Journal of Internet Services and Applications, 2024, 15:1, doi: 10.5753/jisa.2024.3891

Reducing Persistence Overhead in Parallel State Machine Replication through Time-Phased Partitioned Checkpoint

Everaldo Gomes Jr. [**University of São Paulo** | ***everaldogjr@gmail.com***]
Eduardo Alchieri [**University of Brasília** | ***alchieri@unb.br***]
Fernando Dotti [**Pontifical Catholic University of Rio Grande do Sul** | ***fernando.dotti@pucrs.br***]
Odorico Machado Mendizabal [**Federal University of Santa Catarina** | ***odorico.mendizabal@ufsc.br***]

Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Eng. Agronômico Andrei Cristian Ferreira, s/n, Florianópolis, SC, 88040-900, Brazil.

Received: 30 November 2023 • **Accepted:** 14 May 2024 • **Published:** 26 July 2024

Abstract Dependable systems usually rely on replication to provide resilience and availability. However, for long-lived systems, replication is not enough since given a sufficient amount of time, there might be more faulty replicas than the threshold tolerated in the system. In order to overcome this limitation, checkpoint and recovery techniques are used to update and resume failed replicas. In this sense, checkpointing procedures periodically capture snapshots of the system state during failure-free execution, enabling recovery processes to resume from a previously stored and consistent state. Nevertheless, saving checkpoints introduces overhead, requiring synchronization with the processing of incoming requests to prevent inconsistencies. This overhead becomes even more pronounced in high-throughput systems like Parallel State Machine Replication, where workloads dominated by independent requests leverage multi-threading parallelism. This work addresses the costly nature of checkpointing by proposing a novel approach that divides the replica's state into partitions and takes snapshots of only a few partitions at a time. Replicas continue executing requests targeted to other partitions without interruption. Thus, incoming requests experience delays during a checkpoint only if they access a partition currently being saved. Combining this approach with the Parallel State Machine Replication yields reduced snapshot durations and lower client latency during checkpointing. Additionally, the proposed approach accelerates replicas recovery through collaborative state transfer, enabling workload distribution among replicas and parallel execution of transfer and installation of the recovering state.

Keywords: Checkpoint/restore, Recovery, State Machine Replication, High-availability

1 Introduction

Numerous contemporary distributed systems must operate at a high level of availability, ensuring strong consistency even in the face of failures. Particularly in large-scale systems, failures are not uncommon, with service downtime carrying significant repercussions, as documented in cloud services incidents [Amazon, 2012a,b; Gitlab, 2017] and in the literature [Huang *et al.*, 2017].

Replication is a crucial technique for ensuring availability in the presence of faults. State Machine Replication (SMR) [Lamport, 1978; Schneider, 1990] is a widely adopted replication approach, providing both availability and strong consistency. In SMR, client requests are totally ordered and are then delivered to service replicas that execute them in the received order. Given that replicas initiate from the same state and process requests deterministically and in identical order, they progress through identical system states, thereby ensuring linearizability [Herlihy and Wing, 1990].

In order to improve the performance of SMR, various strategies have been proposed, aiming to execute requests concurrently across replicas by benefiting from application semantics. These strategies, referred to as Parallel State Machine Replication (PSMR) [Kotla and Dahlin, 2004; Kapritsos *et al.*, 2012; Marandi and Pedone, 2014; Alchieri *et al.*, 2017; Mendizabal *et al.*, 2017a], parallelize the execution of commands by categorizing them as either *dependent* or *independent*. Commands are *independent* if they operate on disjoint portions of the replica's state or only read shared state without modifying it, and *dependent* otherwise. Maintaining consistency requires dependent commands to be processed in the same relative order across all replicas. Conversely, independent commands can be executed concurrently, benefiting from mainstream multi-core servers.

While replication strategies effectively mask failures, having a quorum of correct replicas operational is indispensable to maintain service functionality. To uphold the desired levels of availability, recovery mechanisms should enable the reintegration of faulty replicas into execution or the introduction of new replicas into the system. Otherwise, it could lead to a situation where, over time, all service replicas may become unavailable.

Recovery mechanisms implement techniques to ensure the durability of the service state, encompassing methods like logging, checkpointing, and state transfer [Elnozahy *et al.*, 2002; Bessani *et al.*, 2014]. Throughout execution, replicas maintain a log of executed operations, allowing a faulty or new replica to restore a consistent state by replaying the log obtained from a correct replica. However, the log may grow indefinitely, rendering both storage and recovery procedures impractical. To impede unbounded log growth, checkpoint procedures are imperative [Goulart *et al.*, 2023a; Egwutuoha *et al.*, 2013]. Replicas periodically save a snapshot of their states to persistent storage and truncate the log, discarding operations that happened before the saved state. Upon recovery, a replica adopts a valid snapshot, processes logged commands post-checkpoint and resumes handling incoming requests just like other replicas.

Checkpointing, however, is expensive and introduces performance degradation. Besides the increased latency and limited throughput caused by intensive I/O operations, creating checkpoints requires synchronization with incoming requests to establish a consistent and recoverable snapshot of the system's state. As the system's state grows, the I/O-intensive operations of creating checkpoints can evolve into a bottleneck. Therefore, checkpoints play a dual role. During regular operation, they adversely impact the average system throughput and introduce latency stalls and hiccups perceived by clients [Bessani *et al.*, 2014]. This calls for sparse checkpoints, as seen in some systems, e.g., [Kapitza *et al.*, 2012; Tiwari *et al.*, 2014; Frank *et al.*, 2021]. Sparse checkpoints, in turn, result in large logs, which can slow down the recovery process.

Beyond these tradeoffs, checkpointing becomes even more challenging with approaches that scale SMR's throughput, such as PSMR. In addressing such systems, our work introduces a checkpointing strategy to reduce overhead during regular operations while maintaining availability levels.[1] The fundamental concept takes advantage of existing PSMR architectures, allowing requests to be executed concurrently with checkpointing. To do so, our strategy assumes a partitioned state, a common feature in PSMR, and coordinates replicas to capture snapshots of subsets of partitions at a time. While commands to partitions undergoing checkpointing experience momentary delays, requests addressed to other partitions can be executed in parallel, preserving strong consistency.

The main contributions of our work are summarized as follows:

- The detailed presentation of a PSMR checkpoint/recovery approach based on the partitioned time-phased checkpoint algorithm;
- The demonstration of correctness arguments showing how our checkpoint algorithm preserves strong consistency among replicas in a crash-recovery model; and
- A performance assessment that compares the time-phased partitioned checkpoint with a traditional checkpointing approach and a low synchronization cost checkpoint using Copy-on-Write (CoW) [Bobrow *et al.*, 1972]. The main benefits of our checkpoint technique include reduced snapshot duration, higher average throughput during regular operation, decreased latency for requests that are blocked waiting for checkpoints to complete and faster recovery.

The rest of the paper is structured as follows. Section 2 introduces the system model and definitions. Section 3 discusses the main aspects of PSMR. Section 4 present our checkpoint/recovery approach. Section 5 describes our experimental evaluation. Section 6 surveys related work, and Section 7 concludes the paper.

[1]This paper is an extended version of previous research [Junior *et al.*, 2023], where we introduced the foundation of the proposed checkpoint strategy.

2 System Model and Assumptions

We assume a distributed system composed of interconnected processes. There is an unbounded set $C = \{c_1, c_2, ...\}$ of client processes and a bounded set $S = \{s_1, s_2, ..., s_n\}$ of n server processes, also referred as replicas. Server processes are multi-threaded. One dedicated thread runs a *scheduler*, while the others are *worker threads*. The total number of worker threads is given by the number of adopted state partitions.

We assume the crash-recovery failure model [Hurfin *et al.*, 1998; Aguilera *et al.*, 2000] and exclude malicious and arbitrary behavior, e.g., no Byzantine failures. When a replica is correct, all associated threads and the scheduler are correct. When a server process crashes, all associated threads also crash. Processes may crash and recover. Processes are equipped with volatile memory and stable storage. In the event of a crash, a process loses the content of its volatile memory. In contrast, the content of its stable storage remains unaffected, *i.e.*, stable storage data cannot be corrupted or lost. Also, logging-based mechanisms ensure consistency by discarding changes from an update interrupted by failure. However, the access speed to stable storage, such as Hard Disks or Solid State Drives, is slower than accessing in-memory variables.

In the crash-recovery model, crashes are not necessarily permanent. Therefore, a process p_i that crashed at time t may recover at a later time $t' > t$ and resume operation. If p_i recovers after t', messages exchanged in the time interval $[t, t']$ can be missed by p_i. Moreover, the content of p_i's regular variables before t' is lost due to the use of volatile memory. To prevent data loss when a process crashes, the essential information must be kept in stable storage. That way, the state information saved on the stable storage device during failure-free execution can be used for recovery. We assume f faulty or recovering servers out of $n = 2f + 1$ servers.

Processes communicate by message passing, using either one-to-one or one-to-many communication. One-to-one communication is through primitives $send(m)$ and $recv(m)$, where m is a message. If a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication is based on atomic broadcast [Défago *et al.*, 2004], whose main primitives are *broadcast*(m) and *deliver*(m).

Atomic broadcast ensures that (i) if a process broadcasts message m and does not fail, then there is some i such that eventually every correct process delivers (i, m); and if a process delivers (i, m), then (ii) all correct processes deliver (i, m), (iii) no process delivers (i, m') for $m \neq m'$, and (iv) some process broadcast m.

We implement atomic broadcast using Paxos [Lamport, 1998]. Paxos requires additional synchronous assumptions, but our protocols rely on the atomic broadcast as a black box and do not explicitly need these assumptions.

The consistency criterion for SMR is linearizability [Herlihy and Wing, 1990]. By design, the SMR approach keeps consistency in the presence of a subset of failed replicas, according to the assumptions. The mechanisms presented in this paper are concerned with re-establishing the replication level to keep system fault-tolerance in the long run: as such, they are required to recover a failed replica to a valid state, enabling it to further process commands delivered by the atomic broadcast to all replicas. A valid state is one achieved after having processed a prefix p of the total order of commands delivered by atomic broadcast, *i.e.*, a finite sequence of commands applied from the initial state. Any two replicas that processed (exactly) p, by the definition of SMR, have identical states. To express in a general way the correctness requirement to the mechanisms discussed in this paper, we state that any two replicas, after having processed the same prefix of the total order, regardless of crash-recovery events, achieve the same state.

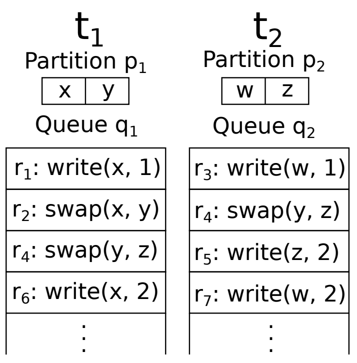
3 Parallel State Machine Replication

In traditional State Machine Replication [Lamport, 1978], a bounded set of replicas receives deterministic requests from an unbounded set of clients. These requests are ordered by using, for example, consensus [Lamport *et al.*, 2001; Ongaro and Ousterhout, 2014] or atomic broadcast [Défago *et al.*, 2004; Hunt *et al.*, 2010; Izraelevitz *et al.*, 2022] algorithms, establishing a total delivery order across replicas. By starting from the same initial state and executing the requests in the same order, replicas remain consistent.

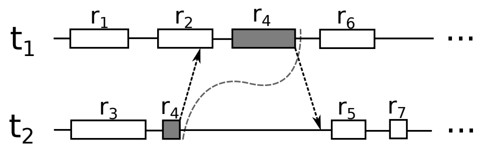
Parallel State Machine Replication (e.g. [Kotla and Dahlin, 2004; Marandi and Pedone, 2014; Alchieri *et al.*, 2017; Mendizabal *et al.*, 2017a]) stems from the observation that only dependent requests need to be executed sequentially, while independent requests can be executed in parallel. Two requests are considered independent if they access different memory position in the application state or if both are read operations. Otherwise, they are dependent.

This work follows an execution model similar to those proposed in Mendizabal *et al.* [2017a]; Marandi and Pedone [2014]; Li *et al.* [2018]. The application state, denoted as S, is partitioned into k disjoint sets, here called partitions. The set of partitions can be represented as $P = \{p_1, p_2, ..., p_k\}$ where $\bigcup_{i=1}^{k} p_k = S$. The system employs k worker threads, with thread t_i responsible for executing requests involving partition p_i. A request r operates on a set of memory positions, denoted as $M(r)$, and each memory position m is assigned to a specific partition, denoted as $P(m)$. Every thread t_i has its own queue q_i. When a request r is delivered by the atomic broadcast, it is placed in the queues of every thread whose memory position are accessed by r, formally represented as $r \rightarrow q_i \forall i | p_i \in \bigcup_{m \in M(r)} P(m)$.

As an example, consider a system with state $S = \{x, y, w, z\}$ and a set of threads $T = \{t_1, t_2\}$. Figure 1(a) represents a possible state partitioning, with threads t_1 and t_2 responsible for updates in partitions $p_1 = \{x, y\}$ and $p_2 = \{w, z\}$, respectively. The figure also depicts the queues of threads after receiving requests r_1 to r_7. Requests are delivered to replicas in the same order and dispatched to the appropriate queues respecting the delivery order. For this illustration, the requests are defined by the commands $write(k, v)$ and $swap(k_i, k_j)$, where $write$ updates the variable specified by the key k with value v, and $swap$ exchange the values of keys k_i and k_j. As shown, r_1 accesses x, and since $x \in p_1$, r_1 is dispatched to t_1. Similarly, r_3 accesses w, and as $w \in p_2$, r_3 is dispatched to t_2.



(a) Requests scheduling across thread queues



(b) An execution trace for the scheduled requests

Figure 1. Example of PSMR requests scheduling and execution

In order to represent a wide range of applications without lacking generality, requests are categorized into two types: *single-variable* and *multi-variable*. Single-variable requests involve accessing a single variable in the application state, such as simple reads or writes. On the other hand, multi-variable requests access two or more variables (e.g. swap or range scan), which can be further divided into *single partition* and *cross partition* requests.

Single-variable requests access a single memory position, so the cardinality of $M(r)$ for a single-variable request r is 1. In Figure 1(a) requests r_1, r_3, r_5, r_6 and r_7 are single-variable requests. As the requests are delivered to the same queue respecting the delivery order, deterministic execution is guaranteed without the need for synchronization.

Multi-variable single partition requests involve accessing multiple variables that belong to the same partition, *i.e.*: given a request r, $\forall x, y \in M(r) | P(x) = P(y)$. These requests can be executed by the same worker thread t. Request r_2 in Figure 1(a) is an example of multi-variable single partition request. Since the request involves variables x and y, both belonging to the same partition of thread t_1, it can be executed without synchronization.

Multi-variable cross partition requests access two or more variables, with at least two of them belonging to different partitions, *i.e.*: $\exists x, y \in M(r) | x \neq y \wedge P(x) \neq P(y)$. Request r_4 in Figure 1(a) is a cross partition request as it accesses variables y from the partition of thread t_1 and z from the partition of thread t_2. These types of requests require threads' synchronization to maintain consistency by ensuring only one thread will update the partitions involved while others are blocked.

Figure 1(b) illustrates an execution trace for threads t_1 and

t_2 based on requests shown in Figure 1(a). Threads run in parallel and single-variable requests r_1, r_3, r_5, r_6 and r_7 are executed as soon as they reach the head of their respective thread queues. The multi-variable request r_2 involves variables x and y, both of which are executed by t_1. Hence, it can be executed without synchronization. The cross partition request r_4 is present in both t_1 and t_2 queues, so they synchronize using a barrier. Here we depict that t_2 retrieves r_4 from its queue and waits for t_1 at the barrier, represented by the dotted curve. Once t_1 retrieves r_4, all involved threads are ready, and r_4 is executed by a single thread (e.g. the one with the the lowest id, in this case, t_1). Afterwards, both t_1 and t_2 are allowed to continue. During the time t_1 executes r_4, t_2 remains idle.

4 Time-Phased Partitioned Checkpointing

This section presents a checkpoint approach aiming at reducing overhead both during normal operation and at recovery time. The idea is to split the service state into partitions, allowing for successive snapshots at the partition-level. At regular intervals, the checkpointing procedure selectively saves the state of a few partitions at a time. During the saving of a partition's state, only requests addressed to those specific partitions are temporarily blocked, while independent requests targeted to other partitions can be executed in parallel with the ongoing partitions checkpoint. Once a partition checkpoint is complete, the blocked requests addressed to that partition are released to execution.

Considering workloads composed mainly by single-partition requests, partitions advance independently. Partition's checkpoints are also taken independently and thus consolidate updates up to different heights of the delivered total order. It is up to the recovery procedure to consistently continue from those heights.

In the presence of cross-partition requests, a checkpoint procedure might overlook the need for cross-partition synchronization, potentially resulting in inconsistent partition checkpoints. For example, suppose a command $swap(x, y)$ is executed, the x partition has checkpointed, storing x with its new value (*i.e.*, the original y value), but y partition is not checkpointed. In this case, the last checkpoint for partition y still has the unmodified y value instead of the x value swapped during the command execution. In such scenarios, resolving checkpoint inconsistencies during recovery becomes necessary, requiring the assistance of advanced logging and recovery mechanisms. This approach bears similarities to the method described in Zheng *et al.* [2014], where fuzzy checkpoints are employed (see Section 6 for more details).

In contrast, our work explores the concept of time-phased, partitioned and consistent checkpoints. We ensure that if a partition is being checkpointed and has executed cross-partition requests since its last checkpoint, the affected partitions must be checkpointed at the same time. Depending on the workload and number of partitions, this approach can result in highly parallel checkpoint and request execution (in the case of workload dominated by single-partition requests) or, on the other side of the spectrum, follow the standard case where all partitions need to be checkpointed simultaneously (in the case of a high rate of cross-partition requests).

4.1 Replicas execution

To explore parallelism at partition level we assume the underlying PSMR model explained in Section 3 and assume that for each request it is known beforehand: (i) whether it is an update (write) operation; and (ii) the partitions it affects. Although any parallel SMR model fulfilling these assumptions could be used, we adopt the PSMR model described in Alchieri *et al.* [2017]. This model provides the necessary information and parallel execution framework to develop the partitioned checkpoint approach. Each replica stores the complete service state, which is divided into static partitions. Each object in the state is mapped to a specific partition. In Alchieri *et al.* [2017] the notion of conflict classes is also proposed. A conflict class defines a set of partitions that have to synchronize to execute a command. Each command c is classified into a conflict class and this information is stamped in $c.type$. Thus, the class indicates the workers threads, *i.e.*, partitions, that have to synchronize to execute c. Given a finite set of workers, each possible combination of workers maps to a single conflict class.

Algorithm 1 outlines the initialization of replicas. The service state is divided into n partitions, with each partition associated with individual checkpoint and log files. We assume that checkpoints are taken every Δ_{cp} scheduled requests. This interval could be expressed in time units without loss of generality. In fact, the frequency of checkpointing partitions could also be different among replicas once the recovery procedure searches for the most updated partitions, not necessarily from the same replica. To handle cross-partition requests, we propose the use of a *conflict matrix* to track which partitions are involved by requests executed. Line 1 initializes the *conflict matrix* as an identity matrix. More details about the conflict matrix and checkpointing execution are presented in Section 4.2. Worker threads' queues are initially empty, and each replica sets the next partition to be saved during a checkpoint according to its identifier (lines 2-4), such that different partitions are saved by different replicas at each checkpoint interval Δ_{cp}. No commands were executed in any partition (line 5) and logs are initially empty (line 6). Before starting the execution, the replica restores a consistent state by running the recovery procedure (line 7). In short, the recovery procedure retrieves the most recent checkpoint and log files for each partition. It installs state partitions from checkpoint files and feeds the input queues of each worker thread with the respective log of commands after the checkpoint. The recovery procedure is presented in detail in Algorithm 8. After recovery, the total order height of the last command processed in each partition is stored in $lstEx[]$. Now the replica requests the consensus (or atomic broadcast) layer to deliver commands starting from the lowest $lstEx[p]$, $p \in 0..n-1$, given by line 8. Finally, the worker threads and scheduler are initialized (lines 9-11), each worker associated to a state partition, and the scheduler delivers commands asked to the consensus (or atomic broadcast) layer.

Algorithm 2 describes the behavior of the *scheduler*. This

Algorithm 1 $init(rId, n, \Delta_{cp})$

```
        {$rId$ : $replicaID$, $n$ : $\#partitions$, $\Delta_{cp}$ : checkpoint Interval}
1: $conf[n][n] \leftarrow \mathcal{I}$                                  {Conflict matrix}
2: $t\_queue[0..n-1] \leftarrow \emptyset$                 {One queue per worker thread}
3: $scheduled \leftarrow 0$                        {Number of scheduled commands}
4: $next\_cp \leftarrow rId \mod n$                 {Next partition to be saved}
5: $lstEx[0..n-1] \leftarrow 0$       {Last command executed in each partition}
6: $\mathcal{L}_i, i \in 0..n-1 \leftarrow \emptyset$                      {A log per partition}
7: $recover()$                              {Synchronize with other replicas}
8: $restartTODeliverFrom(lowestFrom(lstEx[0..n-1]))$
9: for $i = 0..n$ do
10:     start $worker(i)$;                          {One worker per partition}
11: start $scheduler()$;                            {Initialize the scheduler}
```

thread receives a totally ordered sequence of requests from clients and retrieves the list of threads involved c's execution (lines 1-2).[2] Then, the scheduler adds c to the queues associated to each of these partitions (lines 3-5), if c is higher than the last command executed at that partition (line 4). This is to eliminate duplicates since, after recovery, different partitions may be at different heights of the total order and we express here that commands are retrieved from consensus or atomic broadcast as a total order, after the lowest last command executed at each partition – as seen in line 8 of Algorithm 1. To keep consistency among partitioned checkpoints, whenever a checkpoint for partition p_i is created, any partition p_j that executed cross-partition commands accessing p_i, since p_i's last checkpoint, must be checkpointed together with p_i. Therefore, the scheduler keeps track of which threads have to synchronize to execute c. This mapping of conflicts involving multiple partitions is encoded in the conflict matrix (lines 6-8), discussed in detail in Section 4.2.

Every Δ_{cp} commands received, the scheduler decides to take a snapshot of a service partition (line 10). The service partition to be saved is determined by the value of $next\_cp$, which is updated in a round-robin fashion (line 16). The scheduler creates a checkpoint command cp and associates it to every thread responsible for the partitions in conflict as recorded in $conf$ (lines 11-14). Consequently, when executed, these threads will save the partitions they manage in individual image files during the checkpointing process.

Algorithm 2 $scheduler()$

```
1: while deliver(c) do              {atomic broadcast delivers a command $c$}
2:     $cfWrkrs \leftarrow getWrkrs(c.type)$   {one or more workers (cross)}
3:     for $i \in cfWrkrs$ do                      {add c, if not already}
4:         if $after(c.height, lstEx[i])$ then              {...processed to}
5:             $t\_queue[i].add(c)$                        {...the input queue}
6:     for $i \in cfWrkrs$ do                  {record each pair of workers}
7:         for $j \in cfWrkrs$ do                            {...that conflict}
8:             $conf[i][j] \leftarrow 1$        {for single-partition i=j, the identity}
9:     $scheduled \leftarrow scheduled + 1$
10:    if $scheduled \mod \Delta_{cp}$ then            {if it is checkpoint time}
11:        $partSet \leftarrow conflictWith(next\_cp)$                  {Alg 4.}
12:        $cpRreq \leftarrow \langle CP, partSet \rangle$              {the checkpoint request}
13:        for $id \in partSet$ do                       {... is enqueued at each}
14:            $t\_queue[id].add(cpReq)$             {... conflicting partition}
15:        $next\_cp \leftarrow (next\_cp + 1) \mod n$
16: function getWrkrs(conflict class)
17:     returns set of workers for a conflict class
```

[2]For the sake of simplicity, we do not demonstrate algorithms concerning the client side. Typically, they issue requests to a proxy that broadcasts requests to the replicas.

Algorithm 3 describes the worker threads behavior. Each thread retrieves the first command from its queue and extracts the *ids* of all threads associated with the conflict class, adding them to a list called $crWrkrs$ (lines 1-3). The thread with the lowest *id* among them is chosen to execute the command (line 4). A barrier is used to synchronize the execution of the command, ensuring that the thread with the lowest id waits for the other threads (lines 6-7). Once all threads have reached the barrier, the selected thread executes c (line 8) and records this command in the log associated with the corresponding partition (line 9). Finally, the thread signals the other threads to proceed with their execution (lines 10-11). If there are no other threads in $cfWrkrs$, the thread with lowest identifier does not have to wait and can directly execute and log the command. The workers that were not selected to execute the command have to signal the $executor$ thread when they reach the barrier and await the command execution (lines 13-14).

Algorithm 3 $worker(w_{id})$

```
                                           {$w_{id}$ : worker thread Id }
1: while true do
2:     $c \leftarrow head(t\_queue[w_{id}])$;           {get command $c$ to execute}
3:     $cfWrkrs \leftarrow getWrkrs(c.type)$  {find workers to synchronize}
4:     $executor \leftarrow min(cfWrkrs)$      {defines which one executes $c$}
5:     if $w_{id} = executor$ then                   {if this thread executes...}
6:         for all $d \in cfWrkrs \wedge d \neq w_{id}$ do
7:             $wait(t_d)$;            {await signal from all others, if any, ...}
8:         exec($c, w_{id}$);                              {...then execute $c$}
9:         $\mathcal{L}_{w_{id}}.add(c)$                      {log it in the partition's log}
10:        for all $d \in cfWrkrs \wedge d \neq id$ do
11:            $sign(t_d)$                      {signal other threads to continue}
12:    else
13:        $sign(t_{executor})$                        {signal executor - in line 7}
14:        $wait(t_{w_{id}})$           {await it finished exec - signal in line 11}
```

4.2 Conflict matrix

As observed in Algorithm 2 (lines 6-8), the scheduler thread is responsible for annotating which partitions were accessed by cross-partition commands in the current execution interval. These interactions are registered through a conflict matrix, denoted as $conf$. The *conflict matrix* is a square matrix of order n, where n is the number of state partitions. Each line or column index represents a partition, and each cell in the conflict matrix can assume either the value 0 or 1. A cell $conf[i][j] = 1$ indicates that partitions i and j executed a cross-partition command, meaning there was interaction between them during the current execution interval. $conf[i][j] = 0$, otherwise. Initially, the conflict matrix is initialized as an identity matrix.

To exemplify the use of a conflict matrix, consider the requests r_1 to r_7 in Figure 1, along with the following matrix that represents the replica conflict matrix after the execution of these requests. Notice the matrix represents 4 partitions while requests in the example were addressed only to partitions p_1 and p_2.

$$conf = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

From the matrix, it can be observed that cross-partition requests involved partitions p_1 and p_2. This information is obtained from the values of cells $conf[1][2]$ and $conf[2][1]$ which are set to 1. In this case, request r_4 is the cross-partition request that caused these cells to be set to 1. All other requests in the execution example are single-partition.

Whenever Δ_{cp} commands are executed by the replica, the scheduler determines which partitions need to be saved in the current checkpointing (Algorithm 2, line 11). Algorithm 4 describes how to determine which partitions need be saved in the current checkpointing, based on the conflict matrix.

The input to the algorithm is $next\_cp$, which represents the initial partition to be saved in the checkpoint. Starting with $next\_cp$ in line 1, the algorithm calculates the set of conflicting partitions, be they directly conflicting with $next\_cp$ or conflicting with others already identified as conflicts. This is done by repeatedly (line 3) evaluating for new conflicts (lines 5 and 6) whenever a new conflict is added (in line 7).

After collecting the set of partitions directly or indirectly involved with $next\_cp$, these conflicts have to be cleaned from the conflict matrix (lines 9–12) avoiding false-positive interactions among partitions in the next checkpoint. Finally, the list of partitions that must be saved in the current checkpoint is returned (line 13).

Algorithm 4 $conflictWith(next\_cp)$

```
                  {$next\_cp$ : initial partition to be saved in the checkpoint}
 1: $partitions \leftarrow \{next\_cp\}$          {starting with partition to save}
 2: $modif \leftarrow true$
 3: while $modif$ do                       {whenever a conflict is found }
 4:     $modif \leftarrow false$                {reevaluate to find further...}
 5:     for all $p \in partitions, i \in 0..n-1$ do            {conflicts...}
 6:         if $conf[p][i] = 1 \wedge i \notin partitions$ then     {transitively}
 7:             $partitions \leftarrow partitions \cup i$            {add it}
 8:             $modif \leftarrow true$                  {...and check again}
 9: for $i \in partitions$ do                    {clean these conflicts}
10:     for $j \in partitions$ do
11:         if $i \neq j$ then                          {leave identity}
12:             $conf[i][j] \leftarrow 0$
13: return $partitions$          {return list of partition ids to be saved}
```

4.3 Partitioned checkpointing algorithms

The scheduler dispatches *checkpoint* requests to the threads responsible for the partitions whose states need to be saved. These checkpoint requests are handled as regular commands, and their execution is triggered by the execution of the $exec(c)$ function, as described in Algorithm 3, line 8. A typical command execution is illustrated by Algorithm 5. For each command received, the command type is checked and the appropriate procedure for executing the command is performed (lines 1 to 5). If the command is a checkpoint introduced by the scheduler, then Algorithm 6 is activated (lines 6 and 7). After executing the command, the thread updates the index of the last command executed on the partition (line 8).

As described in Algorithm 6, the replica initiates a checkpoint with the next id. For each partition to be saved, informed as parameter, it stores: its current state and the last command executed. Then the partition's log is truncated. After all partitions informed were saved, the checkpoint is

Algorithm 5 $exec(c, p_{id})$

```
              {$c$ : command to be executed, $p_{id}$ : partition to execute $c$}
1: if $c.op = OP_1$ then
2:     ...          {Execution code for the application command $OP_1$}
3:     ⋮
4: else if $c.op = OP_n$ then
5:     ...          {Execution code for the application command $OP_n$}
6: else if $c = \langle CP, partSet \rangle$ then
7:     $checkpoint(partSet)$                               {Algo. 6}
8: $lstEx[p_{id}] \leftarrow c.instance$        {record last executed command}
```

marked complete and finished. Notice that all partitions being saved are blocked for command execution.

Algorithm 6 $checkpoint(partSet)$

```
                                          {$partSet$ : set of partitions}
 1: $chkp_{id} \leftarrow newChkp()$        {next id to chkp, init persistency}
 2: $chkpTh \leftarrow \emptyset$
 3: for all $p \in partSet$ do
 4:     $chkpTh \leftarrow chkpTh \cup$               {new checkpoint thread}
 5:       $new\ thread$ :               {...for each partition, its thread}
 6:         $\mathcal{S} \leftarrow getSnapshot(p)$          {copy the partition state}
 7:         $l \leftarrow lstEx[p]$              {last command executed in p}
 8:         $write(\langle chkp_{id}, p, l, \mathcal{S} \rangle)$        {store it to stable device}
 9:         $truncate(\mathcal{L}_p)$                  {trim partition's log}
10: for each $t \in chkpTh$ do
11:     $wait(t)$                {await all checkpoint threads to finish}
12: $finishChkp(chkp_{id})$                      {mark chkp as complete}
```

Algorithm 7 $recv(ReqMetadata)$

```
Replica $r_{id}$ receives ReqMetadata (Algo. 8) from recovering replica $r_{rec}$
1: $resp \leftarrow \emptyset$
2: for all partition $p$ do
3:     let $chkp_{id} \leftarrow$ id of newest complete checkpoint having $p$
4:     let $l \leftarrow$ the last command in $chkp_{id}$ for $p$
5:     $resp \leftarrow resp \cup \langle p, l, r_{id}, chkp_{id} \rangle$
6: $send$(RespMetadata, $resp$) to $r_{rec}$
```

4.4 Recovery

During its normal execution, a service replica performs periodic checkpoints, enabling recovery replicas to restore their state based on the most recent partition checkpoints among the correct replicas. The fact that the service snapshots are individually stored favors rapid recovery, especially in multi-core architectures, as the snapshots can be transferred and installed in parallel. Besides speeding up recovery, collaborative state transfer balances the workload among the correct replicas during recovery.

Figure 1 illustrates how the recovery procedure works. When initiating recovery, the recovering replica requests all replicas to send metadata related to their partition checkpoints, as shown by the *ReqMetadata* request in the figure. Of particular interest for recovery is information about the last command processed in each partition. Upon receiving the metadata, the recovery replica compares the identifiers for each checkpoint partition in the other replicas, identifying the most up-to-date ones. In this illustration, replica r_0 has the most up-to-date checkpoint for partition 0, indicated by $cp_0 = 70$, while in replica r_1, cp_0 lags at 20. However, r_1 possesses the most up-to-date snapshots for partitions 1 and n ($cp_1 = 50$ and $cp_n = 50$). The recovering replica then requests the snapshot and log of the most updated partitions from the replicas, denoted by the *ReqPartState* request. As it receives the state images, the replica installs the checkpoints and appends the commands in the logs to the worker threads' queues. Notice that transferring and installing partition states occur in parallel, and some partitions may take longer than others to be restored. A barrier ensures that the recovering replica processes log commands only after all partition checkpoints are restored.

Algorithm 8 describes the recovery process. Upon restart (Algorithm 1), the replica triggers recovery, which sends the *ReqMetadata* request to all replicas in the system (lines 2-3). Other replicas respond to this as in Algorithm 7: in short, it responds with the most recent checkpoint for each partition, informing also the last command of the total order included in the state. Since a checkpoint may not include all partitions, information from different checkpoints of the responding replica may be returned. After receiving responses from the other replicas, the recovering replica chooses, for each partition, to recover from the most advanced one (highest last command - line 9). Observe that some replicas might crash before replying with their checkpoint information. It does not affect the recovery as the recovering replica awaits replies for a bounded time limit, represented by a $timeout$ (line 4). Still, suppose the replies take longer than the timeout. In that case, the recovery replica will repeat the operation until at least one correct replica informs its checkpoint metadata (see the while loop, lines 1-6). For each partition, in parallel (line 11), it then requests the respective origin replica for the partition, awaits its response (lines 12-13), installs the state, feeds the recovery log to the input queue of the respective partition's worker thread and records the last executed command at each partition (line 16). The recovery finishes once all partitions passed this process (lines 17-18).

Algorithm 8 recover()

```
1:  while $allRsps = \emptyset$ do                                  {Getting checkpoint metadata}
2:      for each $r_i \in \mathcal{R}$ where $i \neq id$ do          {For all service replicas ...}
3:          $send$(ReqMetadata) to $r_i$                        {...request checkpoint metadata }
4:      for each $r_i \in \mathcal{R}$ where $i \neq id$ or $timeout$ do
5:          if $recv$(RespMetadata, $m$) then                  {...receive metadata $m$ and}
6:              $allRsps \leftarrow allRsps \cup m$
7:  $rcvThrds \leftarrow \emptyset$                                     {set of recovery threads}
8:  for each partition $p$ do
9:      let $\langle p, l, r_{id}, chkp_{id} \rangle$ = entry with highest $l$ for $p$ in $allRsps$
10:     $rcvThrds \leftarrow rcvThrds \cup$                              {starts recovery thread}
11:     $new\ thread$ :                                   {...to request partition $p$ to $r_{id}$}
12:         $send$(ReqPartState, $p$, $chkp_{id}$) to $r_{id}$                       {request and}
13:         $recv$(RespPartState, $\mathcal{C}$, $\mathcal{L}$)         {receives *checkpoint* and *log*}
14:         $install(C)$                                      {installs the received state}
15:         $t\_queue[p] \leftarrow \mathcal{L}$                       {assign *log* to input queue of $p$}
16:         $lstEx[p] \leftarrow$ height of last command in $\mathcal{L}$
17: for each $t \in rcvThrds$ do
18:     $wait(t)$                                   {await all recovery threads to finish}
```

4.5 Correctness arguments

Starting from the initial state, consider that the same prefix p of commands is applied at any two replicas. The SMR approach ensures that at this point replicas have identical states. Here we want to argue this same property considering that a replica may have crashed and recovered before processing the whole prefix p. In the following, we show this considering the use of the partitioned checkpoint and recovery mechanisms proposed.

Besides safety, we also argue about the liveness of the system when the above mechanisms are employed.

4.5.1 Safety

Proposition 1. *A partition's snapshot and log are consistent.*

The system initiates with initial state and empty logs at all replicas, at all partitions. From Algorithm 3 we have that partitions process commands sequentially and from lines 8 and 9 we have that whenever a command is executed at a partition, it is logged at the partitions log. Thus, a partitions log reflects the set of commands applied against the initial state.

From Algorithm 2, line 10, we have that at each Δ_{cp} commands a checkpoint command is scheduled at involved partitions. Partitions process the checkpoint command as any other command, *i.e.*, sequentially (see Algorithm 5) and thus without interference. This means that the snapshot taken is a consistent state and comprises the cumulative effect of all commands before the checkpoint. The checkpoint is in Algorithm 6. Notice from line 8 the snapshot taken keeps, besides the state of the partition, a reference to the last command executed. Notice that a checkpoint is considered valid if it is complete, see line 12. We assume that line 12 is atomic w.r.t. crashes, *i.e.*, either the snapshot is marked complete or not. In the second case it does not exist for recovery. Right after the snapshot, the log is truncated at line 9. Inductively the last snapshot can be regarded as a new initial state, and thus a partitions log reflects at any time the set of commands applied against the respective last snapshot taken.

Proposition 2. *Commands are executed exactly once at partitions, in order, regardless of crash recovery.*

Supposing no crash-recovery events, from Proposition 1 we have that at each partition, at each replica, a command is applied exactly once, in consensus order, against the state.

In the event of a recovery, according to Algorithm 8, the recovering replica searches the highest checkpoint for each partition and then retrieves and installs each one. In line 13 we have that a partition snapshot and respective log are retrieved from a chosen source replica. Due to Proposition 1, the snapshot and log are consistent. The snapshot is installed as state and the log is processed next, as it is feed in the input queue of that partition. Thus, after this, the partition will have executed exactly the prefix of commands until the last one of the retrieved log.

According to Algorithm 1, after processing the log further commands are retrieved from consensus, starting from the lowest one needed to continue processing the snapshots used (line 8). Then, according to the scheduler thread Algorithm 2, line 4, a command arriving from consensus is only scheduled at a partition if it is higher than the last command applied to that partition. With this, it is ensured that commands after the log are correctly applied to each partition as appropriate. Since logging is sequential, according to consensus

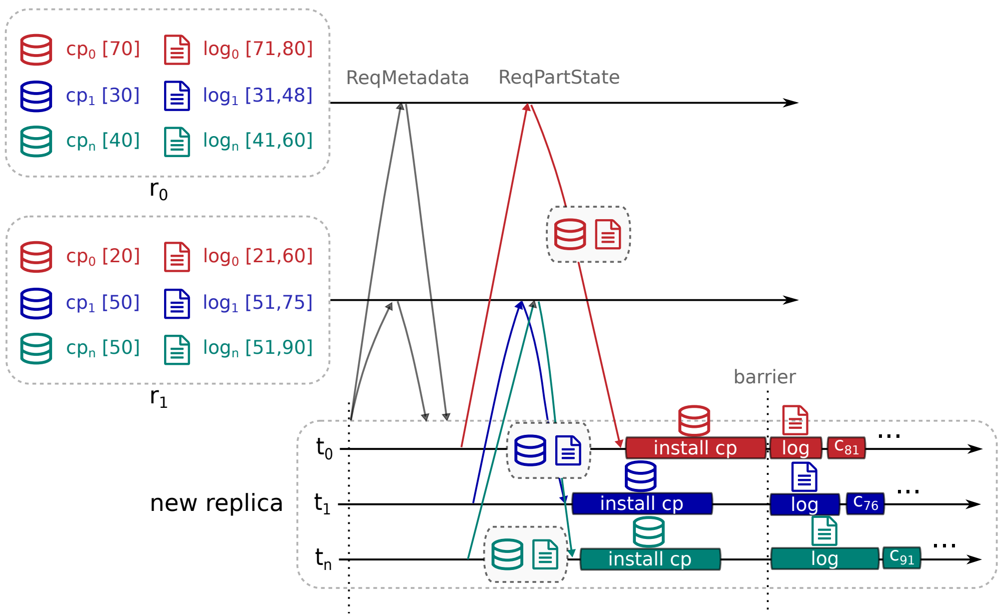


Figure 2. Recovery through collaborative state transfer.

order, log processing is equally sequential, and commands retrieved from consensus also in consensus order, it follows that each command is applied at a partition once and in consensus order regardless of crash-recovery events at the partition.

Proposition 3. *For independent workloads, it is possible to recover from any combination of partition snapshots retrieved.*

If partitions evolve independently, *i.e.*, without cross partition commands, by Proposition 2 one can derive that any combination of consistent pairs ⟨ snapshot, log ⟩ for different partitions can be used for recovery. If all partitions finish processing up to a given height of the total order, then each partition processed exactly once, in order, all commands addressed to it up to this height.

Proposition 4. *Snapshots taken are consistent for cross partition commands.*

From system initial state, or inductively after a checkpoint, a replica applies commands to partitions and records, from the previous point, which partitions are involved in cross-partition commands. This is performed by Algorithm 2, lines 6-8. When the next checkpoint moment comes, the set of conflicting partitions is calculated, line 11. Algorithm 4 returns the conflicting partitions since the last checkpoint (as well as resets the conflict matrix to start recording conflicts from the current checkpoint on). For the conflicting partitions, the checkpoint command *CP* is executed as a conflicting command, *i.e.*, the involved partitions process their inputs up to *CP* and synchronize. This is ensured by Algorithm 3, by the conflicting command execution procedure, and results that any cross-partition commands before *CP* are processed in all partitions involved. Therefore, a replica generates checkpoints to partitions that are consistent with cross-partition commands: *i.e.*, either a cross-partition command is reflected in the snapshot of all partitions involved, or none (it comes after *CP*).

Proposition 5. *Snapshots adopted during recovery are consistent for cross partition commands.*

The recovery procedure chooses the most advanced snapshot of each partition, regardless of replica, to recover from. This, and further processing procedures, are argued sufficient for independent commands at Propositions 2 and 3. Now we argue that it is also sufficient for cross partition commands.

By Proposition 4, a checkpoint generates snapshots of different partitions that are consistent w.r.t. cross-partition commands. Complementing, notice that a checkpoint is considered complete only after all needed partitions were checkpointed as stated in Algorithm 6, last line. Considering complete checkpoints, notice that if any partition p_i has a more advanced checkpoint w.r.t. any other partition p_j it is because p_i progressed with independent commands, otherwise whey would have both the snapshots taken at the same complete checkpoint. Therefore, recovering the most advanced snapshot for each partition – which is what the algorithm specifies – results in checkpoints that consistently preserve cross-partition commands.

From the above, we derive that if a replica has processed up to a given prefix of the total order, irrespectively of cross-partition commands and crash-recovery events, it will reflect, at each partition, exactly the same sequence of commands that another replica without any crash-recovery event.

4.5.2 Liveness

According to SMR a request can be executed and responded to the client once the previous ordered ones were executed. We assume the total order of commands to be ensured by consensus. Therefore, provided consensus instances are decided, liveness at SMR level is ensured. Additional assumptions for consensus termination are outside this discussion as we rely on consensus as a black box.

Regarding the recovery mechanisms proposed, we have to argue that recovery is always possible and finishes. Recall that any past command can be retrieved from consensus.

Also, according to discussed in Section 4.5.1 provided checkpoints are complete, *i.e.*, consistent w.r.t. cross partition commands, the technique can use partition information from different checkpoints by different replicas. With this, it is possible to recover from any complete checkpoint provided, even older ones, in the extreme case the initial state. Obviously more recent checkpoints favor faster recovery. Since $n = 2f + 1$ is assumed, there are always operational replicas that can provide recent partitions snapshots and accompanying logs.

5 Experimental evaluation

This section evaluates the *time-phased partitioned checkpointing* technique, comparing its performance with the traditional checkpointing model. The behavior of both strategies combined with the copy-on-write (CoW) is also evaluated, as this strategy aims to reduce synchronization costs during command execution while saving state. For comparison purposes, the following aspects were considered:

- How the frequency of checkpoints affects the average system throughput;
- The influence of the CoW technique in both partitioned and traditional checkpointing techniques;
- How the checkpointing overhead affects the system for different numbers of partitions; and
- How the recovery time is affected by the technique.

5.1 Prototype implementation

To evaluate the performance, the proposed protocols were integrated in BFT-SMaRt [Alchieri *et al.*, 2017; Bessani *et al.*, 2014] and a key-value store prototype was implemented. It consists of a set of key-value tables, where keys are represented by integer numbers and values are arrays of 1024 bytes, offering the following operations:

- *put(table, key, value)* - inserts into $table$ a $\langle$*key*,*value*$\rangle$ pair;
- *get(table, key)* - retrieves the *value* associated with *key* from *table*. If the key or the table is not found, it returns null;
- *swap(table*$_1$*,key*$_1$*,table*$_2$*,key*$_2$*)* - exchange the values associated with the keys *key*$_1$ and *key*$_2$, belonging to tables *table*$_1$ and *tables*$_2$, respectively;
- *multi_table_put(table[*t_1*,* t_2*, ...], key[*k_1*,* k_2*, ...], value[*v_1*,* v_2*, ...])* - inserts into tables *table*$_i$ the pairs $\langle$*key*$_i$,*value*$_i\rangle$ following the respective order of the parameters;

For the partitioning of the service state, each key-value table is considered a partition. For example, the operation *checkpoint(0)* stores on persistent storage the data contained in *table* 0. In the traditional approach, the *checkpoint* operation saves all tables of the service. This application fits well with representative examples of key-value store services broadly used in many large online services (e.g., Twitter [Schuller, 2014], Amazon [DeCandia *et al.*, 2007], and Facebook [Masti, 2021]) and their states are easy to shard based on the range of keys values.

Any two single-partition operations on the same partition are executed sequentially according to the total delivery order. Any cross-partition operation is ordered at all partitions involved, that cooperate to execute it, following the protocol of Section 4.1. For example, the operation $swap(0, 1, 3, 5)$ performs the swap between the values of *key* 1 in *table* (partition) 0 and *key* 5 in *table* (partition) 3. This operation is enqueued to both respective worker threads of *partitions* 0 and 3 and these workers synchronize when both reach the command to execute it.

Checkpoint operations also force synchronization among partitions. In the case of traditional checkpointing, all worker threads must synchronize. In the proposed technique, only the threads that executed cross-partition commands have to synchronize, as discussed in Algorithm 2, Section 4.

For load generation, a multi-threaded client randomly generates requests to be sent to the service, according to selected *read* and *conflict rates*.[3] Read commands are given by *get* operations and write commands are represented by *put* operations when accessing single-partition or *multi_table_put* when accessing cross partitions.

For this evaluation, the total size of the service state is 1GB divided into 4 equal size partitions (256MB), Table *id*s are unique and belong to the interval $[1..4]$. Tables are created and populated at initialization. To insert the keys, the command $put(table, key, value)$ is used. Operations *put* and *multi_table_put* were used to update the tables. Since each key (key) maps to an array ($value$) of 1024 bytes, each table has $250,000$ distinct keys. The state size does not change over time. This assumption is made to restrict the number of varying elements on the experiments.

5.2 Test environment

We used the Emulab platform [White *et al.*, 2002] with 4 physical machines for the experiments, consisting of 3 server replicas and 1 client replica. The choice of 3 server replicas is due to the fact that the service replicas are part of the consensus protocol, and it requires $2f + 1$ replicas to tolerate f faults. With this configuration, the service can tolerate the failure of 1 node without compromising its availability. The client machine instantiated a load generator with a variable number of threads responsible for generating requests. During the experiment, the Java Virtual Machine (JVM) heap memory was set to 32 GB. The machine specifications for the service replicas and the load generator are the same: 2 Intel Xeon 2.4 GHz 64-bit 8-core processors, 64 GB RAM, and a 200 GB 6Gbps SATA SSD disk.

5.3 Workload characterization

Given the application and deployment above, we ran experiments to ranging the number of client threads to reach the saturation point. These experiments were set up without checkpoints and failures. Each execution lasts 1 minute, consisting of 100% write operations. The service replicas were configured with 4 execution threads (4 partitions). Figure 3 shows the system's saturation point. We can observe the average

[3]We use 'conflict' or 'cross-partition operation' as synonyms.

throughput (horizontal axis) expressed in kilo commands versus the 90th percentile of latency (vertical axis) expressed in milliseconds.

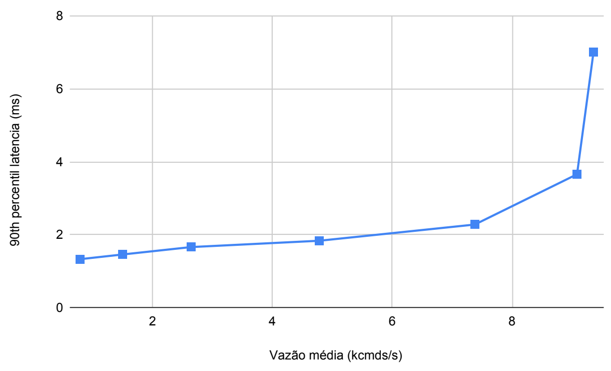


Figure 3. Saturation point for service configuration with 4 workers *threads* and exclusively written workload.

As we observed, after 8k commands/s response times clearly increase, indicating saturation. Further experiments were conducted at 70% of the saturation load, corresponding to 22 client threads.

The characteristics of workloads generated for the experiments differ in two aspects: (i) read or write operations; and (ii) single or cross-partition operations. In total, four different workloads were used, as listed next.

- 100% read, single-partition ($100r0c$) - this workload represents the best-case scenario in terms of performance, as reads are lightweight and the absence of cross-partition operations eliminates the need for thread synchronization;
- 90% read operations and 1% cross-partition ($90r1c$) - meaning, 90% reads and 10% writes, and within the total number of writes, 1% are cross-partition, involving two or more partitions. This workload represents scenarios mainly consisting of reads, with few updates and few conflicts. This setup fits well with the representative examples of key-value stores commented at Section 5.1. For example, Twitter uses a key-value store to store tweets that are usually written once and read multiple times. Consequently, such applications have a workload that contains mostly read operations. Moreover, the choice for 1% of conflicts stems from observations from the literature. Moraru *et al.* [2013] state that from the available evidence, dependency probabilities between 0% and less than 2% are the most realistic. For instance, in Chubby, for traces with 10 minutes of observation, fewer than 1% of all commands could possibly generate conflicts [Burrows, 2006]. In Google's advertising back-end, F1, which uses the geo-replicated table store Spanner, fewer than 0.3% of all operations may generate conflicts [Corbett *et al.*, 2012];
- 0% read operations and 100% cross-partition ($0r100c$) - this workload generates only write operations, all cross partitions, represented by *multi_table_put* involving two randomly selected partitions. This scenario represents the worst-case in terms of synchronization, as all requests require coordination between threads;
- 0% read operations and 0% cross-partition ($0r0c$) - this workload consists only of write, single-partition operations.

Read commands are given by *get* operations and write commands are represented by *put* operations when accessing single-partition or *multi_table_put* when accessing cross partitions. For the defined workloads, read operations do not cause conflicts between partitions. Therefore, a conflict relationship involving multiple partitions can only occur during the execution of write operations on multiple data.

To evaluate the impacts of the checkpoint interval, all previously described workloads were evaluated with intervals Δ_{cp} set at every 50,000, 100,000, 150,000, and 200,000 commands. Considering i as the i-th command received by the service replica, whenever $i \ mod \ \Delta_{cp} = 0$, the checkpoint procedure is executed.

5.4 Performance evaluation

Experiments were conducted using both traditional and partitioned checkpointing techniques. In the first case, a checkpoint containing a full image of the service state is taken every Δ_{cp} requests. In the partitioned approach, a target partition is saved every Δ_{cp} requests. Although known logging/checkpointing optimizations could be incorporated in our prototypes (e.g., optimistic logging, incremental checkpoints, system-level checkpointing, and dynamic or adaptive checkpoint intervals) [Goulart *et al.*, 2023a], we are interested in evaluating the impacts caused by our technique, then avoiding the interference caused by other techniques.

Figure 4 shows how the checkpoint periodicity (horizontal axis) affects the average throughput (vertical axis) for the $100r0c$ workload, which means a workload with 100% read operations and no conflicts between partitions. The graph displays the throughput as a function of the checkpoint interval for the following replica configurations: *Traditional* (traditional checkpointing technique), *Partitioned* (time-phased partitioned checkpointing technique), *GP-Traditional* (traditional checkpointing technique with conflicts involving disjoint groups of partitions), *GP-Partitioned* (partitioned checkpointing technique with conflicts involving only disjoint groups of partitions). Note that we differentiate the chances of conflicts involving any partitions from situations where conflicts always involve the same disjoint groups of partitions. In the second case, a swap command can involve only partitions 1 and 2, or partitions 3 and 4, but never partitions 1 and 3 or 4, or 2 and 3 or 4. In other words, group $\{1, 2\}$ is disjointed from $\{3, 4\}$.

It can be observed that the time-phased partitioned checkpointing outperforms the traditional technique in all evaluated scenarios. Furthermore, with larger checkpoint intervals, the traditional checkpointing achieves higher throughput, as expected. Note that even with partitioned checkpoints and a checkpoint interval of 50,000 commands, the throughput is approximately 4 times higher than that obtained with the traditional technique. Even when considering larger checkpoint intervals, such as 200,000 commands, the partitioned technique still shows improvement compared to the traditional technique. Since as the checkpoint intervals

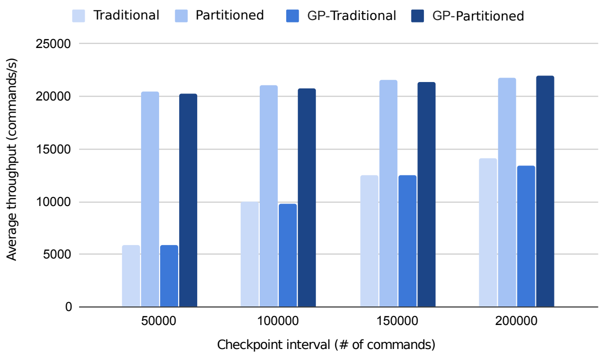


Figure 4. Average throughput using the workload $100r0c$ and 4 partitions.

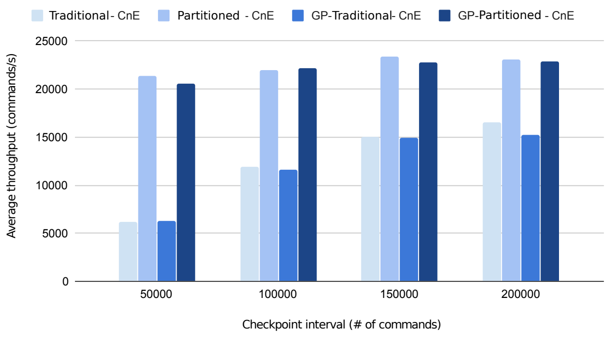


Figure 5. Average throughput using CoW with the workload $100r0c$ and 4 partitions.

become smaller, the cost of log reprocessing during recovery becomes less expensive, the reported result suggests that adopting a more frequent checkpointing intervals with the proposed approach can be advantageous. When comparing strategies with conflicts involving any partitions or disjoint groups of partitions, no significant differences are observed since the workload ($100r0c$) does not cause any conflicts.

Other techniques, such as copy-on-write (CoW) [Bobrow *et al.*, 1972], can reduce the overhead of checkpoint creation. This technique reduces the required synchronization by making copies of the state during checkpoint creation, allowing new commands to be executed in parallel with state saving. With CoW enabled, when a checkpoint thread needs to copy a memory page from the application to the snapshot, it maps it into the target address space and marks it read-only. If a worker thread tries to write to that page, it will notice that the page is a CoW page and thus (lazily) allocate a new page, fill it with the data, and map this new page into the application address space. The checkpoint procedure then continues copying data from its private copy of the page, which is unmodified.

However, a large amount of memory may be needed for duplicating data during state saving. Figure 5 compares the throughput results achieved with the CoW technique and our approach. The graph displays the throughput versus the checkpoint interval for the following replica configurations: *Traditional-CoW* (traditional checkpointing technique with CoW), *Partitioned-CoW* (partitioned checkpointing technique with CoW), *GP-Traditional-CoW* (traditional checkpointing technique with CoW and conflicts involving disjoint groups of partitions), *GP-Partitioned-CoW* (partitioned checkpointing technique with CoW and conflicts involving disjoint groups of partitions).

The use of Copy-on-write benefited the traditional checkpointing strategy in particular. Compared to Figure 4, the traditional technique showed higher throughput for all scenarios with checkpoint intervals equal to or greater than 100,000 commands. This optimization provided a slight gain for the cases where the time-phased partitioned checkpointing technique was used. Another experiment was conducted comparing different heap configurations. It was found that the traditional technique requires a larger amount of memory for state copies, causing the system to fail whenever the heap configuration was less than 6.5 GB. On the other hand, the proposed checkpointing technique requires less memory due to checkpoints at partition-level. It was necessary to configure the memory heap to 4.5 GB for the system to operate normally. For the next evaluated scenarios, we opted to not present the results using CoW. In general, the technique brings performance improvements to both approaches and can be considered an orthogonal optimization for both checkpointing approaches.

Figure 6 presents the result for the $0r0c$ scenario, where all client operations are writes. It can be observed that, due to the absence of conflicts, the partitioned checkpointing technique shows higher average throughput compared to the traditional model. Furthermore, it can be noticed that the average throughput of the partitioned checkpointing technique remains relatively stable, with minor variations in the checkpoint period. As expected, the average throughput is lower than in the previous scenario, where the workload was $100r0c$. It is because write commands are more costly than read commands.

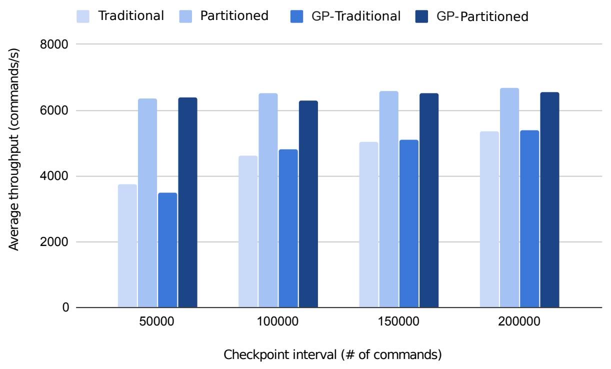


Figure 6. Average throughput using the workload $0r0c$ and 4 partitions.

Figure 7 shows the results obtained with the workload $0r100c$, where 100% of the commands are write and involve multiple partitions. For request generation, variables are randomly selected to access two different partitions using the $multi\_table\_put$ request. This scenario represents the most costly case in terms of synchronization since 100% of the commands involve multiple partitions. The measured throughputs are similar to those observed in Figure 6, where no conflict occurs. This result suggests that despite the need to anticipate checkpoints from other partitions at each interval, the partitioned checkpointing provides gains as it stores state partitions in parallel, reducing the checkpoint duration. Although there are periods when all commands are blocked, this period is shorter with our technique. When a checkpoint ends, queued commands are processed quickly, as the system is able to process more than the 6000 commands/s. What is unnoticed in this graph is the latency impact the clients observe while their requests are blocked during the checkpointing. Figures 9 and 10 demonstrate how latency and throughput values vary over time.

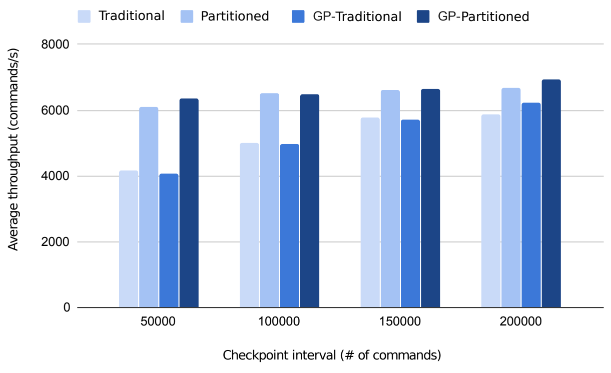


Figure 7. Average throughput using the workload $0r100c$ and 4 partitions.

Figure 8 shows the results obtained for the workload $90r1c$. Even the average throughput is lower compared to the best case $100r0c$ due to the presence of write operations and conflicts between partitions, it is still higher than the traditional model. Once again, the parallelism while saving partition states enables rapid checkpointing, which incurs a higher average throughput with our approach than the traditional one. A slight gain in performance is observed when conflicts involve disjoint groups of partitions. It indicates that some requests are processed while some partitions have their states saved.

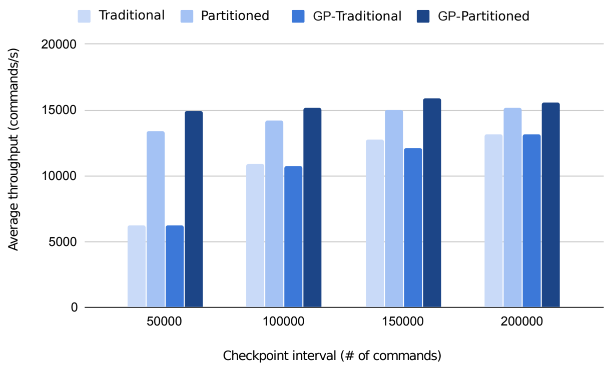


Figure 8. Average throughput using the workload $90r1c$ and 4 partitions.

Figure 9 presents the response latency of requests considering an interval of 50,000 commands between checkpoints and a workload of $90r1c$. The latency is measured per request throughout the execution, with values given in milliseconds. The latency peaks represent the instants when a request was waiting for an ongoing checkpoint to finish. In the traditional technique, saving a checkpoint takes at least 6 seconds to complete (observe the peaks near 9s, 17s, and 26s on the graph). This time for saving the state affects the latency observed by clients. During the checkpointing process, the observed latency with the time-phased partitioned checkpointing is lower than with the traditional technique. This gain is obtained because requests that do not involve saving state partitions can execute in parallel with the execution of the checkpointing procedure.

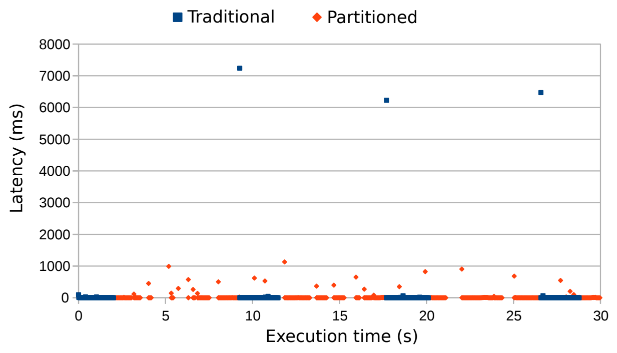


Figure 9. Response latency for a checkpoint interval of every $50,000$ commands, using the workload $90r1c$ and 4 partitions.

Figure 10 presents the throughput behavior, measured in executed commands, over time in seconds. The dashed line indicates the execution of the traditional technique, while the solid line represents the throughput of the technique with partitioned checkpointing. Note that the average throughput for the traditional strategy drops to 0 at the 5-second mark and remains at this level for approximately 6 seconds. The service replica stores its state during this period and stops processing new requests. In the proposed checkpointing strategy, there is a decrease in throughput, but the service replica can still process new requests. The partitioned checkpointing technique produces smaller peaks of response latency because checkpoints of portions of the state are performed at different time instants.

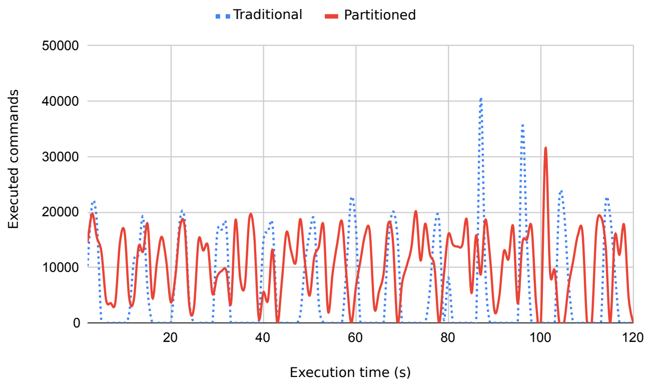


Figure 10. Throughput over time for a checkpoint interval of every $50,000$ commands, using the workload $90r1c$ and 4 partitions.

Table 1 presents a simulation of the percentage of the state stored at each checkpoint. It shows the total number of conflicts involving partition p_1, assuming that p_1 is the target partition to be saved in the checkpoint. Based on the conflict matrix maintained by the scheduler, the total number of partitions that had commands conflicting with p_1, directly or transitively, was calculated. The table shows the total percentage of the state stored at each checkpoint, using intervals between checkpoints of 50,000, 100,000, 150,000, and 200,000 commands. The *# partitions* column presents the partitioning configurations used. The workload $90r1c$ was used for this experiment, meaning that 1% of the generated requests involve pairs of partitions.

With a conflict rate of 1% for 4 and 8 partitions, the entire

Table 1. Simulation of the number of partitions involved before executing a checkpoint for configurations with the workload $90r1c$ and 4 partitions.

# partitions	Percentage of the state stored per interval Δ_{cp}			
	$\Delta_{cp} = 50000$	$\Delta_{cp} = 100000$	$\Delta_{cp} = 150000$	$\Delta_{cp} = 200000$
4	100	100	100	100
8	100	100	100	100
16	87,5	100	100	100
32	31,25	78,12	87,50	96,87
64	7,81	26,56	35,94	40,62
128	0,78	3,90	7,03	17,19
256	0,39	0,78	1,17	1,56

application state needs to be stored regardless of the checkpoint interval used. For 16 partitions and a checkpoint interval of 50,000 requests, 87.5% of the state was stored, while in the other intervals, all partitions were checkpointed in advance. This observation highlights the advantage of using short checkpoint intervals. Furthermore, increasing the number of partitions leads to a greater dispersion in conflicts involving pairs of partitions. Consequently, fewer partitions need to be checkpointed at each interval, allowing the processing of more commands during the checkpoint execution.

As can be seen, even with a low conflict rate, the high throughput to which the system is subjected causes the storage of one partition's state to force the anticipation of storing other partitions' states. Therefore, the throughput of both the traditional and the proposed checkpointing techniques is very similar in configurations with few partitions since the entire state is saved at each checkpoint interval in both cases. By assuming that conflicts involve disjoint groups of partitions, a considerable increase in the average service throughput is observed, especially for small checkpoint intervals.

To evaluate how state partitioning affects the recovery procedure, Figure 11 presents the results. Each vertical bar represents the total time required to perform the recovery procedure. For the partitioned case, the results for each partition are presented separately for better visualization. The total recovery time is divided into two periods. The first period, *Partition Transfer*, represents the time required to request a partition from a correct replica and receive this partition. The *Install CP* indicates the time required to install the received checkpoint.

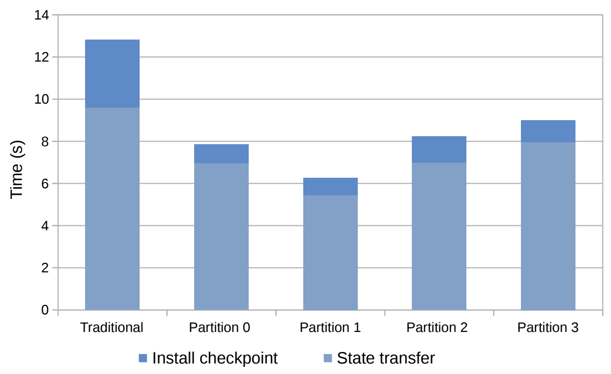


Figure 11. Recovery time with the time-phased partitioned checkpoint.

The partitioned checkpointing technique requires less time for recovery. To complete recovery, the time required for the last partition (*partition 3*) is shorter than in the traditional technique. While the traditional recovery took approximately 13 seconds, the partitioned checkpointing technique took about 9 seconds due to overlapping tasks. Or example, the recovery of partitions 0 and 1 is completed, while the checkpoints of partitions 2 and 3 still need to be received. It is noteworthy that the time required to receive the complete state in the traditional model is similar to the time required to receive a partition. This effect occurs because although state fragments are transmitted, the threads responsible for receiving the partitions compete to access the network controller interface. Performance could be improved by using multiple network interfaces to receive and send state partitions.

A final remark concerns the specifics of the workload patterns, as the application behavior and workload can influence performance. Assuming that the distribution of requests among partitions is evenly distributed, the checkpoint sizes of the partitions would be similar, and consequently, the time to save the state of any partition is comparable. However, in scenarios where a few partitions are heavily updated while others are rarely accessed, the time to save the heavier partitions may represent almost the time to save the entire state of the application, while saving the other partitions occurs almost instantaneously. Although the technique's benefits may be less noticeable in this scenario, the system behavior during checkpointing would still be analogous to the traditional checkpointing approach. Thus, in the worst case, a similar performance can be expected as in traditional checkpointing techniques.

To maximize the benefits of the partitioned checkpointing technique, state partitioning policies should provide a balanced load distribution among partitions while reducing the probability of commands involving multiple partitions. Although it is beyond the scope of this work, efficient partitioning strategies can be combined with the partitioned checkpointing technique [Curino *et al.*, 2010; Kumar *et al.*, 2013; Quamar *et al.*, 2013; Le *et al.*, 2019; Marandi *et al.*, 2014; Li *et al.*, 2018; Coelho and Pedone, 2018; Goulart *et al.*, 2023b]. Regarding the reduction of conflicts involving multiple partitions, several applications fall into the case where conflicts concentrate between disjoint groups of partitions, such as message queue systems and streaming processing systems, where a large portion of messages is directed to subsets of queues, with these subsets being disjoint [Eugster *et al.*, 2003; Sachs *et al.*, 2010; Chen *et al.*, 2011; Viel and Ueda, 2014; Cheng *et al.*, 2017].

6 Related work

This section provides an overview of techniques found in the literature that aim to reduce checkpoint costs and accelerate the recovery of replicas in the system. Finally, we also present checkpointing strategies for PSMR.

6.1 General checkpoint/recovery techniques

In [Bessani *et al.*, 2013], the authors propose a strategy in which each replica of the service performs a checkpoint at a different point in time, ensuring a minimum quorum of available replicas to guarantee progress in choosing new commands to be processed. Towards this end, the proposed protocol assumes a fixed amount, for example, n write operations performed between each checkpoint, plus an interval representing a time lag between the replicas. The authors achieved improved throughput during normal system execution because, at each interval between checkpoints, only one replica interrupts its execution to save the application state while other replicas continue executing normally. Performance was improved by introducing a time lag between checkpoints among the replicas of the service, and the technique proposed in this work is inspired by this approach. However, in the time-phased partitioned checkpointing technique, each service replica performs a checkpoint of a different partition at each checkpoint interval, meaning the time lag is not between independent replicas but between partitions of the same replica.

In Mendizabal *et al.* [2017b], authors propose two strategies to reduce recovery time. The first one, called *speedy recovery*, is based on observing dependencies between the requests in the command log and new requests. If there is no dependency between a new request and the commands in the log, the new request can be executed in parallel with the log processing. Thus, recovering replicas can anticipate the execution of some new commands if they do not affect the state to be reconstructed from the recovery log. Before reprocessing the commands from the log, the replica must restore a state from a recent checkpoint. The second strategy, *on-demand State Recovery*, allows the checkpoint to be installed in parts as needed. With this strategy, a state partition, represented by a checkpoint segment, is installed the first time a request operates on the data space of that segment. Similar to our work, the authors save snapshots of saved partitions in separate files, enabling their transfer and installation in parallel. However, they do not plan saving these partitions with a time lag, which would allow the execution of commands that do not involve the partitions being saved. Although both techniques enable the parallel transfer and installation of checkpoints, saving partitions with a time lag, as proposed in this paper, is less detrimental to the replicas throughput during normal execution.

In the proactive replication strategy [Castro and Liskov, 2000], replicas frequently restore their states based on a quorum of correct replicas to remove potentially compromised states due to malicious processes. Therefore, the efficiency of this process is crucial to enable frequent recoveries. So, to recover a replica, the state transfer mechanism checks the stored state of the replica and determines which portions of the state are updated and uncorrupted. The authors have developed a hierarchical partitioning to reduce the amount of transmitted information. The root partition corresponds to the complete state of the service, and each node corresponds to a partition. Each node is further divided into sub-partitions s of equal size. For each checkpoint, each replica maintains a copy of the partition tree. A checkpoint makes a new copy of the tree and discards it when the checkpoint becomes stable. Nodes of the checkpoint's partition tree store tuples containing metadata with the partition summary, and the copies of the partition tree only store the modified tuples, reducing the space and time required to maintain these checkpoints. This approach improves recovery by reducing the amount of data to be transferred. Partial recovery is possible because state transfer aims to replace only parts of the state that were corrupted or affected by arbitrary or malicious behavior. On the other hand, our approach does not assume Byzantine behavior, and the recovery consists of constructing a complete and consistent state for a new replica in the system.

In UpRight [Clement *et al.*, 2009], authors reduce the throughput overhead during checkpointing by adding helper processes to the system. Primary and helper processes receive the same sequence of requests. In the primary instance, checkpointing is ignored, while in the second instance (the helper), the responses to these requests are disabled, allowing the node to perform checkpointing while the primary is processing new requests. This approach, like the one proposed in this paper, improves the system throughput during normal replica execution. However, in UpRight, the number of processes and communication in the system is doubled, incurring additional costs.

The checkpoint mechanism in the SiloR in-memory database [Zheng *et al.*, 2014] utilizes threads responsible for storing the application's state. Checkpoints execute at predefined time intervals; each thread stores a state partition in different storage units. In this model, a management thread assigns the state partitions to each checkpoint thread responsible for creating the checkpoint. For load balancing, each checkpoint thread traverses a fixed number of database keys. However, in this approach, commands can modify the state while checkpoints are being stored, resulting in inconsistent saved states. This results in fuzzy checkpoints and the need for a more complex recovery protocol. We used only one storage unit to evaluate our checkpointing technique. Therefore, it is impossible to assert whether the technique would present better performance like in Zheng *et al.* [2014]. In future work, we should investigate the impacts on performance by using parallel I/O and multiple storage devices.

6.2 Checkpointing in PSMR

The PSMR model proposed in Marandi *et al.* [2014] parallelizes not only the processing of commands but also the delivery of requests via the consensus protocol. Two checkpointing strategies were proposed for this model, as described in Mendizabal *et al.* [2014]. The first strategy is coordinated, meaning that all service replicas must reach the same common state for a checkpoint to occur. In the second approach, replicas perform checkpoints independently. The coordinated approach introduces a checkpoint command into the consensus protocol to ensure that replicas generate equivalent checkpoints. In the uncoordinated approach, the execution of the checkpoint request occurs at different times since each replica of the service generates it independently, and the consensus protocol does not order it. In our approach, similar to the presented model, the checkpoint request is generated periodically by each service replica. However, there is no divergence between the states created by the service replicas because there is only one sequence of requests delivered by the consensus protocol. Another advantage of the partitioned checkpoints strategy is its ability to be used independently of the PSMR model, unlike the strategy designed for the PSMR model proposed in Marandi *et al.* [2014].

In Kotla and Dahlin [2004], service replicas receive requests through a parallelizing process. The parallelizer is responsible for serializing the execution of dependent commands and ensuring that their execution order respects the order in which the requests were received. Independent commands can be processed in parallel by the set of worker threads. Due to the sequencing of requests created by the consensus protocol, all replicas produce the same states, yielding identical checkpoints. Although the authors do not present a checkpoint approach for this RMEP model, the system allows the definition of commands that conflict with any other command waiting for execution. Therefore, the checkpoint can be executed as a synchronizing command with all others, as discussed in Mendizabal *et al.* [2016]. Thus, during the execution of a checkpoint, it is guaranteed that the worker threads have executed all commands preceding this event. In this strategy, checkpoints can be created at each fixed and deterministic interval, for example, every n executed requests since the last checkpoint. This approach blocks all worker threads while executing the checkpoint, resembling the traditional checkpoint strategy used for comparison with our time-phased partitioned checkpoint.

The checkpointing strategy outlined in Kapritsos *et al.* [2012] also resembles the traditional checkpointing strategy presented in Section 5. Replicas progress through command processing, and the checkpointing process occurs at regular intervals between checkpoints, determined by time intervals or the number of commands executed. However, in this strategy, requests are grouped into batches. Like the other strategies described in this section, this approach requires halting the execution of new commands while saving the application state. In contrast, our approach only suspends the execution of requests involving the state partitions currently being saved. This allows the throughput of each replica to be less impacted by the checkpointing procedure, resulting in lower latency observed by clients. In Mendizabal *et al.* [2016], the authors compare the performance of checkpoint techniques designed for the PSMR models proposed in Marandi *et al.* [2014]; Kotla and Dahlin [2004]; Kapritsos *et al.* [2012].

7 Conclusion

This work presented a new checkpointing and recovery approach that aims to reduce the overhead of checkpointing by partitioning the service state and introducing a lag in the state-saving process for different partitions. Divide the checkpoint execution into smaller state partition checkpoints reduces the throughput overhead of the running replicas. Additionally, processing commands in parallel with partition checkpointing provides lower request latency for clients. Besides the *state-saving* process being faster because only a portion of the state is stored at a time, the time-phased checkpoint enables some partitions continuously to serve client requests.

Another important observation of the proposed strategy is the faster recovery. By partitioning the service state, the state transfer during recovery can involve multiple replicas, distributing the workload among more than one replica. Furthermore, both the transfer and installation of the state can be executed in parallel, making better use of architectures with multiple processing cores.

A performance evaluation compared the impact of our approach on throughput, latency, and recovery time with the traditional checkpointing model. In scenarios with a low incidence of cross-partition requests, the latency of pending requests during checkpointing, as noticed by clients, is reduced proportionally to the number of partitions. In less favorable scenarios, the incidence of cross-partition commands involves multiple service partitions per checkpoint. Thus, triggering a partition checkpoint will force the checkpointing anticipation of the other partitions. In the worst-case scenario, all partitions interact with each other, either directly or transitively. Even so, unlike traditional techniques where a single thread saves the application state, state saving occurs in parallel in our technique. The parallel execution of checkpointing procedures demonstrated less interference in the execution of replicas, and the proposed strategy proved more efficient than the traditional technique, yielding gains in all evaluated scenarios.

Declarations

Funding

This work was financed in part by the Fundação de Amparo à Pesquisa do Estado Do Rio Grande do Sul – FAPERGS PqG 07/21; Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq Universal 18/21; and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES Brasil, Finance Code 001.

Authors’ Contributions

All the authors developed the work as a whole in a collaborative effort.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Data can be made available upon request.

References

Aguilera, M. K., Chen, W., and Toueg, S. (2000). Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13:99–125. DOI: 10.1007/s004460050070.

Alchieri, E., Dotti, F., Mendizabal, O. M., and Pedone, F. (2017). Reconfiguring parallel state machine replication. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 104–113. IEEE. DOI: 10.1109/SRDS.2017.23.

Amazon (2012a). Summary of the december 24, 2012 amazon elb service event in the us-east region. Available at:`https://aws.amazon.com/message/680587/`.

Amazon (2012b). Summary of windows azure service disruption on feb 29th. Available at:`https://azure.microsoft.com/en-us/blog/summary-of-windows-azure\-service-disruption-on-feb-29th-2012/`.

Bessani, A., Sousa, J., and Alchieri, E. E. P. (2014). State machine replication for the masses with bft-smart. In *DSN*, pages 355–362. DOI: 10.1109/DSN.2014.43.

Bessani, A. N., Santos, M., Felix, J., Neves, N. F., and Correia, M. (2013). On the efficiency of durable state machine replication. In *USENIX ATC*, pages 169–180. Available at:`https://www.usenix.org/conference/atc13/technical-sessions/presentation/bessani`.

Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., and Tomlinson, R. S. (1972). Tenex, a paged time sharing system for the pdp-10. *Communications of the ACM*, 15(3):135–143. DOI: 10.1145/361268.36127.

Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA. USENIX Association. Available at:`https://www.usenix.org/legacy/event/osdi06/tech/full_papers/burrows/burrows_html/`.

Castro, M. and Liskov, B. (2000). Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, page 19. USENIX Association. Available at:`https://www.usenix.org/conference/osdi-2000/proactive-recovery-byzantine-fault-tolerant-system`.

Chen, J., Arumaithurai, M., Jiao, L., Fu, X., and Ramakrishnan, K. (2011). Copss: An efficient content oriented publish/subscribe system. In *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 99–110. IEEE. DOI: 10.1109/ANCS.2011.27.

Cheng, D., Chen, Y., Zhou, X., Gmach, D., and Milojicic, D. (2017). Adaptive scheduling of parallel jobs in spark streaming. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE. DOI: 10.1109/INFOCOM.2017.8057206.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009). Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA. ACM. DOI: 10.1145/1629575.1629602.

Coelho, P. and Pedone, F. (2018). Geographic state machine replication. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–230. DOI: 10.1109/SRDS.2018.00034.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Woodford, D., Saito, Y., Taylor, C., Szymaniak, M., and Wang, R. (2012). Spanner: Google's globally-distributed database. In *OSDI*. DOI: 10.1145/249124.

Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57. Available at:`https://dspace.mit.edu/handle/1721.1/73347`.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220. DOI: 10.1145/1323293.1294281.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421. DOI: 10.1145/1041680.1041682.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *Journal of Supercomputing*, 65(3):1302–1326. DOI: 10.1007/s11227-013-0884-0.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408. DOI: 10.1145/568522.568525.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131. DOI: 10.1145/857076.857078.

Frank, A., Baumgartner, M., Salkhordeh, R., and Brinkmann, A. (2021). Improving checkpointing intervals by considering individual job failure probabilities. In *IPDPS*. DOI: 10.1109/IPDPS49936.2021.00038.

Gitlab (2017). Gitlab.com databse incident. Available at:`https://about.gitlab.com/blog/2017/02/01/gitlab-dot-com-database-incident`.

Goulart, H., Álvaro Franco, and Mendizabal, O. (2023a). Checkpointing techniques in distributed systems: A synopsis of diverse strategies over the last decades. In *WTF*, pages 15–28, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/wtf.2023.785.

Goulart, H. S., Trombeta, J., Franco, A., and Mendizabal, O. M. (2023b). Achieving enhanced performance combining checkpointing and dynamic state partitioning. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 149–159. DOI: 10.1109/SBAC-PAD59825.2023.00024.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492. DOI: 10.1145/78969.78972.

Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. (2017). Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155. DOI: 10.1145/3102980.310300.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA. Available at:`https://www.usenix.org/legacy/event/atc10/tech/full_papers/Hunt.pdf`.

Hurfin, M., Mostefaoui, A., and Raynal, M. (1998).

Consensus in asynchronous systems where processes can crash and recover. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*, pages 280–286. IEEE. DOI: 10.1109/RELDIS.1998.740510.

Izraelevitz, J., Wang, G., Hanscom, R., Silvers, K., Lehman, T. S., Chockler, G., and Gotsman, A. (2022). Acuerdo: Fast atomic broadcast over rdma. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11. DOI: 10.1145/3545008.3545041.

Junior, E. G., Alchieri, E., Dotti, F., and Mendizabal, O. (2023). A time-phased partitioned checkpoint approach to reduce state snapshot overhead. In *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, LADC '23, page 100–109, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3615366.3615417.

Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S. V., Schröder-Preikschat, W., and Stengel, K. (2012). Cheapbft: resource-efficient byzantine fault tolerance. In *Eurosys*. DOI: 10.1145/2168836.2168866.

Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., Dahlin, M., *et al.* (2012). All about eve: Execute-verify replication for multi-core servers. In *OSDI*, volume 12, pages 237–250. Available at:`https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kapritsos`.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *DSN*, pages 575–584. IEEE. DOI: 10.1109/DSN.2004.1311928.

Kumar, K. A., Deshpande, A., and Khuller, S. (2013). Data placement and replica selection for improving colocation in distributed environments. *arXiv preprint arXiv:1302.4168*. DOI: 10.48550/arXiv.1302.4168.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565. DOI: 10.1145/3335772.3335934.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169. DOI: 10.1145/279227.279229.

Lamport, L. *et al.* (2001). Paxos made simple. *ACM Sigact News*, 32(4):18–25. Available at:`https://lamport.azurewebsites.net/pubs/paxos-simple.pdf`.

Le, L. H., Fynn, E., Eslahi-Kelorazi, M., Soulé, R., and Pedone, F. (2019). Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1453–1465. IEEE. DOI: 10.1109/ICDCS.2019.00145.

Li, B., Xu, W., and Kapitza, R. (2018). Dynamic state partitioning in parallelized byzantine fault tolerance. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 158–163. IEEE. DOI: 10.1109/DSN-W.2018.00056.

Marandi, P. J., Bezerra, C. E., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 368–377. IEEE. DOI: 10.1109/ICDCS.2014.45.

Marandi, P. J. and Pedone, F. (2014). Optimistic parallel state-machine replication. In *SRDS*, pages 57–66. IEEE. DOI: 10.1109/SRDS.2014.25.

Masti, S. (2021). How we built a general purpose key value store for facebook with zippydb. Available at:`https://engineering.fb.com/2021/08/06/core-infra/zippydb/`.

Mendizabal, O. M., De Moura, R. S., Dotti, F. L., and Pedone, F. (2017a). Efficient and deterministic scheduling for parallel state machine replication. In *IPDPS*, pages 748–757. IEEE. DOI: 10.1109/IPDPS.2017.29.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2016). Analysis of checkpointing overhead in parallel state machine replication. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 534–537, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2851613.285187.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2017b). High performance recovery for parallel state machine replication. In *ICDCS*, pages 34–44. IEEE. DOI: 10.1109/ICDCS.2017.193.

Mendizabal, O. M., Marandi, P. J., Dotti, F. L., and Pedone, F. (2014). Checkpointing in parallel state-machine replication. In *International Conference on Principles of Distributed Systems*, pages 123–138. Springer. DOI: 10.1007/978-3-319-14472-6_9.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *SOSP*. DOI: 10.1145/2517349.2517350.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319. Available at:`https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

Quamar, A., Kumar, K. A., and Deshpande, A. (2013). SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441. DOI: 10.1145/2452376.2452427.

Sachs, K., Appel, S., Kounev, S., and Buchmann, A. (2010). Benchmarking publish/subscribe-based messaging systems. In *International Conference on Database Systems for Advanced Applications*, pages 203–214. Springer. DOI: 10.1007/978-3-642-14589-$6_2$1.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319. DOI: 10.1145/98163.98167.

Schuller, P. (2014). Manhattan, our real-time, multi-tenant distributed database for twitter scale. Available at:`https://blog.x.com/engineering/en/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale`.

Tiwari, D., Gupta, S., and Vazhkudai, S. S. (2014). Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *DSN*, pages 25–36. IEEE. DOI: 10.1109/DSN.2014.101.

Viel, E. and Ueda, H. (2014). Data stream partitioning re-optimization based on runtime dependency mining. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 199–206. IEEE. DOI: 10.1109/ICDEW.2014.6818327.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270. DOI: 10.1145/844128.844152.

Zheng, W., Tu, S., Kohler, E., and Liskov, B. (2014). Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477. Available at:`https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting`.