Datenbank Spektrum (2020) 20:231–241
https://doi.org/10.1007/s13222-020-00358-4

SCHWERPUNKTBEITRAG

Towards Log-Less, Fine-Granular State Machine Replication

Jan Skrzypczak[1] · Florian Schintke[1]

Received: 29 May 2020 / Accepted: 29 September 2020 / Published online: 15 October 2020
© The Author(s) 2020

Abstract

State machine replication is used to increase the availability of a service such as a data management system while ensuring consistent access to it. State-of-the-art implementations are based on a command log to gain linear write access to storage and avoid repeated transmissions of large replicas. However, the command log requires non-trivial state management such as allocation and pruning to prevent unbounded growth.

By introducing in-place replicated state machines that do not use command logs, the log overhead can be avoided. Instead, replicas agree on a sequence of states, and former states are directly overwritten. This method enables the consistent, fault-tolerant replication of basic data management primitives such as counters, sets, or individual locks with little to no overhead. It matches the properties of fast, byte-addressable, non-volatile memory particularly well, where it is no longer necessary to rely on sequential access for good performance. Our approach is especially well suited for small states and fine-granular distributed data management as it occurs in key-value stores, for example.

Keywords Replicated State Machine · Consistency · Consensus · Non-volatile memory

1 Introduction

State machine replication [1] is a general technique to increase service availability in the presence of failures without losing strongly consistent, i.e., linearizable [2], access to it. Its realization requires that multiple processes agree on the same sequence of commands. Consensus protocols such as Paxos [3] or Raft [4] are needed to achieve this reliably in the presence of failures.

Current state-of-the-art designs of replicated state machines (RSM) rely on a command log to keep track of previously agreed-upon commands. While this design enables replicating even large states, the use of command logs incurs state management overhead to prevent the local state from growing unbounded. This overhead includes log allocation, truncation, log recovery, and snapshotting. These mechanisms must be kept in sync with the consensus protocol among replicas, further increasing the complexity of such systems [5]. The challenges and overhead associated with a log make the RSM approach cumbersome to replicate small states.

In this article, we introduce the notion of in-place RSMs. In contrast to their log-based counterpart, in-place RSMs directly agree on a sequence of states without using a log. The resulting architecture only requires a small, constant set of state variables. It enables with little overhead the fault-tolerant, consistent access to small states, such as counters, sets, locks, or small pieces of metadata like current group memberships, elected leaders, and more. These are ubiquitous primitives used in data management systems, ranging from databases over object stores to key-value stores.

To survive large-scale failures such as power outages, the replicated state must be stored on a persistent medium. Traditional persistent storage technologies heavily favor sequential, coarse-grained access, which would make it costly for in-place RSMs to gain persistence. With the recent advent of 3D XPoint based memory [4], such as Intel Optane DC Persistent Memory (DCPMM) [6], we have a persistent storage that features byte-addressability with sub-microsecond latency for the first time. These properties are promising for the fast, fine-granular persistence of in-place RSMs.

✉ Jan Skrzypczak
skrzypczak@zib.de

✉ Florian Schintke
schintke@zib.de

[1] Zuse Institute Berlin, Berlin, Germany

The main contributions of this article are:

- We propose the in-place RSM architecture, which is optimized for fine-granular replication by avoiding the use of a command log (Section 4).
- We discuss how in-place RSMs can be used in a distributed key-value store using a separate RSM instance for every key-value pair (Section 5).
- We compare the expected access pattern of log-based and in-place RSMs on systems equipped with Intel's DCPMM and NVDIMM-N (Section 6.2).

2 State Machine Replication

The simplest network service uses a single, centralized server that exposes an application interface. Clients use the interface by sending requests to the server. While this design is simple, the service becomes unavailable if the server fails or becomes inaccessible, which is often undesirable and can become costly for critical services. A common technique to make the system more fault-tolerant is to replicate the server to several physical machines. This is called state machine replication [1].

In the following, we first give a general introduction to replicate state machines (RSMs). We then discuss two RSM architectures in more detail. First, the log-based design that can replicate states of arbitrary sizes and is commonly used in scientific literature to date. Second, we propose a novel approach called the in-place architecture, which is optimized for small replicated states.

2.1 General Design

The general method of building a replicated state machine is simple. First, the service is represented by a deterministic state machine. A state machine consists of a (possibly unbounded) number of states and a set of transition functions that define changes from the current state to the next state based on some input (we refer to [7, Chapter 8] for a more formal definition). Afterward, multiple copies, called replicas, of the state machine are created and placed on multiple, independent servers. The following steps are then repeated during the operation of the service:

1. Replicas receive commands in client requests, which are treated as inputs to the state machine.
2. Replicas then agree on the command order.
3. Replicas execute the commands individually in the agreed order.
4. After the client's command was executed on enough replicas, the client is notified of the execution result.

Commands can modify or read the state machine's current state. The state machine must act deterministically to ensure that all replicas that have executed the same sequence of commands will be in the same state. Thereby, an RSM can provide strongly-consistent, i.e., linearizable [2], access to the replicated state while tolerating individual replica failures.

2.2 The Consensus Problem

Agreeing on the command ordering is the most critical task of an RSM as it ensures replicas to not diverge over time. Solving this problem safely requires using a *consensus* protocol [8] such as Paxos [3, 9] or Raft [10][1].

A consensus protocol ensures that all (non-faulty) processes agree on a single value. Specifically, it must satisfy the following *safety* properties [9]:

Nontriviality: Some process must have proposed any value learned.
Stability: Any process can learn at most one value. A learned value cannot be changed later.
Consistency: Two different processes cannot learn different values.

Consensus protocols are typically based on *quorum* decisions [12]. A quorum is a set of processes that has a non-empty intersection with all other quorums in the system. Quorums are often defined so that every majority in the system is a quorum. To make progress, consensus protocols require the existence of a non-faulty quorum of processes. With majority based quorums, an RSM can tolerate a minority of failed replicas.

Although the existence of a non-faulty quorum is a necessary condition for progress, it is not sufficient. Fischer et al. [8] have shown that no consensus protocol can exist that guarantees progress in an asynchronous system, even if only a single process fails. Typically, a centralized coordinator called *leader* that orders concurrently submitted commands is elected to alleviate this problem [9]. Progress is then guaranteed as long as the leader does not fail.

3 The Log-Based RSM Architecture

The log-based RSM architecture is the approach commonly discussed in literature (e.g., [5, 9, 10, 13]). It uses a command log that intermediately represents the replicated state as depicted in Fig. 1.

[1] RSMs can also be implemented with atomic broadcasts [11]. Both problems are equivalent from a theoretical perspective.

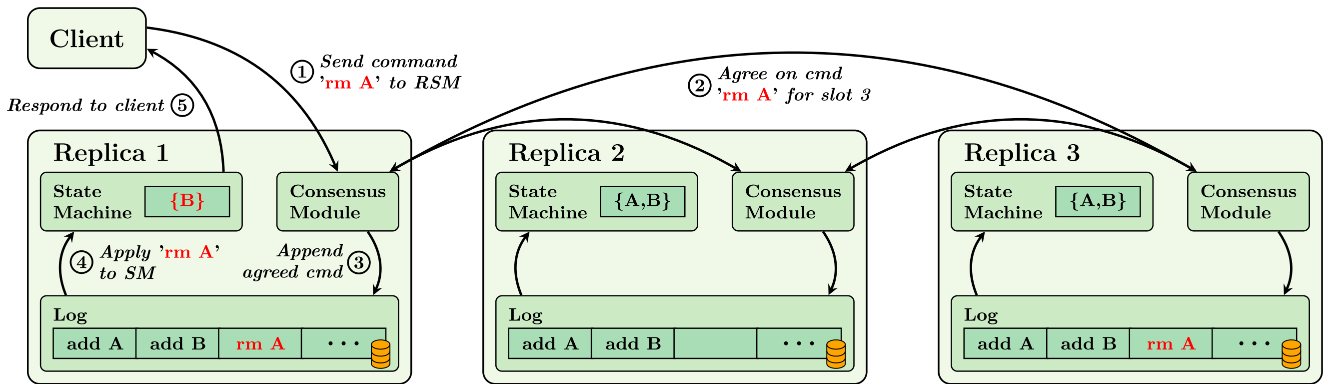

Fig. 1 Overview of the log-based RSM architecture. The replicated state consists of a single set. As replicas work at arbitrary speed, some replicas have not applied the agreed upon command at time the client received a response

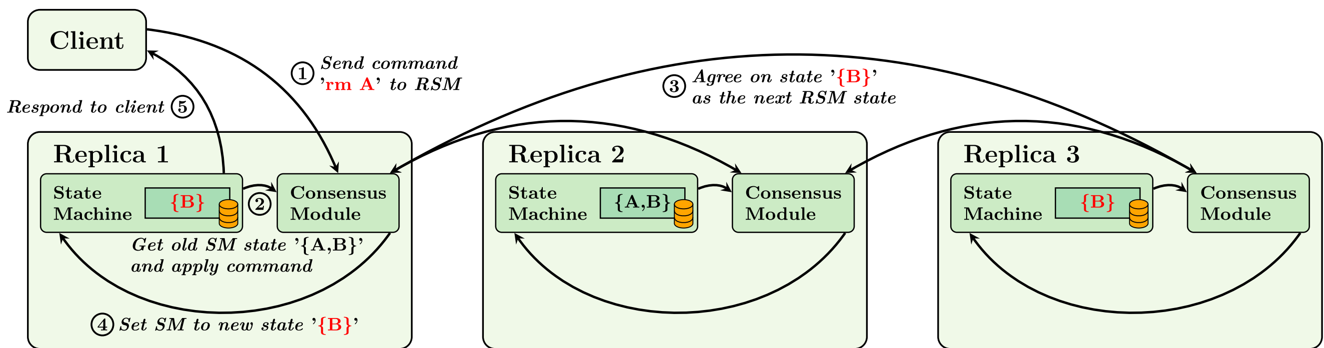

Fig. 2 Overview of the in-place RSM architecture. The replicated state consistent of a single set. In contrast to the log-based architecture, replicas agree on a sequence of replica states instead of a sequence of commands

3.1 Overview

Every replica consists of its local state machine's current state, a command log that contains a sequence of commands that the RSM has agreed upon, and a consensus module that exchanges messages with the other replicas by executing a consensus protocol. The command log is typically placed on a persistent medium so that a crashed replica, e.g., caused by a power outage, can recover by replaying the learned commands.

Clients can access the replicated state by sending a command to one of the replicas (step 1). The consensus module of this replica now proposes the command as a candidate for the next available slot in the command log (step 2). As replica 1 has already learned two commands in the example, it proposes the new command for slot 3. Note that the other replicas may also concurrently propose other commands for the same slot. However, the safety properties of consensus ensure that all replicas agree on the same command for slot 3. Once an agreement is reached, the chosen command is inserted into the log at the respective position (step 3). If the replica has also learned every previous command, it can apply the new command to its local state (step 4) and notify the client of the result (step 5).

Consensus is used in the log-based architecture to learn commands for specific slots in the log instead of directly agreeing on the RSM's next state. The agreed-upon order of commands is explicitly stored in the log of each replica. Thus, log-based RSMs directly realize the general steps outlined in Section 2.1.

This design has the advantage that the amount of transferred data is independent of the replicated state's size—the consensus module agrees on proposed commands. Thus, a command log fits well if the replicated state is much larger than individual commands.

3.2 The Cost of the Command Log

Managing the command log in an RSM is a challenging task. Several factors must be considered in an actual implementation. Most obviously, the command log has to be stored somewhere. If the replicated state is large, the implied storage overhead of managing a command log may be small in comparison. However, this is not the case if the replicated state itself is small. For instance, the RSM used in Fig. 1 only manages a single set. If the set itself remains small, then the command log quickly grows orders of magnitude larger than the actual state.

In addition to the direct storage requirements, a large command log also increases recovery time after a crash, as all commands have to be replayed first. Therefore, it is crucial to prevent the log from growing unbounded. However, it is not sufficient to discard the prefix of the log as then the full state cannot be recovered. A local snapshot of a replica's state is needed before the log can be truncated up to that point.

Adding a snapshotting mechanism first appears to be straight-forward because snapshots do not have to be synchronized across replicas. However, it introduces a high degree of complexity when building a productive system for several reasons [5]. These include:

- Command log and snapshot need to be mutually consistent. The possibility of a failed snapshot must also be considered.
- Serializing a large snapshot to a persistent medium is time and resource-intensive and can impact the replica's capacity to process new commands.
- A fallen-behind replica may need to catch-up by requesting missing commands. If the command was already truncated in other replicas, full snapshots must be transmitted, which can incur high bandwidth costs and delays.

Despite the numerous factors that must be considered when managing a command log, such state management is seldom discussed in literature in the context of RSMs. While some proposals sketch solutions to these issues, they are most often not mentioned. Only a minority of approaches discuss them in more depth. As noted by Chandra et al. [5], there is a gap between literature and production-ready use of log-based RSMs. This gap makes RSMs challenging to implement and results in protocols whose correctness remains unproven.

4 The In-Place RSM Architecture

We propose the in-place RSM architecture, which is optimized to replicate states of small sizes. As it does not rely on a command log, the complex state management outlined in Section 3.2 is not needed.

Although the general structure of the in-place architecture is similar to the log-based approach, the use of consensus is fundamentally different. Instead of agreeing on a sequence of commands for specific slots in a log, an in-place RSM agrees directly on the sequence of state machine states. Thus, it especially shines when the replicated state is small enough to be repeatedly transmitted over the network (i.e., the data management is latency-bound and not bandwidth-bound) and is combined with a storage medium that allows fast, byte-addressable access such as NVRAM (see Section 4.3). For example, in-place RSMs can be used for basic primitives such as counters, sets, locks, or small system-wide metadata such as current process group memberships or leaderships without the complexity of managing a log. Furthermore, the small storage and protocol overhead of an in-place RSM makes it easy to deploy many parallel, independent RSM instances that share the same physical machines. As we will discuss in Section 5 in more detail, this property is beneficial, for example, to replicate individual key-value pairs of a key-value store each in a separate RSM instance.

4.1 Overview

The high-level design of an in-place RSM, depicted in Fig. 2, resembles its log-based counterpart. Initially, a client submits a command to any replica (step 1). The replica's consensus module then proposes a new state as the next state of the RSM. It first fetches the replica's current state (step 2), applies the received command on the state, and then proposes the result by sending messages to the other replicas. If no concurrent proposal of another replica exists, this proposal will eventually be accepted as the next state of the RSM (step 3). Every replica that learns the new state can immediately override its current local state (step 4). The local state must be stored on a persistent medium if replicas need the ability to recover from crashes. Finally, the client is notified that its command succeeded (step 5).

In contrast to using a log, no replica explicitly stores the sequence of commands. Instead, the sequence is only implied by the current replica state. Thus, replicas can quickly catch-up if they have missed past consensus decisions, e.g., due to message loss caused by unreliable communication channels. For example, replica 2 in Fig. 2 may not learn state '$\{B\}$'. However, it can still safely overwrite its local state if it learns a subsequent proposed state. In contrast, replicas in the log-based approach must learn *all* previous agreed-upon commands before new commands can be executed or have to request a potentially large snapshot of the state. Of course, the consensus protocol must be modified for in-place RSMs, ensuring that only up-to-date replicas succeed with their proposal.

4.2 Modified Consensus

Current consensus algorithms are either directly build around a command log [10], generalized dependency graphs that allow multiple equivalent command orderings [14], or can only be used to agree on a single command [9]. Systems that use the latter kind of algorithms typically chain multiple consensus instances—one for each slot in the log. However, the managed state grows in all cases with the number of learned commands, which eventually needs to be actively truncated. To meet the demands of our in-place

RSM architecture, we designed a new consensus algorithm that can agree on an arbitrary number of values in sequence (i.e., states of the RSM) without additional memory requirements.

Our algorithm is based on classical single-decree Paxos [9]. In the following, we describe the high-level approach. We refer to [15] for the full algorithm.

4.2.1 Paxos Consensus

Single-decree Paxos can be used by a set of processes to agree on a single value. Paxos distinguishes the roles of the proposer, acceptor, and learner. Proposers propose values, acceptors vote on the proposals, and learners collect acceptor votes and may learn a proposed value.

Paxos is executed in two phases, as shown in Algorithm 1. In the first phase, a proposer first chooses a (unique) round number. Rounds are used to order concurrent proposals. The proposer then announces to all acceptors its intent to propose a value in this round during the second phase. Acceptors reply with their current vote if they have not seen a higher numbered proposal.

Algorithm 1 Overview of single-decree Paxos (Learner role omitted)

```
    Proposer:
 1: on propose(v) from client:                                  (phase 1)
 2:     r ← a round larger then every previously seen round
 3:     send prepare(r) to all acceptors
 4: on ack(r_ack, r_vote, v_vote) from acceptor quorum:         (phase 2)
 5:     Let ack(r'_ack, r'_vote, v'_vote) be a msg with max r_vote
 6:     ▷ Propose client's value only if no acceptor has voted yet
 7:     v_prop ← (r'_vote = 0) ? v : v'_vote
 8:     send accept(r'_ack, v_prop) to all acceptors
    Acceptor:
 9: on initialise:
10:     r_ack ← 0, r_vote ← 0, v_vote ← ⊥
11: on prepare(r) from proposer p:                              (phase 1)
12:     if r > r_ack then
13:         r_ack ← r
14:         send ack(r_ack, r_vote, v_vote) to p
15: on accept(r, v) from proposer p:                            (phase 2)
16:     if r ≥ r_ack then
17:         r_vote ← r; v_vote ← v
18:         send voted(r_vote, v_vote) to learners
```

If the proposer has received a quorum of ack replies, it proceeds to phase 2. It proposes the most recent value received in phase 1 messages. If no acceptors have voted for a value yet, it can choose its own value, e.g., a value received from a client. The proposer sends the proposal to all acceptors, who vote for it if they have not seen a higher numbered proposal. Learners collect the votes.

Once a learner has received votes for the same proposal from a quorum, the included value was learned by consensus. The protocol ensures that the learned value never changes: Intuitively, as a quorum has voted for value v, every proposer will receive a least one instance of v in a phase 1 ack message. Thus, no other value will be proposed anymore.

4.2.2 RMWPaxos: Enhancing Paxos with Consistent Quorums

Single-decree Paxos can only be used to agree on a single value. If agreement on a sequence of values is needed, e.g., commands in an RSM, multiple single-decree Paxos instances are necessary.

To avoid the need to clean-up old instances, we extended single-decree Paxos so that a single Paxos instance can agree on an arbitrarily long sequence of values. We call this modified protocol RMWPaxos [15], as it supports the consistent application of arbitrary read-modify-write operations on a single value. Informally, the protocol supports the following semantics (see [15] for a formal problem statement):

Given a current agreed-upon state s and a set of concurrently received commands C, all processes agree on the next state s' such that $\exists c \in C : s' = c(s)$.

Already applied commands must never be 'lost'. Once s' was agreed on, the next agreed state in the sequence must be $c(s')$ for some command c. Thus, one of the main challenges in developing the protocol is reliably detecting the current agreed-upon state before applying a new command to propose the next value.

We achieve this by augmenting Paxos with the notion of *consistent quorums*. A quorum is consistent if all acceptors in it have voted for the same state. Otherwise, the quorum is inconsistent. We can use this concept to modify phase 2 of Paxos (see Fig. 3):

After a proposer p has completed phase 1, it checks if the quorum has responded consistently. It does so by comparing all included values in the received ack messages. If they are the same, then the quorum is consistent, the state is reliably established, and the corresponding command is successfully finished. Then, p knows with certainty that the system has agreed on this value, and no newer agreed-upon value exists. Thus, p can apply a client command and propose the result as the next value in the sequence. If p succeeds with its proposal, it can update its local state machine with the proposed value and notify the client that issued the command.

If the quorum is inconsistent, p does not know which was the last agreed-upon value (cf. Fig. 3, step two). In order to prevent the system from 'losing' commands, p cannot immediately propose a new value. Instead, it must propose the most recent value received in phase 1, similar to single-

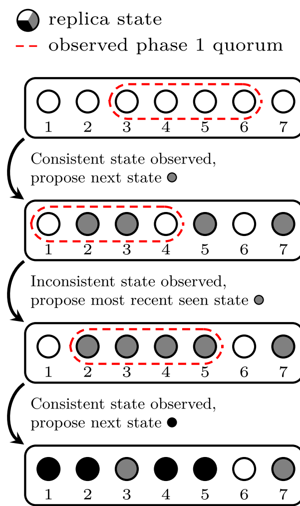

Fig. 3 Using the in-place consensus protocol to propose subsequent states of an RSM with 7 replicas (acceptors). Round numbers are omitted for simplicity

decree Paxos (cf. Algorithm 1, Line 4–8). Afterward, p can retry by executing the protocol from the beginning.

An inconsistent quorum may indicate that some other proposers executed the protocol concurrently. However, it can also be the result of message loss or other failures. In this case, the proposer cannot decide with certainty which (or if any) of the received values was agreed-upon judging from phase 1.

4.3 Persisting the State with NVRAM

The in-place RSM architecture promises less state overhead and protocol complexity than log-based designs when fault-tolerant access is needed on a fine-granular scale. Traditional persistent storage media such as HDDs or SSDs are not the best fit for the needs of in-place RSMs. They can only be accessed on a block granularity. Operating systems (OS) employ a paging mechanism to handle access to these block devices efficiently. Pages often have a size of 4 KiB, but they can also be larger on current systems. To access any data on the persistent storage, the OS fetches the corresponding page into main memory. If the data is modified, the page is marked as dirty. It is eventually written back to the persistent medium, either explicitly by the application or due to periodic synchronizations by the OS.

The use of multiple in-place RSMs, e.g., to replicate individual key-value pairs of a key-value store (Section 5) results in an access pattern dominated by fine-granular random-access writes. However, such an access pattern cannot be efficiently handled by the paging mechanism for several reasons: First, writing a full page upon every modification causes write amplification as the modified data is often much smaller than a single page. Second, different RSM instances may not share the same page, increasing the risk of page faults.

With the recent availability of 3D XPoint based memory, such as Intel's DCPMM, we now have a persistent storage medium that features byte-granular access with sub-microsecond latency. These properties are promising for the needs of in-place RSMs. While several performance studies exist that discuss DCPMM's applicability in various areas, none has evaluated the persistent access patterns that occurs when using in-place RSM's. Therefore, we investigate them in Section 6.2.

5 Use Case: Distributed KV-Store

Our modified consensus protocol RMWPaxos (see Section 4.2.2) makes it possible for an in-place RSM to agree on a sequence of values with a constant number of additional state variables. Periodic state truncation and management is not needed, which makes it easy and affordable to deploy an arbitrary number of RSM instances in parallel on the same set of physical machines. This property can be used to easily replicate individual key-value pairs in a distributed key-value store. We have outlined a possible architecture in Fig. 4.

A fundamental challenge in a distributed kv-store is to divide the key-space across multiple physical machines in a fault-tolerant manner. This can be achieved by using a structured overlay network, such as Chord [16] and Chord# [17].

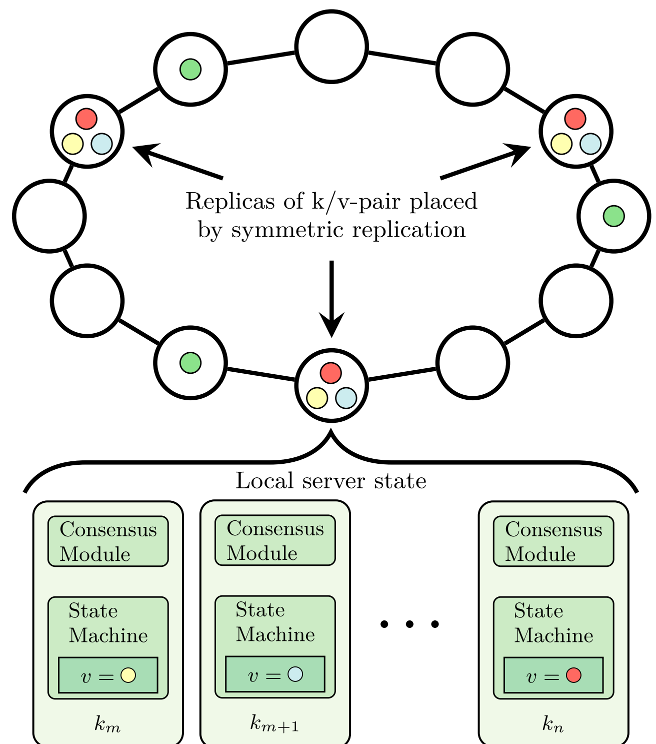

Fig. 4 Design of a replicated key-value store using in-place RSMs for each individual key-value pair

They arrange the keyspace in a virtual ring. Each server is responsible for a part of the keyspace, managing all key-value pairs in the assigned ring segment. A given key can be found by executing the routing protocol of the overlay network.

To improve data availability in case of a server failure, key-value pairs must be placed on independent servers, for example by symmetric replication [18]. The ring segments that servers manage do not have to be of the same size. Thus, two key-value pairs located on the same machine may have their replicas located on differing servers. For simplicity, this is not shown in Fig. 4.

The replicas of a key-value must be kept in sync to prevent clients from receiving inconsistent or outdated information when accessing a key (strong consistency model). For that, each key-value pair and its replicas can be handled as an instance of an in-place RSM. If a client wishes to access or modify a key's value, it can send the respective command to a replica. The replicas execute the consensus protocol outlined in Section 4.2.2 to compute the key-value pair's new state before sending a response back to the client. By this design, client requests can be processed consistently as long as a quorum (e.g., a majority) of replicas is available.

By itself, this approach can handle arbitrary commands that target a single key-value pair. To consistently handle modifications that span multiple key-value pairs, a multi-key transaction protocol [19] can be implemented on top of our in-place RSM.

6 Experiments and Evaluation

In this section, we evaluate two aspects regarding the use of in-place RSMs. First, we discuss and evaluate the effects of deploying in-place RSMs in an in-memory distributed system by using the distributed key-value store Scalaris as an example (Section 6.1).

We then investigate the influence and performance consequences when Intel's DCPMM is used as local persistent storage for in-place RSMs. Here, we compare the performance of the respective access patterns of the different approaches executed on RAM, NVDIMM-N, and DCPMM (Section 6.2).

6.1 Deployment in a Distributed KV-Store

We have implemented RMWPaxos in the context of Scalaris [20], a distributed key-value store written in Erlang. Scalaris features the architecture outlined in Section 5. A detailed evaluation can be found in [15].

The deployment of RMWPaxos in Scalaris to treat every k/v-pair as its own RSM revealed two advantages.

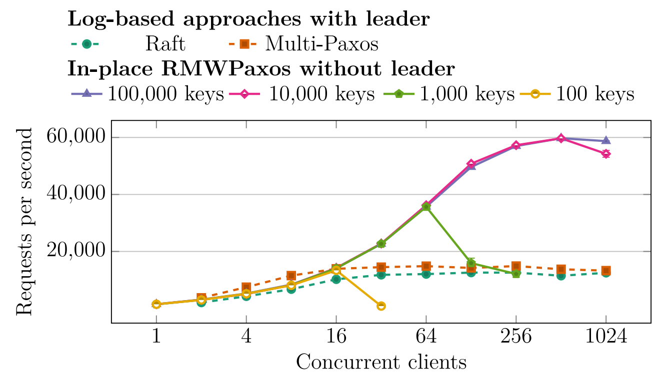

Fig. 5 Throughput comparison of RMWPaxos with leader-based approaches using 5 replicas. RMWPaxos can avoid the leader bottleneck by distributing load across many instances

First, it increases the flexibility of placing replicas of a given key on physical machines. The decision can be made on a per-key basis, which can be useful, e.g., to ensure geographic proximity in systems spanning across multiple data centers. Typically, this placement decision can only be made on the basis of individual shards or partitions when using log-based RSM approaches.

Second, it allows a better load distribution across multiple instances. As mentioned in Section 2.2, consensus algorithms typically benefit from using a leader to handle concurrency efficiently. Paxos also benefits from a leader since two proposers may repeatedly invalidate each other's proposals. However, the leader becomes the bottleneck of the system under high load. Deploying an RSM instance per key reduces the number of requests each individual instance has to handle. Depending on the access pattern exhibited by the system, it can even alleviate the need for a leader entirely.

To quantify this effect, we deployed Scalaris on a cluster with two Intel Xeon E5-2670 v3, 2.40 GHz per cluster node. All nodes were fully connected with 10 Gbit/s Ethernet links. We deployed five Scalaris nodes on separate cluster nodes, each representing one replica. Two separate cluster nodes were used to generate client requests. We deployed RMWPaxos with different system sizes, i.e., number of keys. The keys were accessed using a Pareto distribution with $\alpha = 1.16$ (80 % requests target 20 % of keys) to simulate skew. Fig. 5 shows the results using a read-dominated workload (95 % reads, 5 % writes).

6.2 Evaluating DCPMM for In-Place RSMs

We evaluate the use of DCPMM as local storage backend for in-place RSMs and compare the random-access pattern of multiple in-place RSMs to the sequential access of a log-based RSM implementation. Overall, we study the following three access patterns:

Log: Typical access pattern using a command log. Our log implementation resembles the implementation approach of the log in PMDK [21]. The log consists of a payload and a header containing the current log size. Log entries are written in two steps to ensure failure atomicity. First, the log entry is appended to the log and persisted. Next, the new log size is persisted in the header. For simplicity, we ignore log-truncation. Instead, we use a maximum log size of 500 MB, after which writes wrap-around and overwrite previous commands.

Single In-Place: The expected access pattern of a single in-place RSM. We overwrite the singular value using a simple copy-on-write scheme. The persisted value consists of a header, and two value versions. The header indicates which version was written last. As with our log, writes are executed in two steps. First, the old version is overwritten and persisted. Then, the header is updated in an atomic write.

Multi In-Place: Similar to Single In-Place, but with 100,000 independent values. Values are accessed using a Zipf distribution [22] with a high skew ($\alpha = 0.9$). This mimics multiple, independent in-place RSM instances as used, for example, to manage distinct key-value pairs in a key-value store.

Writes have to be performed in two steps as DCPMM can only guarantee write atomicity up to 8 byte values. Therefore, the use of a small 8 byte header is required to indicate that a larger write completed.

Although our evaluation is motivated by the use of in-place RSMs, these primitives are generally applicable. For instance, database systems often rely on write-ahead logging to ensure the durability of transactions. In contrast, the copy-on-write scheme used in the in-place access pattern can update values of arbitrary (albeit fixed) size in a failure atomic manner.

As a reference frame for the DCPMM performance, we also execute our measurements on both volatile RAM and NVDIMM-N. NVDIMM-N is a traditional DIMM with flash storage on the same module. The DIMM is accessed directly during regular operation. In the event of a power failure, data is copied from the volatile to the non-volatile medium with battery power. An overview of the used hardware is given in Table 1.

Table 1 Experimental setup

	RAM & NVDIMM-N	Optane DCPMM
Env.	CentOS 7 pmdk 1.8, GCC 9.1	CentOS 7 pmdk 1.8, GCC 9.1
CPU	2x Intel Xeon Silver 4116	2x Intel Xeon Platinum 8280L
Memory	192 GB DDR4, 32 GB NVDIMM-N	3 TB Intel Optane DCPMM

To manage access to the persistent media, we used PMDK's [21] `pmem_persist` function with temporal stores [23]. Writes to the volatile storage are simple `memcpy` calls. To avoid cross-NUMA effects, we used process pinning to utilize a CPU core on the same NUMA region as the used memory regions. Furthermore, we configured both DCPMM and NVDIMM-N to use DAX [24] to bypass the page cache of the OS.

Every measurement consists of 50 million writes. We show both the 99th percentile of the observed write latencies and the number of writes per second. Every measurement was repeated 10 times. The 99 % confidence interval is always within 5 % of the reported median. The results are shown in Fig. 6.

The Log and Single In-Place strategy exhibit near-identical performance on volatile main memory. In contrast, the Multi In-Place strategy is noticeably slower in both throughput and latency for writes larger than 256 bytes. This effect results from the significantly higher cache miss rate of the Multi In-Place strategy, caused by its random-access pattern. This effect could not be observed on both NVDIMM-N and Optane DCPMM, as cache lines are always invalidated by the flush performed as part of PMDK's `pmem_persist` call.

For smaller writes, NVDIMM-N outperformed DCPMM by up to a factor of 5. Such a large discrepancy was unexpected. However, we observed that DCPMM has a large penalty for repeatedly flushing the same cache lines. Previous performance studies made a similar observation, reporting an almost 8-fold increase in latency when flushing a single cache line compared to sequential access [25].

This effect impacts the Single In-Place strategy most, as it overwrites the same value repeatedly. This results in a lower throughput compared to the other strategies on DCPMM. However, Log and Multi In-Place are also affected. The header in Log must be updated after every append operation. Due to the high access skew of the Multi In-Place strategy, approx. 30 % of requests target 0.1 % of all RSM instances, whereas the most written instance is targeted by nearly 4.5 % of all requests. As writes are more distributed, Multi In-Place features a slightly higher throughput with lower latencies as the Log strategy.

The same effect causes the performance peak at 64 byte values (cache line size) for both In-Place strategies on

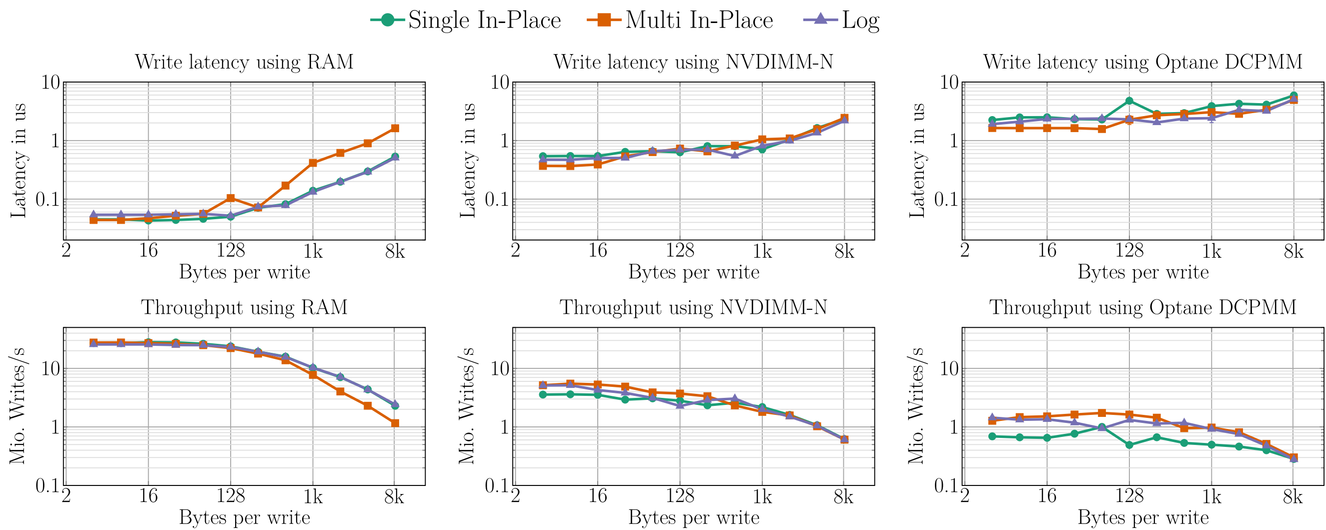

Fig. 6 Comparing different access patterns on volatile main memory and byte-granular, persistent memory

DCPMM. The cache line with the 8 byte header overlaps with multiple values or value versions due to the dense memory layout for smaller values.

Van Renen et al. [25] propose to alleviate this issue by using several fields for the header located on different cache lines, which are written alternating. This solution is also applicable to our In-Place strategies by maintaining more versions than just the latest two.

Even with our non-optimized implementations, Log and Multi In-Place achieved more than one million writes per second with less than 2 microseconds latencies for writes up to a kilobyte. This exceeds the write throughput of every existing consensus system known to us, for both Ethernet interconnects (e.g. [10, 26, 27]), and RDMA-aware systems (e.g. [13, 28]) leveraging low-latency interconnects like InfiniBand [29]. These results are promising for deployments of the in-place RSM architecture on systems with DCPMM or similar storage technologies available in the future. We conclude that persisting fine-granular state is likely no longer a performance bottleneck on such systems.

7 Related Work

7.1 Consensus

Many consensus protocols have been designed for the use in log-based state machine replication. Paxos-derived [3, 9, 14, 30] approaches typically chain a sequence of single-agreement consensus instances to agree on a log. Log pruning and other state management are not part of the core protocol, which Chandra et al. [5] identified as one of the main challenges for designing a productive system based on Paxos. In contrast, approaches like Raft [10] or Zab [31] are directly centered around the command log's semantics. However, this comes at the cost of less flexible protocols. For instance, Raft requires a leader for safety, whereas basic Paxos does not.

In recent years, consensus protocols were adapted to leverage the properties of low-latency, RDMA-aware interconnects [13, 28], programmable switches [27], as well as FPGAs [32]. These approaches are also centered around a log and require additional state management.

7.2 Persistent Main Memory

In recent years, various persistent main memory technologies are under active development. Intel's Optane DC Persistent Memory [6], 3D XPoint [4], became the first emerging technology with a commercially available product. A number of performance studies have been conducted to understand its properties in various settings, such as raw read/write performance [33, 34], basic primitives such as page propagation, and logging [25], as well as more complex index-structures [35–37] and storage engines [38].

However, to our knowledge, no other work exists to date that uses the byte-granularity of persistent main memory in the context of consensus to enable a more fine-granular use of replicated state machines. While the required persistent primitive—repeated in-place updates of a single value—is also used in some persistent date structures, we are not aware of any work that performs a direct evaluation of this primitive in isolation.

8 Conclusion

This article introduced the notion of in-place replicated state machines, which are optimized for replicating fine-granular states by avoiding the state management overhead associated with a command log. We have demonstrated that many independent in-place RSM instances can be deployed by using the distributed key-value store Scalaris as an example. Such granularity alleviates the need for a distinguished leader and improves load distribution across all replicas.

Until now, the use of in-place RSMs was restricted to domains where replication on volatile memory was sufficient, due to the properties of the currently available storage technologies. The first results on Intel's DCPMM indicate that the new upcoming byte-addressable persistent memory technologies provide the properties required for the fine-granular persistence that the in-place architecture offers. We believe that such fine-granular persistence may become more relevant as these new types of memory gain more exposure in future mainstream systems.

Acknowledgements This work received funding from the German Research Foundation (DFG) under Grant RE 1389 as part of the DFG priority program SPP 2037. The authors thank ZIB's core facilities unit for providing the machines and infrastructure for the evaluation.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Schneider FB (1990) Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput Surv 22(4):299–319
2. Herlihy M, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492
3. Lamport L (1998) The part-time parliament. ACM Trans Comput Syst 16(2):133–169
4. Handy J (2015) Understanding the Intel/Micron 3D XPoint memory. SDC, Santa Clara, CA, USA
5. Chandra TD, Griesemer R, Redstone J (2007) Paxos made live: an engineering perspective. ACM PODC, Portland, Oregon, USA, pp 398–407
6. Intel Corporation (2020) Intel Optane Technology. https://www.intel.com/content/www/us/en/products/docs/storage/optane-technology-brief.html. Accessed 28 Sept 2020
7. Lynch NA (1996) Distributed algorithms. Morgan Kaufmann, San Francisco
8. Fischer MJ, Lynch NA, Paterson M (1983) Impossibility of distributed consensus with one faulty process. ACM SIGACT-SIGMOD, Atlanta, Georgia, USA, pp 1–7
9. Lamport L et al (2001) Paxos made simple. ACM Sigact News 32(4):18–25
10. Ongaro D, Ousterhout JK (2014) In search of an understandable consensus algorithm. USENIX ATC, Philadelphia, PA, USA, pp 305–319
11. Agrawal D et al (1997) Exploiting atomic broadcast in replicated databases (extended abstract). Euro-Par, Passau, Germany, pp 496–503
12. Vukolic M (2012) Quorum systems: with applications to storage and consensus. Morgan & Claypool, San Rafael
13. Wang C et al (2017) APUS: fast and scalable Paxos on RDMA. SoCC, Santa Clara, CA, USA, pp 94–107
14. Lamport L (2005) Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33
15. Skrzypczak J, Schintke F, Schütt T (2020) RMWPaxos: fault-tolerant in-place consensus sequences. IEEE Trans Parallel Distrib Syst 31(10):2392–2405
16. Stoica I et al (2001) Chord: a scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM, San Diego, CA, USA, pp 149–160
17. Schütt T, Schintke F, Reinefeld A (2007) A structured overlay for multi-dimensional range queries. Euro-Par, Rennes, France, pp 503–513
18. Ghodsi A, Alima LO, Haridi S (2005) Symmetric replication for structured peer-to-peer systems. DBISP2P, Trondheim, Norway, pp 74–85
19. Schintke F et al (2010) Enhanced Paxos commit for transactions on DHTs. IEEE/ACM CCGrid, Melbourne, Victoria, Australia, pp 448–454
20. Schütt T, Schintke F, Reinefeld A (2008) Scalaris: reliable transactional P2P key/value store. ACM SIGPLAN workshop on ERLANG, Victoria, Canada, pp 41–48
21. Intel Corporation (2020) Persistent Memory Development Kit. https://pmem.io/pmdk/. Accessed 28 Sept 2020
22. Newman ME (2005) Power laws, Pareto distributions and Zipf's law. Contemp Phys 46(5):323–351
23. Intel Corporation (2019) Extended memcpy in PMDK 1.5. https://pmem.io/2019/01/22/extended-memcpy.html. Accessed 28 Sept 2020
24. Wilcox M (2014) DAX: page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/. Accessed 28 Sept 2020
25. van Renen A et al (2019) Persistent memory I/O primitives. DaMoN 2019, Amsterdam, The Netherlands, pp 12:1–12:7
26. Moraru I, Andersen DG, Kaminsky M (2013) There is more consensus in egalitarian parliaments. ACM SOSP, Farmington, PA, USA, pp 358–372
27. Li J et al (2016) Just say NO to Paxos overhead: replacing consensus with network ordering. USENIX OSDI, Savannah, GA, USA, pp 467–483
28. Poke M, Hoefler T (2015) DARE: high-performance state machine replication on RDMA networks. HPDC, Portland, OR, USA, pp 107–118
29. Pentakalos OI (2002) An introduction to the InfiniBand architecture. CMG, Reno, Nevada, USA, pp 425–432
30. Lamport L (2006) Fast Paxos. Distrib Comput 19(2):79–103

31. Junqueira FP, Reed BC, Serafini M (2011) Zab: high-performance broadcast for primary-backup systems. IEEE/IFIP DSN, Hong Kong, China, pp 245–256
32. István Z et al (2016) Consensus in a box: inexpensive coordination in hardware. USENIX NSDI, Santa Clara, CA, USA, pp 425–438
33. Yang J et al (2020) An empirical guide to the behavior and use of scalable persistent memory. USENIX FAST, Santa Clara, CA, USA, pp 169–182
34. Izraelevitz J et al (2019) Basic performance measurements of the Intel Optane DC persistent memory module. arXiv:1903.05714 [cs.DC]
35. Götze P, Tharanatha AK, Sattler K (2020) Data structure primitives on persistent memory: an evaluation. arXiv:2001.02172 [cs.DB]
36. Coburn J et al (2011) NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS, Newport Beach, USA, pp 105–118
37. Yang J et al (2015) NV-tree: reducing consistency cost for NVM-based single level systems. USENIX FAST, Santa Clara, CA, USA, pp 167–181
38. Arulraj J, Pavlo A, Dulloor S (2015) Let's talk about storage & recovery methods for non-volatile memory database systems. ACM SIGMOD, Melbourne, Australia, pp 707–722