r.id)$ (cf. Algorithm 1 line 40). By doing so, it effectively behaves as if receiving a $PREPARE$ message from the same proposer immediately after voting. Therefore, this proposer can skip the first phase when making its next proposal.

This optimisation is useful for single-writer settings or scenarios in which a proposer must execute multiple writes within a short period. As no locking or lease mechanism is used, an ongoing fast write sequence can be interrupted at any time by other proposers. Thus, we avoid the costs and unavailability associated with a leader and its (re-)election.

Fewer Concurrency Conflicts Caused by Reads. If a read observes a consistent quorum after the first phase, it returns a result without interfering with any concurrent request because acceptors do not modify their rounds. If a read observes an inconsistent quorum, a write-through is triggered, which can cause interference. Write-throughs cannot be prevented completely, as a crashing proposer can cause a proposal to be only partially established. Therefore, we adopt the idea of contention management [18], [24] to unreliably detect a crashed writer:

When a reading proposer observes an inconsistent quorum, it stores the highest round it has received. Then, it retries phase 1 without an explicit round. If the quorum is again inconsistent, it checks whether progress was made by comparing the received rounds with the rounds from the previous iteration. If they remain unchanged, then it is possible that the write crashed and a write-through must be triggered. Otherwise, the reader can try again. The proposer can keep collecting replies from its previous attempts as it is possible to reach a consistent quorum with delayed replies.

To prevent a read from starving due to a continuous stream of writes, we define an upper limit on the number of retry attempts. Its effects are evaluated in Section 7.

Batching. Batching is a commonly used engineering technique to reduce bandwidth and contention by bundling multiple commands in a single request at the cost of higher response latency. Every proposer manages separate batches for read and update commands. A batch is processed at regular intervals by starting the protocol. For write batches, all update commands of the batch are applied in-order on the old value before proposing the resulting new value. When processing a read batch, the read value is simply returned to all clients. The size of all messages remains constant, independent of the number of batched commands. This shifts the performance bottleneck from internal communication to the processing speed of the respective proposers.

6 Analysis

In this section, we focus on additional aspects that might be beneficial for practical deployments. An experimental evaluation can be found in Section 7.

Compared to canonical Paxos and Multi-Paxos our registers require a similar number of 2–4 message delays per consensus in the conflict-free case. Two additional message delays are needed by canonical Paxos when a valid round number is not known yet and by our registers when a read using incremental rounds has to help to establish a consensus. Reading a stable, established consensus with our approach only needs 2 message delays, no concurrency control and does not cause acceptor state changes, which is costly if their state must be persisted. Furthermore, our approach works on a fixed set of resources which makes dynamic resource allocation, pruning, and deallocation needless. This makes our register applicable on a more fine-granular level than other consensus-based approaches that rely on a command log.

Relying on consistent quorums does not harm robustness nor performance. Like in canonical Paxos, a single replica with the highest round seen in an inconsistent quorum will suffice to propose its value. But on a *consistent quorum*, we can (a) terminate a read operation early by not needing to write and re-learn the consensus and (b) can base the next consensus in our consensus sequence on that.

Not requiring an explicit leader provides more continuous availability. In our approach, any proposer can issue requests to the register at any time. When a proposer fails, other proposers can immediately proceed and do not need to wait for or elect a new leader. Still, a proposer submitting many requests in sequence without any interference of other proposers can perform each write to the register in just two message delays (no batching), like a leader.

7 Experimental Evaluation

We implemented RMWPaxos in Scalaris [25], a distributed key-value store written in Erlang. The correctness of our implementation was tested using a protocol scheduler [26], which forces random interleavings of incoming messages. We detected no safety violations using this approach.

The primary focus of the evaluation is to show the scalability of our approach under different workloads, as absolute performance is highly dependent on the available hardware environment and engineering efforts that are independent of the actual approach. Our register aims to be a general primitive. Thus, we consider use-case dependent techniques that optimise network traffic and concurrent access, e.g., request batching, being out-of-scope of this paper.

All benchmarks were performed on a cluster with two Intel Xeon E5-2670 v3, 2.40 GHz per node. All nodes are fully-connected with 10 Gbit/s links. Each cluster node hosts a single replica, which is a Scalaris node that encapsulates one proposer and one acceptor process. Load generation was performed on up to two separate cluster nodes using the benchmarking tool Basho Bench [27], which was modified to enable workloads with heterogeneous client processes. In all experiments, Basho Bench clients were distributed evenly across the load generating nodes. All clients submit their requests sequentially, i.e., each client waits for a response before issuing the next request.

All shown measurements ran for 10 minutes with request data aggregation in one-second intervals. We show the mean with 99 percent confidence intervals (CI) and 99th percentile latencies. In almost all cases, the CI lies within two percent of the reported median.

7.1 Comparison With Raft and Multi-Paxos

First, we compare the performance of RMWPaxos with open-source implementations of Multi-Paxos [2], [28] and Raft [4], [29], two commonly used state-of-the-art protocols. To minimise the performance impact of the IO subsystem, we configured both approaches to write their data to RAM disk. In RMWPaxos, data is stored by using Erlang's build-in term storage [30]. All approaches use three replicas. As both Multi-Paxos and Raft make use of a leader, we simulate a leader by randomly selecting one node to which all requests are forwarded to in the case of RMWPaxos. As any leader election protocol can be implemented on top of RMWPaxos, we consider leader election to be out-of-scope.

We measured the throughput of all approaches in scenarios: First, a counter that is accessed by an increasing number of clients (Fig. 3a). Second, a binary value of increasing size accessed by a fixed number of clients (Fig. 3b).

All three approaches handle requests in a single round-trip between leader and a quorum of following nodes. Thus, the observed differences can largely be attributed to their different strategies in handling the data locally. Due to the absence of any state management, RMWPaxos consistently outperforms both the Raft and Multi-Paxos implementation for small state sizes. For the latter two, overhead caused by reading/writing data to the local file system increases request latency, which in turn negatively affects throughput. In addition, the Multi-Paxos and Raft implementations use mechanisms such as checksum validation to protect against disk corruption.

We note that Multi-Paxos has a higher throughput than RMWPaxos in read-heavy workloads with few clients. We attribute this to our method of load generation. As clients submit requests sequentially, both approaches do not reach full capacity. Here, we observe a slightly lower mean read latency for Multi-Paxos (0.6ms vs 0.8ms), which is likely caused by implementation-specific overhead.

For values smaller or equals to 4kB, all approaches exhibit nearly constant read performance. However, the throughput of RMWPaxos decreases for larger values. This is because the full value is always transferred from a quorum of nodes to the proposer when executing a read. This causes high communication costs in settings where individual objects have moderate or large size. However, analysis of existing large-scale key-value stores have shown a heavy skew towards small values of less than a kilobyte [31], [32].

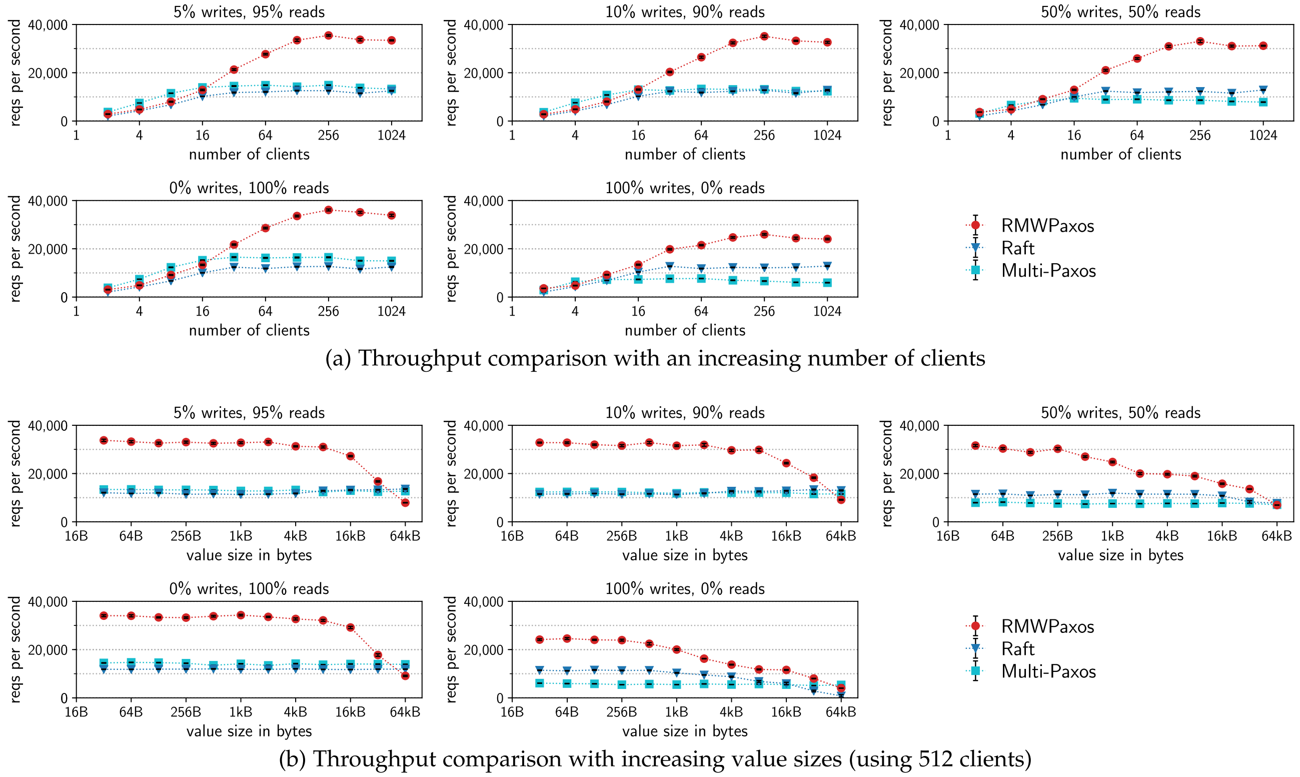
In contrast to RMWPaxos, the Raft and Multi-Paxos implementations include optimisations to keep data transfer costs between nodes constant when executing a read if the leader is stable. In Raft's case, an empty heartbeat log entry must be appended to the command logs to ensure that the data of the leader is up-to-date. This introduces a slight overhead when reading entries.

7.2 Leaderless Performance

RMWPaxos is derived from Paxos. Thus, it does not depend on the existence of a leader to satisfy the safety properties of consensus, in contrast to protocols like Raft, which do not work without a single leader. However, a leader is beneficial for progress because it prevents the duelling proposer problem. For RMWPaxos, we can alleviate the need for a leader as it is trivial to deploy an arbitrary number of concurrent RMWPaxos instances. This way, load on a single instance can be greatly reduced, depending on the workload.

We examined both single-writer (Fig. 4) and multi-writer (Fig. 5) workloads, as previous work in the design of data structures has shown that supporting concurrent modifications often inhibits their performance [24]. To better illustrate the effects of concurrent requests, we increased the system size to five replicas (acceptors).

Single-Writer. To evaluate single-writer performance, we used one writing client and up to 1024 concurrent readers



(a) Throughput comparison with an increasing number of clients

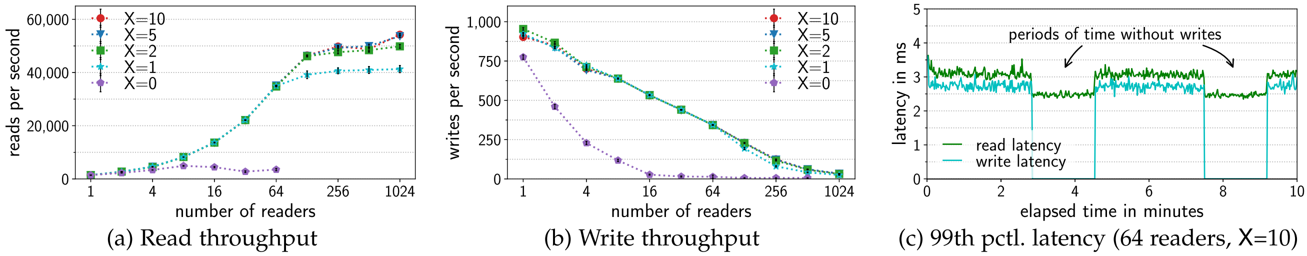
(b) Throughput comparison with increasing value sizes (using 512 clients)

Fig. 3. Comparing the throughput of RMWPaxos with Raft and Multi-Paxos using three replicas.

with a different number of read retries (parameter X). The results are depicted in Fig. 4. We observed that even a single retry (X=1) improves both read and write throughput greatly compared to disabling this optimization (X=0). In the latter case, the register was overloaded due to concurrent write-through attempts by the readers if more than 64 readers where used, dropping throughput to 0 at some times. As these results are not stable, they are not shown in Fig. 4a. Choosing a value for X larger than 2 has only a minor impact on the read throughput. As acceptors must handle more messages with an increasing number of clients, their response latency increases. This leads to a consistent decline of the write throughput, as shown in Fig. 4b. Since the load is distributed more evenly across all replicas, the maximum observed throughput increased by roughly 70 percent compared to our leader-based experiments (cf. Fig. 3a), even though the system size increased from 3 to 5 replicas.

Fig. 4c shows the latency impact of using read retries. Read latency only increases by approx. 0.5 ms in the presence of a concurrent writer. This may contradict the expectation that some reads require multiple round trips as they can observe an inconsistent quorum initially. However, proposers can continue collecting replies from the initial attempt and return a result once they observe a consistent quorum. As there is only a single writer, such a quorum always exists, at the latest after receiving a reply from every acceptor. This also means that reads trigger no write-throughs. Thus, both reads and writes succeed after a single round trip in a single writer setup as long as no acceptor fails. Note that writes exhibit a slightly lower latency as they always succeed with a quorum of replies, wheres reads must potentially wait for all replies in some cases.

Multi-Writer, Single-Register. All clients sent a uniform mix of read and write requests for the evaluation of multiple writers. Fig. 5a compares the throughput of a read-heavy workload (5 percent writes) with a write-heavy workload (50 percent writes) [33]. Performance degradation caused by duelling proposers can be observed for both workloads. The throughput of the read-heavy workload scales up until four concurrent clients. Afterwards, clients begin to invalidate each other's proposals repeatedly. In write-heavy

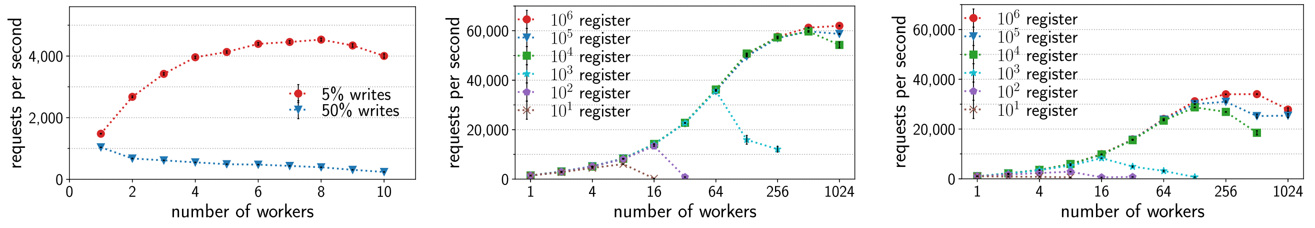


(a) Read throughput

(b) Write throughput

(c) 99th pctl. latency (64 readers, X=10)

Fig. 4. Single-writer performance of RMWPaxos with five replicas.



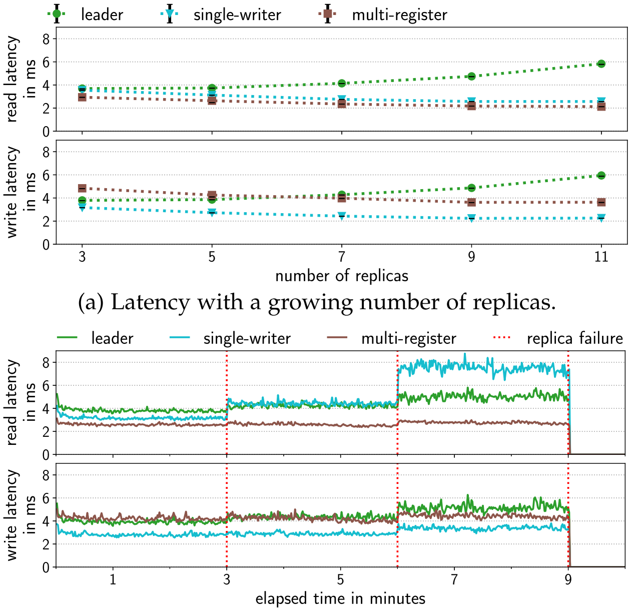
(a) Single register throughput (b) Multi-register read-heavy throughput (c) Multi-register write-heavy throughput

Fig. 5. Leaderless multi-writer performance of RMWPaxos with five replicas.

workloads, even two concurrent clients are enough to have a negative impact on the system's performance. As shown in the previous experiments, a leader at the application level helps to handle write concurrency effectively.

Multi-Writer, Multi-Register. All previous measurements focused on a single register. As highlighted in Section 5.6, the absence of state management overhead easily allows for arbitrary many registers to be used. We benchmarked configurations using up to 10^6 register instances and 512 concurrent clients. The registers were accessed according to a Pareto distribution [34] with $\alpha \approx 1.16$ (80 percent of requests targeted 20 percent of registers). Figs. 5b and 5c show the results for read-heavy (5 percent writes) and write-heavy (50 percent writes), respectively. The results are as expected. More concurrent clients can be handled without performance degradation due to duelling proposers by increasing the number of parallel registers. The load is evenly distributed across all replicas, as no leader is used. In addition, contention is low in settings with a large number of parallel registers. This results in a higher achievable throughput than it is possible with the use of a leader (cf. Fig. 3a).

RMWPaxos performs consistently better under read-heavy workloads, which coincides with the results from the single-register evaluation. We used the read-write ratios of YCSB [33], a benchmarking framework that aims to simulate real-world use-cases. Studies of large-scale distributed systems have shown an even higher skew towards reads, reporting read-write ratios of up to 450:1 [8], [31], [35].



(a) Latency with a growing number of replicas.

(b) Latency with five replicas and process failures.

Fig. 6. 99th percentile latency comparison using 64 clients.

7.3 Impact of Replication Degree and Failures

We investigated the impact of the number of replicas on the response latency of RMWPaxos, as well as its ability to tolerate replica failures. For that, we used different deployment strategies from our previous experiments: (1) A single register accessed by a leader, (2) a single register accessed by a single writer and multiple readers, and (3) a 10.000 register setup accessed by multiple writers and readers with the Pareto distribution used in Section 7.2. We will refer to them as the leader, single-writer, and multi-register strategy, respectively. All measurements were executed using 64 clients. Clients used a read-heavy workload (5 percent writes, 95 percent reads) in the leader and multi-register deployment. The results are shown in Fig. 6.

When using a leader, the number of messages the leader must process increases with a growing number of replicas. This results in an increasing response latency as shown in Fig. 6a. In contrast, the load is distributed evenly among all node in both the single-writer and multi-register setup. Assuming a constant throughput, the number of messages each proposer is sending is independent of the system size. As only replies from a quorum of replicas is needed, fewer messages must be received in total by each proposer to answer all requests. This results in a slightly lower response latency of these strategies with growing system sizes.

To measure the impact of failures, we let one replica crash after every three minutes. Overall, all latencies with the exception of the read latency of the single-writer strategy remained fairly consistent as long as a sufficient number of replicas is available. We observed only a slight increase for each new failure, as proposers must potentially wait for the replies of slower acceptor processes. However, to ensure that reads can be processed in the single-writer setup, answers from all replicas are necessary (see single-writer evaluation in Section 7.2). If a replica fails, proposers do not always observe a consistent quorum after all remaining acceptors reply. They must therefore retry their request. This is more likely to happen as more replicas fail.

7.4 Leader Load and Applicability to NVM

Our results show potential for future improvements. First, we aim to improve the issue of high write contention on a single register while alleviating the bottleneck caused by a leader. As a single register is able to handle high read concurrency (see Section 7.2), only writes have to be forwarded to the leader. This can be coupled with a dynamic leader allocation for only highly contentious registers, which further reduces the load placed on the leader.

Second, we believe that the fine-granular nature of our approach is a promising fit for the use in combination with byte-addressable, non-volatile main memory. With the recent availability of NVRAM, along with the current work in NVMe over Fabrics [36], we believe that our approach can leverage these technologies in the future.

8 Related Work

Starting with Lamport's work on the discovery of the Paxos algorithm [2], [3], numerous Paxos extensions [7], [37], [38], [39] have been proposed—most of them following the design of using multiple Paxos instances to learn a sequence of commands. As a notable exception, Generalized Paxos [13] and its derivatives [40], only use a single Paxos instance but require keeping track of an ever-growing set of commands in its messages. In all cases, pruning in some form must be implemented to prevent unbounded memory consumption, which introduces a considerable amount of complexity to the system. This is identified by Chandra *et al.* [11] as one of the main challenges for using Paxos-based designs in practical systems. Despite numerous efforts of making Paxos more approachable [41], [42], [43], reliable state management with Paxos is seldom discussed in detail. Only a few practical Paxos-based systems exist to this date such as Chubby [9], Spanner [8], Megastore [10], and Scalaris [25].

In recent years, various proposals were made to alleviate the dependence on a single leader. Mencius [5] evenly shares the leader's responsibilities by assigning individual consensus instances to single replicas. In Egalitarian Paxos [7], the replica receiving a command is regarded as its command leader. Each replica can act as a leader simultaneously for a subset of commands. This is achieved by decoupling command commit and application from each other and making use of the dependency constraints of each command. In contrast, we do not need an explicit leader depending on workload and load-distribution.

As of today, few consensus protocols, which are not Paxos-based, exist. Most prominently Raft [4] and the closely related Zab protocol [6]. Both are based on the idea of a central command log. Furthermore, they require a *strong* leader, meaning that at most a single leader is allowed to exist at any given time. In contrast, we perform updates on a distributed state in-place and do not need a strong leader.

To the best of our knowledge, we present the first Paxos-based approach that does not rely on additional state management without requiring a leader to satisfy the safety properties of consensus by implementing an atomic RMW register. The register by Li *et al.* [44] only recasts the original Paxos without modification and provides a regular write-once register. The round-based register proposed by Boichat *et al.* [42] is not atomic and only write-once. It is similar to the approach of Li *et al.* and modular to build several, known Paxos variants such as Multi-Paxos or Fast Paxos [22]. CAS-Paxos [45] provides a Paxos-based linearisable multi-reader multi-writer register by letting clients submit a user-defined function instead of a value. However, when handling concurrent writes it is not guaranteed that all (or any) writes are processed by the register due to duelling proposers, which makes it unsuitable to implement basic primitives like counters. The key-value consensus algorithm Bizur [46] is based on a set of single-writer multi-reader registers and therefore relies on electing a strong leader.

The use of consistent quorums in conjunction with Paxos is first introduced by Arad *et al.* [20] in the context of group membership reconfigurations. In this context, a consistent quorum expresses a consistent *view* of the system in terms of group memberships. Skrzypczak *et al.* [47] use consistent quorums to provide linearisable access to CRDTs. While this approach is similar to the protocol presented here, it heavily relies on the mathematical properties of CRDTs and can therefore not be used for general state machine replication.

Shared register abstractions were first formalized by Lamport [48]. Among them, the atomic register provides the strongest guarantees by being linearisable. Numerous implementations exist today. In particular, the multi-writer generalisation [49, p. 25ff.] of ABD [50] has the greatest resemblance to our approach. However, the properties of atomic registers alone do not suffice to solve consensus, as not every completed write is necessarily applied to the register when being confronted with concurrent access. Moreover, only fixed values can be written. Our register abstractions provide arbitrary value transformations based on the register's previous value and ensure that completed writes are applied at-least-once (consensus sequence register) or exactly-once (RMWPaxos).

9 Conclusion

In this paper, we introduced register abstractions that satisfy the safety properties of consensus and allow consensus sequences. We provided implementations extending the principles of Paxos consensus, to allow a sequence of consensuses 'in-place' using a single set of storage resources, instead of a separate instance for every consensus decision.

Additionally, read operations in RMWPaxos do not interfere with each other (are not serialised with each other) and do not modify any state in the acceptors when the register is stable, i.e., no write operation is induced. This improves the parallel read throughput and saves unnecessary, potentially costly state changes of persistent storage for reads. When reads detect ongoing writes, they can either hope the writer will finish soon and mitigate the chance of duelling proposers by just retrying the read, or can start to support the writing themselves as the writer might have crashed. As we show in our evaluation (Section 7), the trade-off between both strategies and how often one should retry the read before helping the writer depends on the system deployment, the number of expected concurrent readers and writers, etc.

Avoiding the need for costly state management and complex protocols for state pruning, providing fast writing in two message delays and supporting concurrent readers without serialisation opens a wide new spectrum of use-cases for Paxos based fault-tolerance. The protocols we provide are beneficial and applicable on a more fine-grained level than Multi-Paxos or similar approaches, as they have low system overhead and provide good scalability.

Code Availability. The source code for our RMWPaxos implementation [51] and the protocol scheduler [26] can be found on GitHub under the Apache License 2.0.

Acknowledgments

The authors would like to thank Alexander Reinefeld and anonymous reviewers for their dedicated comments and valuable discussions that helped to improve this manuscript. This work received funding from the German Research Foundation (DFG) under Grant RE 1389 as part of the DFG priority program SPP 2037. They would also like to thank ZIB's core facilities unit for providing them the machines and infrastructure for the evaluation.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[2] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 51–58, 2001.

[3] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[4] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.

[5] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machine for WANs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 369–384.

[6] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw.*, 2011, pp. 245–256.

[7] I. Moraru, D. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 358–372.

[8] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 251–264.

[9] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 335–350.

[10] J. Baker *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. Conf. Innovative Data Syst. Res.*, 2011, pp. 223–234.

[11] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proc. 26th Annu. ACM Symp. Princ. Distrib. Comput.*, 2007, pp. 398–407.

[12] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, Berlin, Germany: Springer, pp. 34–37, 2011.

[13] L. Lamport, "Generalized consensus and Paxos," Microsoft Research, Redmond, WA, Tech. Rep. MSR-TR-2005–33, 2005.

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

[15] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[16] N. A. Lynch, *Distributed Algorithms*. Burlington, MA, USA: Morgan Kaufmann, 1996, pp. 244–250.

[17] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[18] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proc. 23rd Int. Conf. Distrib. Comput. Syst.*, 2003, pp. 522–529.

[19] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[20] C. Arad, T. Shafaat, and S. Haridi, "CATS: Linearizability and partition tolerance in scalable and self-organizing key-value stores," SICS, Hyogo, Japan, Tech. Rep. T2012:04, 2012.

[21] J. Skrzypczak, "Weakening Paxos consensus sequences for commutative commands," ZIB, Tech. Rep. 17–64, 2017. Available: http://nbn-resolving.de/urn:nbn:de:0297-zib-65741

[22] L. Lamport, "Fast Paxos," *Distrib. Comput.*, vol. 19, no. 2, pp. 79–103, 2006.

[23] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible Paxos: Quorum intersection revisited," in *Proc. Int. Conf. Principles Distrib. Syst.*, 2016, pp. 25:1–25:14.

[24] W. N. Scherer III, M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proc. 24th Annu. ACM Symp. Principles Distrib. Comput.*, 2005, pp. 240–248.

[25] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: Reliable transactional P2P key/value store," in *Proc. ACM SIGPLAN Workshop ERLANG*, 2008, pp. 41–48.

[26] Scalaris, "Implementation of the protocol scheduler," Last Accessed: Mar. 2, 2020. [Online]. Available: https://github.com/scalaris-team/scalaris/blob/master/src/proto_sched.erl

[27] Basho Technologies, "basho-bench: A load-generation and testing tool for basically whatever you can write a returning Erlang function for," Last Accessed: Mar. 2, 2020. [Online]. Available: https://github.com/basho/basho_bench

[28] Basho Technologies, "riak_ensemble: Multi-Paxos framework in Erlang," Last Accessed: Mar. 2, 2020. [Online]. Available: https://github.com/basho/riak_ensemble

[29] RabbitMQ, "ra: A Raft implementation for Erlang and Elixir that strives to be efficient and make it easier to use multiple Raft clusters in a single system," Last Accessed: Mar. 2, 2020. [Online]. Available: https://github.com/rabbitmq/ra

[30] Ericsson AB, "ETS," Last Accessed: Mar. 2, 2020. [Online]. Available: http://erlang.org/doc/man/ets.html

[31] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *Proc. ACM Sigmetrics/Performance Joint Int. Conf. Meas. Modeling Comput. Syst.*, 2012, pp. 53–64.

[32] R. Nishtala *et al.*, "Scaling memcache at facebook," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.

[33] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[34] M. Newman, "Power laws, Pareto distributions and Zipfs law," *Contemp. Phys.*, vol. 46, no. 5, pp. 323–351, Sep 2005.

[35] Wikimedia, "Wikimedia statistics," Last Accessed: Mar. 2, 2020. [Online]. Available: https://stats.wikimedia.org/#/all-projects

[36] D. Minturn and J. Metz, "Under the hood with NVMe over Fabrics," in *Ethernet Storage Forum. SNIA*, 2015. Available: https://www.snia.org/sites/default/files/ESF/NVMe_Under_Hood_12_15_Final2.pdf

[37] C. Wang *et al.*, "APUS: Fast and scalable Paxos on RDMA," in *Proc. Symp. Cloud Comput.*, 2017, pp. 94–107.

[38] E. Gafni and L. Lamport, "Disk Paxos," *Distrib. Comput.*, vol. 16, no. 1, pp. 1–20, 2003.

[39] P. J. Marandi *et al.*, "Ring Paxos: A high-throughput atomic broadcast protocol," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2010, pp. 527–536.

[40] P. Sutra and M. Shapiro, "Fast genuine generalized consensus," in *Proc. IEEE 30th Int. Symp. Reliable Distrib. Syst.*, 2011, pp. 255–264.

[41] J. Kirsch and Y. Amir, "Paxos for system builders: An overview," in *Proc. Workshop Large-Scale Distrib. Syst. Middleware*, 2008, pp. 1–6.

[42] R. Boichat *et al.*, "Deconstructing Paxos," *SIGACT News*, vol. 34, no. 1, pp. 47–67, 2003.

[43] R. van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Comput. Surv.*, vol. 47, no. 3, pp. 42:1–42:36, 2015.

[44] H. C. Li *et al.*, "The Paxos register," in *Proc. 26th IEEE Int. Symp. Reliable Distrib. Syst.*, 2007, pp. 114–126.

[45] D. Rystsov, "CASPaxos: Replicated state machines without logs," *CoRR*, vol. abs/1802.07000, 2018, Available: http://arxiv.org/abs/1802.07000

[46] E. N. Hoch *et al.*, "Bizur: A key-value consensus algorithm for scalable file-systems," *CoRR*, vol. abs/1702.04242, 2017. Available: http://arxiv.org/abs/1702.04242

[47] J. Skrzypczak, F. Schintke, and T. Schütt, "Linearizable state machine replication of state-based CRDTs without logs," in *Proc. 26th IEEE Int. Symp. Reliable Distrib. Syst.*, 2019, pp. 455–457.

[48] L. Lamport, "On interprocess communication. Part II: Algorithms," *Distrib. Comput.*, vol. 1, no. 2, pp. 86–101, 1986.

[49] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus*. San Rafael, CA, USA: Morgan & Claypool, 2012.

[50] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, 1995.

[51] Scalaris, "Implementation of RMWPaxos," Last Accessed: Mar. 2, 2020. [Online]. Available: https://github.com/scalaris-team/scalaris/tree/master/src/rbr

Jan Skrzypczak received the MSc degree in computer science from the Humboldt University of Berlin. He is currently a research associate at Zuse Institute Berlin with the Department of Distributed Algorithms. His research interests include the design and implementation of distributed algorithms, fault-tolerance, reliability and consensus protocols.

Florian Schintke received the PhD degree in computer science from the Humboldt-Universität zu Berlin. He is currently head of the Distributed Algorithms Research Department at Zuse Institute Berlin. His research interests include fault-tolerance and scalability, distributed protocols and algorithms, transactional key-value stores, and distributed data management in general.

Thorsten Schütt received the PhD degree in computer science from the Humboldt University of Berlin. He is currently a researcher at the Zuse Institute Berlin with the Department of Distributed Algorithms. His research interests include P2P protocols, distributed systems, heuristic search, and NVRAM.