Checkpointing in Parallel State-Machine Replication

Odorico M. Mendizabal[1,2], Parisa Jalili Marandi[3], Fernando Luís Dotti[1], and Fernando Pedone[3]

[1] Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS
Porto Alegre, Brazil
odoricomendizabal@furg.br, fernando.dotti@pucrs.br
[2] Universidade Federal do Rio Grande – FURG
Rio Grande, Brazil
[3] University of Lugano – USI
Lugano, Switzerland
{fernando.pedone,parisa.jalili.marandi}@usi.ch

Abstract. State-machine replication is a popular approach to building fault-tolerant systems, which relies on the sequential execution of commands to guarantee strong consistency. Sequential execution, however, threatens performance. Recently, several proposals have suggested parallelizing the execution model of the replicas to enhance state-machine replication's performance. Despite their success in accomplishing high performance, the implications of these models on checkpointing and recovery is mostly left unaddressed. In this paper, we focus on the checkpointing problem in the context of Parallel State-Machine Replication. We propose two novel algorithms and assess them through simulation and a real implementation.

Keywords: State-machine replication, checkpointing, fault tolerance.

1 Introduction

State-machine replication (SMR) is a well-established approach to implementing fault-tolerant services. Replicas in state-machine replication start in the same initial state and execute an identical and ordered set of client commands sequentially and deterministically [10,16]. Therefore, all the replicas traverse the same sequence of internal states and produce the same outputs. Consensus is often used to ensure that commands are totally ordered across replicas [11].

Sequential execution of commands can be a performance bottleneck and a waste of resources, in particular when replicas have access to multicore processors. To overcome this limitation, techniques that allow concurrent execution of commands in state-machine replication have been proposed [8,9,12,13]. These techniques are based on the observation that some commands are *independent*, that is, they access disjoint portions of the replica's state or do not modify shared parts of the state. Therefore, independent commands can be executed

M.K. Aguilera et al. (Eds.): OPODIS 2014, LNCS 8878, pp. 123–138, 2014.

concurrently without compromising the service's consistency. Dependent commands, however, those that modify shared parts of the state, must be executed sequentially, in the same order across replicas.

This paper focuses on checkpointing in Parallel State-Machine Replication (P-SMR) [12], a scalable multithreaded replication model, whose scalability stems from the absence of a centralized component in the execution path of independent commands (e.g., no local scheduler [9]). In P-SMR, replicas alternate between the execution of concurrent commands (i.e., those mutually independent) and the execution of sequential commands. Recovering a failed replica in classic SMR requires retrieving the commands the replica executed but "forgot" due to the failure and the commands the replica missed while it was down. To speed up recovery, replicas can periodically checkpoint their state against stable storage so that upon recovering, a replica can start with a state not too far behind the other replicas, after reading its local checkpoint from stable storage or retrieving a checkpoint from a remote operational replica. Performing checkpoints efficiently in P-SMR is more challenging than in classic SMR because the checkpoint operation must account for the execution of concurrent commands.

We propose two checkpoint techniques for P-SMR: coordinated and uncoordinated. The coordinated algorithm executes checkpoints when replicas are in sequential mode. The uncoordinated algorithm is more complex but can checkpoint a replica's state during both sequential and concurrent execution modes. The fundamental differences between the two approaches are three-fold: (a) With the coordinated mechanism, any two replicas save the same sequence of checkpoints throughout the execution; with uncoordinated checkpoints, replicas may save different states. Saving the same sequence of checkpoints has performance implications during recovery, as we explain in the paper. (b) Since an uncoordinated checkpoint can be started while a replica is executing commands concurrently, faster threads will be idle for shorter periods when waiting for slow threads in the uncoordinated technique than in the coordinated approach. (c) Coordinated checkpoints incur system-wide synchronization, while uncoordinated checkpoints are local to a replica. We discuss in the paper the implications of each technique using simulation models and an in-memory database service.

This paper makes the following contributions: (a) it discusses recovery of failed replicas in the context of parallel state-machine replication, a topic that has received little attention until now; (b) it proposes two checkpoint techniques for P-SMR, coordinated and uncoordinated, and compares their pros and cons; and (c) it assesses the performance of the two techniques using simulation models and an in-memory database service.

The remainder of this paper is organized as follows. In Section 2, we present the system model and assumptions. In Section 3, we recall parallel state-machine replication and provide a consensus-based algorithm that implements P-SMR. In Section 4, we discuss recovery in classic state-machine replication and introduce two checkpoint algorithms for P-SMR. We assess the performance of our proposed algorithms in Section 5 and relate them to the state of the art in Section 6. We conclude the paper in Section 7.

2 System Model and Assumptions

We assume a distributed system composed of interconnected processes. There is an unbounded set $C = \{c_1, c_2, \ldots\}$ of client processes and a bounded set $R = \{r_1, r_2, \ldots\}$ of replica processes. We do not make any assumptions about the relative speed of processes or message delays, i.e., the system is asynchronous.

We assume the *crash-recovery* model and exclude malicious or arbitrary behavior. A process can be either *up* or *down*, and it switches between these two modes when it fails (i.e., from up to down) and when it recovers (i.e., from down to up). Replicas are equipped with volatile memory and stable storage. Upon a crash, a replica loses the content of its volatile memory, but the content of its stable storage survives crashes.

Processes communicate by message passing, using either one-to-one or one-to-many communication. One-to-one communication is performed through primitives *send(m)* and *receive(m)*, where m is a message. If m's sender transmits m "enough times" and m's destination does not fail, then m is eventually received. One-to-many communication relies on the consensus abstraction, defined next.

The consensus problem can be described in terms of processes that propose values and processes that must agree upon a decided value. Consensus is defined by the primitives *propose(v)* and *decide(v)*, where v is an arbitrary value. A consensus protocol ensures the following safety requirements: (i) any value decided must have been proposed; (ii) a process can decide at most one value; and (iii) two different processes cannot decide different values. Solving consensus requires additional assumptions about the system model [6]. In this paper, we simply assume consensus can be solved without explicitly extending the model with these assumptions (e.g., [3,4]).

State-machine replication can be implemented with a sequence of consensus *rounds*, where the i-th consensus round decides on the i-th command to be executed by the replicas. We identify the decision of the i-th consensus round as *decide(i,v)*. In order to simplify our algorithms, we modify the propose primitive above such that a value proposed by a non-faulty process is eventually decided in some consensus round.

Our consistency criterion is *linearizability*: a system is linearizable if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time ordering of commands across all clients [1].

3 Parallel State-Machine Replication

In contrast to classic state-machine replication (SMR), where the execution of commands is sequential, in parallel state-machine replication (P-SMR) independent commands can be executed concurrently. To understand the interdependencies between commands, assume commands C_i and C_j, where W_i and W_j indicate the commands' writeset and R_i and R_j indicate their readset. According to [9], C_i and C_j are *dependent* if any of the following conditions hold:

(i) $W_i \cap W_j \neq \emptyset$, (ii) $W_i \cap R_j \neq \emptyset$, or (iii) $R_i \cap W_j \neq \emptyset$. In other words, if the writeset of a command intersects with the readset or the writeset of another command, the two commands are dependent. Two commands are independent if they are not dependent.

P-SMR parallelizes the *agreement* and the *execution* of commands. Instead of using a single sequence of consensus rounds to order commands as in SMR, P-SMR uses multiple sequences of consensus. More precisely, if there are $n + 1$ threads at each replica, $t_0, ..., t_n$, P-SMR requires $n + 1$ consensus sequences, $\gamma_0, ..., \gamma_n$, where thread t_0 (at each replica) participates in consensus sequence γ_0 only, and thread t_i, $0 < i \leq n$, participates in consensus sequences γ_0 and γ_i. To ensure that t_i handles commands in the same order across replicas, despite participating in two consensus sequences, t_i orders messages from its two consensus sequences using a deterministic merge procedure (e.g., handling decisions for the sequences in round-robin fashion). To ensure progress, every consensus sequence must have a never-ending stream of consensus rounds, which can be achieved by having one or more processes proposing *nil* values if no value is proposed in a consensus sequence after some time [14]. Obviously, replicas discard *nil* values decided in a consensus round.

P-SMR ensures two important invariants. First, commands decided in consensus sequence γ_0 are serialized with any other commands at a replica and executed by thread t_0 in the same order across replicas (*sequential execution mode*). Second, commands decided in the same round in consensus sequences $\gamma_1, ..., \gamma_n$ are executed by threads $t_1, ..., t_n$ concurrently at a replica (*concurrent execution mode*).

Clients propose a command by choosing the consensus sequence that guarantees ordered execution of dependent commands while maximizing parallelism of independent commands. The mapping of commands onto consensus sequences is application dependent. In the following, we illustrate two such mappings.

- (Concurrent reads and sequential writes.) Commands that read the replica's state are proposed in any arbitrary consensus sequence γ_i, $0 < i \leq n$; commands that modify the replica's state are proposed in sequence γ_0.
- (Concurrent reads and writes.) Divide the service's state into disjoint partitions $P_1, ..., P_n$ so that commands that access partition P_i only are proposed in γ_i and commands that access multiple partitions are proposed in γ_0.

Clients must be aware of the mapping of commands onto consensus sequences and must be able to identify commands that read the service's state only or modify the state, in the first case above, or to identify commands that access a single partition (and which partition) or multiple partitions, in the second case.

Algorithm 1 presents P-SMR in detail. For each thread t_i, $round[i]$ (line 3) indicates the number of the next consensus round to be handled (or being handled) by t_i, for all consensus sequences involving t_i. Threads use semaphores $S[0..n]$ (line 4) to alternate between sequential and concurrent modes and, as shown in the next section, to create a checkpoint. Variable $next[i]$ (line 5) determines whether t_i is in *sequential* or *concurrent* mode.

Thread t_0 tracks decisions in consensus sequence γ_0 only (line 8). The "decided $[\gamma_0](r, \langle - \rangle)$ **and** $r = round[0]$" condition holds when there is a decision in consensus sequence γ_0 that matches $round[0]$. If the value decided in $round[0]$ is a command (line 9), t_0 waits for every other thread t_i (line 10) and then handles the request (line 11). After the command is executed, t_0 signals the other threads to continue their execution (line 12). Whatever value is decided in the round, $round[0]$ is incremented (line 13). Note that a *nil* decision in consensus sequence γ_0 does not cause threads to synchronize.

Algorithm 1: P-SMR

```
 1: Initialization:
 2:   for i : 0..n do                                   {for each thread t_i:}
 3:     round[i] ← 1                                    {all threads start in the same round}
 4:     S[i] ← 0                                        {semaphore used to implement barriers}
 5:     next[i] ← SQ                                    {start in sequential mode}
 6:   start threads t_0, ..., t_n

 7: Thread t_0 at a replica executes as follows:
 8:   upon decided [γ_0](r, ⟨cid, cmd⟩) and r = round[0]
 9:     if cmd ≠ nil then                               {if cmd is a real command...}
10:       for i : 1..n do wait(S[0])                   {barrier: wait for threads t_1, .., t_n}
11:       execute cmd and reply to cid                 {execute command and reply to client}
12:       for i : 1..n do signal(S[i])                 {let threads t_1, .., t_n continue}
13:     round[0] ← round[0] + 1                        {pass to the next round}

14: Thread t_i in t_1, ..., t_n at a replica executes as follows:
15:   upon decided [γ_0](r, ⟨cid, cmd⟩) and r = round[i] and next[i] = SQ
16:     if cmd ≠ nil then                              {if decided on a real command...}
17:       signal(S[0])                                 {barrier: signal semaphore S[0] (see line 10)}
18:       wait(S[i])                                   {...and wait to continue (see line 12)}
19:     next[i] ← CC                                   {set execution mode as concurrent}

20:   upon decided [γ_i](r, ⟨cid, cmd⟩) and r = round[i] and next[i] = CC
21:     if cmd ≠ nil then                              {if decided on a command...}
22:       execute cmd and reply to cid                 {execute command and reply to client}
23:     next[i] ← SQ                                   {set execution mode as sequential}
24:     round[i] ← round[i] + 1                        {pass to the next round}
```

Each thread t_i, $0 < i \leq n$, alternates between executing in sequential and concurrent modes (lines 15 and 20). If t_i decides a value in consensus sequence γ_0 for its current round and the current execution mode is sequential (line 15), t_i checks whether the command is not *nil* (line 16) and in such a case t_i signals thread t_0 (line 17) and waits for t_0 to continue (line 18). Thread t_i then sets $next[i]$ to CC (line 19), meaning that it is in concurrent mode now. When t_i decides a value in consensus sequence γ_i for round $round[i]$ and $next[i] =$ CC (line 20), t_i executes the command if it is not *nil* (lines 21–22), sets the execution mode as sequential (line 23), and passes to the next round (line 24).

4 Checkpointing in P-SMR

Recovery in classic SMR is conceptually simple: Replicas log commands before executing them (e.g., as part of consensus) and periodically (e.g., after k commands) save the application state or the changes made since the last recorded checkpoint in stable storage. When a replica recovers from a failure, it retrieves a checkpoint from its local storage or from a remote replica and resumes operation after installing this checkpoint. The recovering replica also needs to recover the value decided in consensus rounds not included in the installed checkpoint.

Checkpoints speed up recovery and save storage space. Checkpoints shorten recovery time since a recovering replica does not need to start with an empty state and (re-)execute every decided command to catch up with the other replicas. Checkpoints save storage space since commands decided in "old" consensus rounds can be garbage collected. A sufficient condition to remove data related to the i-th consensus round is that all replicas have recorded a checkpoint containing the effects of the command decided in the i-th round.

We propose next two novel checkpointing algorithms for P-SMR. In the first algorithm, coordinated checkpointing, replicas must converge to a common state before taking a checkpoint; in the second algorithm, uncoordinated checkpointing, replicas take checkpoints independently and may not be in an identical state when the checkpoint takes place. We conclude the section with a comparison between the two algorithms.

4.1 Coordinated Checkpointing

The idea behind our coordinated checkpointing algorithm is to force replicas to undergo the same sequence of checkpointed states. To this end, we define a checkpoint command *CHK* that depends on all other commands. Therefore, *CHK* is executed in sequential mode in P-SMR and ordered by consensus sequence γ_0. Since replicas implement a deterministic strategy to merge consensus sequences, command *CHK* is guaranteed to be executed after each replica reaches a certain common state.

Algorithm 2 presents the coordinated checkpoint algorithm in detail. When a replica recovers from a failure (line 1), it first retrieves the latest checkpoint stored at the replica or requests one from a remote replica (line 2). Tuple $\langle last\_rnd[0] \rangle$ identifies the retrieved checkpoint. (Every replica stores an initialization checkpoint, empty and identified by $\langle 1 \rangle$.) The replica then initializes variables S, *round* and *next* (lines 3–10) and starts all threads (line 11).

Thread t_0's only difference with respect to Algorithm 1 is that it must check whether a decided command is a checkpoint request (line 16), in which case t_0 stores the replica's state on stable storage and identifies the checkpoint as $\langle round[0] \rangle$ (line 17). Threads $t_1, ..., t_n$ execute the same pseudocode in Algorithms 1 and 2.

Algorithm 2: Coordinated checkpoint

```
 1: upon starting or recovering from a failure
 2:   retrieve latest/remote checkpoint, which has id ⟨last_rnd[0]⟩
 3:   for i : 0..n do                                   {for each thread t_i...}
 4:     S[i] ← 0                                {semaphore used to implement barriers}
 5:     if i = 0 then                                          {thread t_0...}
 6:       round[i] ← last_rnd[0] + 1                  {goes to the next round in...}
 7:       next[i] ← SQ                                          {...sequential mode}
 8:     else                                                 {threads t_1, ..., t_n...}
 9:       round[i] ← last_rnd[0]                        {stay in this round in...}
10:       next[i] ← CC                                          {...concurrent mode}
11:   start threads t_0, ..., t_n

12: Thread t_0 at a replica executes as follows:
13:   upon decided [γ_0](r, ⟨cid, cmd⟩) and r = round[0]
14:     if cmd ≠ nil then                  {if cmd is a command/checkpoint request...}
15:       for i : 1..n do wait(S[0])          {barrier: wait n times on semaphore}
16:       if cmd = CHK then                    {if cmd is a checkpoint request...}
17:         store checkpoint with id ⟨round[0]⟩                   {take checkpoint}
18:       else                                                            {else...}
19:         execute cmd and reply to cid          {execute command and reply to client}
20:       for i : 1..n do signal(S[i])                {let each thread t_i continue}
21:     round[0] ← round[0] + 1                        {one more handled decision}

22:   each Δ time units do                  {ideally done by a single replica only:}
23:     propose[γ_0](⟨t_0, CHK⟩)                  {request a system-wide checkpoint}

24: Thread t_i in t_1, ..., t_n at a replica executes as follows:
25:   upon decided [γ_0](r, ⟨cid, cmd⟩) and r = round[i] and next[i] = SQ
26:     if cmd ≠ nil then                  {if cmd is a command/checkpoint request...}
27:       signal(S[0])                                  {implement barrier (see line 15)}
28:       wait(S[i])                          {...and wait to continue (see line 20)}
29:     next[i] ← CC                                {set execution mode as concurrent}

30:   upon decided [γ_i](r, ⟨cid, cmd⟩) and r = round[i] and next[i] = CC
31:     if cmd ≠ nil then                          {if cmd is an actual command...}
32:       execute cmd and reply to cid          {execute command and reply to client}
33:     next[i] ← SQ                                {set execution mode as sequential}
34:     round[i] ← round[i] + 1                        {one more handled decision}
```

4.2 Uncoordinated Checkpointing

We now present an alternative algorithm that does not coordinate checkpoints across replicas: each replica decides locally when checkpoints will happen. Unlike the coordinated checkpointing algorithm, where all replicas record identical checkpoints, with the uncoordinated algorithm the checkpoints vary across the replicas.

The main difficulty with uncoordinated checkpoints is that a checkpoint request may be received any time during a thread's execution. Thus, one thread

may receive a checkpoint request when in sequential execution mode while another thread receives the same request when in concurrent execution mode. Essentially, this happens because we do not order checkpoint requests with consensus decisions, as in the coordinated version of the algorithm.

In brief, our algorithm works as follows. First, thread t_0 requests a checkpoint by sending a local message to the other threads. Second, the handling of a checkpoint request at a replica does not change the sequence of commands executed by threads t_i, $0 < i \leq n$, which still alternate between sequential and concurrent execution modes in each round. To guarantee this property, when t_0 requests a checkpoint it tracks the signal it receives from t_i: If t_i signals t_0 upon receiving the checkpoint request, then after the checkpoint, t_0 releases t_i so that t_i can proceed with the next command. If t_i signals t_0 because it started the sequential execution mode, after the checkpoint t_0 keeps t_i waiting until t_0 also goes through the sequential execution of commands. In this case, when t_i later receives the checkpoint request, it simply discards it.

Algorithm 3 presents the uncoordinated checkpointing algorithm in detail. When a replica recovers from a failure, it retrieves the last saved checkpoint from its local storage or from a remote replica (line 2). This checkpoint identifies the round and the execution mode the thread must be in, after the checkpoint is installed (lines 4–5). (A replica is initialized with an empty checkpoint, identified as $\langle 2, \text{SQ}[, 1, \text{CC}]_{\times n}\rangle$.) Variable $last\_sync[i]$ contains the last round when t_i started in sequential mode and signaled t_0 (line 8); $waiting[i]$ tells whether upon executing a command t_0 must wait for t_i (line 9).

The execution of a sequential command by t_0 is similar in both the coordinated and uncoordinated algorithms, with the exception that t_0 only waits for t_i if it is not already in waiting mode (line 14); this happens if t_i signals t_0 because it started sequential execution mode but t_0 started a checkpoint. After the execution of the sequential command, all threads are released (lines 17–18). To execute a checkpoint, t_0 sends a message to all threads and waits for them (lines 21–23). If t_i signaled t_0 because it entered sequential mode in t_0's current round or some round ahead (line 26), which happens if the value decided in t_0's current round is nil, t_0 keeps track that t_i is waiting (line 27); otherwise t_0 signals t_i to continue (line 29).

The execution of commands for threads $t_1, ..., t_n$ is similar in both checkpoint algorithms, with the exception that before signaling the start of sequential execution mode, t_i sets $last\_sync[i]$ with its round number (line 33). Upon receiving a checkpoint request $\langle r, CHK \rangle$ that satisfies condition $last\_sync[i] < r \leq round[i]$ (line 42), t_i signals t_0 and waits for t_0's signal (lines 43–44). If $last\_sync[i] \geq r$, then it means that t_i has already signaled t_0 when entering sequential execution mode; thus, it does not do it again. If $r > round[i]$, then the checkpoint request is for a round ahead of t_i's current round. This request will be considered when t_i reaches round r.

Algorithm 3: Uncoordinated checkpoint

```
 1: upon recovering from a failure
 2:    retrieve checkpoint, which has id ⟨rnd[0], nxt[0], ..., rnd[n], nxt[n]⟩
 3:    for i : 0..n do                                   {for each thread t_i, 0 ≤ i ≤ n:}
 4:       round[i] ← rnd[i]                                          {t_i's round and...}
 5:       next[i] ← nxt[i]                    {... execution mode when checkpoint taken}
 6:       S[i] ← 0                              {semaphore used to implement barriers}
 7:    for i : 1..n do                                   {for each thread t_i, 1 ≤ i ≤ n:}
 8:       last_sync[i] ← 0                      {last round t_i entered sequential mode}
 9:       waiting[i] ← false                                 {initially t_i isn't waiting}
10:    start threads t_0, ..., t_n

11: Thread t_0 at a replica executes as follows:
12:    upon decided [γ_0](r, ⟨cid, cmd⟩) and r = round[0]
13:       if cmd ≠ nil then                                    {if decided on a command...}
14:          for i : 1..n do if ¬waiting[i] then wait(S[0])    {wait for each active t_i}
15:          execute cmd and reply cid                {execute command and reply to client}
16:          for i : 1..n do
17:             waiting[i] ← false                  {after sequential mode no thread waits}
18:             signal(S[i])                                                  {ditto!}
19:       round[0] ← round[0] + 1                          {t_0 passes to the next round}

20:    each Δ time units do                        {t_0 periodically triggers a local checkpoint}
21:       for i : 1..n do
22:          send ⟨round[0], CHK⟩ to t_i                 {send checkpoint request to t_i}
23:          if ¬waiting[i] then wait(S[0])              {wait for each active thread t_i}
24:       store checkpoint with id ⟨round[0], next[0], round[1], ...⟩      {take checkpoint}
25:       for i : 1..n do                                                  {for each t_i}
26:          if last_sync[i] ≥ round[0] then          {if t_i entered sequential mode...}
27:             waiting[i] ← true                    {keep t_i waiting until t_0 catches up}
28:          else                                                          {else...}
29:             signal(S[i])                                              {let t_i proceed}

30: Thread t_i in t_1, ..., t_n at a server executes as follows:
31:    upon decided [γ_0](r, ⟨cid, cmd⟩) and r = round[i] and next[i] = SQ
32:       if cmd ≠ nil then                                {if decided on a real command...}
33:          last_sync[i] ← round[i]                {take note that entered sequential mode}
34:          signal(S[0])                                                {implement barrier}
35:          wait(S[i])                                          {...and wait to continue}
36:       next[i] ← CC                                {set execution mode as concurrent}

37:    upon decided [γ_i](r, ⟨cid, cmd⟩) and r = round[i] and next[i] = CC
38:       if cmd ≠ nil then                                {if cmd is an actual command...}
39:          execute cmd and reply to cid             {execute command and reply to client}
40:       next[i] ← SQ                                {set execution mode as sequential}
41:       round[i] ← round[i] + 1                                  {pass to the next round}

42:    upon receive ⟨r, CHK⟩ from t_0 and last_sync[i] < r ≤ round[i]
43:       signal(S[0])                          {checkpoints are done in mutual exclusion}
44:       wait(S[i])                                                        {ditto!}
```

4.3 Coordinated versus Uncoordinated Checkpointing

With coordinated checkpoints, a checkpoint only happens after each thread receives a *CHK* request and finishes executing all the commands decided before the request. With uncoordinated checkpoints, a checkpoint is triggered within a replica and is not ordered with commands. These mechanisms have important differences, as we discuss next.

First, with coordinated checkpoints every replica saves the same state upon taking the k-th checkpoint. Saving the same state across replicas is important for *collaborative state transfer* [2], a technique that improves performance by involving multiple operational replicas in the transferring of a saved checkpoint to the recovering replica, each replica sending part of the checkpointed state. Collaborative state transfer is not possible with uncoordinated checkpoints.

Second, coordinated checkpoints take place when replicas are in sequential execution mode; hence, no checkpoint contains a subset of commands executed concurrently. Uncoordinated checkpoints, however, can save states of a replica during concurrent execution mode. The implication on performance is that threads that execute commands more quickly when in concurrent mode do not have to wait for slower threads to catch up so that a checkpoint can be taken.

Third, the interval between the time when a checkpoint is triggered at a replica and the time when it takes place in the replica in the uncoordinated technique is lower than in the coordinated technique. In addition to requiring a consensus execution, which introduces some latency, a checkpoint request in the coordinated technique can only be handled after previously decided commands are executed at the replicas.

5 Performance Analysis

In this section, we assess the impact of the proposed approaches on the system performance by means of a simulation model and a prototype. Our simulations focus mostly on the cost of synchronization due to checkpointing. Aspects inherent to recovery (e.g., state transferring) are highly dependent on the application and sensitive to the data structures used by the service, the workload, and the size of checkpoints. We consider such aspects with our prototype, which implements an in-memory database with operations to read and write database entries. In our experiments, we generate sustained workloads with independent commands only. With this strategy we maximize the use of threads to execute commands, removing the possibility of thread idleness due to the synchronization needed by dependent commands.

5.1 Simulations

We implemented a discrete-event simulation model in C++ and configured each experiment to run until the 98% confidence interval of the command response time was a small fraction of the average value. We evaluated replicas without

checkpointing enabled and with the two proposed checkpoint algorithms, and considered different classes of workload in terms of requests execution time: (i) fixed-duration commands (i.e., all commands take the same time to execute), (ii) uniformly distributed command duration, and (iii) exponentially distributed command duration. In the last case, a majority of commands have low execution times, while a small number of commands take long to execute.

We start by evaluating the scalability of both techniques. Figure 1 shows the maximum throughput achieved by a replica according to the number of threads, where each thread is associated with a processing unit (i.e., core). In these experiments, we used workloads (i), (ii), and (iii), described above, with average command execution time of 0.5 units. Checkpoints are taken every 200 time units, and the checkpoint duration is 0. By not considering the time taken to create a checkpoint, the results reveal the overhead caused exclusively by checkpoint synchronization. The throughput of P-SMR without checkpoints scales proportionally to the number of threads. The overhead of uncoordinated checkpointing is lower than the overhead of the coordinated technique and the difference between the two increases with the number of threads.

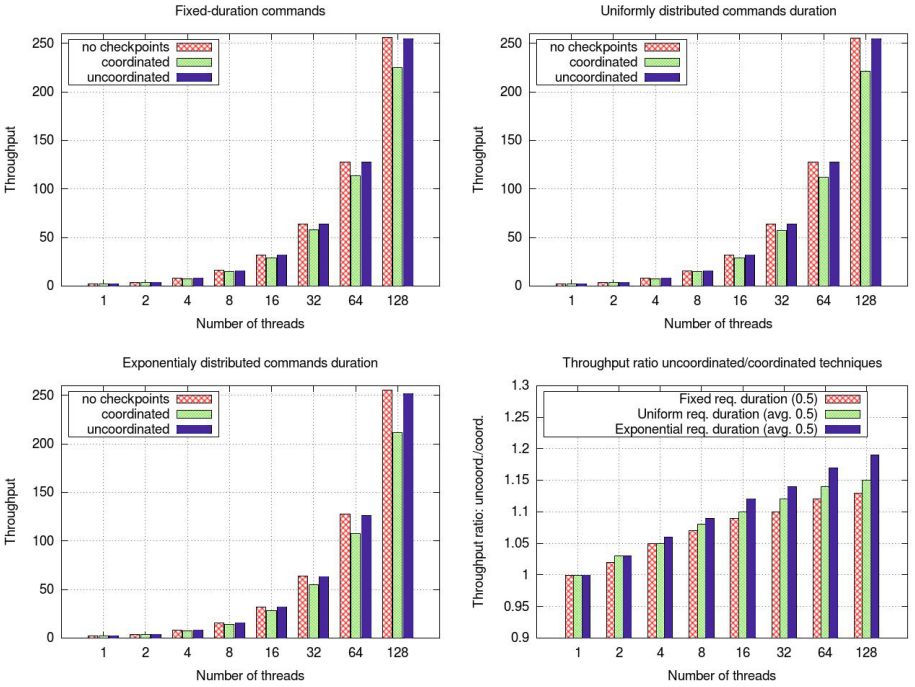

Fig. 1. Throughput of a replica with the number of threads for different commands execution duration workloads and the ratio of the two techniques with the number of threads

The bottom right graph of Figure 1 depicts the throughput ratio between the uncoordinated and the coordinated techniques under different workloads, as we increase the number of threads. Two facts stand out: First, uncoordinated checkpointing outperforms coordinated checkpointing in all scenarios and the difference increases with the number of threads. Second, the difference between the two techniques is more important when there is more variation in the command execution time. This happens because "faster threads" (i.e., those executing shorter commands) wait longer for "slow threads" during a checkpoint in the coordinated technique than in the uncoordinated approach.

Next, we evaluate the impact caused by the checkpoint frequency. Figure 2 shows the throughput and latency of replicas with 16 threads. In this experiment, the command duration follows the exponential distribution. The checkpointing interval varies from 12 to 1600 time units and the checkpointing duration is 0. The workload generated for this experiment reaches a throughput equivalent to 75% of the maximum. Although the uncoordinated checkpointing algorithm outperforms the coordinated algorithm in all configurations, the difference between the two decreases as checkpoints become more infrequent.

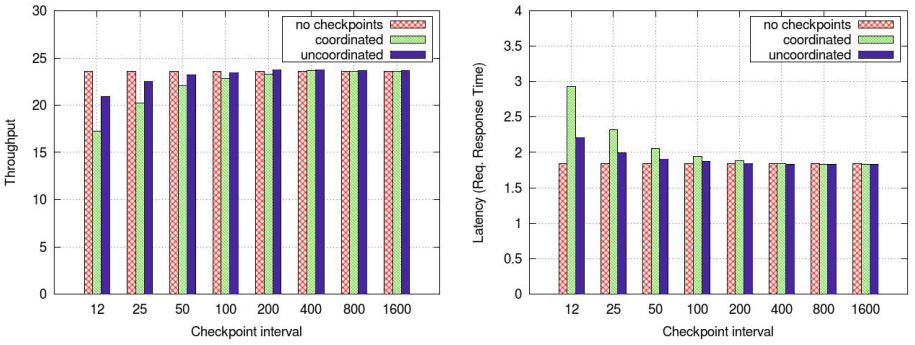

Fig. 2. Throughput and latency of a replica executing commands with an exponentially distributed execution time (average of 0.5 time units)

Figure 3 depicts the throughput and latency results for scenarios in which checkpoints take 5 time units to execute. The overhead introduced by a checkpoint has the effect of decreasing the throughput and increasing the average response time of commands. However, the checkpoint overhead did not change the trend seen in the previous experiments: uncoordinated checkpointing consistently performs better than coordinated checkpointing, and the difference between the two reduces as checkpoints are taken less often.

5.2 Implementation

We implemented consensus using Multi-Ring Paxos [14], where each consensus sequence is mapped to one Paxos instance. To achieve high performance, each

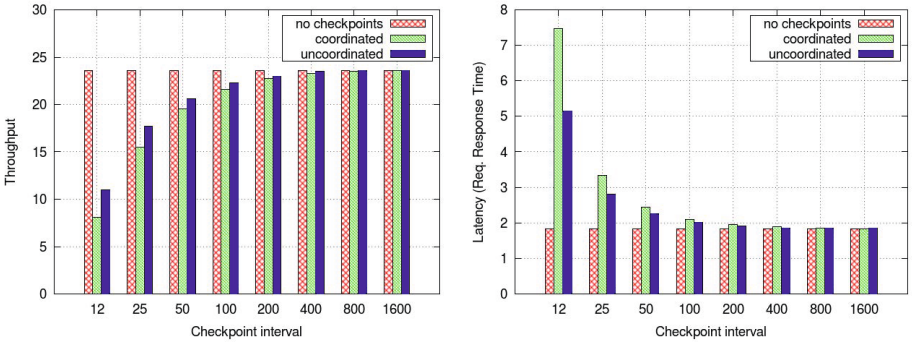

Fig. 3. Throughput and latency of a replica executing commands with an exponentially distributed execution time (average of 0.5 time units) and checkpoint duration of 5 time units

thread t_i decides several times on consensus sequence γ_i before deciding on sequence γ_0. Moreover, multiple commands proposed to a consensus sequence are batched by the group's coordinator (i.e., the coordinator in the corresponding Paxos instance) and order is established on batches of commands. Each batch has a maximum size of 8 Kbytes. The system was configured so that each Paxos instance uses 3 acceptors and can tolerate the failure of one acceptor.

The service is a simple in-memory database, implemented as a hash table, with operations to create, read, write, and remove entries. Each entry has an 8-byte key and an 8-byte value. A checkpoint duplicates the hash table in memory (using copy-on-write) and writes the duplicated structure to disk, either synchronously or asynchronously. We ran our experiment on a cluster with Dell PowerEdge R815 nodes equipped with four octa-core AMD Opteron processors and 128 GB of main memory (replicas), and Dell SC1435 nodes equipped with two dual-core AMD Opteron processors and 4 GB of main memory (Paxos's acceptors and clients). Each node is equipped with one 1Gb network interface. The nodes ran CentOS Linux 6.2 64-bit with kernel 2.6.32.

Figure 4 shows the throughput and the corresponding 90% percentile of the response time of both techniques. Checkpoints are taken once every 5 seconds and each one takes approximately 3.2 seconds to complete. When a checkpoint happens the database has approximately 10 million entries. The results show that uncoordinated checkpointing has a slight advantage over coordinated checkpoint in some of the configurations. Given the high rate of commands executed per second and the frequency of checkpoints, these results corroborate those presented in the previous section.

6 Related Work

In this section, we briefly review other models for parallel state-machine replication and compare their checkpointing mechanisms. A comprehensive survey of

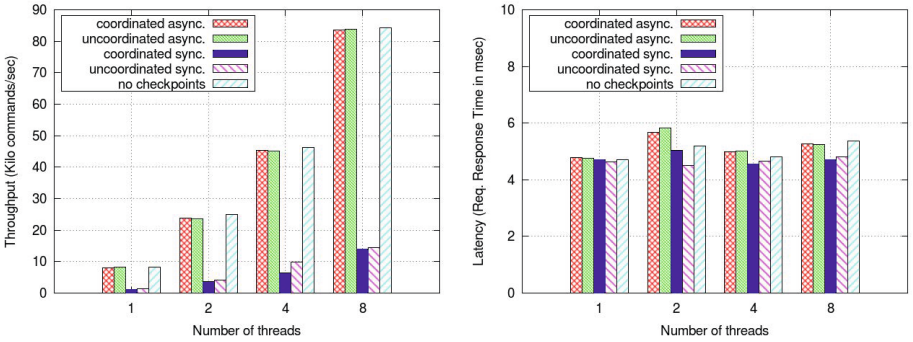

Fig. 4. Throughput and response time of coordinated and uncoordinated checkpointing with asynchronous and synchronous disk writes

checkpointing and recovery techniques for message-passing environments can be found in [5].

In [9], a parallelizer on each replica delivers commands in total order across replicas and distributes them among a set of threads for parallel execution. The parallelizer serializes the execution of dependent requests and ensures that their execution order follows the order decided by the agreement layer. This work also proposes a synchronization primitive executed on the replicas, but invoked by the agreement layer, to select a sequence number for checkpoints. Each replica blocks the execution of all the requests delivered after this sequence number until the checkpoint is completed. Since the selected sequence numbers may vary across the replicas, the recorded checkpoints are not identical across the replicas, similarly to our uncoordinated algorithm.

Eve [8] is a parallel replication model in which replicas first execute commands and then verify the equality of their states through a verification stage. Eve distinguishes one of the replicas as the primary to which clients send their requests. The primary groups commands into batches, assigns to each batch a unique sequence number, and transmits the batched commands to the other replicas. All the replicas, including the primary, are equipped with a deterministic mixer that converts a batch of requests into a set of parallel batches such that all the requests in a parallel batch can be executed in parallel. Once the execution of a batch terminates, replicas calculate a token based on their current state and send it to the verification stage. The verification stage checks the equality of the tokens. If the tokens are equal, replicas commit the executed batch, update their most recent stable sequence number, and respond to the clients. Otherwise, replicas must roll back the execution and re-execute the commands in the order determined by the primary. In Eve, checkpointing happens after the execution of each batch of commands and thus is more frequent than in traditional state-machine replication approach. Similar to a diverging replica, a recovering replica can request state changes from other replicas to build a consistent state.

In [7], the authors present Rex, a replication approach that benefits from multicore architectures. In Rex, one of the replicas, the primary, serves incoming client requests concurrently. There is no ordering requirements on delivering and executing client requests. The relative order among conflicting requests is given by synchronization primitives that coordinate access to shared data. The primary records causal dependencies among synchronization events during the execution and periodically proposes up-to-date traces to the secondary replicas, which reach a consensus on a sequence of traces to be executed. Secondary replicas faithfully follow the traces to replay the primary's execution. Checkpoints are used by Rex to allow a replica to recover from failures, to implement rollback on a downgrading replica (i.e., a leader replica that switches to a secondary role), and to facilitate garbage collection [7]. When a checkpoint is created, the primary pauses all threads before taking on any new requests and marks this particular point in the trace. A secondary receiving such a trace waits until the replay of the marked checkpoint and creates a snapshot. Once the checkpoint is created, the secondary replica resumes its execution and copies the checkpoint to other replicas in background. Since the checkpoint mark proposed by the primary establishes a consistent cut among replicas, the checkpoint generated is identical to each replica, similarly to our coordinated algorithm.

Another parallel replication technique is proposed in [15], where consensus is implemented to benefit from the multicore processors. Since the execution of commands is not parallelized on the replicas and follows the order decided by consensus, recovery can be implemented as in classic state-machine replication.

7 Conclusion

In this paper, we proposed two novel algorithms to address checkpointing in parallel state-machine replication [12]. The difference between our algorithms lies in the way checkpoint requests are synchronized with service commands. In coordinated checkpointing, checkpoints happen either before or after the execution of a batch of concurrent commands decided in a round. In uncoordinated checkpointing, a checkpoint can contain states of a replica in between two serialized commands. These two techniques have implications on performance, which increase in importance as the number of threads augments and checkpoints become more frequent.

Acknowledgments. We would like to thank the anonymous reviewers for their suggestions to improve the paper. This work was supported in part by CAPES PVE - Special Visiting Researcher - Grant No. 88881.062190/2014-01.

References

1. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley-Interscience (2004)

2. Bessani, A., Santos, M., Felix, J., Neves, N., Correia, M.: On the efficiency of durable state machine replication. In: ATC (2001)
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
4. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) 35(2), 288–323 (1988)
5. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3), 375–408 (2002), http://doi.acm.org/10.1145/568522.568525
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) 32(2), 374–382 (1985)
7. Guo, Z., Hong, C., Yang, M., Zhou, D., Zhou, L., Zhuang, L.: Rex: Replication at the speed of multi-core. In: Proceedings of the Ninth European Conference on Computer Systems, p. 11. ACM (2014)
8. Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., Dahlin, M.: All about eve: execute-verify replication for multi-core servers. In: OSDI, pp. 237–250. USENIX Association (2012)
9. Kotla, R., Dahlin, M.: High throughput byzantine fault tolerance. In: DSN (2004)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
11. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS) 16(2), 133–169 (1998)
12. Marandi, P.J., Bezerra, C.E.B., Pedone, F.: Rethinking state-machine replication for parallelism. In: ICDCS (2013)
13. Marandi, P.J., Primi, M., Pedone, F.: High performance state-machine replication. In: DSN (2011)
14. Marandi, P.J., Primi, M., Pedone, F.: Multi-Ring Paxos. In: DSN (2012)
15. Santos, N., Schiper, A.: Achieving high-throughput state machine replication in multi-core systems. In: ICDCS (2013)
16. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) 22(4), 299–319 (1990)