PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

FAST RECOVERY IN PARALLEL STATE MACHINE REPLICATION

ODORICO MACHADO MENDIZABAL

Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Fernando Luís Dotti
Co-Advisor: Prof. Fernando Pedone

Porto Alegre
2016

REPLACE THIS PAGE WITH
THE LIBRARY CATALOG
PAGE

REPLACE THIS PAGE WITH THE COMMITTEE FORMS

Dedico este trabalho à minha familia.

“The art of simplicity is a puzzle of complexity.”
(Douglas Horton)

ACKNOWLEDGMENTS

This thesis is the product of an interesting, albeit long and challenging, journey. It would not be possible without the help of many people. I am extremely grateful to all of them.

First and foremost, I want to thank my advisors, Fernando Dotti and Fernando Pedone. Professor Dotti helped me whenever I needed and offered guidance and encouragement at critical points in time. I am truly thankful for his selfless dedication to both my personal and academic development. His attention to detail taught me the importance of clear thinking and rigor in science. Professor Pedone showed strong commitment and readiness to help despite the distance. His intuition and knowledge in systems building is remarkable, and I learned much from our collaboration. Pedone has also been a source of productive inspiration and instruction for me. I am very fortunate to have worked with them.

The other members of the thesis committee (Avelino Francisco Zorzo, Eduardo Adilio Pelinson Alchieri, Elias Procópio Duarte Jr., and during the qualification exam, Joni da Silva Fraga) have provided detailed feedback. Their insights and comments on the work have been greatly appreciated and improved the quality of this document.

I have also enjoyed the collaboration with Leila Ribeiro. She always had something to say about my work, and I appreciated this very much. For the elaboration of this thesis, she shared important ideas and helped improving the quality of correctness proofs.

I would like to emphasize my gratitude to colleagues at FURG, they supported me greatly and allowed me to keep the focus exclusively in this thesis during the last years. I would like to thank my fellow graduate students from PUCRS, who were a constant source of help, comfort, and knowledge. By risking leaving someone out, I want to acknowledge Carlos Morateli, Thiago Paes, Dalvan Griebler, Rasha Hasan, Alan dos Santos, Joaquin Assunção, Rômulo Reis, Leandro Costa, Maicon Bernardino, Aline Zanin, Kassiano Matteussi, Walter Ritzel, Túlio Baségio, Ricardo Moreira, Thiago Sarturi, Rudá de Moura, Daniel Menzen, Felipe Kuentzer, Henrique Chamorra, and Bruno Ferreira. I am also grateful to PUCRS staff, who made my life less stressful, mainly by taking care of everything when I was living abroad. Thanks to Régis Escobal, Diego Cintrão, Vanessa Torres and Thiago Lingener. I cannot forget to thank Beatriz Franciosi, who demonstrated very supportive, and always had some positive words to share.

I like to thank Pedone's research group for their hospitality and friendship during my visit to USI, Lugano. Special thanks go to Tu Dang, Daniele Sciascia, Paulo Coelho, Edson

Tavares, Samuel Benz, Long Hoang Le, Leandro Pacheco, Eduardo Bezerra, Daniel Cason, and Parisa Marandi, for spending many unforgettable moments together, and for making my stay so pleasurable and productive. Samuel and Parisa had an important influence on this work. They did not hesitate in sharing their Paxos implementation and gave a lot of very helpful advices. I have learned tremendously from them and I highly appreciate these contributions. It is also worth noting that USI infrastructure was always available for me to run experiments. Thanks, Pedone, Leandro, Paulo and Tu, for the kindness and team spirit.

While living in Lugano, some friends exceeded their obligations and played the role of the family in some very special cases. That is something I will never forget. Thanks, Paulo Butzen, Raquel Silveira, Jônata Carvalho, Lara Carvalho, Giuseppe Bellanti, Monica Santos, and Greta Lemos.

I would like to thank my wonderful wife. Cristina was co-author in my first works in the university, and now she is my co-author in life. During the last years, she employing an extraordinary amount of patience, understanding, and love. Accomplishing this thesis would be much harder without her support.

I am profoundly grateful to my family for encouraging me and give unconditional support throughout my whole life. Especially for my parents, who always gave me good advices, believing in me and pushing me forward. Their example has always been the motivating force behind all my academic and professional accomplishments. This thesis is dedicated to them.

Last, but not least, I would like to acknowledge and thank the institutions that provided financial support for this thesis, including the HP Inc under grant HP-PROFACC, and the Swiss Federal Commission for Scholarships for Foreign Students under grant Swiss Government Excellence Scholarship.

FAST RECOVERY IN PARALLEL STATE MACHINE REPLICATION

RESUMO

A replicação máquina de estados é uma técnica bem estabelecida para desenvolvimento de sistemas tolerantes a faltas. Em parte, isso é explicado pela simplicidade da abordagem e sua garantia de consistência forte. O modelo de replicação máquina de estados tradicional baseia-se na execução sequencial de requisições para garantir consistência forte entre as réplicas. A sequencialidade da execução, no entanto, compromete a escalabilidade. Recentemente, algumas propostas sugeriram paralelizar a execução de algumas requisições visando um aumento na vazão. Apesar do sucesso da replicação máquina de estados paralela em obter alto desempenho, as implicações deste modelo em procedimentos de recuperação são desprezadas. Mesmo para a abordagem de replicação máquina de estados tradicional, poucos estudos têm considerado as questões envolvidas na recuperação de réplicas defeituosas. A motivação desta tese é elucidar os desafios e implicações no desempenho decorrentes de mecanismos de pontos de verificação e recuperação em replicação máquina de estados paralela. A tese também avança no estado-da-arte, propondo novos algoritmos para pontos de verificação e recuperação no contexto de replicação máquina de estados paralela. Criar pontos de verificação de forma eficiente em tais modelos é mais desafiador do que na replicação máquina de estados clássica porque deve-se considerar a execução concorrente de comandos. Nesta tese, nós revisitamos as técnicas para pontos de verificação em abordagens paralelas de replicação máquina de estados e comparamos o impacto destas no desempenho através de simulação. Além disso, nós propomos duas técnicas de ponto de verificação para um destes modelos paralelos. Recuperar uma réplica requer: (a) obter e instalar o estado de um ponto de verificação de uma réplica atualizada, e (b) recuperar e re-executar os comandos não refletidos no ponto de verificação. Técnicas paralelas para replicação máquina de estado tornam a recuperação de réplicas particularmente difícil uma vez que a vazão de processamento durante a execução normal (isto é, na ausência de falhas) é muito alta. Consequentemente, o registo de

comandos que precisa ser re-executado é tipicamente grande, o que atrasa a recuperação. Nós apresentamos duas novas técnicas para otimizar a recuperação em replicação máquina de estados paralela. A primeira técnica permite que novos comandos sejam executados em paralelo com a re-execução dos comandos não refletidos no ponto de verificação. Isto ocorre antes da réplica estar completamente atualizada. A segunda técnica introduz recuperação de estado sob-demanda, permitindo que segmentos de um ponto de verificação possam ser recuperados apenas quando necessários, ou ainda, concorrentemente. Nós avaliamos o desempenho de nossas técnicas de recuperação usando um protótipo completo para replicação máquina de estados paralela e comparamos o desempenho destas técnicas com mecanismos tradicionais de recuperação em diferentes cenários.

Palavras-Chave: Replicação Máquina de Estados, pontos de verificação, recuperação, alta disponibilidade.

FAST RECOVERY IN PARALLEL STATE MACHINE REPLICATION

ABSTRACT

A well-established technique used to design fault-tolerant systems is state machine replication. In part, this is explained by the simplicity of the approach and its strong consistency guarantees. The traditional state machine replication model builds on the sequential execution of requests to ensure consistency among the replicas. Sequentiality of execution, however, threatens the scalability of replicas. Recently, some proposals have suggested parallelizing the execution of replicas to achieve higher performance. Despite the success of parallel state machine replication in accomplishing high performance, the implication of such models on the recovery is mostly left unaddressed. Even for the traditional state machine replication approach, relatively few studies have considered the issues involved in recovering faulty replicas. The motivation of this thesis is clarifying the challenges and performance implications involved in checkpointing and recovery for parallel state machine replication. The thesis also aims to advance the state-of-the-art by proposing novel algorithms for checkpointing and recovery in the context of parallel state machine replication. Performing checkpoints efficiently in such parallel models is more challenging than in classic state machine replication because the checkpoint operation must account for the execution of concurrent commands. In this thesis, we review checkpointing techniques for parallel approaches to state machine replication and compare their impact on performance through simulation. Furthermore, we propose two checkpoint techniques for one of these parallel models. Recovering a replica requires (a) retrieving and installing an up-to-date replica checkpoint, and (b) restoring and re-executing the log of commands not reflected in the checkpoint. Parallel state machine replication render recovery particularly challenging since throughput under normal execution (i.e., in the absence of failures) is very high. Consequently, the log of commands that need to be applied until the replica is available is typically large, which delays the recovery. We present two novel techniques to optimize recovery in parallel state machine replication. The first technique allows new commands to execute concurrently with the execution of logged

commands, before replicas are completely updated. The second technique introduces on-demand state recovery, which allows segments of a checkpoint to be recovered concurrently. We experimentally assess the performance of our recovery techniques using a full-fledged parallel state machine replication prototype and compare the performance of these techniques to traditional recovery mechanisms under different scenarios.

Keywords: State machine replication, checkpointing, recovery, high availability.

LIST OF FIGURES

Figure 1.1 – Log size estimation. 26

Figure 3.1 – Classical versus parallel state machine replication 36

Figure 3.2 – CBASE scheduling example. 38

Figure 3.3 – “Execute-then-verify” design. 39

Figure 3.4 – Eve execution example. 39

Figure 4.1 – 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. 50

Figure 4.2 – Throughput and latency of a replica executing commands with an exponentially distributed execution time (average of 0.5 time units). 51

Figure 4.3 – 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. 51

Figure 4.4 – Throughput and response time of coordinated and uncoordinated checkpointing with asynchronous and synchronous disk writes. 52

Figure 4.5 – Maximum normalized throughput with instantaneous checkpoints (top) and 5-time unit checkpoints (bottom). Checkpoint interval of 400 commands and command execution duration exponentially distributed with average 0.5. 54

Figure 4.6 – Throughput (top) and latency (bottom) of various techniques. Replicas configured with 16 threads, commands execution exponentially distributed with average commands of 0.5 and checkpoint duration of 5 time units. 56

Figure 5.1 – Four representations of a dependency graph with six commands. (a) The original dependency graph, where edge $x \rightarrow y$ means that commands x and y are dependent and x was delivered before y. (b) The original graph grouped in batches of two commands. (c) The abridged dependency graph; notice that commands b and c are serialized in the abridged graph. (d) The stored dependency graph; batches B_1 and B_2 are assigned to worker thread t_1 and batch B_3 is assigned to worker thread t_2; $b(B)$ is a digest of the variables accessed by commands in B, used to track dependencies between command batches. The stored dependency graph preserves all dependencies defined in the original dependency graph. 60

Figure 5.2 – Resulting bitmap for a batch of commands. The bitmap size is m and it encodes 2 keys. 65

Figure 5.3 – Maximum throughput versus number of threads. 67

Figure 5.4 – Throughput versus latency of 1k-byte commands. 67

Figure 5.5 – Throughput (top) and latency (bottom) according to the batch size. . . 68
Figure 5.6 – Throughput variation according to dependency probability. 69
Figure 5.7 – Throughput variation according to the checkpoint interval. 69
Figure 5.8 – Checkpoint duration according to the checkpoint size. 70
Figure 5.9 – Log size increase over time. 70
Figure 6.1 – Resulting bitmap item for a group of batches. The bitmap item contains encoded information for batches *i* to *j*. 77
Figure 6.2 – Dependency log structure. 77
Figure 6.3 – Throughput and latency for crash-recovery trace using: (a) classical recovery; (b) speedy recovery; (c) speedy recovery combined with on-demand state transfer 80
Figure 6.4 – Throughput using speedy recovery and on-demand state transfer with 0% of dependency probability. 81
Figure 6.5 – Throughput using speedy recovery and on-demand state transfer with 5% of dependency probability. 82
Figure 6.6 – Time taken during recovery for workloads with dependency probability 0% (top) and 5% (bottom) 84
Figure 6.7 – Throughput using speedy recovery and on-demand state transfer with real dependency probability. 85
Figure 6.8 – Pending requests structures: (a) a list of batches combined with a consolidated bitmap; (b) a vector pointing to lists tuples containing batches and their bitmaps. 85
Figure 6.9 – Access to pending requests stored in *v_seq*. 86
Figure 6.10 – Throughput using speedy recovery and on-demand state transfer with real dependency probability and rescheduling of pending requests enabled. 87
Figure 6.11 – Throughput using speedy recovery and on-demand state transfer with real dependency probability and rescheduling of pending requests enabled (commands size is 1024 bytes). 88
Figure A.1 – Simulation execution example 100
Figure A.2 – CBASE simulation model 101
Figure A.3 – P-SMR simulation model 102

LIST OF TABLES

Table 4.1 – Maximum throughput with instantaneous checkpoints 54
Table 5.1 – Smallest log size estimation according to the checkpoint size 71
Table 6.1 – Recovery time and service downtime 80
Table 6.2 – Recovery techniques comparison 83
Table 6.3 – Recovery techniques comparison for workload with real dependency probability 88

LIST OF ALGORITHMS

3.1 **P-SMR** 41
4.1 **Coordinated checkpoint** 46
4.2 **Uncoordinated checkpoint** 48
5.1 **Efficient Parallel SMR** 63
6.1 **Recovery** 75
A.1 **CBASE – execution** 103
A.2 **CBASE – parallelizer** 104
A.3 **P-SMR – execution** 105

CONTENTS

1 INTRODUCTION 25
1.1 CONTRIBUTIONS 27
1.2 ROADMAP 29

2 SYSTEM MODEL 31
2.1 PROCESSES AND COMMUNICATION 31
2.2 MULTICAST AND BROADCAST 32
2.3 CONSISTENCY 33

3 BACKGROUND 35
3.1 STATE MACHINE REPLICATION 35
3.2 PARALLEL STATE MACHINE REPLICATION 36
3.2.1 CBASE 36
3.2.2 THE EXECUTION-VERIFY APPROACH (EVE) 37
3.2.3 P-SMR 40

4 CHECKPOINTING IN PARALLEL STATE MACHINE REPLICATION 43
4.1 CHECKPOINTING IN CBASE 43
4.2 CHECKPOINTING IN EVE 44
4.3 CHECKPOINTING IN P-SMR 44
4.3.1 COORDINATED CHECKPOINTING 45
4.3.2 UNCOORDINATED CHECKPOINTING 45
4.3.3 COORDINATED VERSUS UNCOORDINATED CHECKPOINTING 47
4.3.4 PERFORMANCE ANALYSIS 49
4.4 PERFORMANCE EVALUATION OF CHECKPOINTING FOR PARALLEL SMR MODELS 52
4.5 RELATED WORK 57

5 EFFICIENT PARALLEL STATE MACHINE REPLICATION 59
5.1 OVERALL IDEA 59
5.2 ALGORITHM IN DETAIL 62
5.3 EVALUATION 64
5.3.1 IMPLEMENTATION 64

5.3.2 ENVIRONMENT AND CONFIGURATION 66
5.3.3 PERFORMANCE ANALYSIS 66

6 FAST RECOVERY IN PARALLEL STATE MACHINE REPLICATION 73
6.1 SPEEDY RECOVERY OF LARGE LOGS 73
6.2 ON-DEMAND STATE RECOVERY 75
6.3 EVALUATION 76
6.3.1 IMPLEMENTATION 76
6.3.2 GOALS AND METHODOLOGY 77
6.3.3 PERFORMANCE ANALYSIS 78
6.4 RELATED WORK 88
6.4.1 RECOVERY IN CLASSICAL STATE MACHINE REPLICATION 89
6.4.2 RECOVERY IN PARALLEL STATE MACHINE REPLICATION 89
6.4.3 RECOVERY IN TRANSACTIONAL SYSTEMS 90

7 CONCLUSION 91
7.1 CONTRIBUTIONS 91
7.2 FUTURE WORK 92

REFERENCES 95

APPENDIX A – Simulation Tool **99**
A.1 SIMULATION OF PARALLEL SMR MODELS 100
A.1.1 LOAD GENERATION 101
A.1.2 CBASE MODEL 102
A.1.3 P-SMR MODEL 104

APPENDIX B – Correctness of Proposed Algorithms **107**
B.1 CORRECTNESS OF P-SMR (ALGORITHM 3.1) 107
B.2 CORRECTNESS DISCUSSION OF COORDINATED CHECKPOINT (ALGORITHM 4.1) 109
B.3 CORRECTNESS DISCUSSION OF UNCOORDINATED CHECKPOINT (ALGORITHM 4.2) 109
B.4 CORRECTNESS DISCUSSION OF THE PROPOSED PARALLEL SMR (ALGORITHM 5.1) 110
B.5 CORRECTNESS DISCUSSION OF SPEEDY RECOVERY (ALGORITHM 6.1) . 113

1. INTRODUCTION

Many Internet services have strict availability and performance requirements. High availability requires tolerating component failures and can be achieved with replication. State machine replication (SMR) is a classical approach to managing replicated servers [Lam78, Sch90]. In state machine replication replicas start in the same initial state and deterministically execute an identical sequence of client commands. Consequently, replicas traverse the same states and produce the same responses. To boost the performance of the service, one can deploy replicas in high-end servers (scale up). Since modern servers increase processing power by aggregating multiple processors (e.g., multi-core architectures), to benefit from these architectures, replicas need to parallelize the execution of commands. Despite the fact that concurrent execution of commands seems at odds with SMR's requirement of deterministic execution, some proposals have revisited the classical approach to introduce parallelism (e.g., [KD04, KWQ+12, MBP14]).

Parallel state machine replication techniques exploit service semantics and are based on the observation that independent commands can execute concurrently while only dependent commands must be serialized and executed in the same order by the replicas. Two commands are dependent *directly* if they access any common variable v and one of them changes v or *indirectly*, through transitivity (i.e., if commands c_i and c_j, and c_j and c_k are pairwise dependent then c_i and c_k are dependent). Otherwise, commands are independent. Executing dependent commands concurrently may result in unpredictable and inconsistent states across replicas. Although the performance of parallel state machine replication techniques depend on specifics of the technique, available hardware resources, and the workload mix of independent and dependent commands, studies report large improvements in performance (e.g., [KWQ+12, MP14]).

This thesis takes a close look at parallel state machine replication, focusing specially on recovery, a topic that has received little attention in the literature. Although one could use recovery techniques designed for classical state machine replication (e.g., [BSF+13, CL99, CKL+09]), these are not optimized for parallel state machine replication. To understand why, we must review the basics of recovery in state machine replication. During normal operation (i.e., in the absence of failures), replicas log the commands they execute and periodically checkpoint the service state against stable storage. When a replica creates a new checkpoint, it can trim the log, removing commands that are already reflected in the checkpoint. Upon recovery, the recovering replica retrieves a current checkpoint and the log of "old commands", that is, commands that were already executed by the other replicas but are not included in the retrieved checkpoint. The recovering replica is available and can execute "new commands" after it has installed the checkpoint and executed the retrieved log of old commands. This

procedure can be optimized in many ways, as we discuss later, but existing optimizations are orthogonal to the techniques we propose.

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 parallel state machine replication is more challenging than in classic state machine replication because the checkpoint operation must account for the execution of concurrent commands. In this thesis we review checkpointing techniques for parallel state machine replication and compare their impact on performance by means of simulation. Furthermore, we propose two checkpoint techniques for P-SMR [MBP14]: coordinated and uncoordinated. Coordinated checkpoints incur system-wide synchronization, while uncoordinated checkpoints are local to a replica.

During normal operation, checkpoints hurt performance since they introduce inherent overheads (e.g., execution stalls). This calls for sparse checkpoints, a strategy adopted by some existing systems (e.g., [CL99, KBC+12]). Sparse checkpoints, however, result in large logs of old commands, which slows down recovery. Even though some techniques can reduce the overhead of checkpoint creation (e.g., copy-on-write), the frequency of checkpoints is also impacted by practical concerns. A replica typically performs checkpoints sequentially, only starting one checkpoint after the previous one has finished. Thus, checkpoint frequency is ultimately limited by how quickly a single checkpoint can be performed. Figure 1.1 illustrates a replica performing checkpoints every $cp_{interval}$ and checkpoints take cp_{dur} time units to be completely saved. The dark gray area indicates the minimum number of requests in a log, i.e., the smallest possible log. The longest log includes the requests processed during checkpoint creation plus the requests processed in the last checkpoint interval (i.e., the light gray area plus the dark gray area).

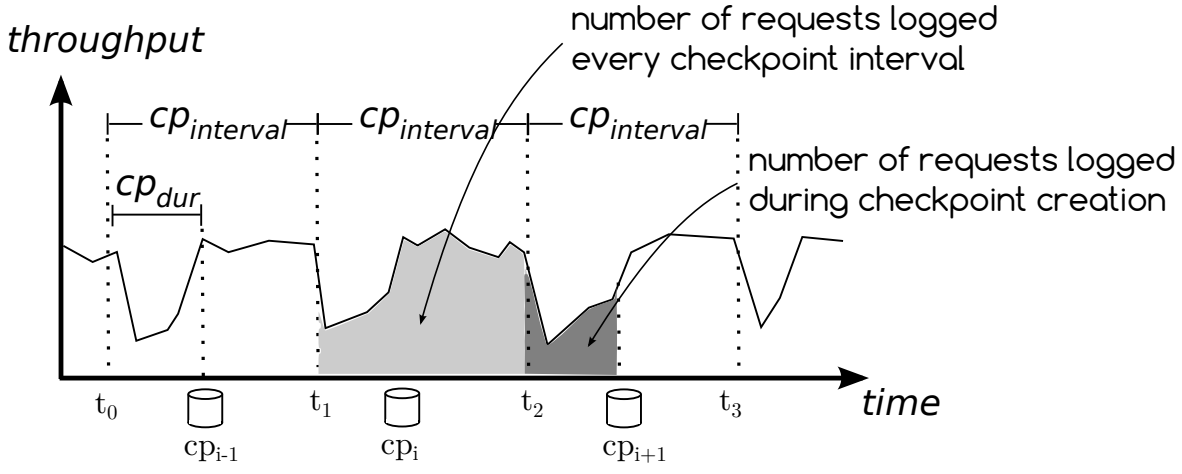


Figure 1.1 – Log size estimation.

Even when considering the smallest possible logs, the high throughput achieved by parallel approaches to state machine replication incur an expressive number of logged requests. In our analysis (details in Chapter 5), it takes about 15 seconds to perform a 512M-byte checkpoint, including serialization of the state. At this checkpoint frequency, under peak load, our parallel state machine replication prototype can execute nearly one million commands between two checkpoints, amounting to almost 1G-byte worth of logged old commands.

While one may attempt to resort to techniques to increase the frequency of checkpoints (e.g., copy-on-write) and reduce the size of the log of old commands, we approach recovery from a different perspective. Instead of attempting to increase the frequency of checkpoints to reduce the log of old commands and lower the downtime of a recovering replica, we rethink recovery in state machine replication and propose a more fundamental approach. We introduce two optimizations tailored to parallel state machine replication. The first optimization allows new commands to execute before old commands have been processed. This is based on the observation that a new command does not need to wait for an old command to be executed if the two commands are independent. The second optimization is inspired by the fact that a considerable amount of recovery time is due to state transfer and installation. We propose to divide a checkpoint into segments, and retrieve and install each segment only when it is needed for the execution of a command. We also allow a segment to be concurrently retrieved and installed with other segments. Therefore, checkpoint segments are handled on demand and possibly concurrently. To integrate these two optimizations in parallel state machine replication, we introduce a protocol that handles command dependencies effectively (details in Chapter 5). Our protocol encodes command dependencies in batches, using a data structure that trades the overhead involved in processing commands individually for concurrency in the execution of these commands, and minimizes synchronization among worker threads at a replica. We have assessed these techniques experimentally and observed that they can reduce recovery duration by more than 10 times when compared to traditional recovery techniques.

1.1 Contributions

The thesis makes the following contributions:

- We propose two checkpoint techniques for P-SMR [MBP14], coordinated and uncoordinated. In P-SMR, replicas alternate between the execution of concurrent commands (i.e., those mutually independent) and the execution of sequential commands. Our 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 Chapter 4; (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 the implications of each technique using simulation models and an in-memory database service.

- We review variations of parallel state machine replication and checkpointing strategies for each variation, 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, techniques are not affected equally.

- We present a new parallel state machine replication protocol that schedules commands for execution efficiently. When assessing dependencies among commands, our protocol considers batches of commands, instead of commands individually. While batching reduces the overhead of command handling, it decreases concurrency among independent commands, since dependencies are tracked at the unit of batches. This is inspired by the fact that some serialization must be introduced in the execution path of commands anyway since the number of threads that execute commands is bounded (i.e., typically a multiple of the number of physical cores). Moreover, calculating the dependency among batches requires a fixed-size number of comparisons, determined by the number of worker threads.

- We propose high performance recovery techniques tailored to parallel state machine replication. These techniques allow speedy recovery of large logs and on-demand state recovery (i.e., checkpoint). The first technique aims at overlapping the execution of new commands with old commands, albeit respecting their dependencies. The second technique quickly brings needed state to a recovering replica and allows concurrent retrieval of checkpoint segments. On-demand recovery could be combined with existing optimizations described in the literature, such as collaborative state transfer [BSF+13]. Both speedy and on-demand state recovery aim to minimize replica downtime.

- We experimentally assess the performance of our recovery techniques using a full-fledged parallel state machine replication prototype and compare the performance of these techniques to traditional recovery mechanisms under different scenarios. In particular, we exercise the system in contention-free workloads (i.e., independent

commands only), to understand inherent limitations, and in contention-prone workloads (i.e., mix of independent and dependent commands).

1.2 Roadmap

The rest of this thesis is organized as follow. In Chapter 2 we present our system model and assumptions. In Chapter 3 we refer to the classic and parallel versions of state machine replication. A comparative analysis of checkpointing overhead on parallel versions of state machine replication appears in Chapter 4. In Chapter 5 we propose an efficient parallel state machine replication. A new recovery protocol for parallel state machine replication is the topic of Chapter 6. The thesis concludes in Chapter 7 that summarizes main findings and identifies areas for future research. Appendix A provides detailed information about the simulator designed to evaluate checkpointing techniques, and Appendix B presents the correctness discussion for algorithms proposed in this thesis.[1]

[1] We would like to thank Leila Ribeiro for the support and insightful ideas that helped formulate correctness proofs of Section B.4 in Appendix B.

2. SYSTEM MODEL

In this chapter, we present our system model. Except where explicitly mentioned, the rest of the text relies on the assumptions here stated.

2.1 Processes and Communication

We assume a distributed system composed of interconnected processes $\Pi = \{p_1, p_2, ...\}$. More precisely, 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 server processes. Server processes in S behave as active replicas by implementing the state machine replication approach. For this reason, we use the terms *server process* and *replica* with the same meaning.

We assume the *crash-recovery* failure model and exclude malicious and arbitrary behavior (e.g., no Byzantine failures). 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 may crash and recover, although they are not obligated to recover once they failed.

Processes do not have access to shared memory. They communicate uniquely by message passing, using either one-to-one or one-to-many communication. One-to-one communication is through primitives send(m) and receive(m), where m is a message. Messages can be lost but not corrupted. If a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication is based on atomic multicast or atomic broadcast primitives, both defined next using the abstraction of the consensus problem [CT96].

Protocols proposed in this thesis ensure safety under both asynchronous and synchronous execution periods. However, since the FLP impossibility result proved that consensus cannot be solved deterministically in asynchronous systems where at least one process may crash [FLP85], we assume that the system is partially synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time* (GST) [DLS88] and it is unknown to the processes. Before GST, there are no bounds on process execution speed or message delivery delays. After GST, such bounds exist but they are unknown. In order to prove liveness, we assume that there is a time (e.g., after GST) when a quorum of processes is correct. A process is *correct* if it eventually remains *up*. Otherwise, the process is *faulty*. We assume f faulty servers, out of $n = 2f + 1$ servers.

2.2 Multicast and Broadcast

One-to-many communication adopts the consensus abstraction [CT96] to implement atomic multicast or atomic broadcast. 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 (*non-triviality*);

ii. a process can decide at most one value (i.e., a learner cannot change its mind about what value it has learned) (*stability*);

iii. two different processes cannot decide different values (*consistency*).

iv. provided that some of consensus participants are non-faulty (e.g., $f + 1$), some proposed value is eventually decided (*progress*).

The consensus abstraction is suitable to discuss crash-recovery because although a faulty process possibly has not participated on some consensus instances, it is allowed to participate on newer instances once it has recovered (becomes up).

The atomic broadcast protocol ensures the following safety properties:

i. a process delivers a message m only if m was previously broadcast by some process;

ii. if both processes p and q deliver messages m and m', then p delivers m before m', if and only if q delivers m before m'.

While the first property is derived directly from consensus properties, the second one results from the total order notion expressed by the execution of successive instances of consensus. Depending on the failure pattern, a set of processes would crash and recover infinitely many times and, as a consequence, processes would not make progress. Therefore, in order to provide liveness, we assume that eventually a set of *up* processes remains up long enough to reach consensus.

We define atomic broadcast by the primitives *broadcast*(m) and *deliver*(i, m), where i refers to the consensus instance in which message m was decided upon. This definition implicitly assumes that atomic broadcast is implemented with a sequence of consensus instances identified by natural numbers $1, 2, 3, ...$ (e.g., [CT96, Lam98]). This definition of atomic broadcast differs from more common definitions (e.g., [DSU04]), but allows a recovering server to retrieve messages that the server delivered before the failure. By introducing the consensus instance in the delivery event, a server can easily determine the messages it needs to retrieve upon recovering from a failure.

Atomic multicast is defined by the primitives *multicast*(γ, *m*) and *deliver*(*m*), where γ is a group of destinations. Atomic multicast ensures that:

i. a process that belongs to γ delivers a message *m* only if *m* was previously multicast by some process from γ;

ii. if both processes *p* and *q* deliver messages *m* and *m′*, then *p* delivers *m* before *m′*, if and only if *q* delivers *m* before *m′*.

It is worth noting that both atomic broadcast and atomic multicast do not obligate all replicas to deliver every broadcast message. Once there is a quorum of up replicas participating in consensus involving message *m*, they may decide to deliver *m* despite some other replicas are down.

2.3 Consistency

Our consistency criterion is linearizability [HW90]. A system is linearizable if there is a way to reorder the client commands in a sequence that [AW04]:

i. respects the semantics of the commands, as defined in their sequential specifications;

ii. respects the real-time ordering of commands across all clients.

Linearizability can be achieved by ensuring that each replica executes commands in the same order, for instance. That is the case of the traditional state machine replication approach. As we discuss in Appendix B, the parallel state machine replication approach adopted in this thesis also ensures linearizability.

3. BACKGROUND

This chapter introduces the state machine replication (SMR) approach and its concurrent variations. We start by presenting the traditional SMR approach and its characteristics. Finally, we discuss how recent work has increased throughput of SMR by allowing execution of independent commands in parallel.[1]

3.1 State Machine Replication

State machine replication is a general approach to implementing fault-tolerant services by replicating servers and coordinating client interactions with server replicas [Lam78, Sch90]. The service is defined by a state machine and consists of *state variables* that encode the state machine's state and a set of *commands* that change the state (i.e., the input). The execution of a command may (i) read state variables, (ii) modify state variables, and (iii) produce a response for the command (i.e., the output).

SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas. This last aspect is captured by *linearizability* [AW04, HW90]. In classical SMR, linearizability can be achieved by having clients atomically broadcast commands and replicas execute commands sequentially in the same order. In order to ensure that the execution of a command will result in the same state changes and results at different replicas, commands must be *deterministic*: the changes to the state and response of a command are a function of the state variables the command reads and the command itself. Therefore, if servers start from the same state and execute commands in the same order, they will produce the same state changes and results after the execution of each command.

Figure 3.1(b) illustrates a classical SMR, where commands issued by clients are handled by the client proxy, which broadcasts the commands to all replicas and waits for the response from one replica. Client proxies translate client invocations into requests that include a command identifier and its parameters. Requests are delivered by the server proxies, which issue them against the local service. The agreement layer is responsible for ordering requests in the same total order across replicas (e.g., using the Paxos protocol [Lam98]).

[1] Part of the contribution of this chapter appears in [MMDP14, MDP16].

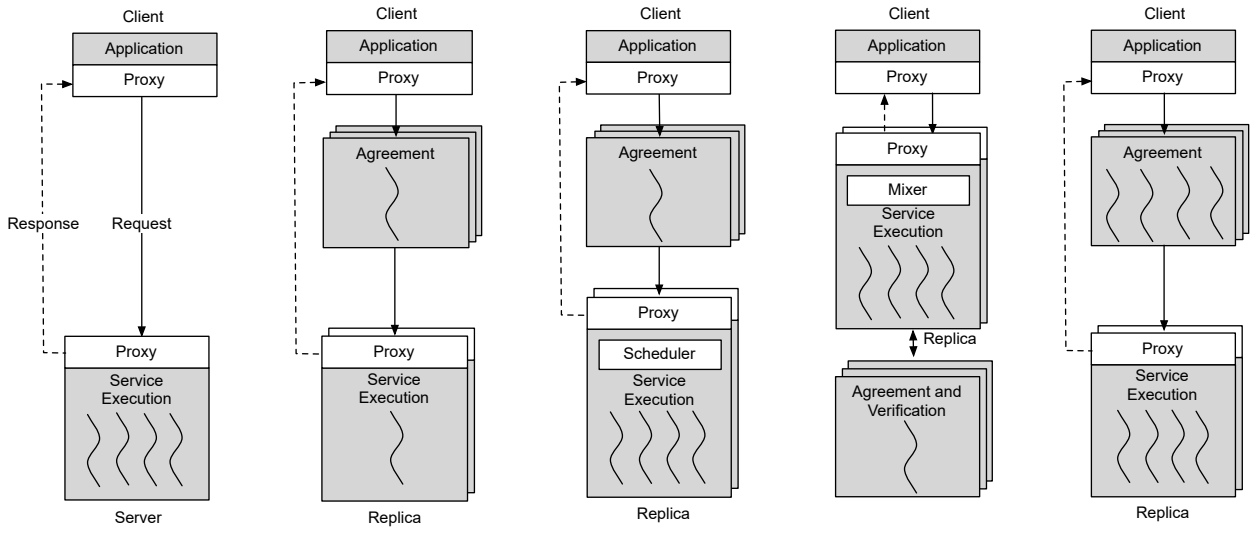


Figure 3.1 – Classical versus parallel state machine replication

3.2 Parallel State Machine Replication

Classical SMR makes poor use of multi-processor architectures since deterministic execution normally translates into (single-processor) sequential execution of commands. Although (multi-processor) concurrent command execution may result in non-determinism, it has been observed that “independent commands” (i.e., those that are neither directly nor indirectly dependent) can be executed concurrently without violating consistency [Sch90].

A few approaches have been suggested in the literature to execute independent commands concurrently with the goal of improving SMR performance (e.g., [KD04, KWQ+12, MBP14]). In this section, we describe these proposals. One of these approaches, CBASE [KD04], motivated us to implement an efficient parallel state machine replication protocol described in Chapter 5.

3.2.1 CBASE

To parallelize the execution of independent commands, CBASE [KD04] adds a deterministic scheduler, also known as parallelizer, to each replica (see Figure 3.1(c)). Clients atomically broadcast commands and the parallelizer at each replica delivers commands in total order, examines command dependencies, and distributes them among a pool of worker threads for execution.

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 [KD04], C_i and C_j are *dependent* if any of the following conditions hold: $W_i \cap W_j \neq \emptyset$; $W_i \cap R_j \neq \emptyset$; or $R_i \cap W_j \neq \emptyset$. 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.

The parallelizer uses a dependency graph to maintain a partial order across all pending commands, where vertices represent commands and directed edges represent dependencies. While dependent commands are ordered according to their delivery order, independent commands are not directly connected in the graph.[2]

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

Worker threads receive independent commands from the parallelizer (i.e., vertices with no incoming edges) to be concurrently executed. When a worker thread completes the execution of a command, it removes the command from the graph and responds to the client that submitted the command.

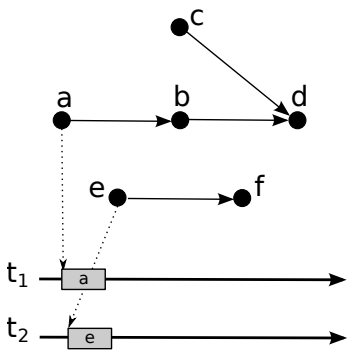
Figure 3.2 shows an illustrative dependency graph created by the parallelizer. In this example, six commands are delivered in the order $a, b, ..., f$. Commands a, c and e are independent, so they can be scheduled to execute concurrently. Threads t_1 and t_2 start processing independent commands a and b in parallel (Figure 3.2 (a)). Next, t_1 and t_2 process commands b and f (Figure 3.2 (b)). Command c is executed by thread t_1 while t_2 is waiting for an independent command to be available (Figure 3.2 (c)). Notice that d cannot be scheduled while c has not been completed (c and d are dependent but c was delivered first, so, c must execute before d). Finally, d becomes the only command pending execution and is scheduled to thread t_2 (Figure 3.2 (d)).

3.2.2 The Execution-Verify approach (Eve)

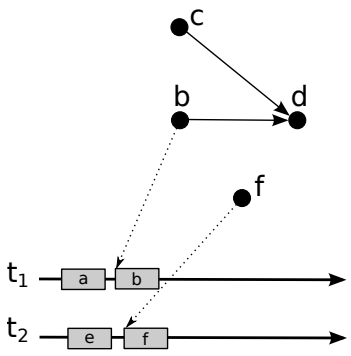
Another approach that allows SMR to scale to multi-core servers is Eve (Execution–Verify) [KWQ+12]. Different from other SMR approaches, Eve replicas first execute commands and then verify the equality of their states through a verification stage (see Figure 3.1(d)).

In Eve, clients send their requests to a primary execution replica. Before execution, the 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 replicas' state, as

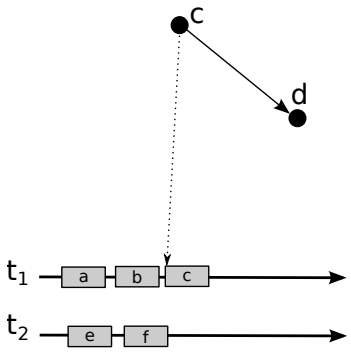
[2]Notice that dependent commands may induce an order on independent commands in the dependency graph. For example, if C_1 and C_2 are independent but C_1 and C_3, and C_2 and C_3 are dependent.



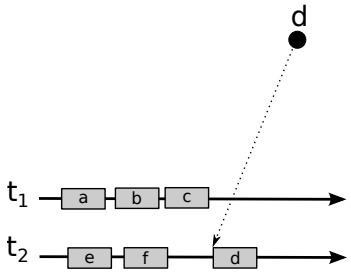
(a) Independent commands *a* and *e* are scheduled and executed by threads t_1 and t_2, respectively.



(b) Independent commands *d* and *f* are scheduled and executed by threads t_1 and t_2, respectively.



(c) Independent commands *c* are scheduled and executed by thread t_1.



(d) Independent commands *d* are scheduled and executed by thread t_2.

Figure 3.2 – CBASE scheduling example.

defined by the common state reached by a majority of replicas. If a few replicas diverge in their state and output, a correct state agreed by a majority of replicas is transferred and installed on those divergent replicas. If too many replicas diverge, replicas roll back to the last verified state and re-execute the commands sequentially and deterministically. After reaching a verified state, the execution replicas send the responses for that batch to the clients.

Figure 3.3 represents the “Execute–verify” design. Replicas r_1 to r_n receive the same batches in the same order (illustrated by batches *i* to *k*). Every replica executes a batch of commands (execution stage) and then verify their state validity (verification stage). While batch execution does not require replica synchronization, the verification stage runs an agreement protocol to determine the final state and outputs of all correct replicas after each batch of requests.

Figure 3.4 shows an illustrative sequence of batches being processed by a replica *r*. In this example, commands *a*, *b*, and *c* belong to batch *B*1, and *d*, *e*, and *f* belong to batch *B*2. Replica’s threads, t_1 and t_2, start processing commands in *B*1 in parallel (the order of

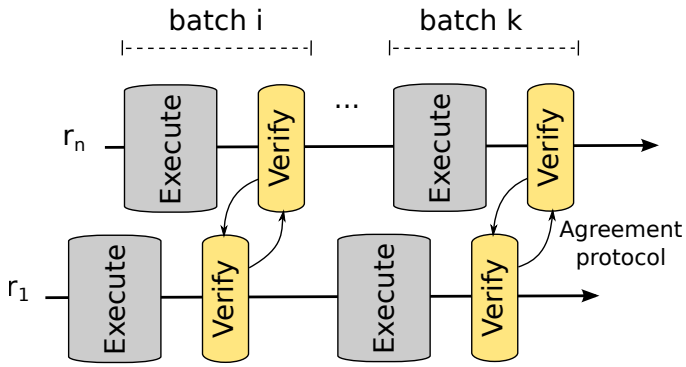
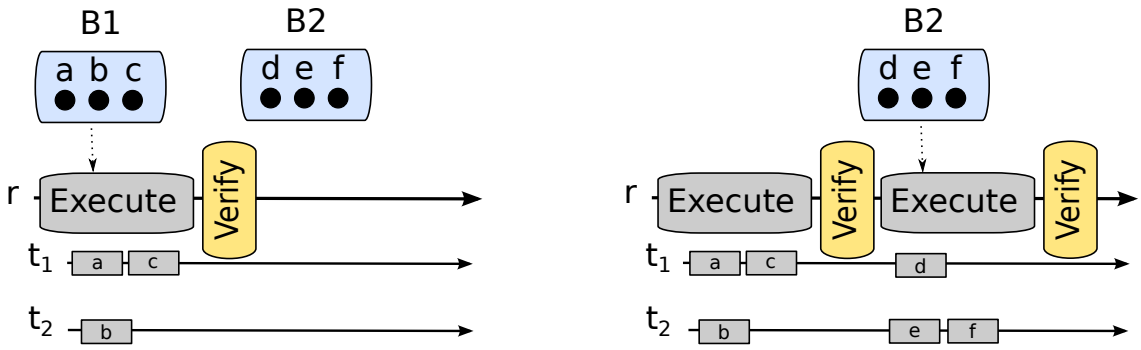


Figure 3.3 – “Execute-then-verify” design.

execution may diverge across replicas). After processing $B1$, r verifies the validity of replica’s state. This procedure is repeated for the execution of each successive batch. Figure 3.4(b) shows the execution of batch $B2$.



(a) Batch $B1$ is scheduled by replica r and executed by threads t_1 and t_2.

(b) Batch $B2$ is scheduled by replica r and executed by threads t_1 and t_2.

Figure 3.4 – Eve execution example.

In order to avoid costly rollbacks, the frequency in which replicas need to reconcile must be reduced. 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 [KWQ+12] (e.g., independent commands). If conflicting commands a and b both modify object x, the mixer will place them in different batches. The mixer and the parallelizer

presented in [KD04] play a similar role. However, even if the mixer may unintentionally include dependent commands in the same batch and cause replicas to produce different states, the verification stage ensures that such divergences cannot affect safety (although they may hamper performance).

3.2.3 P-SMR

In [MBP14], the authors propose P-SMR, a variation of parallel state machine replication where the execution and the delivery of commands occur in parallel. Instead of using a single sequence of consensus rounds to order commands, the approach in [MBP14] uses multiple sequences of consensus (see Figure 3.1(e)). 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 [MPP12]. 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 3.1, extracted from [MMDP14], adapts the original P-SMR algorithm proposed in [MBP14]. 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 chapter, to create a checkpoint. Variable *next*[*i*] (line 5) determines whether t_i is in *sequential* or *concurrent* mode.

Algorithm 3.1 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}
```

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.

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).

The correctness of Algorithm 3.1 is discussed in detail in Appendix B.

4. CHECKPOINTING IN PARALLEL STATE MACHINE REPLICATION

In order to improve fault-tolerance, the SMR approach can be enhanced by supporting recovery. Recovery protocols ensure durability of the data managed by SMR services. In case of replica failures, the recovery service enables a recovering replica to re-establish a consistent state and catch up with the other replicas [BM93, EAWJ02].

Recovery can be achieved in many ways. In this work, we focus on *transparent* techniques, i.e., the recovery happens without intervention of application or programmer. 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 parallel SMR is more challenging than in classic SMR because the checkpoint operation must account for the execution of concurrent commands. Although checkpoints are expected to reduce the throughput of replicas, it is not clear how they impact the existent parallel SMR protocols. In this chapter, we describe checkpointing strategies for CBASE and Eve, and propose two checkpointing approaches for P-SMR. Moreover, we discuss how checkpointing affects the performance of these models, and assess by means of simulations the impact of checkpointing on performance.

4.1 Checkpointing in CBASE

To recapitulate, in [KD04] 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 and ensures that their execution order adheres to the delivery order. Independent commands are dispatched to be processed in parallel by a set of worker threads.

The parallelizer approach built for CBASE relies on the totally ordered delivery of commands, that is also used to generate checkpoints. After an agreed number k of commands are delivered at a replica, the parallelizer stops assigning commands to worker threads. Once all commands in execution are finished, the replica takes a checkpoint and then resumes normal execution [KD04]. Replicas will produce identical checkpoints since any two replicas take checkpoints at fixed and deterministic intervals (e.g., whenever the k-th command is executed since the last checkpoint).

Besides the costs inherent to the checkpointing itself, such as state serialization and maintenance of checkpoint structures, checkpoints in the parallelizer approach induce additional overhead caused by thread synchronization. This overhead originates from the fact that at the moment a checkpoint is invoked, new incoming commands cannot be executed and, as a consequence, worker threads may be idle until all threads finish their work and the checkpoint is taken.

4.2 Checkpointing in Eve

Even though checkpointing and recovery are not explicitly described in [KWQ+12], 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 checkpointing strategy is similar to that presented in Section 4.1, but instead of taking checkpoints after every *k* commands have been processed, in Eve checkpoints are taken after commands in *n* batches have been executed and the effects of these commands on the replica's state verified. Although there exists a synchronization cost associated to checkpoints in Eve, this cost is part of the verification stage, i.e., right after executing batched commands, worker threads may sit idle until the verification stage terminates.

4.3 Checkpointing in P-SMR

Checkpointing solutions for CBASE [KWQ+12] and Eve [KD04] rely on a single stream of totally ordered messages being delivered to the replicas. However, in P-SMR [MBP14] not only the execution, but also the delivery of messages occurs in parallel. Therefore, solutions analogous to the ones described previously are not possible in P-SMR.

We propose next two novel checkpointing algorithms for P-SMR.[1] 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. The correctness discussion for both algorithms appears in Appendix B.

[1] These results also appear in [MMDP14].

4.3.1 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 4.1 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 3.1 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 3.1 and 4.1.

4.3.2 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

Algorithm 4.1 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}
```

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 4.2 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 $\langle 2, \mathrm{SQ}[, 1, \mathrm{CC}]_{\times n}\rangle$.) 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.

4.3.3 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* [BSF+13], 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

Algorithm 4.2 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!}
```

some latency, a checkpoint request in the coordinated technique can only be handled after previously decided commands are executed at the replicas.

4.3.4 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 a prototype, which implements an in-memory database with operations to read and write database entries. In our experiments, we generate 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.

Simulations

We implemented a discrete-event simulation model[2] in C++ and configured each experiment to run until a confidence interval of 98% is reached. We evaluated replicas without checkpointing enabled and with the two proposed checkpoint algorithms, and considered different classes of workload in terms of request 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 4.1 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), mentioned 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.

The bottom right graph of Figure 4.1 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

[2]More details about our simulation tool are discussed in Appendix A.

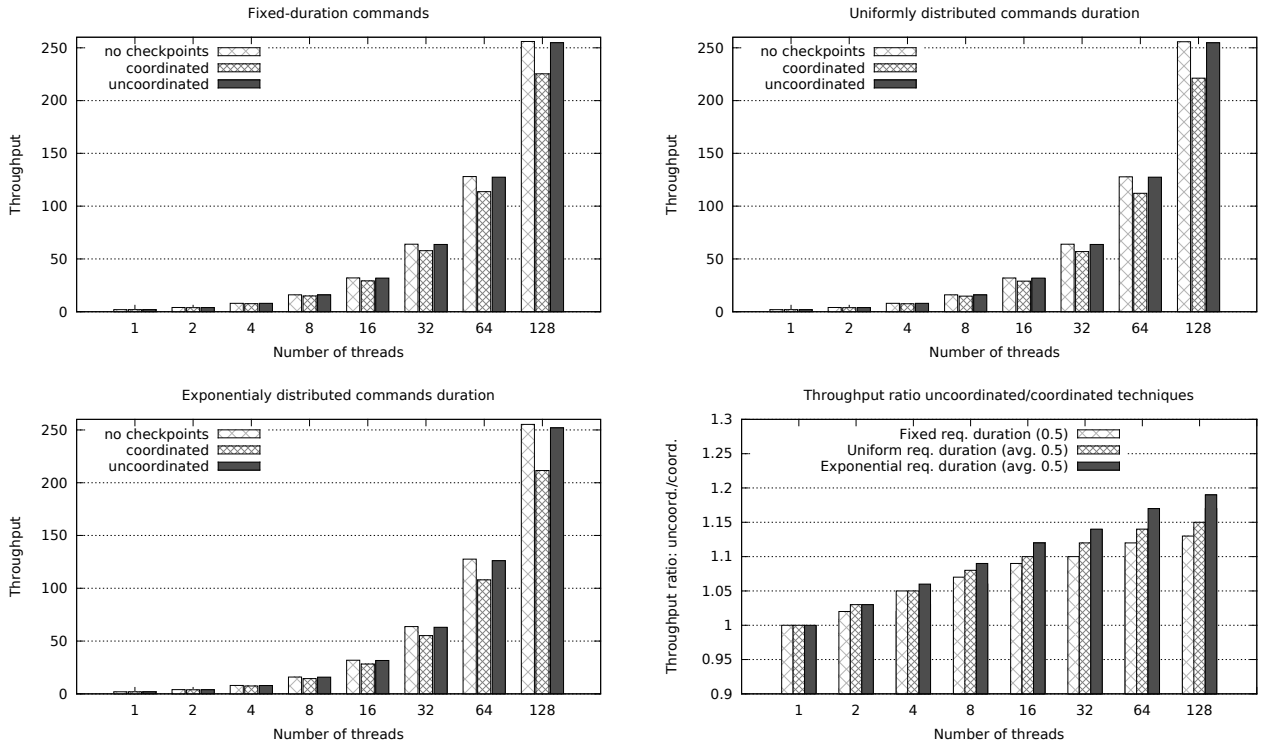


Figure 4.1 – Throughput of a replica with the number of threads for different commands execution duration workloads and the ratio of the two techniques with the number of threads.

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 4.2 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 most of the configurations, the difference between the two decreases as checkpoints become more infrequent.

Figure 4.3 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.

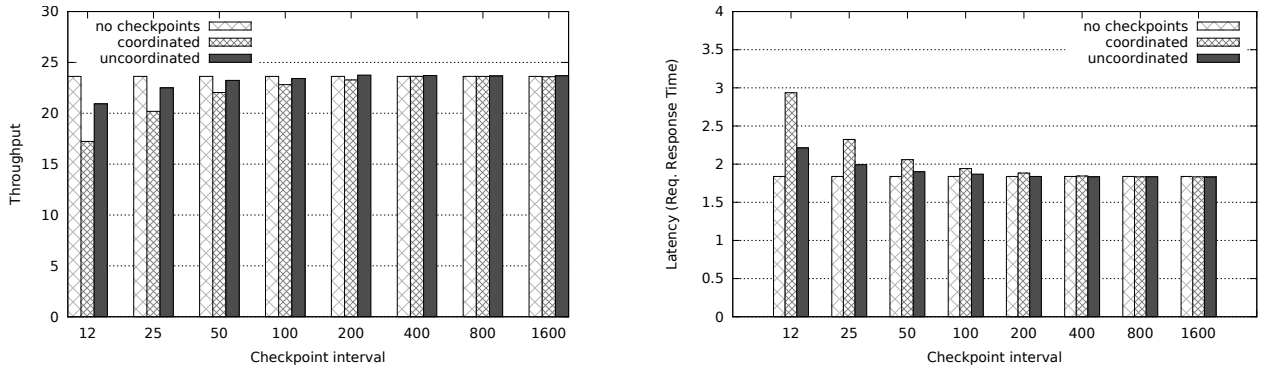


Figure 4.2 – Throughput and latency of a replica executing commands with an exponentially distributed execution time (average of 0.5 time units).

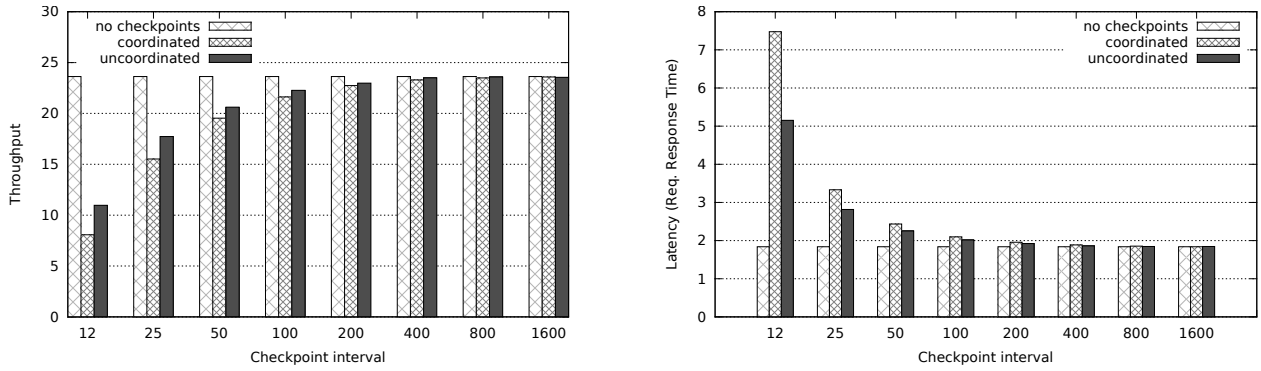


Figure 4.3 – Throughput and latency of a replica executing commands with an exponentially distributed execution time (average of 0.5 time units) and checkpoint duration of 5 time units.

Implementation

We implemented consensus using Multi-Ring Paxos [MPP12], where each consensus sequence is mapped to one Paxos instance. To achieve high performance, each 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 4.4 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.

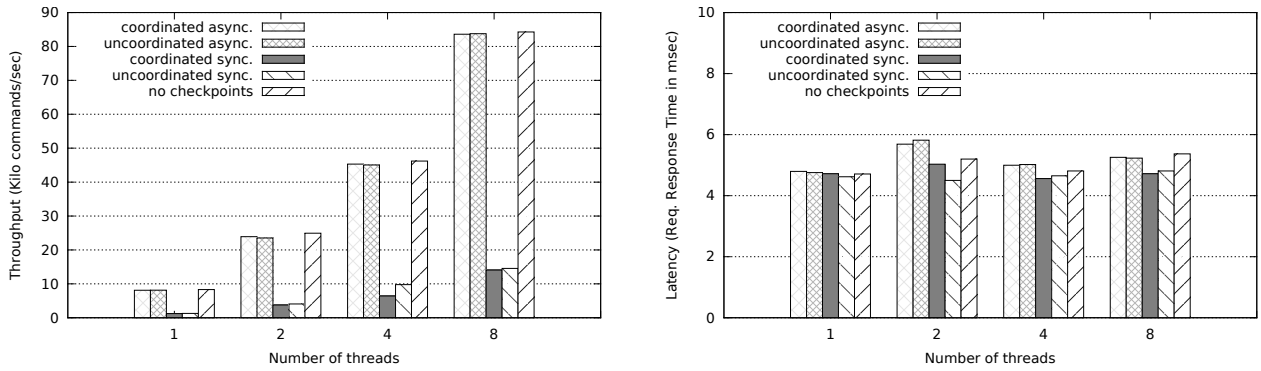


Figure 4.4 – Throughput and response time of coordinated and uncoordinated checkpointing with asynchronous and synchronous disk writes.

4.4 Performance Evaluation of Checkpointing for Parallel SMR Models

In this section, we evaluate the impact of checkpointing in every parallel approaches to SMR discussed in this chapter. As observed, checkpoints introduce an overhead in the normal execution of parallel SMR approaches. Our analysis aims to quantify the cost of synchronization due to checkpoints through simulation.

Setup

We implemented a discrete-event simulation model in C++ and configured each experiment to run until a confidence interval of 98% is reached. We built simulation models for CBASE [KD04], Eve [KWQ+12], and P-SMR [MBP14]. The classical SMR model is a special case of parallel SMR where just one thread executes commands. The behavior of classical SMR corresponds to executions of the CBASE and P-SMR with a single executing thread.[3]

[3]This holds for the CBASE since we do not consider the overhead of the scheduler in our simulations.

We ran simulations with and without checkpointing enabled, 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 following, we report results using an exponentially distributed command duration only. Our conclusions about the relative performance of each technique also apply to experiments using the other distributions.

Analogously to the previous experiments, we generate 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 caused by dependent commands. In doing so, our analysis focuses on the impact of synchronization due to checkpointing.

The effects of the number of threads

Figure 4.5 exhibits the maximum normalized throughput achieved by replicas as the number of threads varies. The normalized throughput 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, as stated in Equation 4.1.

$$norm\_tput(x) = measured\_tput(x)/(norm\_tput(1) \times x) \tag{4.1}$$

In the graphs, command durations follow an exponential distribution with an average command execution time of 0.5. For all three techniques, we show performance of executions without and with checkpoints, in which case checkpoints are taken every 400 commands. The throughput values measured for each technique are presented in Table 4.1.

The graph on the top of Figure 4.5 considers executions with “instantaneous” checkpoints (i.e., no execution time) and the graph on the bottom of the figure shows results when checkpoints take 5 time units. Note that differently from executions without checkpoints, instantaneous checkpoints require threads to synchronize.

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 4.5 (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 CBASE and P-SMR scale proportionally to the number of threads (not seen in the graphs due to normalization). Eve presents lower throughput than the other techniques due to its verification stage, which periodically forces thread synchronization. When checkpoints

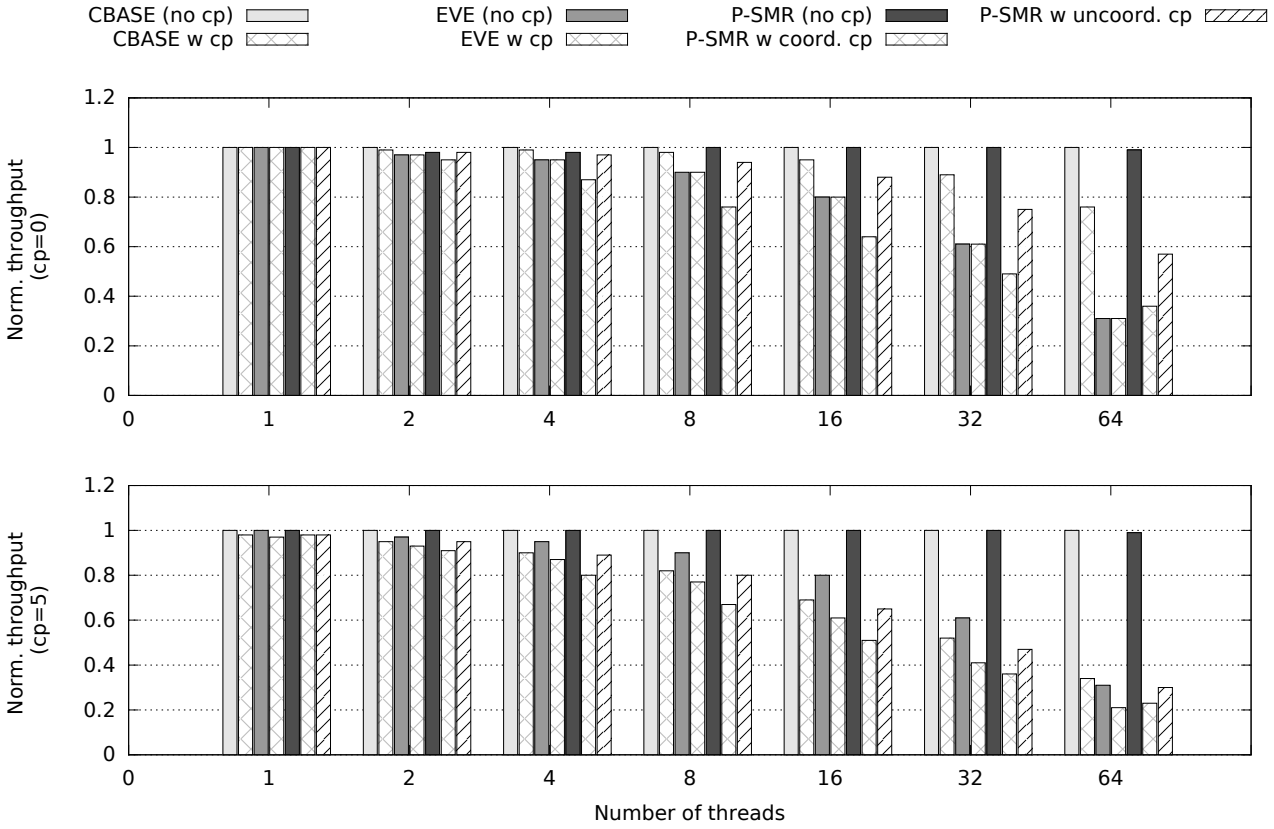


Figure 4.5 – Maximum normalized throughput with instantaneous checkpoints (top) and 5-time unit checkpoints (bottom). Checkpoint interval of 400 commands and command execution duration exponentially distributed with average 0.5.

Table 4.1 – Maximum throughput with instantaneous checkpoints

threads	**CBASE**		**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

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 CBASE is smaller than in P-SMR uncoordinated because the scheduler in the CBASE can perform a more efficient scheduling of commands than P-SMR, where clients must decide which thread should execute a command when the command is broadcast. P-SMR is more advantageous

than CBASE in executions where the scheduler becomes the bottleneck, and the technique cannot scale with additional threads [MBP14].

Figure 4.5 (bottom) depicts the maximum throughput for scenarios in which checkpoints take 5 time units. As observed before with instantaneous checkpoints, performance degrades with the number of threads. In CBASE, 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% with command and checkpoint execution, respectively. This is a consequence of the fact that checkpoints happen more often with CBASE 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.

The effects of the checkpoint frequency

The following experiments evaluate the impact caused by the frequency in which checkpoints are taken by replicas in configurations with 16 cores. Figure 4.6 shows the throughput and latency for workloads where request duration follows an exponential distribution with average 0.5. We configured the workload in this experiment to reach 75% of the maximum throughput reachable by each model. Checkpoint interval varies from 400 to 6400 requests and checkpoints take 5 time units.

Checkpoints have an impact on the throughput and response time of replicas (top and bottom of Figure 4.6, respectively), although the overhead caused by checkpoints decreases as checkpoints become more infrequent, independently of the parallel SMR approach under analysis.

From the results, CBASE and uncoordinated P-SMR present the higher throughput rates. Regarding latency, CBASE outperforms P-SMR, presenting lower response time. This happens due to thread scheduling. While in the CBASE approach incoming commands can be dispatched to any free worker thread, in P-SMR clients send commands directly to a given thread. This way, a command may end up enqueued if the thread assigned to execute the command is busy executing another command, even if some other thread is available. Although CBASE and Eve schedule commands more efficiently within replicas, they are subject to a single point of contention, the scheduler in CBASE 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, with its performance consequences. In our simulation we do not associate scheduling costs to any of the models.

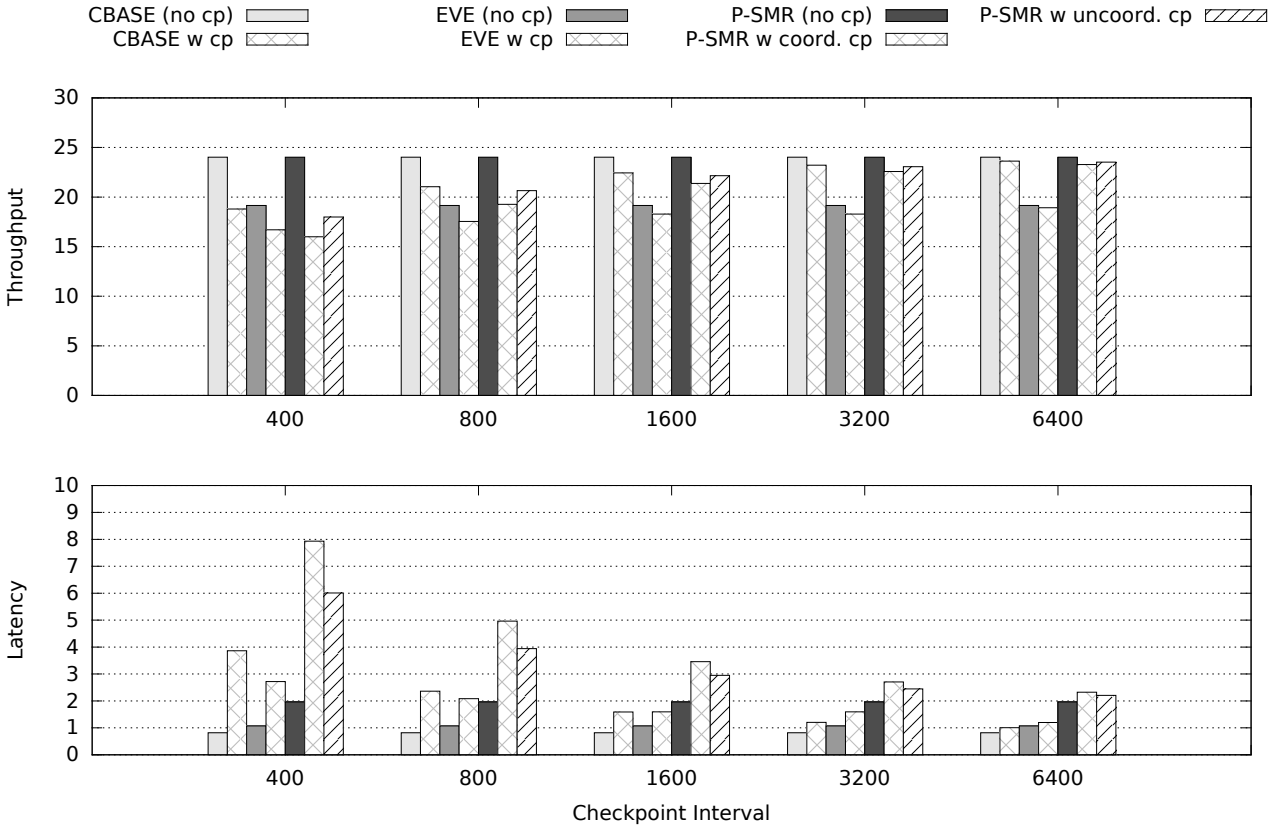


Figure 4.6 – Throughput (top) and latency (bottom) of various techniques. Replicas configured with 16 threads, commands execution exponentially distributed with average commands of 0.5 and checkpoint duration of 5 time units.

Summary

In the following, we summarize our findings.

- 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 CBASE 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, CBASE 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.

In this section, we reviewed and evaluated the performance of checkpointing in existing parallel state machine replication approaches. Our analysis focused mostly on the checkpointing synchronization overhead. The proposed simulation models capture the inherent differences among existing approaches and allowed us to measure the way checkpoints affect performance in each case.

4.5 Related Work

While there is a rich body of research on the algorithmic aspects and implementation of SMR addressing *crash faults* [Bur06, CWO+11, CGR07, CDE+12, HKJR10, RST11], *Byzantine faults* [CL99, KAD+07, CKL+09, CWA+09, KBC+12], reports on checkpointing and recovery procedures for SMR are relatively scarce. In a recent work, Bessani *et al.* [BSF+13] reviewed the literature and pointed out weaknesses of the common durability techniques (logging, checkpointing and state transfer) applied to the SMR model. Further, in [AK08, CGR07] authors discuss challenges and performance limitations of checkpointing in practical SMR implementations. With regard to checkpointing to parallel state machine replication, the only works addressing the topic are [MMDP14, MDP16] (the contribution of both works is essentially the subject of this chapter).

Our research focuses in the challenges involved to correctly take checkpoints despite the concurrency inherent to parallel state machine replication models. We proposed checkpointing mechanisms and evaluated the impact on performance caused by extra synchronization required by these techniques. A complementary discussion about checkpointing concerns the way as replica's state is converted into a checkpoint image, and how a replica state is restored from a checkpoint. In this sense, some works propose optimizations for logging, checkpointing and state transferring.

General optimizations on durability techniques can minimize checkpointing and recovery overhead, improving the overall system's performance. For instance, instead of logging single operations, some works log batches of operations [BSF+13, CWA+09, CL99, SFK+09, KAD+07]. This strategy explores the fact that disks are block-devices, so it is preferable to write a batch of operations in the same block instead of writing multiple operations in several blocks. Another logging improvement proposed in [BSF+13] is the processing and logging of batches in parallel. This technique benefits mainly those applications in which the time of processing batches is equal to or higher than the logging time. In [RST11], the authors demonstrate that the logging overhead can be dramatically reduced by the use of solid-state disks (SSD) instead of magnetic disks.

The generation of checkpoints may degrade the performance of the service. Taking a snapshot after processing a certain number of requests, as proposed in most works in SMR (*e.g.* [CL99, Lam98, SFK+09, KAD+07, RST11]), can delay the system momentarily. This happens because requests are no longer processed while replicas save their state. Even when replicas are not fully synchronized, delays may also occur because the necessary agreement quorum might not be available. In [BSF+13] the authors force replicas to take checkpoints at different times. Since the system makes progress as long as a quorum of $n - f$ replicas is available,[4] there are f spare replicas in fault-free executions. Thus, $n - f$ replicas can continue processing client requests and up to f replicas could be taking state snapshots. The use of a helper process for taking checkpoints asynchronously is proposed in [CKL+09]. Two similar threads, the primary and the helper, execute in parallel. While the primary continuously processes requests and sends replies to the clients, the helper periodically take checkpoints. The helper thread pauses the incoming requests so that it is quiescent while it is producing a checkpoint.

During a state transfer, at least one replica has to spend resources to send its own state to another replica. Extra delays can occur due to the transmission of the state through the network and because of the disk accesses. One way to avoid performance degradation is to ignore state transfer requests until the workload is low enough to process both the state transfer and regular messages [HKJR10]. Other way to minimize this overhead is through the reduction of the amount of information transferred. State can be efficiently represented by data structures based on hierarchical state partitions, as proposed in [CL99], incremental checkpoints[CKL+09, CRL03], or compression techniques. In [BSF+13], the authors present a collaborative state transfer protocol, so the burden imposed on replicas is evenly distributed among them.

A comprehensive survey covering checkpointing and recovery techniques for general message-passing applications is presented in [EAWJ02]. Although some concepts and terminology discussed in [EAWJ02] are common to this work, authors do not address state machine replication directly.

[4] n is the number of all replicas in the system and f is the number of replicas allowed to crash.

5. EFFICIENT PARALLEL STATE MACHINE REPLICATION

In this chapter, we introduce a protocol to efficiently handle command dependencies and schedule independent commands to execute concurrently. The main goal of this protocol is to reduce the overhead needed to keep tracking of dependencies and assign commands to worker threads. Our design is motivated not only to speed up the execution in the absence of failures, but also to boost the recovery of replicas (detailed in Chapter 6).

Before diving into details of our protocol, we first describe the basic operation of CBASE scheduling [KD04]. In CBASE, the parallelizer uses a directed acyclic graph to form a *dependency graph*. The dependency graph tracks dependencies among all pending commands, where vertices represent commands and directed edges represent dependencies. Independent commands do not need to be connected in the graph. Dependent commands are ordered according to their delivery order (i.e., edges connecting dependent commands in the dependency graph). Worker threads receive independent commands from the parallelizer (i.e., vertices with no incoming edges) to be concurrently executed.

In CBASE approach, the number of disconnected components in the dependency graph determines how many commands can be executed concurrently at any given time. Figure 5.1 (a) depicts an illustrative dependency graph with six commands, delivered in the order $a, b, ..., f$. Commands a, c and e are the next ones to be scheduled for execution and can execute concurrently. Commands a and b are dependent but a was delivered first, so, a must execute before b. Intuitively, fewer interdependencies between commands in the dependency graph favor concurrency. However, the cost of adding a new command in the dependency graph is proportional to the number of commands in the graph that are independent of the new command. For example, a new command g will be first compared to commands d and f; if g is independent of d, it will be compared to c and b, and so on. If g is independent of every command in the graph, it will be compared against all vertices.

As illustrated by this example, the dependency analysis cost increases with the dependency graph size. In order to reduce the scheduling cost, our proposed scheme penalizes concurrency in the execution of independent commands for reduced overhead when tracking command dependencies.

5.1 Overall idea

In summary, our scheme combines the following strategies:

- The scheduler assigns batch of commands to the worker threads, as opposed to individual commands.[1]

[1] Batching to SMR was originally proposed by [FVR97].

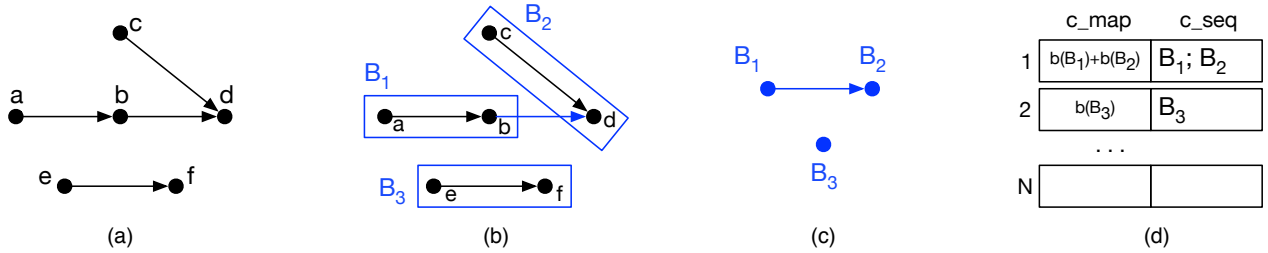


Figure 5.1 – Four representations of a dependency graph with six commands. (a) The original dependency graph, where edge $x \rightarrow y$ means that commands x and y are dependent and x was delivered before y. (b) The original graph grouped in batches of two commands. (c) The abridged dependency graph; notice that commands b and c are serialized in the abridged graph. (d) The stored dependency graph; batches B_1 and B_2 are assigned to worker thread t_1 and batch B_3 is assigned to worker thread t_2; $b(B)$ is a digest of the variables accessed by commands in B, used to track dependencies between command batches. The stored dependency graph preserves all dependencies defined in the original dependency graph.

- The dependency graph is stored as sequences of command batches, with one sequence per worker thread.
- The scheduler minimizes synchronization with worker threads when scheduling command batches.

Batched commands. Clients submit commands through a client proxy, which groups commands from different clients and broadcasts the commands for execution as batches. When the proxy receives responses for all commands in a batch, it can submit a new batch of commands. There can be any number of client proxies, each one handling a group of clients.

The abridged dependency graph. The parallelizer delivers batches of commands and builds an abridged dependency graph, where vertices are command batches and edges are dependencies induced by the commands in the batches. More precisely, there is an edge from batch B_i to B_j in the graph if B_i is delivered before B_j and B_j contains a command that depends on a command in B_i (see Figure 5.1 (b) and (c)). Moreover, each batch of commands contains a bitmap with a digest of the variables read and written by the commands in the batch. The idea is that given two bitmaps, $b(B_i)$ and $b(B_j)$, we want to be able to determine whether there is a command in batch B_i that depends on some command in batch B_j by comparing their bitmaps. The way bitmaps are encoded to satisfy this property is application dependent and can be achieved in different ways. In our prototype, we consider write commands in a database, where each operation includes the key of the entry written in the database. Therefore, we create bitmaps by hashing the key provided in the command; the hashed value corresponds to a bit set in the bitmap. Checking whether two batches contain dependent commands boils down to a bit-wise comparison of their bitmaps.

The overhead versus concurrency tradeoff. The abridged dependency graph establishes a tradeoff. On the one hand, batching reduces the overhead needed to handle commands (e.g., fewer system calls to deliver commands, fewer edges to store the graph, fewer comparisons to determine dependencies), which improves performance. On the other hand, the abridged dependency graph reduces concurrency since it may induce dependencies among independent commands. In Figure 5.1(b), independent commands *a* and *c* are serialized since commands in B_1 must execute before commands in B_2 because *b* is in B_1, *d* is in B_2 and *b* precedes *d*.

The stored dependency graph. We now describe how the abridged dependency graph is stored at each replica. It introduces an optimization that allows new command batches to be added to the graph in $O(tb)$, where t is the number of worker threads, and b is the bitmap size. The parallelizer and the worker threads share two data structures, each one with one entry per worker thread (see Figure 5.1 (d)). For each worker thread t_i, $c\_seq[i]$ contains the sequence of command batches the parallelizer assigned to t_i and $c\_map[i]$ contains a bitmap that encodes all commands in $c\_seq[i]$. When the parallelizer delivers a new batch B, it checks B's bitmap, $b(B)$, against each $c\_map[i]$ to determine which worker threads need to coordinate in order for commands in B to be executed. If no interdependencies are detected, the parallelizer chooses one worker thread t_i (e.g., the least loaded one), assigns B to t_i by appending B to $c\_seq[i]$, and updates t_i's bitmap $b\_map[i]$. Otherwise, the parallelizer determines all worker threads that have been scheduled commands on which B depends, adds B to the end of c_seq of such threads and updates their bitmaps.

The execution of commands. Each worker thread t_i executes commands following their order in $c\_seq[i]$, where commands in the same batch are handled in the order they appear in the batch. When a worker reaches a command batch B that requires coordination among workers, all involved workers coordinate so that a single worker executes the commands in B. After the thread executes all commands in B, it signals the other worker threads involved to proceed. When thread t_i is finished with B, t_i removes B from $c\_seq[i]$ and recomputes $c\_map[i]$ based on the batches currently in $c\_seq[i]$. As a consequence of this procedure, the parallelizer and the worker threads must access structures c_seq and c_map in mutual exclusion.

Concurrent access of shared structures. Our final optimization aims to reduce the synchronization needed between the parallelizer and the worker threads to access c_map. In order to achieve this, the parallelizer maintains a "shadow copy" of c_map, s_map, which it uses to detect dependencies between a delivered batch B and the commands previously assigned to worker threads. Structure s_map is only accessed by the parallelizer. Each entry $s\_map[i]$ contains a superset of the commands in $c\_seq[i]$. This is because when worker

thread t_i executes a batch of commands, it updates *c_map*[*i*] but not *s_map*[*i*]. Therefore, the parallelizer may detect false positives when determining *B*'s dependencies and the worker threads that need to coordinate to execute commands in *B*. Once these worker threads are determined, for each such a thread t_i, the parallelizer requests exclusive access to *c_map*[*i*] and *c_seq*[*i*]. The parallelizer then appends *B* to *c_seq*[*i*], updates *c_map*[*i*] with *b*(*B*), and copies the value of *c_map*[*i*] into *s_map*[*i*].

5.2 Algorithm in detail

Algorithm 5.1 details the behavior of the parallelizer and the worker threads, respectively. Data structures *c_seq*[*i*] and *c_map*[*i*] are accessed in mutual exclusion by the parallelizer and worker thread t_i. When the parallelizer delivers a new batch of commands *B* (line 26), it compares *B*'s bitmap, *b*(*B*), against each entry in *s_map*, the shadow copy of *c_map* (lines 12–15). If *b*(*B*) and *s_map*[*i*] intersect (line 14), then worker thread t_i has been scheduled to execute a batch of commands that depend on commands in *B* and is added to set *dep_list* (line 15). If no dependencies are found (line 16), the parallelizer can choose any of the worker threads to execute commands in *B* (lines 17–18). If there are dependencies, they are all in *dep_list*, and one worker thread among the ones in *dep_list* will execute the commands in *B* (lines 19–20). For each worker thread t_i involved in the execution of *B* (line 21), the parallelizer appends $\langle dep\_list, B \rangle$ to t_i's sequence of command batches (line 22), updates *c_map*[*i*], t_i's bitmap (line 23), and stores a fresh copy of *c_map*[*i*] in *s_map*[*i*] (line 24).

Every worker thread t_{id} iterates through its list of assigned commands (line 30). For each entry $\langle dep\_list, B \rangle$ in t_{id}'s sequence *c_seq*[*id*] (line 31), t_{id} deterministically selects one worker thread *exec_t* to execute *B* (line 32). The selected thread executes *B* (line 36) and then signals the other threads in *dep_list* (lines 37–40), which simply wait for the signal (line 41). After handling *B*, every involved thread recomputes its bitmap with the command batches in *c_seq* (lines 42–44).

Why it works. Correctness is discussed in detail in the Appendix B. Here we highlight the main intuition. From the total delivery order of batches $<_B$ and knowing the dependency between commands in different batches it is possible to obtain a batch sequence dependency relation $\prec_B$, an irreflexive partial order which is transitive, antisymmetric and acyclic.

Replica consistency. This holds since (i) all batches are enqueued at some thread; (ii) queues of working threads are compatible with $<_B$, that is, the batches enqueued at a working thread appear in the queue coherently with $<_B$; (iii) when batches appear in more than one queue then they depend on previous batches in those queues, in which case the

Algorithm 5.1 Efficient Parallel SMR

```
1:  Main data structures:
2:      c_map[1..N] {bitmap vector, shared by parallelizer and worker threads, accessed in mutual exclusion}
3:      c_seq[1..N]  {vector of sequence of command batches, shared by parallelizer and worker threads,
        accessed in mutual exclusion}
4:      s_map[1..N]  {shadow copy of bitmap vector, accessed by parallelizer only}

5:  procedure Initialization()
6:      for i = 1..N do                                  {for each worker thread...}
7:          c_seq[i] ← ∅                                 {set command set for thread i}
8:          c_map[i] ← 0                                 {set bitmap for thread i}
9:          s_map[i] ← 0                                 {shadow of the bitmap}
10:     k ← 1

11: procedure schedule(B)
12:     dep_list ← ∅                                     {workers that depend on B}
13:     for i = 1..N do                                  {for each worker}
14:         if b(B) ∩ s_map[i] then                      {check dependencies and}
15:             dep_list ← dep_list ∪ {i}                {keep track of dependency}
16:     if dep_list = ∅ then                             {if no dependencies...}
17:         choose next_t in 1..N                        {select one worker}
18:         workers ← {next_t}                           {ditto!}
19:     else                                             {if there are dependencies...}
20:         workers ← dep_list                           {request workers to sync}
21:     for all i ∈ workers do                           {for each concerned worker}
22:         c_seq[i] ← c_seq[i] ⊕ ⟨dep_list, B⟩          {append B to sequence}
23:         c_map[i] ← c_map[i] ∨ b(B)                   {logic OR of bitmaps}
24:         s_map[i] ← c_map[i]                          {update shadow copy}

25: The parallelizer executes as follows:
26: when deliver(k, B)                                   {when deliver new batch of commands}
27:     schedule(B)
28:     k ← k + 1

29: Each worker thread t_id executes as follows:
30: when c_seq[id] ≠ ∅                                   {while there are commands to execute}
31:     ⟨dep_list, B⟩ ← remove first element in c_seq[id]
32:     exec_t ← smallest i in dep_list                  {worker to execute command}
33:     if id = exec_t then                              {if t_id was selected}
34:         for all i ∈ dep_list ∧ i ≠ id do             {involved workers...}
35:             wait for signal from t_i                 {...use barrier before command}
36:         execute commands in B                        {execute commands}
37:         for all i ∈ dep_list ∧ i ≠ id do             {involved workers...}
38:             signal t_i                               {...use barrier after command}
39:     else                                             {else, if t_id wasn't selected...}
40:         signal t_exec_t                              {first barrier}
41:         wait for signal from t_exec_t                {second barrier}
42:     c_map[id] ← 0                                    {recompute worker's bmap}
43:     for all ⟨dep_list, B⟩ in c_seq[id] do            {for each batch}
44:         c_map[id] ← c_map[id] ∨ b(B)                 {logic OR of bitmaps}
```

involved threads synchronize to solve all dependencies of a common batch before processing the batch, that is, a batch is only processed after its dependencies were solved. A batch that appears in a subset of thread queues (possibly in only one queue) is independent of batches in all other queues it does not appear. Moreover, replica progress is granted. A first observation is that the queues of working threads with commands that need synchronization will not deadlock because $<_B$ is an irreflexive total order, and thus acyclic. Since the queues of working threads are coherent with $<_B$, there is no inversion in their elements to provoke a cycle. A second related observation is that it is always possible to remove a batch B of the queue of a working thread because their queues are coherent with $<_B$ and thus the lowest element is well-defined (this is argued in Appendix B).

Consistency across replicas. As batches arrive at different replicas respecting $<_B$, it is possible to calculate dependencies from previous batches still enqueued (i.e., calculate $\prec_B$ for the previous delivered batches still enqueued). Different replicas may progress at different speeds and thus may have different sets of enqueued batches. Consider independent batches B_i and B_j such that $B_i <_B B_j$. While a replica R_1 could have processed B_j concurrently with B_i and finish B_j before B_i, leading to $B_j \bullet B_i$, in another replica R_2, B_i could already have been processed when B_j is delivered, leading to $B_i \bullet B_j$. However this is not harmful since it is possible only for independent batches. If batches were dependent, R_1 would enforce $<_B$ which is naturally followed by R_2 since it already processed B_i in the appropriate order, respecting the total order of dependent batches.

5.3 Evaluation

In this section, we describe our prototype, present the experimental environment, identify the parameter space, and discuss the results of our performance study.

5.3.1 Implementation

In order to evaluate our parallel implementation of state machine replication, we developed a key-value store service. The service implements commands to create, read, update and remove keys from an in-memory database. Atomic broadcast is provided by the primitives broadcast and deliver implemented in Ring Paxos [MPSP10], a high-throughput atomic broadcast protocol. In our prototype, we used the URingPaxos[2] library, a Multi-Ring Paxos implementation in Java [BMPG14, BPP15].

Bitmaps are used to encode the keys of every command batched in a request. A hash function maps the keys into bitmap positions. Figure 5.2 illustrates a batch B and its

[2]https://github.com/sambenz/URingPaxos

respective bitmap representation. Batch *B* contains 2 commands and commands' keys are mapped into a bitmap of size *m*, *b*(*B*). The second and the last but one *b*(*B*) bits are set to 1, indicating that keys *x* and *y* belong to this bitmap.

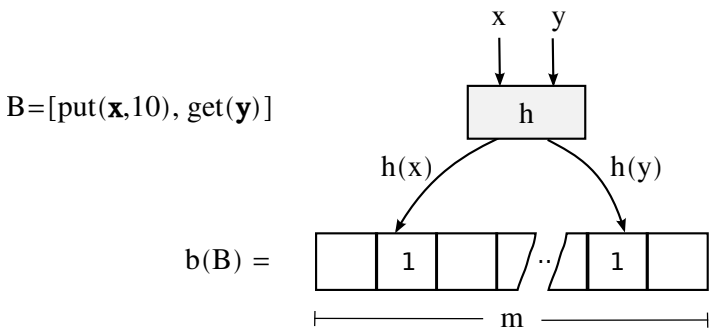


Figure 5.2 – Resulting bitmap for a batch of commands. The bitmap size is *m* and it encodes 2 keys.

The number of bits in a bitmap does not necessarily represent the number *n* of commands encoded in that batch. Since any two commands may access the same key, the same bit can map more than one command key. In addition, depending on the size of *n* and *m*, there is a certain probability that the hash function maps two different keys to the same bitmap position. The probability of two encoded keys occupying the same bit increases as *n* increases and *m* decreases. While this approach is subject to false positives (i.e., it may detect a conflict when none exists), it is not prone to false negatives (i.e., it does not miss real conflicts).

Client commands are forwarded to a client proxy, which is responsible for batching those commands in a single request. To alleviate the burden on the parallelizer, the bitmaps for a batch are computed by the client proxy and encapsulated in the request message. Client proxies broadcast a request to the replicas and wait for the first reply from a replica for every command in the batch before broadcasting another batch.

With the aim of shrinking batches and reducing message size, the values retrieved from our key-value store service are compressed before being transferred through the network. Towards this end, we use primitives `compress(byte[] value)` and `decompress(byte[] value)` on the client side. For instance, update and read operations should be in the form `db.put(x, compress("Hello world!"))` and `result = decompress(db.get(x))`, respectively. In this example, `db` is a reference to the key-value store, `x` is a key, and `put(key x,byte[] value)` and `get(key x)` are update and read operations, respectively.

Upon receipt of a batch, the replica proceeds as in Algorithm 5.1. To execute commands (line 32), each worker thread decompresses and extracts command values from received batches before executing them. In the experiments, we opted for using only update commands.

To reduce the overhead of checkpointing, two main optimizations are implemented: copy-on-write [LNP90, EJZ92, Rod08], and sequential checkpointing [BSF+13]. In the copy-on-write technique, all key-value store objects are write-protected during checkpointing. If the replica attempts to modify one of these objects while it is still protected, the object is duplicated and the protection is removed, allowing modifications on it. The newly allocated object is accessible only by the checkpoint procedure to write the original content to stable storage. After writing each object to stable storage, the write-protection is removed and duplicate objects deallocated. The copy-on-write technique permits the checkpoint procedure to run in parallel with the execution of commands. In sequential checkpointing, the intuition is to make replicas store their state at different times, to ensure that a quorum of replicas can continue processing without suffering from the checkpointing overhead. Although replicas checkpoint their state at different points of the execution, the checkpointing interval is the same for all of them. This means that every checkpoint interval only f of the correct replicas take a checkpoint.

5.3.2 Environment and configuration

All experiments were executed on a cluster with two types of nodes: HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory; and Dell PowerEdge R815 nodes equipped with four 16-core AMD Opteron 6366HE processors running at 1,8 GHz and 128 GB of main memory. The HP nodes were connected to an HP ProCurve switch 2920–48G gigabit network switch, and the Dell nodes were connected to another, identical network switch. The switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 6.5 and had the Oracle Java SE Runtime Environment 8. Paxos proposer, acceptors, and clients were deployed on HP nodes, while replicas were deployed on Dell nodes. Our prototype was set up to tolerate one failure, requiring three acceptors and two replicas.

5.3.3 Performance Analysis

Our parallel state machine replication is a complex system, whose performance is determined by application characteristics (e.g., command size, mix of read and write commands, percentage of dependent and independent commands) and configuration parameters (e.g., number of worker threads, batch size, bitmap size, checkpoint frequency). Consequently, assessing performance under the complete space of configuration parameters is impractical. Our approach is to start with a series of experiments covering a range of scenarios helpful to identify meaningful indicatives of performance.

Our prototype limits client requests size to 64k bytes. The message size is calculated by the sum of the bitmap size plus the sum of key and value sizes for each batched command (i.e., command's key and value size are multiplied by the number of commands in a batch). In our experiments, the bitmap size varies from 1024 to 20480 bits. We used keys of 8 bytes and values sizes vary from 8 bytes to 4096 bytes.

Our first set of experiments aims to understand how normal execution in parallel state machine replication is affected by the number of worker threads and message size. From Figure 5.3, both 1k- and 4k-byte commands replicas reach best performance with 8 threads. With fewer than 8 threads, the bottleneck is on the execution of commands; with more than 8 threads the bottleneck is on the scheduling of commands (parallelizer). As we could expect, performance is higher with smaller values. Based on these results, hereafter we consider experiments with 8 worker threads and commands' value of 1k-byte. Figure 5.4 depicts the throughput versus latency graph for this configuration. For the experiments that follow, we consider an operational load (i.e., number of clients) that corresponds to 70% of peak performance, approximately 43k commands per second.

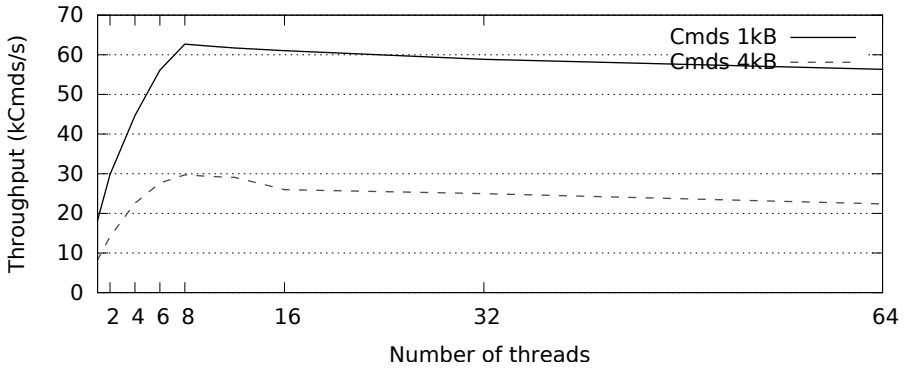


Figure 5.3 – Maximum throughput versus number of threads.

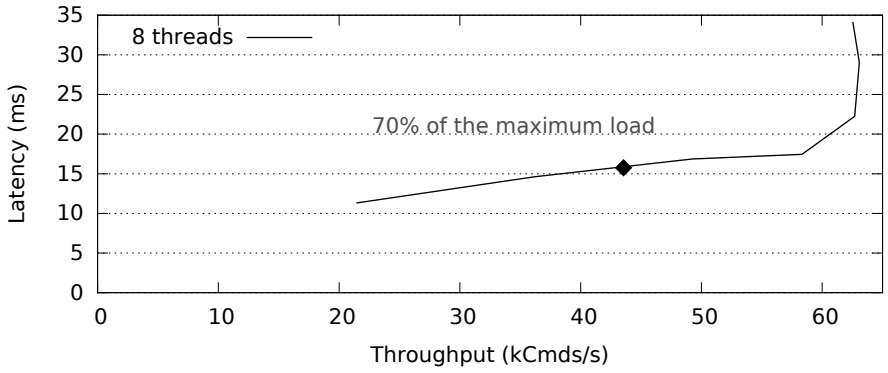


Figure 5.4 – Throughput versus latency of 1k-byte commands.

The experiments above assume batches with 50 commands, independent commands only (i.e., 0% dependency probability), and no checkpoints. We now proceed to

understand how each one of these parameters impacts performance. From Figure 5.5, as expected, both throughput and latency increase with batch size. Throughput increases with batch size since there is less scheduling overhead per command executed, while latency increases with batch size since clients receive responses when all commands in a batch have completed. We notice an important throughput gain with batch size 50 if compared to smaller batches, while presenting reasonable latency; therefore we continue the experiments with a fixed batch size of 50 commands.

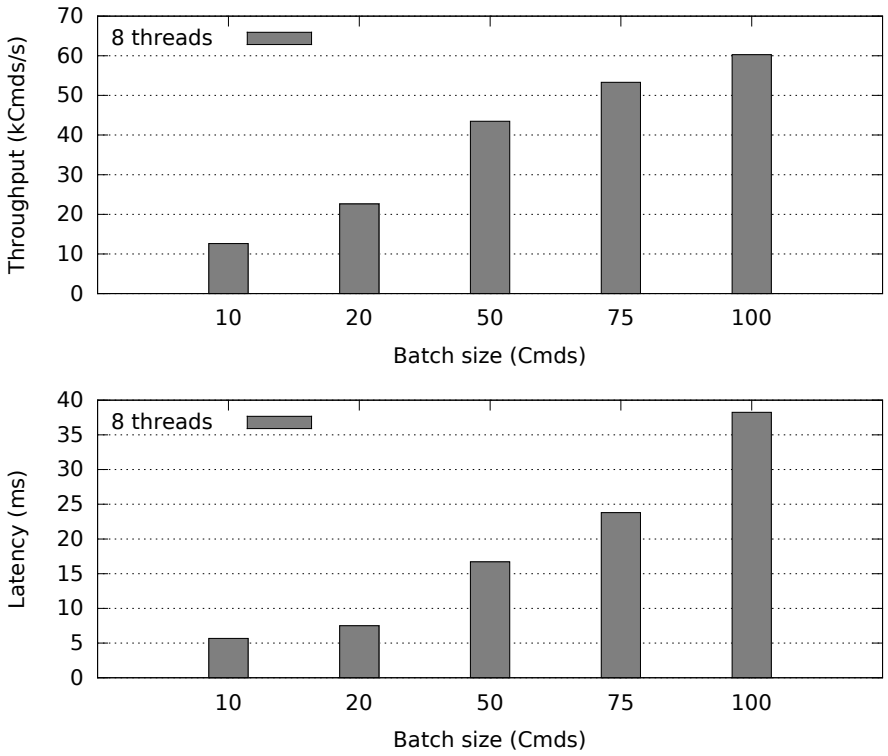


Figure 5.5 – Throughput (top) and latency (bottom) according to the batch size.

Further aspects that impact throughput in our parallel state machine replication approach are dependency probabilities and checkpoint interval. Two batches of commands are dependent if there are at least two commands, one in each batch, that are dependent; conversely, the batches are independent if no command in one batch depends on commands in the other batch. We denote dependency probability the probability that two batches are dependent. As it can be observed in Figure 5.6, throughput decreases as the dependency probability increases. Lower dependency probabilities show little effect on the throughput while, after 10% dependency probability throughput drops considerably until it reaches another plateau, approaching sequential batch processing throughput. When the dependency probability is 100%, all batches are sequentially executed leading to throughput coherent with the plot of Figure 5.3 for 1 thread, both around 18K commands per second.

The checkpoint interval also affects performance, as observed in Figure 5.7. In our prototype, checkpoint intervals are measured in number of batches executed between two

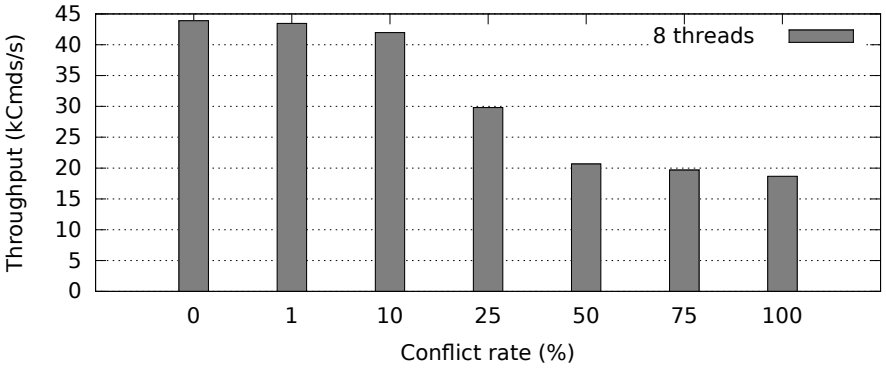


Figure 5.6 – Throughput variation according to dependency probability.

consecutive checkpoints. The bigger the intervals, the higher the throughput, but also the longer the log of old commands during recovery. The configuration with checkpoint interval of 20k batches has provided a throughput around 36k commands per second. Although higher checkpoint intervals achieve higher throughput, they may incur long logs.

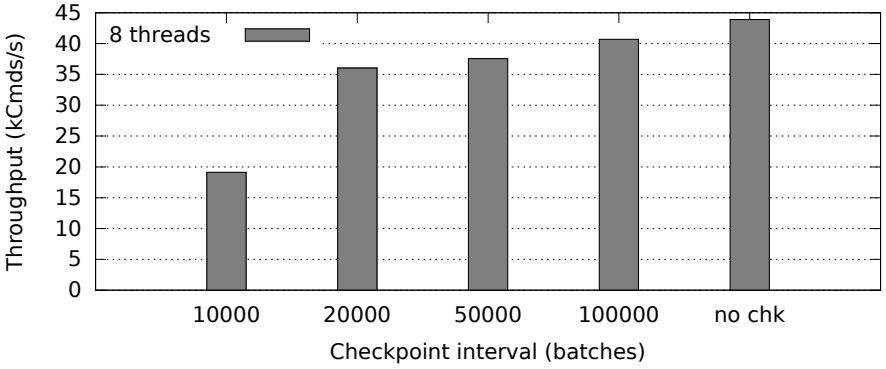


Figure 5.7 – Throughput variation according to the checkpoint interval.

In previous experiments, we set the space of generated keys in a way that checkpoint size was around 512M bytes. In our prototype, replicas store their checkpoints in a remote stable storage. The checkpoint duration is the time taken to serialize the entire service tables into a checkpoint image plus the time to transfer the checkpoint image to the stable storage through FTP.

Next, we evaluate how checkpoint size affects performance. Towards this end, we measure the time taken for saving checkpoints with different sizes and analyze how the checkpoint duration affects the log growth.

Figure 5.8 shows the checkpoint duration according to the checkpoint size. As observed, creating a checkpoint of 512M-bytes takes approximately 15 seconds, while a checkpoint of 1G-byte takes 30 seconds. We observe a processing rate around 35M-bytes per second while generating checkpoints.

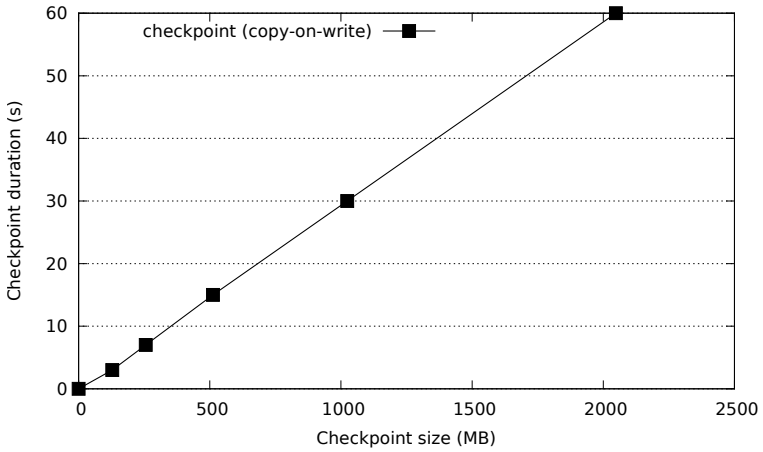


Figure 5.8 – Checkpoint duration according to the checkpoint size.

Our prototype implements copy-on-write [LNP90, EJZ92, Rod08] and sequential checkpointing [BSF+13] to minimize the effects of checkpointing. While the first technique allows the processing of new commands in parallel with checkpoint creation, the second one reduces throughput hiccups caused by checkpointing. Because of that, we expect little reduction in the overall throughput when checkpoints are in progress.

Figure 5.9 shows the growth of log over time for executions with 1, 2, 4, and 8 worker threads. These results consider an operational load that corresponds to the peak performance for each threads' configuration scenario (see Figure 5.3 for a baseline). From Figure 5.8, performing a 512M-byte checkpoint takes about 15 seconds. During this interval, for the 8 threads configuration, our prototype can execute nearly one million of commands, amounting to almost 1G-byte worth of logged old commands. For the same set up (8 threads), while performing a 2G-byte checkpoint (which takes about 60 seconds), our prototype can execute more than 3.5 millions of commands, amounting to more than 3.5G-byte worth of logged commands.

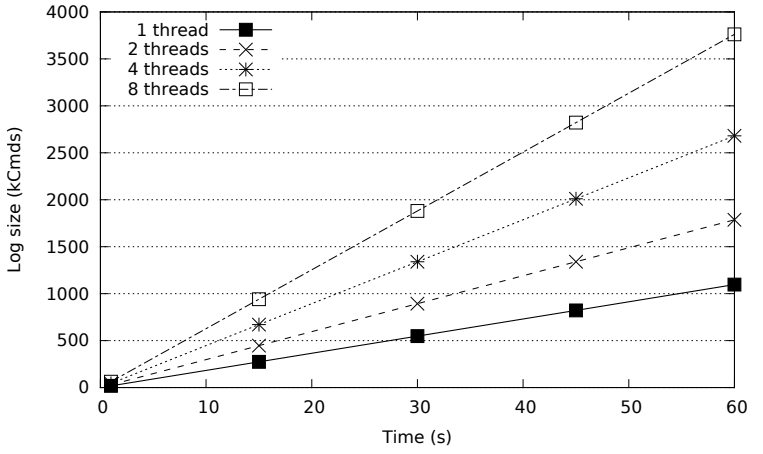


Figure 5.9 – Log size increase over time.

Table 5.1 presents the smallest log size expected for several combinations of checkpoints size and number of worker threads. As the checkpoint size increases, the checkpoint procedure takes longer to finish. The number of commands processed during checkpoint duration indicates the minimum number of requests in a log, i.e., the smallest possible log (see Figure 1.1). The throughput varies with the number of worker threads in execution, as does the log size. In the table, the log size given in M-byte refers to commands with keys of 8-bytes and values of 1k-byte.

Table 5.1 – Smallest log size estimation according to the checkpoint size

Checkpoint size (MB)	Checkpoint duration (s)	Log size (kCmds)				Log size (MB)			
		1 thread	2 threads	4 threads	8 threads	1 thread	2 threads	4 threads	8 threads
128	3.77	69	112	168	236	68	110	166	233
256	7.55	138	225	337	473	136	221	332	466
512	15.09	276	449	674	946	271	442	663	931
1024	30.19	551	899	1349	1893	543	884	1327	1863
2048	60.38	1103	1797	2697	3785	1086	1769	2655	3725

6. FAST RECOVERY IN PARALLEL STATE MACHINE REPLICATION

Recovering a crashed server boils down to fetching and installing a service checkpoint and retrieving and (re-)executing commands that are not included in the checkpoint. With standard recovery techniques, a recovering replica can only execute “new commands” after it has fetched and installed a checkpoint and retrieved and executed “old commands”, that is, commands that were already ordered and perhaps even executed but are not included in the installed checkpoint. In this chapter, we introduce techniques to reduce recovery time.

6.1 Speedy recovery of large logs

Recovery typically faces the following tradeoff. In order to minimize the number of old commands a recovering replica needs to retrieve and execute, checkpoints must be frequent. Checkpoints, however, may degrade performance during normal execution; thus, for performance checkpoints should be infrequent. Even though some techniques strive to reduce the impact of checkpointing on normal execution (e.g., in our prototype we use a copy-on-write data structure [LNP90, EJZ92, Rod08] and sequential checkpointing [BSF+13] to minimize the effects of checkpointing), checkpoint frequency is limited by other factors. Since concurrent checkpoints may overload the system and further complicate the design, checkpoints are typically configured to happen sequentially, that is, one checkpoint only starts after the previous one has finished. Therefore, checkpoint frequency is ultimately limited by how quickly a single checkpoint can be performed.

In systems designed for high performance, such as parallel state machine replication, a practical consequence of the recovery tradeoff mentioned above is that a recovering replica needs to handle a large sequence of old commands before it can execute new commands. This situation renders the replicated system more vulnerable to failures since a recovering replica is only available once it can process new client commands. Instead of trying to reduce the sequence of old commands (e.g., by increasing checkpoint frequency), we approach the recovery tradeoff from a different perspective: we allow new commands to execute before old commands have been processed. In brief, our strategy is based on the observation that a new command does not need to wait for an old command to be executed if the two commands are independent. In the rest of this chapter, we explain how we integrate this strategy in the parallel state machine replication scheme described in Chapter 5.

During normal operation, replicas create a dependency log, which contains bitmaps of command batches the replica executed since its last checkpoint. When a replica creates a new checkpoint, it trims its dependency log. To recover from a failure, a replica retrieves a recent checkpoint and the dependency log from an operational replica (or from remote

storage). Old commands not included in the restored checkpoint will be delivered with atomic broadcast. But since the dependency log contains a digest of the old commands, it can be retrieved much more efficiently than the actual commands.

With the dependency log, the recovering replica can execute new commands before processing all old commands. The replica splits the delivery of commands into two flows: one flow with new commands and one flow with old commands. Old commands are scheduled for execution as during normal execution. A new command is scheduled for execution if it is independent of every old command that has not been scheduled yet. The replica uses the dependency log to check whether a new command is independent of pending old commands.

Algorithm 6.1 complements Algorithm 5.1 and introduces the recovery of parallel state machine replication. This algorithm presents the steps of a starting replica (lines 2–10) and redefines the steps of the parallelizer (lines 12–25 in Algorithm 6.1 replace lines 26–28 in Algorithm 5.1). When a replica starts its execution, it first retrieves a checkpoint and the checkpoint identifier (line 2). The checkpoint identifier is the largest delivery instance of a command in the checkpoint. The replica then installs the checkpoint (line 3).

After the checkpoint is installed, the replica calls the initialization procedure defined in Algorithm 5.1 (line 5), queries the atomic broadcast module to determine the latest instance of a delivery event (line 5), retrieves the dependency log with bitmaps of batches containing old commands, that is, commands in batches delivered before the latest instance (line 6), initializes variables *i* and *k*, which will keep track of delivery events for old and new commands, respectively (lines 7 and 8), and variables *n_seq* and *n_map*, which contain the sequence of batches of new commands that cannot yet be executed and their bitmap representation (lines 9 and 10).

The parallelizer handles two flows of delivery events, one for old commands and one for new commands. When the parallelizer delivers a batch *B* of old commands (line 13), it schedules *B* for execution and gets ready for the next batch (lines 14 and 15). If there are no more batches of old commands, the parallelizer schedules all batches of new commands that have been delivered but could not yet be scheduled because they contain commands that depended on old commands, either directly or indirectly (lines 16–18).

When the parallelizer delivers a batch *B* of new commands, it checks whether *B*'s bitmap intersects with the bitmaps of batches with old commands that have not been scheduled yet (i.e., *d_map* structure) and with new commands that precede *b* (i.e., *n_map* structure). If there is no intersection, *B* is scheduled for execution (lines 20–21); otherwise, *B* is appended to *n_seq* to be executed later (line 23) and its related bitmap *n_map* is updated (line 24). Finally, the parallelizer is ready to deliver the next batch of new commands (line 25).

Why it works. Correctness of our recovery technique is discussed in Appendix B. We regard the total delivery order $<_B$ as being a concatenation of $<_{B_c}$, $<_{B_o}$ and $<_{B_n}$, respectively the batches processed in the checkpoint, the batches received while the replica was not

Algorithm 6.1 Recovery

```
 1: Upon starting execute as follows:
 2: (chk, i) ← lastCheckpoint()                          {retrieve checkpoint and its identifier}
 3: install chk                                          {install checkpoint}
 4: initialization()                                     {initialize structures in Algorithm 5.1}
 5: j ← currentDeliveryInstance()                        {instance of last delivered message}
 6: d_map[i..j] ← dependencyLog(i, j)                    {bitmaps of delivered batches}
 7: i ← i + 1                                            {instance of next batch of old commands}
 8: k ← j + 1                                            {instance of next batch of new commands}
 9: n_seq ← ∅                                            {sequence of batches with new commands}
10: n_map ← 0                                            {bitmap associated with sequence above}

11: The parallelizer executes as follows:
12: when (deliver(i, B) and i ≤ j) or deliver(k, B)
13:    if delivered (i, B) then                          {if delivered batch of old commands}
14:       schedule(B)                                    {schedule batch for execution}
15:       i ← i + 1                                      {set next old batch to be retrieved}
16:       if i > j then                                  {if done with batches of old commands}
17:          for each B ∈ n_seq, in order do             {pending batches of...}
18:             schedule(B)                             {...new commands are scheduled now}
19:    else                                              {if delivered batch B of new commands}
20:       if b(B) ∩ (d_map[i] ∨ ... ∨ d_map[j] ∨ n_map) = ∅ then
21:          schedule(B)                                {schedule B for execution if possible}
22:       else                                          {if B depends on batches that precede B}
23:          n_seq ← n_seq ⊕ ⟨B⟩                         {add B to pending sequence}
24:          n_map ← n_map ∨ b(B)                       {update related bitmap}
25:       k ← k + 1                                     {set next new batch}
```

available and have to be processed to catch up with the current state, and new batches being received by the replica as soon as they are available again. To allow new batches to execute concurrently with old batches, we ensure that any batch in $<_{B_n}$, which is independent of other batches in $<_{B_n}$, can be executed concurrently with batches in $<_{B_o}$ only if they are independent, otherwise they wait for batches in $<_{B_o}$ to be completed. This ensures that all batches, both old and new, respect $\prec_B$, the dependency relation.

6.2 On-demand state recovery

Although processing new commands concurrently with old commands improves the availability of the system by bringing a recovering replica back up more quickly, old and new commands can only execute after the recovering replica has transferred and installed a checkpoint. On-demand state recovery addresses this shortcoming. The overall idea is to divide the service state (i.e., checkpoint) into segments, and retrieve and install each segment only when it is needed for the execution of a command, be it a new or an old command.

In order to implement on-demand state recovery, we have moved the logic involved in the retrieval and installation of a checkpoint to the worker threads, instead of performing it

as the first action of a recovering replica. As soon as a recovering replica has retrieved the dependency log, it can schedule commands, as described in the previous section. Before a worker thread executes a command, it checks whether the needed segment is already installed or not. If the segment is not installed, the worker thread retrieves the segment from an operational replica (or remote storage), installs the segment, and then executes the command. If the segment is being installed by another thread, then the worker thread blocks until the segment has been completely installed. When finalizing the installation, the segment is marked as installed and each blocked thread is notified to resume its execution.

This scheme improves performance in two aspects. First, it defers the transferring and installation of segments to when they are needed. Second, it allows to parallelize these operations on different worker threads.

6.3 Evaluation

In this section, we describe our prototype, explain our assessment goals and methodology, and present the results of our performance study.

6.3.1 Implementation

In order to evaluate our recovery technique, we improved the prototype presented in Chapter 5 by implementing speedy recovery and on-demand recovery techniques. The implementation of these features is discussed in more detail below.

The dependency log is implemented as a list of consolidated bitmaps for groups of delivered batches. Instead of keeping one bitmap per batch, several batches are grouped and associated to a single bitmap that is computed as the union of the bitmaps of batches in the group. We call this consolidated bitmap *log bitmap item*.

Figure 6.1 illustrates batch bitmaps *i* to *j* being added to a log bitmap item. A log bitmap item is composed of the cumulative result of OR operations for batch bitmaps *i* to *j*.

For performance reasons, after every *j* batches have been received and added to a log bitmap item, replicas create a new log bitmap item that will be used to store information about the next *j* batches. The list of log bitmap items constitutes the *dependency log*, as depicted in Figure 6.2. The first element in the list contains a consolidated bitmap for the first *j* batches, the second element is a consolidated bitmap for the next *j* subsequent items, and so on. In order to check dependencies between an incoming request and the batches already processed, a replica tests whether the incoming bitmap intersects with any of the log bitmap items in the log bitmap list.

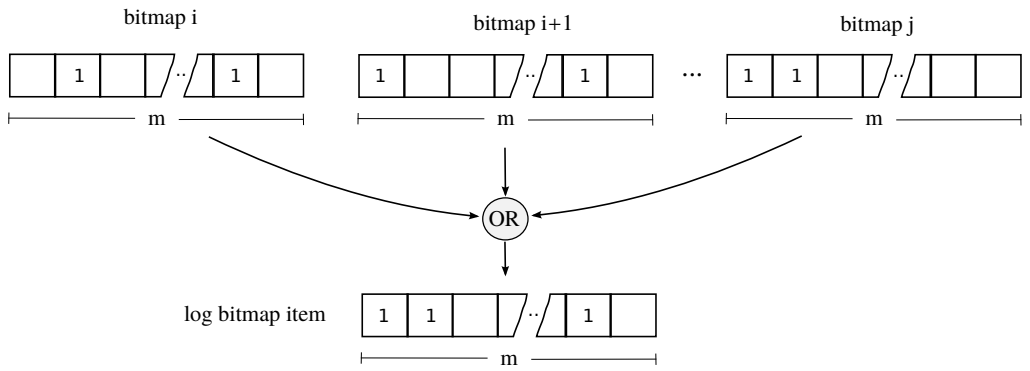


Figure 6.1 – Resulting bitmap item for a group of batches. The bitmap item contains encoded information for batches *i* to *j*.

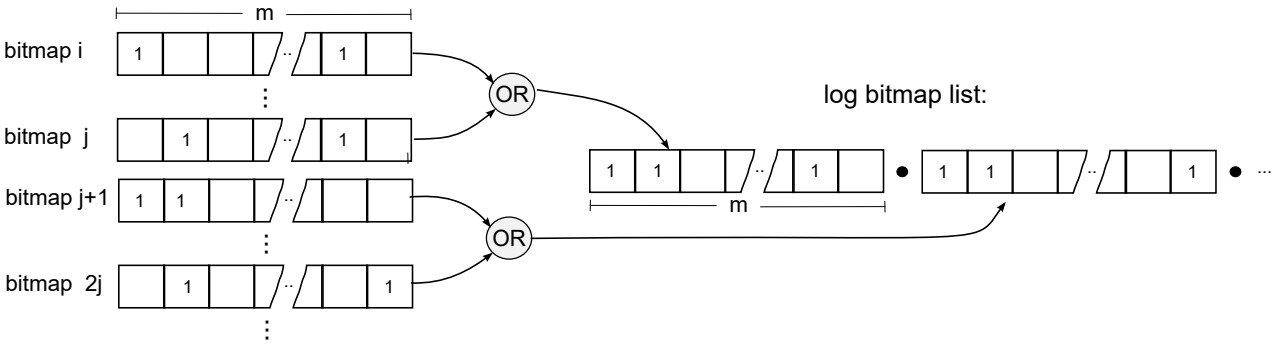


Figure 6.2 – Dependency log structure.

When a checkpoint is created, the replica uses the identifier of the last batch present in that checkpoint to trim unnecessary log bitmap items from the dependency log. Every log bitmap item that contains only batch instances lower than that identifier can be removed. As a consequence, the dependency log contains only information about requests that are not included in the replica's most recent checkpoint.

We divide the state of our key-value store service in partitions such that each partition is responsible for a range of keys. Key ranges are of the same size and workloads generated in our experiments choose keys uniformly distributed among the partitions. While checkpointing, a replica serializes each of the partitioned tables into a checkpoint segment (using copy-on-write). Since the partitioned tables are independent, checkpoint segments can be created concurrently by multiple threads.

6.3.2 Goals and methodology

The high performance recovery techniques introduced in this thesis aim to speed up recovery of large logs and reduce state transfer. Ultimately, both techniques are designed to increase a replica's availability and introduce modifications whose impact on server and

client sides have to be observed. We wish to quantify the effects of these optimizations with emphasis on the following goals.

- **Recovery time.** A replica is recovered as soon as it can process new commands. We assess the time it takes for a replica to recover using the techniques introduced in this work and compare them to classical recovery.
- **Throughput during recovery.** Speedy recovery allows new commands to be processed before recovery is finished, differently from classical recovery techniques. We determine the throughput of new commands during the restart of a replica.
- **Recovery breakdown.** We investigate the interplay of the steps involved in speedy recovery and on-demand state transfer and how each step contributes to recovery duration.

We focus our recovery study on three strategies: (a) classical SMR recovery, (b) speedy recovery, and (c) speedy recovery combined with on-demand state transfer. All experiments were executed on the same cluster described in Chapter 5.

6.3.3 Performance Analysis

Once we surveyed the performance of our prototype for several important parameters (see Chapter 5), we fix the system with 8 threads, operational load at 70% of peak throughput, batch size of 50 commands, checkpoint interval at 20k batches and investigate the behavior of recovery for 0% and 5% dependency probabilities. The reason for choosing dependency probability of 0% is to understand the potential of the technique (i.e., it results in the best performance), while 5% is more than one should expect in a typical application, according to the literature. Moraru et al. [MAK13] state that from the available evidence, dependency probabilities between 0% and 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 [Bur06]. In Google's advertising back-end, F1, fewer than 0.3% of all operations may generate conflicts [CDE+12].

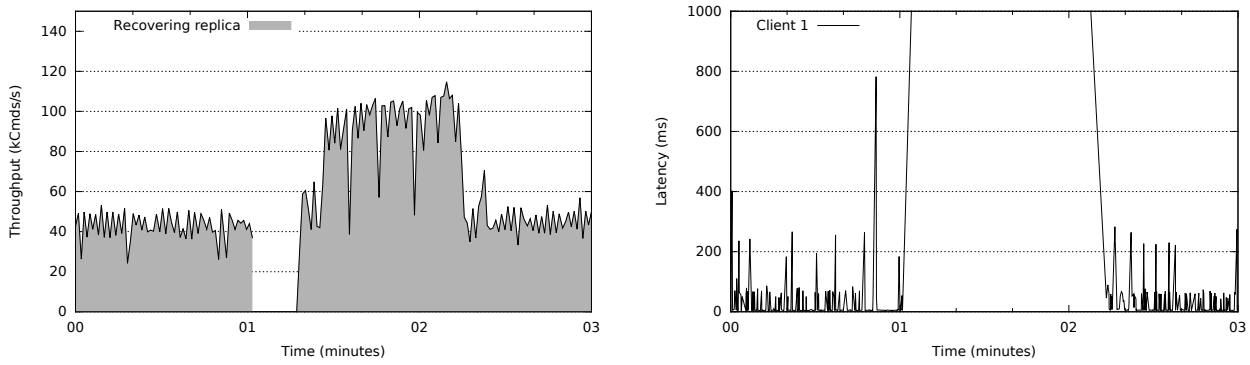
Keeping the dependency probability rate in a controlled range is very challenging or even impractical, since it is highly dependent on systems parameters and susceptible to workload fluctuations. In order to allow greater control over dependencies observed in our experiments, we extended our prototype to induce dependencies according to synthetic rates. By doing so, although our prototype performs all bitmaps comparisons resulting from a regular execution, dependencies are detected according to a probabilistic rate configured in advance. The following experiments adopt synthetic rates. Experiments with real dependencies are left to the end of this section.

We start by analyzing the effects on replica's downtime due to the anticipation of new requests processing. To facilitate our analysis, only one replica is active during the executions. Thus, service unavailability will be perceptible to the clients as requests timeout. In our scenario, a replica executes for 3 minutes, fails and, as soon as possible, it starts recovering. By starting the recovery procedure immediately after a failure, and running a single replica, we reduce unpredictable effects that could be caused by failure detection or reconfiguration mechanisms. In order to avoid execution stalls, we disable the generation of checkpoints throughout the execution. Consequently, the whole log of old commands has to be processed by the recovering replica.

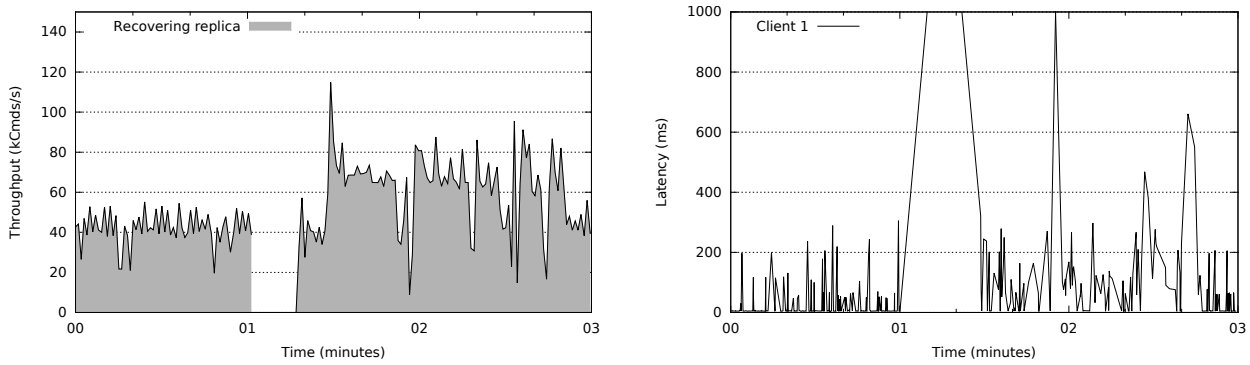
Figure 6.3 depicts excerpts of execution traces for (a) classical recovery, (b) speedy recovery, and (c) speedy recovery combined with on-demand state transfer. The execution interval shown in the graphs has 3 minutes of duration and it starts in the last minute before replica failure. Graphs on the left show replica throughput and graphs on the right show the response time measured at one of the client proxies. The dependency probability was set to 0%. For all execution traces, the recovery log contains around 80.000 batches, i.e., four millions of commands.

Replica crashes at the instant 1 minute and starts recovering immediately. The processing of requests is resumed at the point where throughput returns to growth. Between these two points in time, the replica downloads and installs a checkpoint image. The higher throughput observed when the replica recovers is explained by the large log of commands that have to be processed and by the fact that the replica can handle load levels higher than that experienced before failure. After the recovery log has been completely executed, the throughput falls to the same level observed during normal operation. This is not clear in scenarios (b) and (c) because we trimmed the execution trace to fit in a 3 minutes interval, however in scenario (b) this happens around 3 minutes and (c) around 2 minutes and 49 seconds. As expected, the time to resume processing of requests is smaller when on-demand state transfer is enabled. This happens because downloading a checkpoint segment takes just a fraction of the time to download the whole image, and some requests can be processed right after the segment is installed.

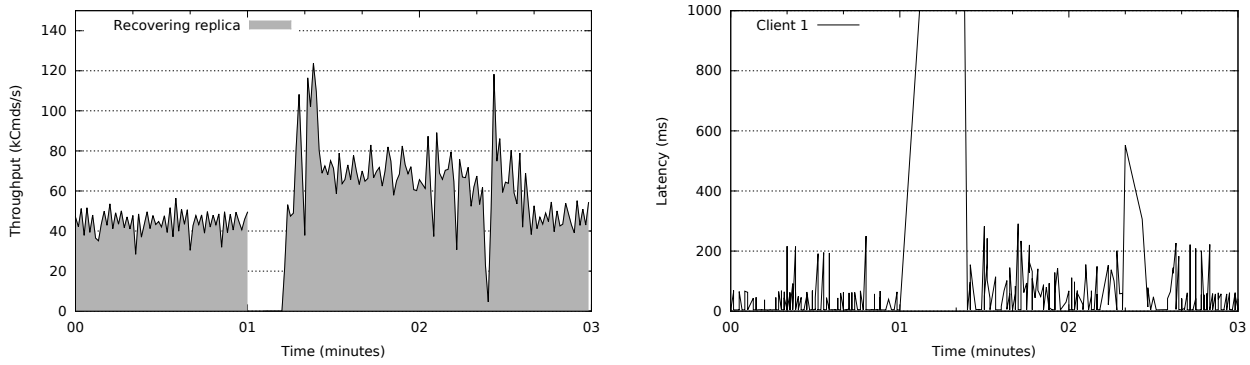
After sending a request, client proxies wait for 1 second to receive a reply. If no reply is received, a timeout expires and the request is resent. Measurements around $1000ms$ in the latency graphs identify client requests that expired. As expected, speedy recovery and speedy recovery combined with on-demand state transfer are faster than classical recovery. This can be clearly observed by the longer recovery time for the classical recovery. While in the classical recovery new commands are processed after approximately 63 seconds, with our recovery strategies the recovery time is less than 15 seconds. Table 6.1 shows the recovery time (measured at replica) and service downtime (measured at client proxy) for each technique. The service downtime is the period in which client proxy requests are expired.



(a) Classical recovery



(b) Speedy recovery



(c) Speedy recovery combined with on-demand state transfer

Figure 6.3 – Throughput and latency for crash-recovery trace using: (a) classical recovery; (b) speedy recovery; (c) speedy recovery combined with on-demand state transfer

Table 6.1 – Recovery time and service downtime

	Recovery time (s)	Service downtime (s)
Classical recovery	63.43	69.72
Speedy recovery	14.89	28.66
Speedy + on-demand state recovery	10.46	24.71

To better understand how the proposed recovery strategies perform, we drill down into the throughput analysis. The next results depict the flows of delivery instances for both new and old commands (commands in the log). Moreover, we mark the timestamps for important steps during recovery. The next experiments consider scenarios with one correct replica and one recovering replica. Just like the previous experiments, the recovering replica crashes and recovers immediately after failure. Replicas are configured to take checkpoints at intervals of 20.000 batches.

Figure 6.4 shows the execution of a recovering replica combining speedy recovery with on-demand state transfer. We set the space of generated keys in a way that checkpoint size was around 512M bytes and the dependency probability was 0%. The recovering replica takes about 7 seconds to recover, a period that corresponds to the time to download and install at least one checkpoint segment. Right after that, the replica can process new commands in the installed segment while downloading and installing other checkpoint segments on-demand. Once the first checkpoint segment is installed, the replica starts processing new and old commands assigned to that segment. The total throughput is given by the sum of the black and gray areas, representing the throughput of old and new commands, respectively.

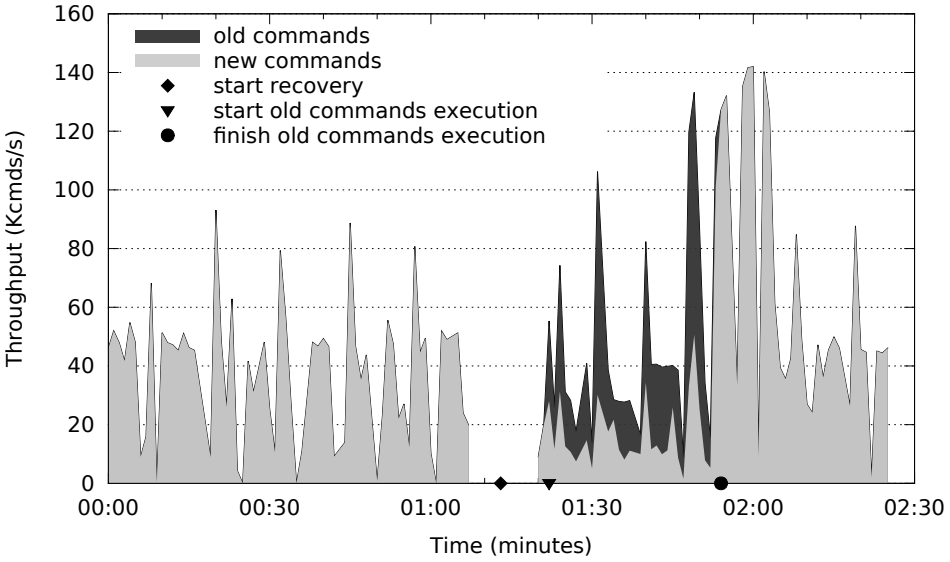


Figure 6.4 – Throughput using speedy recovery and on-demand state transfer with 0% of dependency probability.

Figure 6.5 shows a similar execution, but with dependency probability of 5%. The recovering replica also takes about 7 seconds to recover, but the throughput of new commands processed during recovery is lower. As expected, as the dependency probability increases, fewer new commands can be scheduled for execution in parallel with old commands. In the worst case, when all new commands conflict with commands in the dependency log, only old commands would be processed during recovery. This behavior resembles the classic recovery.

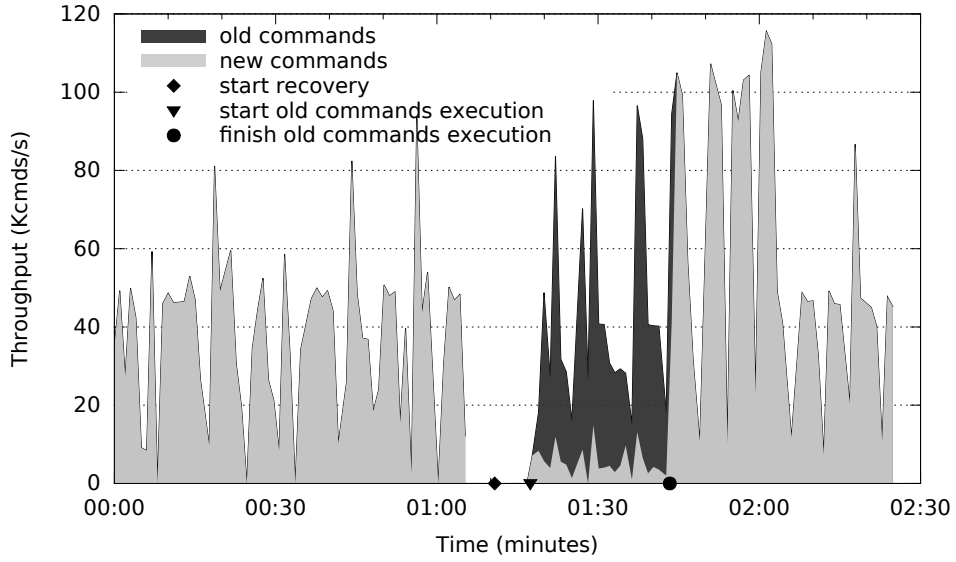


Figure 6.5 – Throughput using speedy recovery and on-demand state transfer with 5% of dependency probability.

Table 6.2 quantifies the recovery behavior for workloads with dependency probability of 0% and 5%. The time to download and install a checkpoint in both classical and speedy recovery techniques is similar. However, when using the on-demand state transfer technique, these times can be considerably reduced. For instance, checkpoint segment P3 takes less than half of the time taken by other techniques. When the dependency probability increases, we expect lower throughput of new commands after replica restart. This can be observed in the row “New command throughput (recovery)”. Nonetheless, the average throughput of new commands during recovery was around 10k when dependency probability was set to 5% for both speedy recovery and speedy recovery combined to on-demand state transfer techniques.

Since our techniques aim at minimizing the unavailability of a replica, we further investigate the behavior in time when we apply our techniques, accounting for the time needed to transfer the checkpoint, to install the checkpoint, to process the log, and the moment when the first new command is serviced, denoting that the service is available for new requests. Figure 6.6 provides a graphical representation of the recovery cost breakdown.

An important metric is the time to execute the first new command, since the replica is able to process new commands from that moment on. Obviously, in the classical approach, the replica processes the first new command after processing the whole log. Thus, the time to execute the last old command and the first new command are practically the same. Depending on the dependency probability, speedy recovery can drastically reduce the time to execute the first new command. Regarding the time to execute the first new command, our experiments demonstrated that speedy recovery is three times faster than classical recovery

Table 6.2 – Recovery techniques comparison

	Dependency probability = 0%						Dependency probability = 5%					
	Classical recovery	Speedy recovery	Speedy + on-demand state recovery				Classical recovery	Speedy recovery	Speedy + on-demand state recovery			
			P1	P2	P3	P4			P1	P2	P3	P4
Checkpoint download duration (s)	5.17	5.24	5.44	4.16	2.91	5.23	4.99	5.63	4.27	1.72	2.24	3.98
Checkpoint installation duration (s)	4.57	4.60	1.08	1.10	1.49	1.07	4.74	4.68	1.07	1.05	1.56	1.11
Old command execution duration (s)	22.25	35.53	32.97	34.22	35.36	33.55	22.65	40.19	27.12	25.96	28.73	33.21
Time for last old command execution	33.41	45.37	39.49	39.48	39.76	39.85	32.37	50.50	32.46	28.73	32.53	38.30
Time for first new command execution	33.41	10.43	6.54	5.26	4.40	6.30	32.37	11.38	5.44	2.89	3.80	5.10
Recovery speedup	**1**	**3.2**			**7.6**		**1**	**2.8**		**11.2**		
Throughput during normal execution	34742	34705	36000				34446	34138	36100			
New command throughput (recovery)	0	26242	19702				0	11930	9546			

in contention-free workloads. When on-demand state transfer is combined, recovery becomes more than 7 times faster.

The throughput of new commands during recovery is impacted by the dependency probability and the incoming rate of old commands. Higher rates increase the number of old commands competing with new ones to be processed. In the extreme case, that behavior approximates classic recovery, where old commands are processed first. By reducing the incoming rate of old commands, the system slows down processing of old commands and consequently increases the throughput of new commands. In previous experiments, we set the incoming rate of old commands to 200 batches every 0.2 seconds.

After assessing the performance of the proposed techniques under controlled dependency probabilities, we consider real dependency probability in our analysis. Figure 6.7 shows the throughput of old and new commands in a scenario with real dependency probability. In this experiment, we set the relearning of old commands to 100 batches every 0.2 seconds, bitmap size of 20k bits, keys of 8 bytes, values size of 128 bytes, batches of 50 commands, and checkpoint interval at 8000 batches. The operational load for this experiment results in an average throughput of 9k commands per second during normal execution.

When considering real dependency probability and long recovery logs, we noticed that only a small number of new requests are executed concurrently with old requests (see the tiny spike of new commands processed around 3:10 minutes). The inability to execute new requests in parallel with old requests in such scenarios is caused by a high dependency probability induced by the list of pending requests. As presented in Algorithm 6.1, every new request that depends on old or pending requests is added to a list of pending requests, *n_seq* (line 23), and a bitmap encoding dependencies of pending requests, *n_map*, is updated with the new request bitmap. Notice that as more requests are added to *n_map*, more bits are set

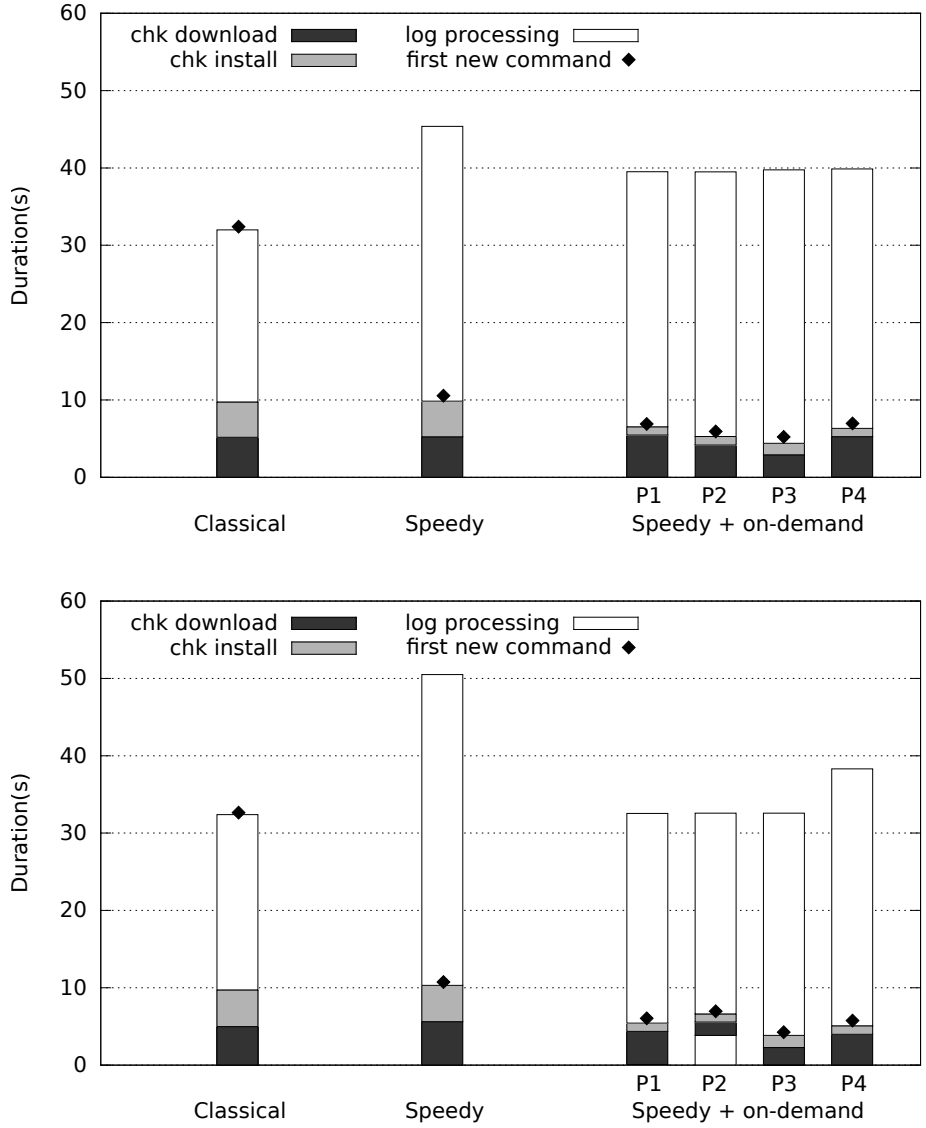


Figure 6.6 – Time taken during recovery for workloads with dependency probability 0% (top) and 5% (bottom)

and the dependency probability increases. Possibly there is an instant in which *n_map* has all bits set to 1, which will make any new request dependent (see Algorithm 6.1, line 20).

The adoption of a single bitmap structure to encode dependency information for all pending requests has demonstrated to be inefficient. To circumvent this issue, we devised a rescheduling mechanism that allows pending requests to be reevaluated and possibly scheduled before the whole log has been processed. We replaced the list of pending requests and its dependency annotation (respectively *n_seq* and *n_map*, as presented in Algorithm 6.1) by a structure we call *v_seq*. The new structure splits a single list of pending requests into smaller lists, while the dependency information is kept in the level of batches, instead of using a single consolidated bitmap.

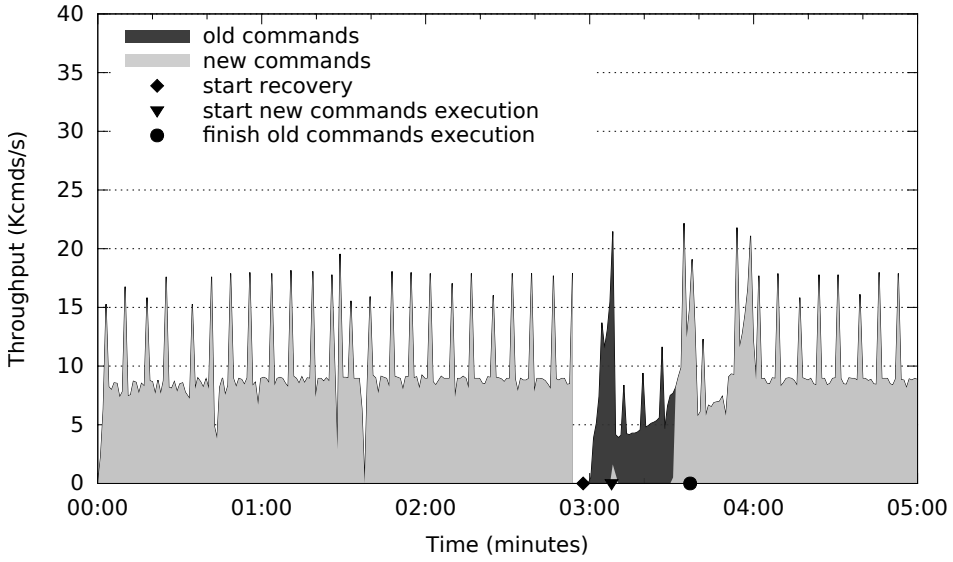
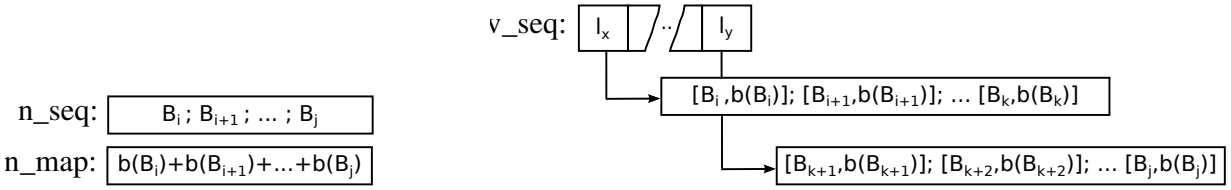


Figure 6.7 – Throughput using speedy recovery and on-demand state transfer with real dependency probability.

Figure 6.8 depicts (a) *n_seq* and *n_map* structures, and (b) *v_seq*. In the former proposal, for each new batch B that cannot be processed, B is appended to *n_seq* and its bitmap, $b(B)$, is merged into *n_map*. *v_seq* is a vector of pointers to lists which contain pending batches and their bitmaps. Two pairs $[B_x, b(B_x)]$ and $[B_y, b(B_y)]$ in *v_seq* respect the total delivery ordering. Thus, if B_x is delivered before B_y, then: (i) $[B_x, b(B_x)]$ and $[B_y, b(B_y)]$ belong to the same list in *v_seq*, such that $[B_x, b(B_x)]$ precedes $[B_y, b(B_y)]$; or (ii) $[B_x, b(B_x)]$ belongs to list l_x and $[B_y, b(B_y)]$ belongs to list l_y, such that *v_seq* position pointing to l_x precedes the position pointing to l_y.



(a) *n_seq* (b) *v_seq*

Figure 6.8 – Pending requests structures: (a) a list of batches combined with a consolidated bitmap; (b) a vector pointing to lists tuples containing batches and their bitmaps.

Before diving into details of the proposed rescheduling approach, we recap the strategy adopted on the implementation of the dependency log. Our prototype implements the dependency log as a list of bitmap items. Every item in the list consolidates the information of j successive request bitmaps (see Figure 6.1). Let o represent the total number of old requests to be processed, then the number of bitmap items bi in the list is given by $n = \lceil o/j \rceil$, and a dependency log DL can be represented as $DL = \langle bi_1, \ldots, bi_n \rangle$. During recovery, whenever

j successive old requests are processed, the head of *DL* is removed. Pending requests that depend on the bitmap removed from *DL* may have the chance to be reevaluated and rescheduled provided they do not depend on the remaining requests encoded by *DL*. For this reason, we optimized our protocol to reschedule pending requests whenever *DL* is trimmed.

The rescheduling approach works as follow. For each list l_j in *v_seq*, for each batch B_y in l_j, if $b(B_y)$ does not intersects with log bitmaps items in *DL*, and $b(B_y)$ does not intersects with $b(B_x)$, where $b(B_x)$ belongs to l_i and l_i precedes l_j in *v_seq*, then B_y is scheduled for execution and removed from l_j. Scheduling B_y for execution consists in executing the procedure *schedule(B_y)* (see Algorithm 5.1).

Since the rescheduling procedure runs in parallel with the recovery, new requests could be added to *v_seq* lists while batches in *v_seq* lists are under evaluation for rescheduling.

In order to provide mutual exclusion on access to *v_seq* without incurring synchronization, rescheduling and recovery procedures access distinct parts of *v_seq*. The rescheduling procedure evaluates all except the last list pointed by *v_seq*. The last list in *v_seq* is accessed exclusively by the recovery procedure. After adding a certain number of requests to the list, the recovery procedure creates a new list and updates *v_seq* with a reference to the new list given by the pointer in the last *v_seq* position. Figure 6.9 illustrates *v_seq* and highlights which lists contain requests candidate to be rescheduled. The first two bits of the vector point to empty lists. This means that pending requests on those lists were already rescheduled. Except by the last list (indicated as “current” in Figure 6.9), requests in all other lists pointed by *v_seq* are candidates for rescheduling.

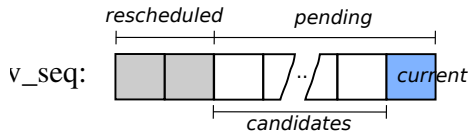


Figure 6.9 – Access to pending requests stored in *v_seq*.

Figure 6.10 shows the throughput of old and new commands for the same workload observed in Figure 6.7. The only difference is that the rescheduling mechanism is enabled. As expected, rescheduling of pending requests allowed new requests to be processed in parallel to the old requests. It is worth noting that the throughput of new commands during recovery stays very close to the throughput during normal operation.

In order to have results comparable with our previous experiments, we changed the command size from 128 to 1024 bytes. Furthermore, we evaluated what is the peak throughput for the adopted workload (i.e., relearning 100 old batches every 0.2 seconds, bitmap size of 20k bits, keys of 8 bytes, values size of 1024 bytes, batches of 50 commands, and checkpoint interval at 8000 batches). In the setup with 8 worker threads, the peak

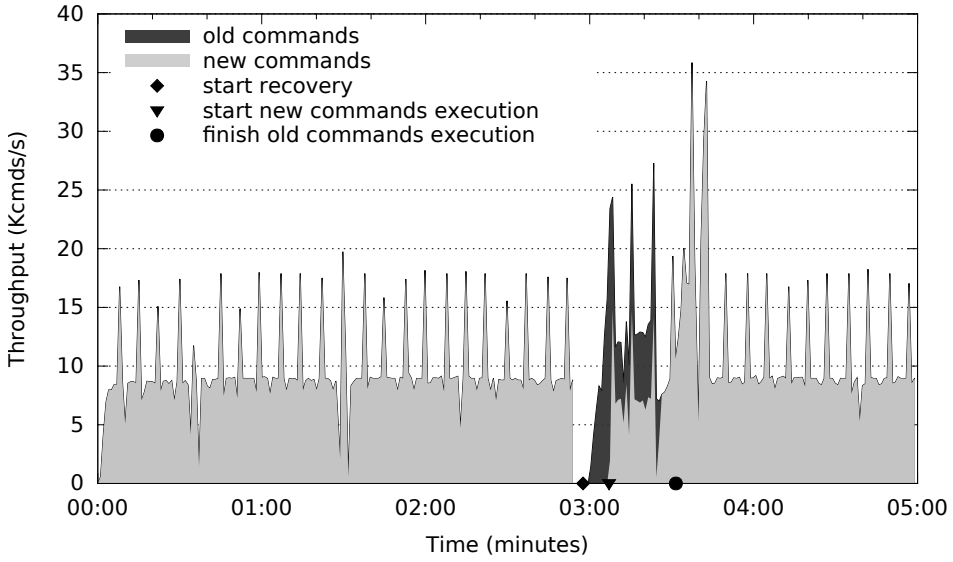


Figure 6.10 – Throughput using speedy recovery and on-demand state transfer with real dependency probability and rescheduling of pending requests enabled.

throughput measured is around 37k commands per second. For the next experiment, we applied approximately 25% of the maximum load.

Figure 6.11 shows the throughput of old and new commands for the aforementioned scenario. As observed, the rescheduling allows parallel execution of old and new commands during recovery. Differently from the scenario with commands size of 128 bytes (see Figure 6.10), where the throughput of new commands during recovery approximates the optimum, workloads with larger commands incur higher processing cost. As a consequence, worker threads queues become higher as well as the dependency probability. Although the throughput of new commands is lower than the throughput of old commands during recovery, the processing of new commands is constant and it allows replicas to recover fast.

Table 6.3 compares recovery techniques for workloads with real dependency probability. The time to download and install a checkpoint in both classical and speedy recovery techniques is similar, although classical recovery has shown a slight advantage. When using on-demand state transfer technique, these times drop considerably. Downloading and installing the segment P4 takes a little more than 3 seconds while downloading and installing a checkpoint in traditional recovery takes around 8 seconds. As expected, speedy recovery and speedy recovery combined with on-demand state transfer improve availability by reducing the recovery time. While new commands are processed after 1 minute and 15 seconds in the classical recovery, speedy recovery starts processing new commands after 40 seconds and this time is reduced to 30 seconds when on-demand state transfer is enabled. Finally, it is possible to observe the average throughput of new command processed during recovery for the proposed techniques. Although speedy recovery demonstrated a superior throughput, around 4.5k commands per second against 2.2k, the processing of new commands during

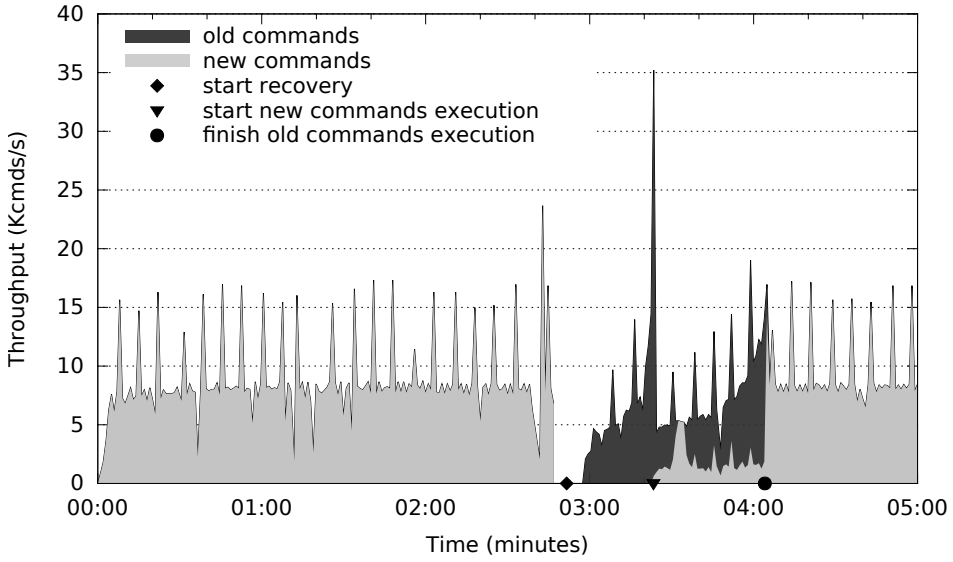


Figure 6.11 – Throughput using speedy recovery and on-demand state transfer with real dependency probability and rescheduling of pending requests enabled (commands size is 1024 bytes).

recovery in speedy recovery approach had a shorter duration. The cumulative number of new commands processed during recovery for speedy recovery was 37392 and for speedy recovery combined with on-demand state transfer was 93023.

Table 6.3 – Recovery techniques comparison for workload with real dependency probability

	Real dependency probability					
	Classical recovery	Speedy recovery	Speedy + on-demand state recovery			
			P1	P2	P3	P4
Checkpoint download duration (s)	3.97	4.20	2.98	3.04	2.85	2.03
Checkpoint installation duration (s)	4.10	4.27	1.23	1.12	1.33	1.06
Old command execution duration (s)	66.25	79.27	68.33	68.34	68.35	67.94
Time for last old command execution	74.32	87.74	72.54	72.50	72.53	71.03
Time for first new command execution	74.32	40.92	31.84	31.77	31.79	30.62
Recovery speedup	**1**	**1.82**				**2.43**
Throughput during normal execution	8987	8900	8941			
New command throughput (recovery)	0	4540	2253			

6.4 Related Work

In this section, we review existing approaches to recovery in classical and parallel state machine replication. We conclude with a brief account of recovery in replicated database systems based on group communication. We first focus on existing approaches to recovery in

classical state machine replication and then consider recovery techniques adapted to parallel models of state machine replication.

6.4.1 Recovery in classical state machine replication

In Chapter 4, we presented the basics of recovery in state machine replication. More advanced techniques have been proposed to improve the efficiency of logging, checkpointing and recovery in SMR (details in Section 4.5). In [BSF+13] three techniques are proposed: parallel logging, sequential checkpointing and collaborative state transfer. The key ideas of parallel logging are to log group operations instead of individual operations, and process operations in parallel with their storage. Grouping operations is conceptually similar to our batched commands. Taking advantage of replication in SMR, sequential checkpointing coordinates replicas such that they do not checkpoint their states at the same time to avoid hiccups during normal execution. While one replica is taking a checkpoint, other replicas continue to process requests. Instead of the traditional state transfer from one single replica, collaborative state transfer proposes that several replicas may send part of their checkpointed state to a recovering replica. These ideas are orthogonal to the ones we propose and could be used in parallel state machine replication. Some works have also discussed how replica recovery can be integrated with group communication primitives [BMPG14] and how to minimize the effects of a recovering replica on normal execution [BPP15].

6.4.2 Recovery in parallel state machine replication

Although the recovery techniques described in the previous section could be used in parallel approaches to state machine replication, some proposals leverage specifics of the protocol to perform checkpoints and recovery efficiently. None of these proposals allow new commands to execute concurrently with old commands or implements on-demand checkpoint transferring.

We have already described CBASE's normal operation in Chapter 3. Checkpointing is briefly discussed in [KD04] and recovery is not mentioned. To ensure that all replicas build the same sequence of checkpoints, a synchronization primitive executed at the replicas, but invoked by the agreement layer, is used 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. As we argued in Chapter 4, checkpointing in Eve seems straightforward, but is not discussed in [KWQ+12]. Two checkpointing mechanisms to P-SMR [MBP14] were presented in Chapter 4 (they also appear in [MMDP14]). Notice, though, that recovery has not been addressed in the literature regarding parallel state machine replication.

In Rex [GHY+14], a single server receives requests and processes them in parallel. While executing, the server logs a trace of dependencies among requests based on the shared variables accessed (locked) by each request. The server periodically proposes the trace for agreement to the pool of replicas. Together it also periodically proposes cuts in the computation, whose corresponding states are saved in a secondary replica. The other replicas receive the traces and replay the execution respecting the partial order of commands. If a cut is provided, a secondary replica, when achieving that cut, creates a snapshot of the state and propagates it to all other replicas. Recovery is performed with the installation of a recent snapshot followed by the replay of the logged commands according to their dependencies. During recovery of a replica, the throughput experienced is around 20% of normal operation and takes around 25 seconds to complete. Strictly, Rex does not implement state machine replication since only one replica executes commands, while the others follow its execution.

6.4.3 Recovery in transactional systems

The problem of efficiently recovering a failed replica has been largely considered in the context of database systems. Replication protocols based on group communication are the closest to our approach in that transactions are ordered before they can be committed. Some of these protocols explicitly address recovery. In [KBB01] the authors discuss how to recover a crashed replica (or start a new one) without stopping transaction processing. The recovered replica only accepts new transactions once recovery has finished. In [JPPMA02] crashed replicas can recover in parallel and at the same time several active replicas can serve them the needed data. The protocol in [LK08] proposes an adaptive approach which allows a recovering replica to catch up with operational replicas by transferring either the recent values of data items or the sequence of missed updates. In these works, transactions can only be processed once old transactions have been recovered. One exception is the approach proposed in [CPPW10] in which new transactions can be executed before recovery has completed. The solution proposed in [CPPW10] builds a data structure that resembles the dependency graph, which is inappropriate in environments subject to very large performance.

7. CONCLUSION

A well-established approach to providing a highly available service is state machine replication (SMR). By replicating a service on multiple servers, clients are guaranteed that even if some replica fails, the service is still available. However, a performance limitation of this approach is the total ordering requirement, which imposes replicas to execute requests in the same order to ensure consistency. To take advantage of multi-core architectures and increase overall throughput, parallel approaches to state machine replication relax the ordering requirement to allow independent commands to be processed in parallel.

This thesis delves into the problem of recovering replicas in the context of parallel state machine replication. Recovering failed replicas quickly is important for availability. If failed replicas recover quickly, their down-time can be substantially reduced. Research on parallel SMR is relatively recent and recovery is rarely addressed in the literature. This thesis reviews parallel variations of state machine replication and discusses how checkpointing and recovery procedures apply to existent proposals. Furthermore, we presented novel checkpointing techniques for P-SMR, an efficient protocol for parallel state machine replication, and the fast recovery approach.

In the following sections, we briefly summary the contributions of this thesis and give some research directions for future work.

7.1 Contributions

We started looking at P-SMR [MBP14], a parallel state machine replication model, whose scalability stems from the absence of a centralized component in the execution path of independent commands (e.g., no local scheduler [KD04]). We proposed two checkpoint techniques for P-SMR: coordinated and uncoordinated. While in the coordinated mechanism any two replicas save identical checkpoints throughout the execution, with uncoordinated checkpoints, replicas do not need to agree upon the same state represented by checkpoints. As showed in our results, by preventing replicas from taking checkpoints under a common and deterministic state, the uncoordinated approach reduces the synchronization cost of checkpointing mechanisms.

We then extended our experiments to evaluate the cost of checkpointing in other parallel SMR approaches. By means of simulation we compare the impact of checkpointing in CBASE [KD04], Eve [KWQ+12], and P-SMR [MBP14]. Our analysis focuses mostly on the synchronization overhead. Although the checkpointing overhead increases with the number of worker threads in all considered techniques, techniques are not affected equally. Both CBASE and P-SMR experience a reduction in performance with checkpoints, but P-SMR

is more vulnerable to checkpoint synchronization. Since Eve requires coordination even in the absence of checkpoints, checkpoint synchronization does not affect its performance. Although the system's performance can be negatively affected by frequent checkpoints, the checkpointing synchronization overhead is sensitive to the checkpointing interval and it can approach the optimal when checkpoints are not so frequent.

A drawback of adopting infrequent checkpoints in parallel state machine replication is the sharp increase in the number of commands logged between two consecutive checkpoints. Large logs may slow down recovery and affect system availability. To address this shortcoming, we propose fast recovery in parallel state machine replication, a set of coordinated techniques to reduce a crashed replica's unavailability period. Speedy recovery is a technique that naturally benefits from command dependencies, allowing new commands to be processed concurrently with old commands, if they are independent. On-demand state recovery enables to recover segments of the state when they are needed instead of recovering the whole state at once. Both techniques have proved to considerably reduce recovery time, when compared to traditional recovery in SMR.

In order to evaluate the fast recovery protocol, we implemented a new parallel state machine replication protocol. Our protocol introduces mechanisms to efficiently represent and calculate dependency among commands, complemented by an efficient scheduling mechanism that considers both dependencies and resource availability (number of working threads) and thus computes dependencies only when useful to parallelize command execution. The computational complexity for scheduling commands with our approach is bounded by the fixed number of worker threads and by the size of bitmaps. Providing low complexity comes at the cost of non-minimal dependency detection, i.e., some requests may be considered dependent although they are not.

7.2 Future Work

With support of bitmaps, we have proposed mechanisms to efficiently represent and calculate dependency among commands. Such mechanisms demonstrated advantages over existent scheduling mechanisms. For instance, updating scheduler dependency information with new command information requires a fixed-size number of comparisons. This constant overhead is not achieved by other schedulers. Although the high throughput achieved by our parallel state machine replication prototype confirms the mechanism efficiency, there are still room for improvements.

While encoding commands into bitmaps, there is no distinction between read and write commands. This means that two read commands accessing the same key are evaluated as dependent. Although this conservative approach does not violate safety, it introduces additional ordering constraints on requests.

As one of the directions for optimizing mechanisms proposed in this thesis, an enhanced approach should distinguish updating from read-only requests while encoding and calculating dependencies. By doing this, read intensive workloads (e.g., name services) would benefit greatly by improving throughput of read-only requests. Moreover, read-only requests do not need to be re-executed during recovery. Hence, the dependency log should keep only information about updating requests.

Another opportunity for improving performance of recovery mechanism is to create a controller mechanism that bounds the load in worker threads queues. In our prototype, the number of batches dispatched to workers queues might be very elevated, especially during recovery. Notice that the dependency probability increases with the number of enqueued requests, and since requests in a thread queue cannot be designated to another thread, then one optimization is to limit the maximum number of requests in a thread queue. The idea is to slow down the parallelizer when threads queues approximate to a threshold. By doing this, worker threads would be kept busy without causing unnecessary dependencies.

REFERENCES

[AK08] Amir, Y.; Kirsch, J. “Paxos for system builders: An overview”. In: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, 2008, pp. 1–6.

[AW04] Attiya, H.; Welch, J. “Distributed Computing: Fundamentals, Simulations, and Advanced Topics”. Wiley-Interscience, 2004.

[BM93] Babaoğlu, O.; Marzullo, K. “Distributed systems (2nd ed.)”. , ACM Press/Addison-Wesley Publishing Co., 1993.

[BMPG14] Benz, S.; Marandi, P. J.; Pedone, F.; Garbinato, B. “Building global and scalable systems with atomic multicast”. In: Proceedings of the 15th International Middleware Conference, 2014, pp. 169–180.

[BPP15] Benz, S.; Pacheco, L.; Pedone, F. “Stretching multi-ring paxos”. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2015, pp. 492–499.

[BSF+13] Bessani, A.; Santos, M.; Felix, J.; Neves, N. F.; Correia, M. “On the efficiency of durable state machine replication.” In: Proceedings of the 24th Conference on Annual Technical Conference, 2013, pp. 169–180.

[Bur06] Burrows, M. “The chubby lock service for loosely-coupled distributed systems”. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, 2006, pp. 335–350.

[CDE+12] Corbett, J. C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; et al.. “Spanner: Google’s globally-distributed database”. In: Proceedings of the 10th Symposium on Operating Systems Design and Implementation, 2012, pp. 251–264.

[CGR07] Chandra, T.; Griesemer, R.; Redstone, J. “Paxos made live: An engineering perspective”. In: Proceedings of the 26th Symposium on Principles of Distributed Computing, 2007, pp. 398–407.

[CKL+09] Clement, A.; Kapritsos, M.; Lee, S.; Wang, Y.; Alvisi, L.; Dahlin, M.; Riche, T. “UpRight cluster services”. In: Proceedings of the 22nd Symposium on Operating Systems Principles, 2009, pp. 277–290.

[CL99] Castro, M.; Liskov, B. “Practical byzantine fault tolerance”. In: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, 1999, pp. 173–186.

[CPPW10] Camargos, L.; Pedone, F.; Pilchin, A.; Wieloch, M. “On-demand recovery in middleware storage systems”. In: Proceedings of the 29th Symposium on Reliable Distributed Systems, 2010, pp. 204–213.

[CRL03] Castro, M.; Rodrigues, R.; Liskov, B. “BASE: Using abstraction to improve fault tolerance”, *ACM Transactions on Computer Systems*, vol. 21–3, 2003, pp. 236–269.

[CT96] Chandra, T. D.; Toueg, S. “Unreliable failure detectors for reliable distributed systems”, *Journal of the ACM*, vol. 43–2, 1996, pp. 225–267.

[CWA+09] Clement, A.; Wong, E. L.; Alvisi, L.; Dahlin, M.; Marchetti, M. “Making byzantine fault tolerant systems tolerate byzantine faults.” In: Proceedings of the 6th Symposium on Networked Systems Design and Implementation, 2009, pp. 153–168.

[CWO+11] Calder, B.; Wang, J.; Ogus, A.; Nilakantan, N.; Skjolsvold, A.; McKelvie, S.; Xu, Y.; Srivastav, S.; Wu, J.; Simitci, H.; et al.. “Windows Azure storage: a highly available cloud storage service with strong consistency”. In: Proceedings of the 23rd Symposium on Operating Systems Principles, 2011, pp. 143–157.

[DLS88] Dwork, C.; Lynch, N.; Stockmeyer, L. “Consensus in the presence of partial synchrony”, *Journal of the ACM*, vol. 35–2, 1988, pp. 288–323.

[DSU04] Défago, X.; Schiper, A.; Urbán, P. “Total order broadcast and multicast algorithms: Taxonomy and survey”, *ACM Computing Surveys*, vol. 36–4, 2004, pp. 372–421.

[EAWJ02] Elnozahy, E. N. M.; Alvisi, L.; Wang, Y.-M.; Johnson, D. B. “A survey of rollback-recovery protocols in message-passing systems”, *ACM Computing Surveys*, vol. 34–3, 2002, pp. 375–408.

[EJZ92] Elnozahy, E. N.; Johnson, D. B.; Zwaenepoel, W. “The performance of consistent checkpointing”. In: Proceedings of the 11th Symposium on Reliable Distributed Systems, 1992, pp. 39–47.

[FLP85] Fischer, M. J.; Lynch, N. A.; Paterson, M. S. “Impossibility of distributed consensus with one faulty process”, *Journal of the ACM*, vol. 32–2, 1985, pp. 374–382.

[FVR97] Friedman, R.; Van Renesse, R. “Packing messages as a tool for boosting the performance of total ordering protocols”. In: Proceedings of the 6th International Symposium on High Performance Distributed Computing, 1997, pp. 233–242.

[GHY+14] Guo, Z.; Hong, C.; Yang, M.; Zhou, L.; Zhuang, L.; Zhou, D. “Rex: Replication at the speed of multi-core”. In: Proceedings of the 9th European Conference on Computer Systems, 2014, pp. 1–14.

[HKJR10] Hunt, P.; Konar, M.; Junqueira, F. P.; Reed, B. “ZooKeeper: wait-free coordination for internet-scale systems”. In: Proceedings of the 21st Conference on Annual Technical Conference, 2010, pp. 1–14.

[HW90] Herlihy, M. P.; Wing, J. M. “Linearizability: A correctness condition for concurrent objects”, *ACM Transactions on Programming Languages and Systems*, vol. 12–3, 1990, pp. 463–492.

[JPPMA02] Jimenez-Peris, R.; Patino-Martinez, M.; Alonso, G. “Non-intrusive, parallel recovery of replicated data”. In: Proceedings of the 21st Symposium on Reliable Distributed Systems, 2002, pp. 150–159.

[KAD+07] Kotla, R.; Alvisi, L.; Dahlin, M.; Clement, A.; Wong, E. “Zyzzyva: speculative byzantine fault tolerance”, *ACM SIGOPS Operating Systems Review*, vol. 41–6, 2007, pp. 45–58.

[KBB01] Kemme, B.; Bartoli, A.; Babaoğlu, O. “Online reconfiguration in replicated databases based on group communication”. In: Proceedings of the 31st International Conference on Dependable Systems and Networks, 2001, pp. 117–126.

[KBC+12] Kapitza, R.; Behl, J.; Cachin, C.; Distler, T.; Kuhnle, S.; Mohammadi, S. V.; Schröder-Preikschat, W.; Stengel, K. “CheapBFT: resource-efficient byzantine fault tolerance”. In: Proceedings of the 7th European Conference on Computer Systems, 2012, pp. 295–308.

[KD04] Kotla, R.; Dahlin, M. “High throughput byzantine fault tolerance”. In: Proceedings of the 34th International Conference on Dependable Systems and Networks, 2004, pp. 575–584.

[KWQ+12] Kapritsos, M.; Wang, Y.; Quema, V.; Clement, A.; Alvisi, L.; Dahlin, M. “All about Eve: execute-verify replication for multi-core servers”. In: Proceedings of the 10th Symposium on Operating Systems Design and Implementation, 2012, pp. 237–250.

[Lam78] Lamport, L. “Time, clocks, and the ordering of events in a distributed system”, *Communications of the ACM*, vol. 21–7, 1978, pp. 558–565.

[Lam98] Lamport, L. “The part-time parliament”, *ACM Transactions on Computer Systems*, vol. 16–2, 1998, pp. 133–169.

[LK08] Liang, W.; Kemme, B. “Online recovery in cluster databases”. In: Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, 2008, pp. 121–132.

[LNP90] Li, K.; Naughton, J. F.; Plank, J. S. “Real-time, concurrent checkpoint for parallel programs”. In: Symposium on Principles and Practice of Parallel Programming, 1990, pp. 79–88.

[MAK13] Moraru, I.; Andersen, D. G.; Kaminsky, M. “There is more consensus in egalitarian parliaments”. In: Proceedings of the 24th Symposium on Operating Systems Principles, 2013, pp. 358–372.

[MBP14] Marandi, P. J.; Bezerra, C. E. B.; Pedone, F. “Rethinking state-machine replication for parallelism”. In: Proceedings of the 34th International Conference on Distributed Computing Systems, 2014, pp. 368–377.

[MDP16] Mendizabal, O. M.; Dotti, F. L.; Pedone, F. “Analysis of checkpointing overhead in parallel state machine replication”. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016, pp. 534–537.

[MMDP14] Mendizabal, O. M.; Marandi, P. J.; Dotti, F. L.; Pedone, F. “Checkpointing in parallel state-machine replication”. In: *Proceedings of the 18th International Conference on Principles of Distributed Systems*, 2014, pp. 123–138.

[MP14] Marandi, P. J.; Pedone, F. “Optimistic parallel state-machine replication”. In: Proceedings of the 33rd International Symposium on Reliable Distributed Systems, 2014, pp. 57–66.

[MPP12] Marandi, P. J.; Primi, M.; Pedone, F. “Multi-Ring Paxos”. In: Proceedings of the 42nd International Conference on Dependable Systems and Networks, 2012, pp. 1–12.

[MPSP10] Marandi, P. J.; Primi, M.; Schiper, N.; Pedone, F. “Ring paxos: A high-throughput atomic broadcast protocol”. In: Proceedings of the 40th International Conference on Dependable Systems and Networks, 2010, pp. 527–536.

[Rod08] Rodeh, O. “B-trees, shadowing, and clones”, *ACM Transactions on Storage*, vol. 3–4, 2008, pp. 2:1–2:27.

[RST11] Rao, J.; Shekita, E. J.; Tata, S. “Using paxos to build a scalable, consistent, and highly available datastore”, *VLDB Endowment*, vol. 4–4, 2011, pp. 243–254.

[Sch90] Schneider, F. B. “Implementing fault-tolerant services using the state machine approach: A tutorial”, *ACM Computing Surveys*, vol. 22–4, 1990, pp. 299–319.

[SFK+09] Singh, A.; Fonseca, P.; Kuznetsov, P.; Rodrigues, R.; Maniatis, P.; et al.. “Zeno: Eventually consistent byzantine-fault tolerance”. In: Proceedings of the 6th Symposium on Networked Systems Design and Implementation, 2009, pp. 169–184.

APPENDIX A – SIMULATION TOOL

We implemented a discrete-event, process-oriented simulation model in C++. The basic elements of our simulation are *processes* and *resources*. *Processes* are active entities that interact with other processes and manipulate structures of our model. They interact with other processes through shared memory or mailboxes, and they may use *resources*. *Resources* represent system's components, such as processing units, storage, etc. Although resources can be shared among different processes, only one process access a resource at a time.

Our model maintains a simulation clock whose value represents the current time in the model. The simulation time is virtual, i.e., there is no relation between the simulation time and the hardware CPU time elapsed during the simulation execution. Simulation time starts at zero and advances unevenly, according to the times at which the state of the model changes. Processes may retrieve the current time for their own purposes (e.g. to collect statistics), but it is not possible for a model to directly assign a value to the simulation clock.

The simulation clock advances exclusively as a consequence of the function `hold` being executed. The function `hold(t)` delays a process for `t` unit times. For example, when a process executes `hold(1.0)`, if there are other processes waiting to run, the calling process will be suspended. Otherwise, time will immediately advance by the specified amount. A process can delay until a specified point in simulated time by calling hold with a parameter value equal to the specified time minus the current time.

Similarly to real processes, a simulation process can be in one of four states: active, ready to run, holding (allowing simulated time to pass), waiting for an event to happen or a resource to become available. Processes in ready state are restarted when the time specified in a hold statement passes. While a process is active, no simulated time passes.

Resources represent a service or resource that can be used by processes. Every resource consists of a server and a single queue (for processes waiting to gain access to the server). Processes access resources in mutual exclusion. All of the waiting processes are placed in the queue until the server becomes available.

Through `reserve(r)` function, processes reserve a given resource `r`. When a process executes a reserve, it either gets use of the server immediately or it is suspended (if the server is busy) and placed in a queue of processes waiting to get use of the server. When a process executes a `release(r)` function, it releases `r`. If there is at least one process in `r` queue, the process at the head of the queue is given access to the server and that process is then reactivated and will proceed by executing statements following its reserve statement.

Figure A.1 exemplifies a simulation of two processes, p_1 and p_2. Process p_1 executes a statement st_1, holds for 0.7 time units, executes another statement (st_2), and attempts to use a resource r. The time when r is allocated by p_1 is represented by the function `hold(1.2)`.

Process p_2 waits 1.0 time units before executes a statement st_a. Since the resource r was not being used at the moment p_1 reserve it, the resource is set as busy and serves process p_1. The diagram illustrates the usage of time during the simulation.

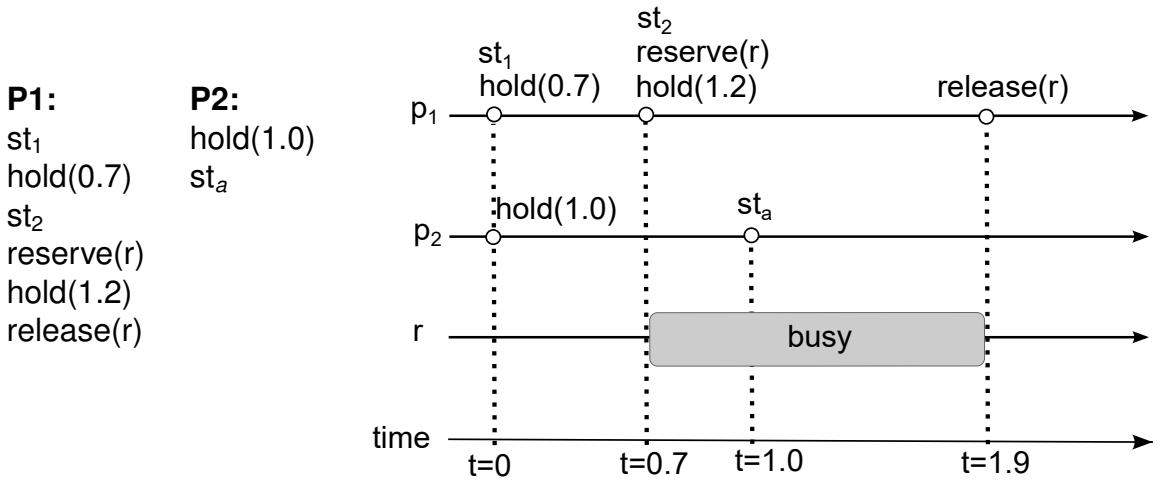


Figure A.1 – Simulation execution example

The communication among processes is given by two distinct mechanisms: shared memory and message passing. The first mechanism allows processes to write and read global variables and it is useful to represent multiple threads or processes accessing some resource at a same node. The second approach mimics the sending and reception of messages by distributed processes. By using a mailbox, the sender process posts a message to be read by the receiver process. A mailbox is a list of messages and it follows a FIFO (First In, First Out) order.

In addition to the aforementioned components, events can be used to synchronize processes operations. Through the primitive `set(e)` processes can signal the occurrence of an event `e`. The primitive `wait(e)` keeps a process blocked until the occurrence of event `e`.

Finally, queuing statistics, such as *queue length*, *utilization*, *throughput*, and *response time* are used in order to analyze resources usage.

A.1 Simulation of parallel SMR models

This chapter introduces the simulation model developed with the aim of analyzing the impact of checkpointing overhead in parallel approaches to SMR. We represented CBASE and P-SMR models by simulating the approaches proposed by Kotla *et al.* [KD04] and Marandi *et al.* [MBP14], respectively. Checkpointing mechanisms behave as described in Chapter 4. Since the traditional SMR model lies in a special case of CBASE and P-SMR where just one thread is available, we did not implement a dedicated model for it.

A.1.1 Load Generation

The load generator module simulates application's clients. Throughout the simulation execution, every client periodically creates and sends new requests to the replicas. Every request contains a command and the client waits the reception of the command output before generate a new request.

Through simulation settings, it is possible to define the total number of clients, a workload ramp up (it is the time between the generation of new clients), and the thinking time (the time taken between the reception of a command output and the generation of a new request by the same client).

Commands are defined as a structure and they contain: an unique identifier, used to differentiate every command; a client id, used to identify the sender of the command; a buffer, that contains the command data; and an execution time, that informs the execution time of the command. The command execution time is set up by the load generator and it can be fixed, uniformly distributed, or exponentially distributed.

The sending of requests by the load generator is performed by the mailbox's primitive send. As it is explained later, in the CBASE model, only one mailbox is used by every replica, since incoming requests are decided by a single sequence of consensus. Every client randomly selects one among the mailboxes available before sending a request. As can be seen in Figure A.2, the client side is composed by a load generator, a set of clients and requests. The workload characterization changes according to the load generator and requests settings.

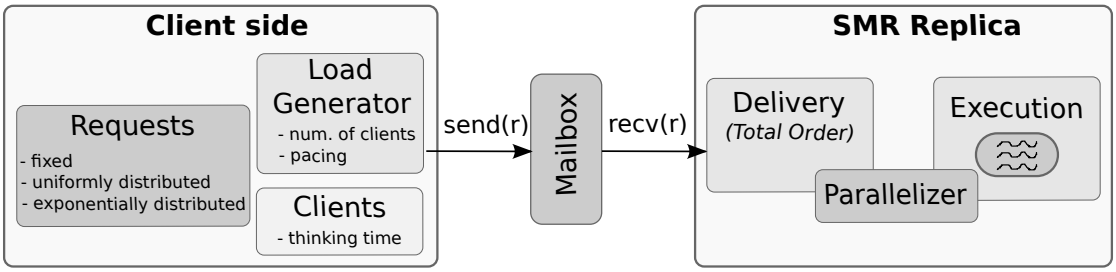


Figure A.2 – CBASE simulation model

In the P-SMR model, incoming requests can be decided by multiple sequences of consensus. Figure A.3 exhibits a P-SMR model, where multiple mailboxes are used to represent the different sequences of consensus. Since messages are added to a mailbox according to a FIFO order, the consensus' agreement requirement is satisfied.

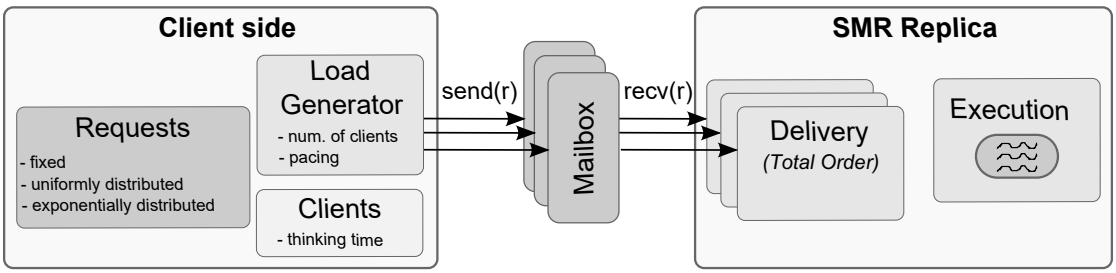


Figure A.3 – P-SMR simulation model

A.1.2 CBASE Model

The CBASE is composed by the delivery module, responsible for reading messages from a single mailbox; a parallelizer, responsible for the dependence analysis and dispatching of requests; and the execution module, composed by multiple threads and responsible for executing requests.

Upon receiving a request, the parallelizer checks if the request depends on other request waiting for processing. For the sake of simplicity, in our simulation model, we did not use a dependency graph as proposed in [KD04] to differentiate dependent and independent commands. Instead, we keep commands in a table.

The parallelizer implements two main primitives: `add_cmd(t_queue *t, int cmd_type, command cmd)` and `command next_cmd(t_queue *t, int cmd_type)`. The first primitive inserts a command `cmd` to a table `t`. The command type `cmd_type` indicates whether a command is independent or depends on other commands. In our experiments, all commands are set up as independent. A table of type `t_queue` is used to store requests delivered by the agreement module. Table's entries have a flag that indicates if their respective commands are being executed by some thread or if they are eligible for dispatching. Commands from different rows are independent, whereas commands in the same row depend each other, i.e., they have to be processed in a sequential order. Therefore, only the first command of each row can be executed.

The primitive `command next_cmd(t_queue *t, int cmd_type)` randomly selects one among all independent commands eligible for dispatching. If none command matches this condition, or table is empty, the primitive returns a nil command.

The execution module is composed by a set of working threads, which are responsible for processing client requests (see Algorithm A.1). Every working thread asks the parallelizer for incoming requests to be processed (line 3). Upon getting a command, a working thread first set a flag indicating that the command is being processed. By setting the command as busy in the table, the thread guarantees that the command will not be dispatched to another thread. Next, the command is executed (lines 7). Then, the work-

ing thread returns the command's output to the client. This is done by setting and event *req_done*[*cmd*.*client_id*]. The client is waiting for this event and as soon as it is set, the client proceeds its execution. Finally, the thread removes the executed command from the table.

Algorithm A.1 CBASE – execution

```
 1: working_thread(thread_id):
 2:    while ¬finish_simulation do
 3:       index ← parallel.next_cmd(cmd)
 4:       if cmd.is_null() then
 5:          hold(busy_waiting)
 6:       else
 7:          set(t, index, busy)
 8:          exec_command(cmd, thread_id)
 9:          set(req_done[cmd.client_id])
10:          remove(t, index)

11: exec_command(cmd, thread_id):
12:    reserve(core[thread_id])
13:    hold(cmd.execution_time)
14:    total_requests + +
15:    release(core[thread_id])
```

The execution of a command is represented by procedure *exec_command(Command cmd, int thread_id)* in Algorithm A.1. It basically reserves the processing unit mapped to the working thread with *thread_id*, keeps the processing unit reserved for *cmd*.*execution_time* time units, increases the number of requests processed, and releases the processing unit.

The checkpointing coordination is managed by the parallelizer. Algorithm A.2 shows the parallelizer procedure. The parallelizer remains blocked waiting for client requests (line 4). Upon receiving a client requests from mailbox, the parallelizer extracts the command carried by the client request, add this command to the table of commands and increment the command counter (lines 5 to 8). When checkpointing is enabled, the parallelizer checks if the last command received is a multiple of *cp_interval*, i.e., if *cp_interval* commands were delivered since the last checkpoint (line 10). In this case, to guarantee that all replicas will take identical checkpoints, the parallelizer waits until all commands in the command table have been executed before taking a checkpoint (line 12). Only the time taken for creation of a checkpoint is represented. The command `hold(cp_duration)` blocks the process for *cp_duration* time units, i.e., the time taken by creation of a checkpoint. Metrics for the number of checkpoints, time elapsed for taking a checkpoint and the number of commands processed per checkpoint are calculated by the parallelizer (lines 15 to 17).

Through a configuration file, it is possible to set up the checkpointing interval (*cp_interval*), checkpoint duration (*cp_duration*), besides informing whether checkpointing is enabled or disabled.

Algorithm A.2 CBASE – parallelizer

```
 1: Initialization:
 2:    cmd_counter ← 0
 3: parallelizer:
 4:    while ¬finish_simulation do
 5:       receive(mailbox, msg)
 6:       cmd ← get_cmd(msg)
 7:       insert(t, cmd)
 8:       cmd_counter + +
 9:       if cp_enabled then
10:          if cmd_counter = (cp_interval × (num_cp + 1) then
11:             measured_cp_duration = clock
12:             while ¬table_is_empty(t) do
13:                hold(busy_waiting)
14:             hold(cp_duration)
15:             num_cp + +
16:             measured_cp_duration ← clock − measured_cp_duration
17:             num_reqs_until_last_cp ← total_requests
```

A.1.3 P-SMR Model

P-SMR parallelizes the *agreement* and the *execution* of commands. Instead of using a single mailbox, P-SMR uses multiple mailboxes. More precisely, there are n mailboxes and n threads at each replica, where thread t_0 (at each replica) receives messages from $mailbox_0$ only.

Commands from different mailbox are independent, whereas commands enqueued in the same mailbox depend each other, i.e., they have to be processed in a sequential order. Since mailbox' receive primitive follows a FIFO order, all replicas process commands from different mailboxes in a same order.

Algorithm A.3 describes P-SMR worker threads executing commands when checkpointing is disabled. Basically, threads receive requests from its mailboxes, extract the request's command, and execute the command. Finally, the worker thread returns the command output to the client. This is done by setting the event *req_done*[*thread_id*]. The client is waiting for this event and, as soon as it is set, the client resumes its execution.

In P-SMR, two types of checkpointing are allowed: coordinated and uncoordinated. In the coordinated checkpointing, replicas must converge to a common state before taking a checkpoint; in the uncoordinated checkpointing, replicas take checkpoints independently and may not be in an identical state when the checkpoint takes place. For a detailed presentation of P-SMR checkpointing algorithms, see Chapter 4.

The simulator is set up through a configuration file. The parameter *sim_duration* specifies a duration threshold to the simulation execution. Workload characterization is de-

Algorithm A.3 P-SMR – execution

```
 1: working_thread(thread_id):
 2:     while ¬finish_simulation do
 3:         receive(mailbox[thread_id], msg)
 4:         cmd ← get_cmd(msg)
 5:         exec_command(cmd, thread_id)
 6:         set(req_done[cmd.client_id])

 7: exec_command(cmd, thread_id):
 8:     reserve(core[thread_id])
 9:     hold(cmd.execution_time)
10:     total_requests++
11:     release(core[thread_id])
```

fined through the parameters *num_clients*, *think_time*, *req_exec_distribution*, and *req_duration*. The number of threads in a replica is given by *num_cores*. The checkpointing strategy is configured through the parameters *checkpoint*, *cp_interval*, and *cp_duration*. Next, we present a sample of configuration file.

```
sim_duration 100000
num_clients 270
think_time 5.0
req_exec_distribution uniform
req_duration 0.05,0.95
num_cores 16
checkpoint uncoordinated
cp_interval 6400
cp_duration 5
verbose 0
logging all
save_file scenario270_uncoord.out
```

APPENDIX B – CORRECTNESS OF PROPOSED ALGORITHMS

This discussion of correctness is made adapting the one of [HW90].

B.1 Correctness of P-SMR (Algorithm 3.1)

In the following, we argue that every execution $\mathcal{E}$ of P-SMR including command invocations and responses is linearizable.

Definition 1 (Command Conflict)**.** Let c_i and c_j be commands, W_i and W_j, R_i and R_j their write- and read-sets, the conflict relation $\#_C \subseteq C \times C$ among commands is given by: $(c_i, c_j) \in \#_C$ if $(W_i \cap W_j \neq \emptyset) \vee (W_i \cap R_j \neq \emptyset) \vee (R_i \cap W_j \neq \emptyset)$.

Definition 2 (History and Subhistory)**.** The execution of a P-SMR is modeled by a *history* which is a finite sequence of command events *delivery*(γ, c) and *reply*(γ, c), respectively representing command request and response, where the arguments represent the consensus sequence and the command, respectively. A subhistory of a history H $|\gamma$, or H_γ, is the subsequence of all events in H for a consensus sequence γ.

Well-formedness: we restrict our discussion to histories where exactly one event reply comes after (possibly not immediately) the matching delivery event. The match is given by events having same arguments. An event follows immediately another event in H if there is no other event between them in H.

Definition 3 (Sequential and Concurrent histories)**.** A history is sequential if each delivery event is immediately followed by its matching reply. A history that is not sequential is concurrent.

Proposition 1. *Given a P-SMR replica with history H, the subhistories H_{γ_i} of the worker threads $t_i, i : 0..n$ are sequential.*

Proof: as can be seen in lines 8 and 11, for every decide (deliver) of valid command there is an immediate reply in the history of t_0; the same is valid for all other threads analogously from lines 20 and 22.

Proposition 2. *Given a P-SMR replica with history H, and subhistories H_{γ_i} of the worker threads $t_i, i : 0..n$, events from t_0 interleave with events from $t_i, i : 1..n$ at the granularity of complete commands, i.e. a subsequence [delivery$(\gamma_0, c) \cdot$ reply(γ_0, c)] appears in H only if the last event of every other thread was a reply.*

Proof: as can be seen in the algorithm, threads $t_i, i : 1..n$ alternate between SQ and CC modes. When they switch to SQ mode it means that each thread has finished a command, and while in SQ mode each thread $t_i, i : 1..n$ synchronizes according to lines 10-12 and 17-18 with t_0 for t_0's isolated execution of a complete command.

Theorem 1. *A P-SMR replica generates linearizable histories.*

Proof sketch: from the definition of linearizability, we must show that there is a sequential history S of commands from H that respects: (i) the real-time ordering of commands across all clients, and (ii) the semantics of the commands.

(i) To address the real-time ordering of commands, we introduce the relation $\prec_H \subseteq C \times C$*, an irreflexive partial order on commands of a history H where,* $\forall c_i, c_j \in H \cdot c_i \prec_H c_j \iff$ *reply*(_, *ci*) *precedes delivery*(_, c_j) *in H. A sequential history S, a linearization of H, is a total order* $<_S$ *compatible with* $\prec_H$*. Since it is always possible to obtain a total order compatible with a partial order, H is linearizable and S obtainable w.r.t. this aspect. In a more concrete level, if reply*(_, *ci*) *precedes delivery*(_, c_j) *in H then* c_i *and* c_j *are not concurrent and their order is fixed in S.*

(ii) The sequential history S, a linearization of H, must also respect the semantics of the commands. Commands may either conflict according to $\#_C$ *or not.*

[a.] To respect command semantics, conflicting commands must be sequentially executed. According to our algorithm, conflicting commands are executed exclusively in t_0 *and by Proposition 1 each thread has a sequential subhistory. Thus, the sequential history S keeps these commands in the same order as in the sequential subhistory.*

[b.] Non-conflicting commands can be executed in different orders without violating their semantics. Threads $t_i, i : 1..n$*, when in CC mode, execute each one a non-conflicting command. Due to their independency, any linearization of these n commands is semantically acceptable in S.*

As discussed in Proposition 2, thread t_0 *only executes a conflicting command after all other threads finished their ongoing non-conflicting commands (CC mode). Therefore H is naturally composed of a sequence of conflicting commands that execute exclusively at the replica, where between any two conflicting commands up to n concurrent commands may execute in threads* $1..n$*. The execution of a batch of (up to) n independent commands can be linearized in S in any possibly way provided they stay between two consecutive conflicting commands in H.*

Proposition 3. *Given a set of P-SMR replicas, all replicas in the set generate a partial order compatible with* $\prec_H$*.*

Proof sketch: All replicas start in identical states and have the same inputs in all consensus sequences. It remains to argue that the replicas take deterministic decisions to order delivery events across consensus sequences. By using a deterministic scheduling procedure, the sequence of delivery events per thread will be the same to all replicas. Note that for every replica there is a succession of same delivery events bounded by two consecutive transitions to SQ mode. A SQ mode starts with the execution of a sequential

command by thread t_0 and it is followed by every thread $t_i, i : 1..n$ processing a command in CC and increasing its round[i] counter by 1. Threads $t_i, i : 0..n$ synchronize for the execution of the next sequential command. This behavior is repeated, ensuring that the set of commands delivered by each replica every two consecutive SQ transitions is the same. Since all variables are deterministically updated, and since any update is triggered by a decision which is homogeneous to all replicas, with the same inputs, they generate the same result.

B.2 Correctness discussion of Coordinated Checkpoint (Algorithm 4.1)

Assuming Algorithm 3.1 is correct, in Algorithm 4.1 we have that in δ time intervals instead of executing sequential command, a checkpoint is generated. Since the CHK msg is decided in consensus sequence γ_0, every replica will take the checkpoint at the same relative moment. Since in t_0 commands are executed sequentially, no other execution will take place during the checkpoint. When the checkpoint finishes the replica resumes the same execution.

B.3 Correctness discussion of Uncoordinated Checkpoint (Algorithm 4.2)

Algorithm 4.2 eliminates synchronization among replicas avoiding a checkpoint to be taken at the same consensus instance. Thus, heterogeneous checkpoints may be taken by different replica. It remains to argue that replicas take valid checkpoints and that this operation does not interfere with the execution model.

Thread t_0 either executes a sequential command or takes a checkpoint. The checkpoint procedure assures that other threads t_i are not executing any command during a checkpoint. If a thread is not executing any command, lines 42-44 define that they block for the execution of a checkpoint (coordinated by t_0). If a thread is executing a *CC* or a *SQ* command, it finishes its execution atomically to process any other signal or decision. Thus the checkpoint is taken between any two complete commands of each thread.

Since a thread executing in *SQ* blocks for t_0's execution, and since t_0 may decide to execute a sequential command or to take a checkpoint, to avoid deadlock t_0 manages the other threads blocking: if t_0 decides to execute a sequential command, other threads are signaled to unblock as in Algorithm 3.1; if t_0 decides to execute a checkpoint then blocked threads (awaiting t_0 to execute a *SQ* command) are not signaled, t_0 takes the checkpoint and let those threads remain blocked until it executes the next sequential command and unblocks all threads. In any case, this is transparent to other threads and the checkpoint is completely taken before the sequential command execution.

Checkpoints are valid since they are taken without concurrent execution and between any two commands of a thread.

The procedure does not interfere with the execution model since it keeps the original synchronization among threads provided by the consensus sequences and alternated execution of *CC* and *SQ* commands.

B.4 Correctness discussion of the proposed parallel SMR (Algorithm 5.1)

Definition 4 (Batch)**.** A *batch* is a sequence of commands. Two batches *non-overlap* if their command sets are disjoint.

Definition 5 (Batch sequence)**.** A *batch sequence* is a pair $(B, <_B)$ where B is a set of non-overlapping batches and $<_B \subseteq B \times B$ is an irreflexive total order. Given a batch sequence and the conflict relation $\#_C \subseteq C \times C$ among commands of batches in B, the derived *batch sequence conflict relation* $\#_B$ is obtained by lifting the conflicts over C to conflicts between batches involving those commands. The *batch sequence dependency relation* $\prec_B$ is the transitive closure of $<_B \cap \#_B$. An *execution* of $\prec_B$ is any total order that is compatible with $\prec_B$.

Proposition 4. *$\prec_B$ is an irreflexive partial order.*

Proof. By construction $\prec_B \subseteq <_B$ is transitive. Since $<_B$ is irreflexive, and antisymmetric, so must be $\prec_B$. □

Definition 6 (Replica)**.** Given a batch sequence $(B, <_B)$, a *replica* R is a finite set of worker threads that execute this batch sequence, that is $R = \{WT_i | 1 \leq i \leq n, n \in \mathbb{N}, WT_i = (B_i, <_{B_i}$) is a batch sequence$\}$ such that:

(i) all batches in B are enqueued to some worker thread: $\bigcup_{1 \leq i \leq n} B_i = B$;

(ii) the queues of worker threads are compatible with the batch sequence order: $<_{B_i} = <_B |_{B_i}$;

(iii) a batch is in multiple threads if and only if it depends on items on these threads: $b \in B_i$ and $b \in B_j$ with $i \neq j \Leftrightarrow (\exists b_i \in B_i, b_j \in B_j . b_i \prec_B b$ and $b_j \prec_B b)$.

Note that since $<_{B_i}$ is an image of $<_B$, the dependency relation on batches of B_i must be $\prec_B$ restricted to the items in B_i.

Discussion of the Implementation. Take Algorithm 5.1. $<_B$ is represented by the total delivery order, as stated in lines 22 to 24. The total order of $<_{B_i}$ is represented by having worker threads processing commands in sequence (lines 26 to 28) and the scheduler appending commands to threads (lines 16 to 20), building a FIFO with commands already enqueued for the thread.

Condition (i), all batches are enqueued to some thread, is granted since every delivered batch (line 22) is scheduled (line 23) and when enqueueing it (lines 16 to 20),

workers is necessarily non empty due to (lines 12 to 16). That is, a delivered batch is certainly enqueued.

Condition (ii) states that the batch sequence of a thread is compatible with the delivery order. Since *schedule* is performed sequentially in the total order of batches delivery, by scheduling each batch to one or more queues of corresponding threads the processing in each thread is compatible with the total delivery order.

Condition (iii), states that a batch is enqueued to multiple threads whenever the batch depends on items in the input queue of those threads. This is ensured by *schedule*. In lines 13 to 15 it checks for all N threads if any already enqueued batches conflict with the current one. In lines 21 to 24 it enqueues B to all threads where dependency is identified. In lines 21 to 24 the *s_map*[*i*] is updated to represent dependency of future batches w.r.t. the current one being enqueued.

Based on the above, Algorithm 5.1 implements the conditions stated. Now we discuss the replica execution model.

Definition 7 (Replica run)**.** Given a replica R with n worker threads WT over a batch sequence $(B, <_B)$, a (complete) *run* π *of replica* R is a list defined inductively as follows ($\bullet$ denotes the inclusion of an element in front of a list):

(i) π is the empty list if all worker threads are empty;

(ii) $\pi = b \bullet \pi'$ if b is the first batch of a set of m worker threads of R. π' is the replica run of the replica R' obtained by removing b from all worker threads in which it appeared.

Discussion of the Implementation.

The basic step is trivial. In the induction step we have that b must be the next batch to be processed in all m threads where it appears. And once it is processed, all m threads remove b and process the next batch. This behavior is implemented in Algorithm 5.1, lines 30 to 44. Every involved worker thread eventually dequeues b and enters a barrier with other threads in *dep_list* (lines 31 and 34–35), only the thread with smallest *id* executes b and all other threads wait for it to complete (lines 36, and 35 and 41). This ensures that the above described execution model is followed.

The next proposition guarantees that a replica run is well-formed, that is, it always possible to obtain a replica run (i.e., a sequence involving all batches) for any replica.

Proposition 5. *Replica run is well-defined (Liveness– it is always possible to obtain a replica run).*

Proof. First, note that if any two batches are in two different worker threads, they must be in the same order, which is the order given in the underlying batch sequence $<_B$*, because the orders in all WTs are restrictions of* $<_B$ *(by Definition 6, item (ii)). This implies that the union of all orders of all worker threads cannot have cycles (because* $<_B$ *is an irreflexive total*

order). Now, since the run is constructed inductively by removing the first batch of one or more worker threads, we have to assure that, if there are non-empty worker threads, it is always possible to remove the first batch of some worker thread. Let b be the batch appearing in some WT (or WTs) of replica R such that b is the lower element according to the batch sequence order $<_B$ that appears in some WT (this element exists since $<_B$ is a total order and B is finite). Then b must be the first batch in all WTs that contain this batch because the local orders of worker threads must be compatible with $<_B$. Therefore this element can be removed according to item (ii) in Definition 7. □

Now, we prove that a replica run is an execution compatible with the partial order describing the dependency relation of a batch sequence.

Theorem 2. *Let R be a replica with n worker threads WT over a batch sequence $(B, <_B)$. Then a run of replica R is an execution compatible with $\prec_B$ (Safety).*

Proof. Let π be a run of replica R. The proof will be done by induction on π:

(i) π is the empty list: since item (i) of Definition 6, the execution is the empty total order when *B* is the empty set.

(ii) $\pi = b \bullet \pi'$: This is the case in which we add a batch *b* in front of a list π'. By induction hypothesis, π' is an execution compatible with the partial order involving the remaining batches. Since *b* is the first batch of a non-empty set of WTs and does not appear in any other WT, there cannot be a batch b' in any WT such that $b' \prec_B b$: if this would be the case either b' would precede *b* in some of the WTs containing *b*, which is absurd because *b* is the first batch in these WTs, or *b* is not present in a WT containing b', which is ruled out by item (iii) in Definition 6 (a batch must follow any other batch it is dependent on). This means that *b* precedes all batches currently in all WTs wrt $\prec_B$ and thus putting it in front of any execution of the rest of the batches will lead to an execution that is compatible with $\prec_B$. □

So far we have discussed that the implementation of one replica *R* executes a possible executions compatible with the partial order given by the dependency relation of a batch sequence.

Proposition 6. *Consistency Across Replicas.*

Two replicas are consistent if, having processed a batch sequence, they converge to identical states.

Proof. Given any two replicas R_i and R_j that process a batch sequence $(B, <_B)$ respectively with runs r_i and r_j, according to Theorem 2 both runs are executions compatible with the batch sequence dependency relation $\prec_B$. Therefore these runs respect the total order of dependent batches while independent batches may be processed concurrently, which leads to identical states. □

Complementary discussion Consistency across replicas is ensured if, given any set of command batches, all dependent batches in this set are executed in the same order in all replicas. Independent batches may be processed in different orders. This stems from the fact that, by definition, commands from independent batches do not interfere with each other. Thus, we focus on how the algorithm ensures that dependent batches are processed in the same order at any two replicas R_1 and R_2. When a command batch B_i is delivered at these replicas, two cases can arise: (1) both replicas have the same set of delivered batches to be executed; and (2) replicas have different sets of batches to be executed, meaning that one replica, say R_1, has already executed some batches before the other replica, R_2. In the first case, B_i will be checked for dependencies against the same set of batches in both replicas, which arrive to the same result and process B_i in the same relative order. In the second case, R_1 has processed some batches that R_2 still holds in its internal queues. Let B_j be one such a batch at R_2. When R_1 delivers B_i, it has already processed B_j. Given that replicas deliver batches in total order, when B_i is delivered at R_2, it is checked against B_j. Here, two possibilities exist: either B_i depends on B_j and then B_i will execute after B_j; or B_i and B_j are independent and can be executed in any order. Thus, replicas process dependent batches in the same order, regardless of the different replicas internal queue states.

B.5 Correctness discussion of Speedy Recovery (Algorithm 6.1)

When discussing classical recovery, the recovering replica has to ensure that it installs the checkpoint, processes missing (old) commands delivered after the checkpoint and before it received the first message; and then processes the new commands (see Chapter 6 for the meaning of old and new commands). To assure that the replica is correct, it has to process these commands in this order, which is the total delivery order. With speedy recovery we take the advantage introduced in the efficient scheduling technique we presented and offer the possibility of concurrently processing new batches if they are independent.

Proposition 7. *Let R be a replica that crashed and recovered during execution over a batch sequence* $(B, <_B)$*, a run of replica R is an execution compatible with* $\prec_B$ *(Recovery safety).*

Proof: Taking $(B, <_B)$, assume three consecutive, disjoint segments of $(B, <_B)$: $(B_c, <_{B_c})$, $(B_o, <_{B_o})$ and $(B_n, <_{B_n})$ where $B_c \cup B_o \cup B_n = B$. B_c contains up to the ith batch according to $<_B$, meaning the checkpoint contents. B_o has the $i+1$th to the jth batch in $<_B$, meaning old batches. B_n contains the $j+1$th to the zth batch meaning new commands. Assuming a correct checkpoint, $\prec_B$ is respected for the segment $<_{B_c}$. In Algorithm 6.1, after restoring a checkpoint (lines 2 to 3), the last delivered instance j is obtained (line 5); $d\_map[i..j]$ obtains dependency log for batches i to j, i.e. batches in B_o. In line 12, either the next batch according to $<_{B_o}$ or to $<_{B_n}$ is delivered. Regarding $<_{B_o}$, its batches come naturally after the checkpoint and can be scheduled (line 14), respecting $\prec_B$. If the batch is the next

one in B_n it's delivery follows the order in $<_{B_n}$ but not $<_B$, since some old batches may be missing. Therefore this batch is checked for dependency with old batches (line 20) and if it is independent from old batches it is scheduled (line 21) as discussed before. If it does depend then it waits for processing in *n_seq* (lines 22 to 24). After all old batches have been processed (line 16) then, according to $<_B$, all new batches waiting in *n_seq* can be scheduled (lines 16 to 18). Therefore, the recovery algorithm follows $\prec_B$ on all batches of a run including crash and recovery.

Proposition 8. *A replica's execution is consistent with the linearizability criterion.*

Proof. *linearizability* states that:

(i) it respects the real-time ordering of commands across all clients. Suppose non overlapping commands c_i and c_j, one command, say c_i is submitted by a client and responded by the service before c_j is submitted by another client. In this case c_i is executed before c_j which only exists in the system after c_i and thus it is not possible to disrespect this order. If c_i and c_j overlap in time, then there is a moment in which both commands were submitted and none of them responded. That is, both commands exist in the system. In such case the ordering protocol decides any (total) order among the commands, which is followed by all replicas. In our specific case, we may choose to further change the order of independent commands but in any case this decision is restricted to the batches not yet responded, and thus possible.

(ii) it respects the semantics of the commands as defined in their sequential specifications. This holds since: (a) non overlapping commands are naturally submitted and responded sequentially; (b) overlapping in time, dependent commands are executed at all replicas in a same order fixed by the ordering protocol; (c) overlapping in time, independent commands can be executed in different moments in different replicas, but leading to the same result since they are independent on the current batches being processed.

□