Analysis of Checkpointing Overhead in Parallel State Machine Replication

Odorico M. Mendizabal
Pontifícia Universidade Católica do Rio Grande do Sul
Brazil

Fernando Luís Dotti
Pontifícia Universidade Católica do Rio Grande do Sul
Brazil

Fernando Pedone
University of Lugano
Switzerland

ABSTRACT

State machine replication (SMR) is a well-established technique to fault-tolerant systems. In part, this is explained by the simplicity of the approach and its strong consistency guarantees. Recently, several proposals have suggested parallelizing the execution of state machine replicas to achieve high throughput. Concurrent execution of commands has many implications, including the recovery of replicas from failures. Conventional checkpointing techniques, for example, must be revisited in parallelized models. In this paper, we review parallel variations of state machine replication and discuss how checkpointing procedures apply to these models. Moreover, we evaluate the impact caused by checkpointing techniques on recovery through simulations.

Categories and Subject Descriptors

Computer systems organization [**Dependable and fault-tolerant systems and networks**]; Software and its engineering [**Extra-functional properties**]: Software fault tolerance— *Checkpoint / restart*

General Terms

State machine replication, checkpointing

Keywords

Distributed Systems, fault tolerance

1. INTRODUCTION

State machine replication (SMR) is a common technique used in the design of fault-tolerant systems. The idea is for replicas to start in the same initial state and deterministically execute the same sequence of clients commands. This way, replicas traverse the same sequence of internal states and produce the same output throughout their execution [11, 15]. Due to its simplicity and strong consistency guarantees (i.e., linearizability [8]), SMR has been largely studied in the last decades (e.g., [2, 3, 4, 5, 6]).

In order to improve SMR's throughput, some recent approaches propose to execute commands in parallel at each replica by exploiting application semantics [9, 10, 13]. In brief, these approaches classify commands as dependent or independent. Commands are *independent* if they access disjoint portions of the replica's state or only read shared state and *dependent* otherwise. Dependent commands must be processed in the same relative order at every replica to avoid inconsistencies. Independent commands can be executed in parallel and benefit from multi-core servers.

One central aspect of SMR concerns the ability to recover from replica failures. Recently, Bessani et al. [2] reviewed the literature and discussed shortcomings of common recovery techniques applied to SMR. In [6], performance implications of checkpointing in SMR are discussed. Just like sequential SMR, parallel SMR approaches must account for replica failure and recovery. Differently from sequential SMR, however, little is known about the implications of checkpointing and recovery on parallel SMR. While checkpoints are expected to reduce the throughput of replicas, it is not clear how they impact parallel SMR protocols.

In this paper, we review parallel SMR variations and their checkpointing strategies, discuss how checkpointing affects the performance of these models, and assess by means of simulations the impact of checkpointing on performance. We study the effects of the number of threads and the frequency of checkpoints on performance. Our results show that while checkpoints impact the performance of all techniques reported, techniques are not affected equally.

The remainder of this paper is organized as follows. In Section 2 we discuss the system model and assumptions. An overview of existent proposals for parallel SMR appears in Section 3. A performance evaluation of checkpointing mechanisms for parallel SMR is presented in Section 4. Section 5 concludes the paper.

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 $R = \{r_1, r_2, \ldots, r_n\}$ of replica processes. Clients in C send requests to replicas in R. We make no assumptions about the relative speed of processes or message delays, i.e., the system is asynchronous.

We assume the *crash-recovery* failure 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.

Clients submit command requests to the replicas, which execute the commands and reply to the clients. Command execution at the replicas is *deterministic*: the output of a command depends only on its input parameters and on the state read by the command. Replicas use an agreement protocol (e.g., Paxos [12]) to order command requests submitted by the clients.[1]

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 [8].

3. PARALLEL APPROACHES TO SMR

In contrast to classical SMR, in parallel replication techniques some commands can be executed concurrently. Next, we present parallel approaches to SMR and discuss how checkpointing can be handled by each of them.

3.1 The parallelizer approach

In [10] replicas are augmented with a *parallelizer* that bridges the delivery and execution of commands. Based on application semantics, the parallelizer serializes the execution of dependent commands according to the delivery order and dispatches independent commands to be processed in parallel by a set of working threads (see Figure 1). To understand the interdependencies between commands, assume two commands c_i and c_j, where W_i and W_j indicate the commands' write-set and R_i and R_j indicate their read-set. According to [10], c_i and c_j are *dependent* if any of the following conditions holds: (i) $W_i \cap W_j \neq \emptyset$; (ii) $W_i \cap R_j \neq \emptyset$, or (iii) $R_i \cap W_j \neq \emptyset$. Two commands are independent if they are not dependent.

The parallelizer follows a producer-consumer model. When a working thread asks for a command to be processed, the parallelizer searches for a delivered command c that has not been assigned to any working thread yet and is independent of all commands currently in execution. If c exists, the parallelizer assigns it to the working thread; otherwise, it blocks the working thread until a command satisfies the requirements described.

Checkpointing. Relying on the total order, after an agreed number k of commands are delivered at a replica, the parallelizer stops assigning commands to working threads. Once all commands in execution are finished, the replica takes a checkpoint and then resumes normal execution [10]. Replicas will produce identical checkpoints since any two replicas take checkpoints at fixed and deterministic intervals.

Besides the costs inherent to the checkpointing itself, such as logging and maintenance of checkpoint structures, checkpoints in the parallelizer approach induce additional overhead since at the moment a checkpoint is invoked, new incoming commands cannot be executed and, as a consequence, working threads may be idle until all threads finish their work and the checkpoint is taken.

3.2 The Execution-Verify approach (Eve)

Another approach that allows SMR to scale to multi-core servers is Eve (Execution–Verify) [9]. Different from other SMR approaches, Eve replicas first execute commands and

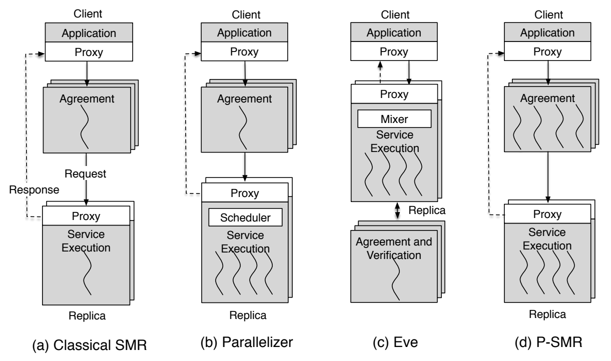

Figure 1: Approaches to parallel SMR.

then verify the equality of their states through a verification stage. Before execution, a primary replica groups client commands into batches and transmits the batched commands to all replicas. Then, replicas speculatively execute batched commands in parallel. After the execution of a batch, the verification stage checks the validity of replica's state, as defined by the common state reached by a majority of replicas. If too many replicas diverge, replicas roll back to the last verified state and re-execute the commands sequentially. Eve minimizes divergence through a mixer stage that applies application-specific criteria to produce batches of commands that are unlikely to interfere with each other [9].

Checkpointing. Even though checkpointing and recovery are not explicitly described in [9], taking checkpoints along a sequence of bounded batches is straightforward. After the execution of a batch, replicas check the equality of their states and diverging replicas assume the state reached by the majority of the replicas. The verification stage causes replicas to reach identical states. For this purpose, every replica should periodically create a checkpoint right after the n-th batch since the last checkpoint has been created and verified. This strategy is similar to that presented in the previous section, but instead of taking checkpoints every k commands have been processed, in Eve checkpoints are taken after commands in n batches have been executed.

3.3 The Parallel SMR approach (P-SMR)

In [13], the authors propose a variation of parallel SMR, where the execution and the delivery of commands occur in parallel. Instead of using a single sequence of consensus executions to order commands, multiple sequences of consensus are used. For each sequence of consensus γ_i, there is a working thread t_i responsible for processing commands decided in γ_i. Independent commands proposed in different sequences of consensus are executed concurrently. Commands decided in a given sequence of consensus γ_i are processed in the same relative order across replicas. Another consensus sequence, γ_{all}, shared among all threads, allows threads to order commands across sequences. In short, multiple consensus sequences, one per thread, result in commands ordered and executed concurrently. Dependent commands are proposed either in the same sequence or in the sequence shared among all threads; in both cases, dependent commands are executed in the same order across replicas.

The mapping of commands onto consensus sequences is application-dependent and the responsibility of clients. Clients propose commands by choosing the consensus sequence that guarantees ordered execution of dependent commands while maximizing parallelism of independent commands.

[1]Solving agreement in the model described above requires additional assumptions (e.g., [1, 7]) In the following, we simply assume the existence of an agreement oracle.

Checkpointing. In [13] not only the execution, but also the delivery of messages occur in parallel. Thus, solutions analogous to the ones described are not possible in P-SMR since replicas evolve through different states. In [14], two checkpointing and recovery algorithms for P-SMR are proposed, called *coordinated* and *uncoordinated* checkpoints.

The coordinated checkpointing algorithm makes use of a special checkpoint message *CHK* that depends on all commands. In order to generate new checkpoints, a coordinator replica periodically proposes a *CHK* message. Upon deciding on a *CHK* message, the execution model provided by P-SMR ensures that every replica processes exactly the same set of messages before processing the checkpoint, which takes place in the same order in every replica. One consequence of coordinated checkpointing is that replicas build the same sequence of checkpoints.

In the uncoordinated algorithm replicas evolve independently with possibly different checkpoints, enhancing concurrency and throughput. Periodically, one coordinator thread in each replica takes a checkpoint, which contains the most recent state modifications. To take a checkpoint, threads in a replica must coordinate their execution, but no coordination across replicas is needed. Since the k-th checkpoint taken by any pair of replicas is possibly different, together with a checkpoint, replicas keep a vector with the number of commands processed by each thread at the moment the checkpoint was taken. Uncoordinated checkpointing is expected to introduce less overhead in a replica's normal execution than coordinated checkpointing. On the downside, with uncoordinated checkpoints, replicas create different sequences of checkpoints and therefore cannot benefit from *collaborative state transfer* [2].

4. PERFORMANCE EVALUATION

In this section, we evaluate the impact of checkpointing in parallel SMR approaches. Our analysis aims to quantify the cost of synchronization due to checkpoints.

We implemented a discrete-event simulation model in C++ and run each experiment until the average value measured for the service latency lies in a 98% confidence interval. We built simulation models for parallel variations of SMR proposed by Kotla *et al.* [10], Kapritsos *et al.* [9], and Marandi *et al.* [13], respectively called *Parallelizer*, *Eve*, and *P-SMR*. The classical SMR model is a special case of parallel SMR where just one thread executes commands.

We ran simulations with and without checkpointing, and considered different requests execution time: fixed-duration; uniformly distributed, and exponentially distributed. We report results using exponentially distributed command duration. Our conclusions apply to the other distributions.

To focus the analysis on the impact of synchronization due to checkpointing, we generate only independent commands in the workload, removing the possibility of thread idleness due to the synchronization needed by dependent commands.

4.1 The effects of the number of threads

The maximum normalized throughput (Figure 2) for x threads, $norm\_tput(x)$, was calculated as the ratio between the value measured with x threads, $measured\_tput(x)$, and the ideal throughput in a perfectly scalable system, that is:

$$norm\_tput(x) = measured\_tput(x)/(norm\_tput(1) \times x)$$

In the graphs, command durations follow an exponential distribution with an average command execution time of 0.5. We evaluate executions without and with checkpoints, in which case they are taken every 400 commands. Table 1 presents the throughput for each technique.

Table 1: Maximum throughput with instantaneous checkpoints, in commands per time unit

threads	Parallelizer		Eve		P-SMR		
	no cp	w cp	no cp	w cp	no cp	coord cp	uncoord cp
1	2	2	1.98	1.98	2	2	2
2	4	3.97	3.89	3.89	3.9	3.79	3.9
4	8	7.89	7.61	7.61	7.81	6.92	7.73
8	16	15.6	14.43	14.43	15.92	12.22	15.09
16	31.94	30.35	25.57	25.57	31.94	20.39	28.08
32	63.95	56.93	39.27	39.27	63.68	31.59	48.08
64	127.86	97	39.43	39.43	126.77	45.67	73

We first consider executions where, if enabled, checkpoints are instantaneous. By ignoring the time taken to create a checkpoint, the results reveal the overhead caused exclusively by checkpointing synchronization. As can be seen in Figure 2 (top), the synchronization overhead increases with the number of threads. When replicas are configured with one single thread there is no synchronization costs, a behavior that corresponds to classical SMR.

When checkpoints are disabled (i.e., "no cp" in the graphs), the throughput of Parallelizer and P-SMR scales proportionally to the number of threads (not seen in the graphs due to normalization). Eve presents lower throughput than the other techniques due its verification stage, forcing thread synchronization. When checkpoints are enabled, the overhead added by P-SMR coordinated checkpointing and Eve are the most impacting among the evaluated approaches. Moreover, the overhead grows as the number of threads increases. The overhead caused by checkpoints in the Parallelizer is smaller than in P-SMR uncoordinated because it can perform a more efficient scheduling than P-SMR, where clients decide which thread should execute a command. P-SMR is more advantageous than the Parallelizer in executions where the scheduler becomes the bottleneck, and the technique cannot scale with additional threads [13].

Figure 2 (bottom) depicts the maximum throughput for scenarios in which checkpoints take 5 time units. Again, performance degrades with the number of threads. In the Parallelizer, the most efficient technique, with 64 threads, replicas spend 65% of the time executing commands and 35% executing checkpoints. In Eve, with configurations with 64 threads, replicas spend 75% and 25%, respectively. This is a consequence of the fact that checkpoints happen more often with the Parallelizer since it has higher throughput than Eve. Practical implementations can consider the tradeoff between checkpoint duration and checkpoint frequency (i.e., number of commands processed per checkpoint) to tune the use of resources with command and checkpoint execution.

4.2 The effects of the checkpoint frequency

We evaluate checkpoint frequency in 16-core replicas, varying checkpoint interval from 400 to 6400 requests. Figure 3 shows the throughput and latency for workloads where request duration follows an exponential distribution with average 0.5 and checkpoints take 5 time units. We configured the workload in this experiment to reach 75% of the maximum throughput reachable by each model.

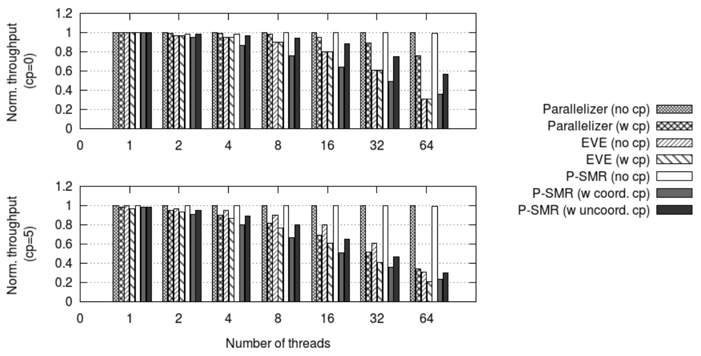

Figure 2: Max. normalized throughput with instantaneous checkpoints and 5-time unit checkpoints.

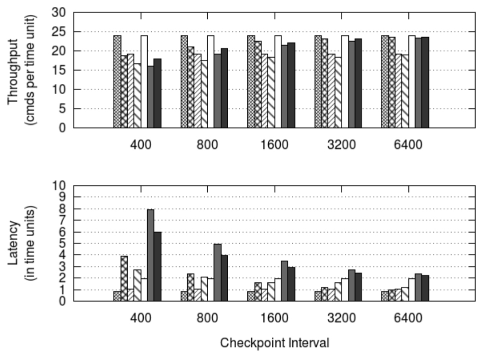

Figure 3: Throughput and latency of various techniques. Replicas configured with 16 threads.

Checkpoints have an impact on the throughput and response time of replicas (top and bottom of Figure 3, respectively), although the overhead caused by checkpoints decreases as checkpoints become more infrequent. The Parallelizer and uncoordinated P-SMR present the higher throughput rates. Regarding latency, the Parallelizer outperforms P-SMR, presenting lower command's response time. This happens due to thread scheduling. While in the Parallelizer approach incoming commands can be dispatched to any free working thread, in P-SMR clients send commands directly to a given thread which may be busy even if some other thread is available. Although the Parallelizer and Eve schedule commands more efficiently within replicas, they are subject to a single point of contention, the scheduler in the Parallelizer and the mixer in Eve. By relying on clients to distribute commands across threads, P-SMR avoids this potential bottleneck, although it becomes exposed to sub-optimal scheduling. In our simulation we do not associate scheduling costs to any of the models.

5. CONCLUSION

We reviewed and evaluated the performance of checkpointing in existing parallel SMR approaches. Our analysis focused mostly on the synchronization overhead. The proposed simulation models capture the inherent differences between existing approaches and allowed us to measure the way checkpoints affect performance in each case.

Checkpoints reduce the performance of all considered techniques and the overhead due to checkpoints increases with the number of threads, even though techniques are not affected equally. Both the Parallelizer and P-SMR experience a reduction in performance with checkpoints, but P-SMR is more vulnerable to checkpoints. Since Eve requires coordination even in the absence of checkpoints, instantaneous checkpoints do not affect its performance, although real checkpoints do reduce its throughput.

The frequency of checkpoints has a bigger impact on the latency of the various approaches than on their throughput. When the checkpoint frequency varies from 400 to 800 commands, the Parallelizer experiences a throughput improvement from 18.79 to 21.04 commands per time unit, while latency is reduced from 3.86 to 2.36 time units. P-SMR's latency is particularly vulnerable to checkpoint overhead due to its sub-optimal scheduling of commands.

Acknowledgements: This work was supported in part by CAPES PVE project 88881.062190/2014-01.

6. REFERENCES

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125, 2000.

[2] A. Bessani, M. Santos, J. Felix, N. F. Neves, and M. Correia. On the efficiency of durable state machine replication. In *USENIX ATC*, 2013.

[3] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[4] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, 1999.

[6] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *PODC*, 2007.

[7] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[8] A. D. Fekete and K. Ramamritham. Consistency models for replicated data. In *Replication*, pages 1–17. Springer, 2010.

[9] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: execute-verify replication for multi-core servers. In *OSDI*, 2012.

[10] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *DSN*, 2004.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[12] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[13] P. J. Marandi, C. E. B. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *ICDCS*, 2014.

[14] O. M. Mendizabal, P. J. Marandi, F. L. Dotti, and F. Pedone. Checkpointing in parallel state-machine replication. In *OPODIS*, 2014.

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