TRADE-OFFS IN IMPLEMENTING CONSISTENT DISTRIBUTED STORAGE

Nicolas C. Nicolaou

M.S., Computer Science and Engineering, University of Connecticut, 2006
B.S., Computer Science, University of Cyprus, 2003

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2011

Copyright by

Nicolas C. Nicolaou

2011

APPROVAL PAGE

Doctor of Philosophy Dissertation

TRADE-OFFS IN IMPLEMENTING CONSISTENT DISTRIBUTED STORAGE

Presented by

Nicolas C. Nicolaou, M.S., B.S.

Major Advisor ______________________________
Alexander A. Shvartsman

Associate Advisor ______________________________
Alexander Russel

Associate Advisor ______________________________
Aggelos Kiayias

Associate Advisor ______________________________
Chryssis Georgiou

University of Connecticut

2011

TRADE-OFFS IN IMPLEMENTING CONSISTENT DISTRIBUTED STORAGE

Nicolas C. Nicolaou, Ph.D.

University of Connecticut, 2011

Distributed data services use replication to ensure data availability and survivability. With replication comes the challenge of guaranteeing data consistency when multiple clients access the replicas concurrently, and various consistency models have been proposed and studied. Atomicity is the strongest consistency model, providing the illusion that data is accessed sequentially. A basic problem in distributed computing is the implementation of atomic objects (registers) that support read and write operations. This thesis explores the communication costs of atomic read/write register implementations in asynchronous message-passing systems with crash-prone processors. It considers such implementations under three assumptions.

First, we consider implementations in the single writer, multiple reader (SWMR) setting. It is known that under certain restrictions on the number of readers, it is possible to obtain implementations where each read and write operation terminates after a single round-trip message exchange with a set of replicas. Such operations are called fast. This thesis removes any restrictions on the number of readers and introduces a new implementation where writes are fast, and at most one read operation performs two round-trips per write operation. Subsequently, we show that such SWMR implementations impose limitations on the number of replica failures and that multiple writer, multiple reader (MWMR) implementations with such characteristics are impossible.

Then, we consider implementations in the SWMR setting where operations access the replicated register by sending messages to a predefined sets of replicas with non-empty intersections, called quorums. We show that more than one two-round read operation may be required for each write in this setting and that general quorum-based implementations are not fault-tolerant. Then we explore trading operation latency for fault-tolerance and introduce a new decision tool that enables some read operations to be fast in any general quorum construction.

Finally, we examine the latency of read and write operations in the MWMR setting. First, we study the connection between fast operations and quorum intersections in any quorum-based implementation. The decision tools introduced in the SWMR setting are then adapted to the MWMR setting to enable fast read operations. Lastly, the thesis develops a new technique leading to a near optimal implementation that allows (but does not guarantee) both fast read and write operations in the MWMR setting.

ACKNOWLEDGEMENTS

Reaching to the end of a long journey I feel the need, with these final keystrokes, to express my gratitude to the people without whom this journey would have been impossible.

First, I gratefully and sincerely thank Dr. Alexander Shvartsman that believed in me, supported me, and allowed my educational growth from the position of my major advisor. Dr. Shvartsman, thank you for mentoring me and teaching me how to deliver meaningful, valuable and important research. You accepted me as a student and with your guidance, encouragement, and patience you crafted and delivered a young scholar. You will always be my inspiration, mentor, guide, and eternal advisor to any path my fate will lead me. By giving me the opportunity to work with autonomy you helped me become an independent thinker and establish a strong academic foundation on which I can build a successful academic career. I am honored to be your student.

I am also grateful to my associate advisors, Dr. Alexander Russell, Dr. Aggelos Kiayias, and Dr. Chryssis Georgiou. I thank you all for the time and energy you spend into my study and for the valuable feedback, ideas, and enthusiasm that made this study a better product. I was honored to have you all in my committee.

A special thank you goes to Dr. Georgiou for being a great collaborator, mentor and most importantly a great friend. I thank him for the endless discussions we had over problems presented in this work, for his dedication, his patience, his guidance and continuous feedback in every step of my studies.

Equally importantly, I thank my parents, extended family, colleagues and friends in Cyprus and in the US. You all made a foreign country feel like home. Your support and faith in me kept me going forward through this tough process. You will always have my gratitude and a place in my heart.

Most of all I would like to thank my wife Andri and my baby girl Raphaela. Andri, had the endurance to cope with my “crashes” , “restarts”, and my arbitrary behavior, throughout my Ph.D. program. Thank you my love for your patience, tolerance, for being by my side, for pushing me, for supporting me, for believing in me, for loving me. But most of all I thank you for bringing to life our beautiful daughter Raphaela. My little Raphaela, tiny as you are, you gave me the strength, meaning, and endurance to overcome the last and most difficult part of this journey.

Finally, I would like to acknowledge the funding support for this work. The study was partially supported by NSF, the Cyprus Research Promotion Foundation grant ΠΕΝΕΚ/0609/31, and the European Regional Development Fund (ERDF). Special thanks to Drs. Alexander Shvartsman and Chryssis Georgiou, without whom I would have not received funding for this work.

CREDITS

This dissertation incorporates research results appearing in the following publications:

[43, 45] This is a joint work with C. Georgiou and A. Shvartsman that appears in the *Journal of Parallel and Distributed Computing* [45]. A preliminary version [43] appears in the Proceedings of *SPAA'06*. This work is included in Sections 4.1.1, 4.2.1-4.2.4, 4.3 and parts of 4.2.5.

[40] This is a joint work with C. Georgiou, S. Kentros and A. Shvartsman that appears in *PDCS'09*. Parts of Section 4.2.5 correspond to this work.

[44] This paper is a joint work with C. Georgiou and A. Shvartsman that appears in *DISC'08*. This work corresponds to part of Sections 5.2-5.4.

[32, 33] This is a joint work with B. Englert, C. Georgiou, P. Musial, and A. Shvartsman. It appears in *OPODIS'09* and corresponds to parts of Sections 6.2-6.4.

TABLE OF CONTENTS

Chapter 1: Introduction **1**

1.1 Motivation 1

1.2 Background 5

1.3 Thesis Contributions 8

1.3.1 SemiFast Implementations with Unbounded Number of Readers . . . 9

1.3.2 Examining Robustness of (Semi)Fast Implementations 10

1.3.3 Fast Operations in the MWMR environment 12

1.4 Thesis Organization 13

Chapter 2: Related Work **15**

2.1 Consistency Semantics 15

2.2 Consensus 18

2.3 Group Communication Services 19

2.4 Quorum Systems 20

2.5 Consistent Memory Under Crash Failures 23

2.5.1 Fastness under Crash Failures 28

2.6 Consistent Memory Under Byzantine Failures 30

2.6.1 Fastness Under Byzantine Failures 34

2.7 Partitionable Networks 37

Chapter 3: Model and Definitions **38**

3.1 Model of Computation 38

3.1.1 Input/Output Automata and Executions 39

3.1.2 Communication 41

3.1.3 Failures 43

3.1.4 Quorum Systems 45

3.2 Consistency and Object Semantics - Atomic Read/Write Registers 47

3.3 Complexity Measures 50

Chapter 4: Trading Speed for Reader Participation 53

4.1 Fast and Semifast Implementations 53

4.1.1 Semifast Implementations 54

4.2 Semifast Implementation: Algorithm SF 55

4.2.1 Grouping Reader Participants – Virtual Nodes 56

4.2.2 High Level Description of SF 57

4.2.3 Formal Specification of SF 58

4.2.4 Correctness of SF 68

4.2.5 Quantifying the Number of Slow Reads 91

4.3 Limitations of Semifast Read/Write Register Implementations 104

4.3.1 Constraints on the Virtual Nodes and Second Round Communication . 104

4.3.2 Impossibility of Semifast Implementations in MWMR environment . . 115

Chapter 5: Trading Speed for Fault-Tolerance 124

5.1 Revisiting Replica Organization of (Semi)Fast Implementations 125

5.2 On the Fault-Tolerance of (Semi)Fast Implementations 126

5.2.1 Fast Implementations are Not Fault Tolerant 126

5.2.2 SemiFast Implementations are Not Fault Tolerant 133

5.2.3 Common Intersection in Fast and Semifast Implementations 138

5.3 Weak Semifast Implementations 140

5.4 Weak-Semifast Implementation: Algorithm SLIQ 141

5.4.1 Examining Value Distribution – Quorum Views 142

5.4.2 High Level Description of SLIQ 144

5.4.3 Formal Specification of SLIQ 145

5.4.4 Correctness of SLIQ 153

5.4.5 Empirical Evaluation of SLIQ 162

Chapter 6: Trade-offs for Multiple Writers 166

6.1 Introducing Fastness in MWMR model 167

6.2 Inherent Limitations of the MWMR Environment 168

6.3 Enabling Fast Read Operations - Algorithm CWFR 177

6.3.1 Incorporating Prior Techniques – Quorum Views 178

6.3.2 High Level Description of CWFR 179

6.3.3 Formal Specification of CWFR 180

6.3.4 Correctness of CWFR 191

6.4 Server Side Ordering - Algorithm SFW 202

6.4.1 New Way of Tagging the Values 204

6.4.2 High Level Description of SFW 205

6.4.3 Formal Specification of SFW 208

6.4.4 Correctness of SFW 219

Chapter 7: Summary and Future Directions **246**

7.1 Summary 246

7.2 Future Directions 248

7.2.1 Dynamism 249

7.2.2 Byzantine Failures 256

7.2.3 Other Environments 260

Bibliography **261**

Appendix A: SLIQ Simulation: Plots **268**

LIST OF FIGURES

1 External Signature of a Read/Write Atomic Memory Service. 47

2 Virtual Nodes. 56

3 SF_w Automaton: Signature, State and Transitions 60

4 SF_r Automaton: Signature and State 61

5 SF_r Automaton: Transitions 62

6 SF_s Automaton: Signature, State and Transitions 66

7 Scenarios (i) $rInt = 2.3s$, $wInt = 4.3s$, and (ii) $rInt = 4.3s$, $wInt = 4.3s$. . 100

8 Scenario (iii) $rInt = 6.3s$, $wInt = 4.3s$. 100

9 Left: Physical communication between w and the servers in $\phi(wr)$ and $\phi(wr_4)$.
Right: Same communication using block diagrams. 106

10 Execution fragments $\phi(A)$, $\phi(B)$, $\phi(C)$, $\phi(D)$. 110

11 Intersections of three quorums Q', Q'', Q'''. 135

12 Graphical representation of quorums Q_3, Q_4 and Q_5. 139

13 (a) $\mathbf{QV1}$, (b) $\mathbf{QV2}$, (c) $\mathbf{QV3}$ with incomplete write, (d) $\mathbf{QV3}$ with complete write. . 143

14 SLIQ_w Automaton: Signature, State and Transitions 147

15 SLIQ_r Automaton: Signature, State and Transitions 150

16 SLIQ_s Automaton: Signature, State and Transitions 152

17 Simple runs using Crumbling Walls 164

18 CWFR_w Automaton: Signature, State and Transitions 182

19 CWFR_r Automaton: Signature and State 185

20 CWFR_r Automaton: Transitions 186

21 CwFr$_s$ Automaton: Signature, State and Transitions 189

22 Traditional Writer Side Ordering Vs Server Side Ordering 203

23 SFW$_s$ Automaton: Signature, State and Transitions 210

24 SFW$_w$ Automaton: Signature, State and Transitions 213

25 SFW$_r$ Automaton: Signature, State, and Transitions 216

26 Simple runs 269

27 Crumbling Walls - Quorum Diversity Runs 270

28 Matrix - Quorum Diversity Runs 271

29 Crumbling Walls - Failure Diversity Runs ($cInt \in [0 \ldots 50]$) 272

30 Matrix - Failure Diversity Runs ($cInt \in [0 \ldots 50]$) 273

31 Crumbling Walls - Failure Diversity Runs ($cInt \in [10 \ldots 60]$) 274

32 Matrix - Failure Diversity Runs ($cInt \in [10 \ldots 60]$) 275

LIST OF SYMBOLS

$\mathcal{I}$	set of process identifiers	38
$\mathcal{W}$	set of writer identifiers in $\mathcal{I}$	38
$\mathcal{R}$	set of reader identifiers in $\mathcal{I}$	38
$\mathcal{S}$	set of replica host (server) identifiers in $\mathcal{I}$	38
p	any reader, writer or server process	39
s	a server process	39
r	a reader process	39
w	a writer process	39
ρ	a read operation	39
ω	a write operation	39
π	a read or write operation	39
A	automaton of an algorithm	39
A_p	automaton assigned to process p for algorithm A	39
$states(A)$	set of all states of automaton A	40
$actions(A)$	set of all actions of automaton A	41
σ	a state of the automaton A	41
$\sigma[p]$	the state of the automaton A_p in the state σ of A	41
$\sigma[p].var$	value of variable var in the state $\sigma[p]$ of automaton A_p	41

$\langle \sigma, \alpha, \sigma' \rangle$ — a transition (or step) from the state σ of A to the state σ' of A as a result of action α in $actions(A)$ 41

$trans(A)$ — set of all transitions (steps) of A 41

ϕ — an execution fragment which is a finite or infinite sequence of steps of A 41

ξ — an execution of A which is an execution fragment that begins with the initial state of A 41

$execs(A)$ — set of all executions of automaton A 41

$\xi | A_p$ — an execution in $execs(A_p)$ which is extracted from the the execution ξ of A and contains all the steps $\langle \sigma, \alpha, \sigma' \rangle$ such that $\langle \sigma[p], \alpha, \sigma'[p] \rangle$ is contained in $trans(A_p)$ 41

$m(\pi, c)_{p,p'}$ — message sent from p to p' during round c of operation π 43

$m(\pi, c)_{p,p'}.var$ — value of variable var contained in the message $m(\pi, c)_{p,p'}$ 43

$cnt(\pi, \mathcal{G})_p$ — p sends and receives messages (contacts) to every process of a set $\mathcal{G} \subseteq \mathcal{I}$, for operation π 43

$scnt(\pi, \mathcal{G})_p$ — p contacts every process in $\mathcal{G} \subseteq \mathcal{I}$ and no other process during operation π 43

f — number of maximum replica host failures 44

$\mathbb{Q}$ — a quorum system 45

Q — a quorum in $\mathbb{Q}$ 45

$\mathbb{Q}^i$ — set of i quorums from $\mathbb{Q}$ 45

V set of all allowed to be written on the register 47

$inv(\pi)$ the step in an execution $\xi$where the unique operation π is invoked . 47

$res(\pi)$ the step in an execution $\xi$where the unique operation π completes . . 47

$\sigma_{inv(\pi)}$ state of A before $inv(\pi)$ step of π 48

$\sigma_{res(\pi)}$ state of A after the $res(\pi)$ step of π 48

$\mathcal{V}$ set of all virtual nodes 56

ν_r virtual identifier assigned to process r 56

Chapter 1

Introduction

This dissertation investigates latency-efficient algorithms for consistent and fault-tolerant distributed storage. In this chapter, we first present the motivation for this work and the challenges for implementing atomic data services in distributed message-passing asynchronous and failure-prone systems. To survey the current results, we review the research in this area. Next, we identify open problems derived from previous works and present the research contributions of this dissertation.

1.1 Motivation

Availability of network storage technologies (e.g., SAN, NAS [46]) and cheap commodity disks increased the popularity of reliable distributed storage systems. To ensure data availability and survivability, such systems replicate the data among multiple basic storage units – disks or servers. A popular method for data replication and maintenance uses redundant arrays of independent disks (RAID) [21, 80]. Although a RAID system may sometimes offer both performance boosting and data availability, it usually resides in a single physical location, is

controlled via a single disk controller, and is connected to the clients via a single network interface. Thus, this single physical location with its single interface constitutes a single point of failure and a performance bottleneck. In contrast, a distributed storage system implements reliable data storage by replicating data in geographically dispersed nodes, ensuring data survivability even in cases of complete site disasters. Researchers often focus on implementing abstract objects that allow primitive operations, like read and write registers. Read/write registers can be used as building blocks for more complex storage systems or to directly implement file storage systems, making them interesting in their own right.

A distributed read/write register implementation involves two distinct sets of participating entities: the *replica hosts* and the *clients*. Each replica host maintains a copy of the replicated register. Each client is a *reader* or a *writer* and performs *read* or *write* operations on the register, respectively. In the message-passing environment, clients access the replicated register by exchanging messages with the replica hosts. A reader performs a read operation as follows: (i) accepts a read request from its environment, (ii) exchanges messages with the replica hosts to obtain the value of the register, and (iii) returns the value discovered to the environment. Similarly, a writer performs a write operation as follows: (i) accepts a value to be written on the register, (ii) exchanges messages with the replica hosts to write this value on the register, and (iii) reports completion to the environment.

Replication allows several clients to access different replicas of the register concurrently, leading to challenges in guaranteeing replica consistency. To define the exact operation guarantees in situations where the register can be accessed concurrently, researchers introduced different *consistency models*. The strongest consistency model is *atomicity* that provides the illusion

that operations are performed in a sequential order, when in reality they are performed concurrently. In addition to atomicity, atomic register implementations must ensure *fault-tolerance*. That is, any operation that is invoked by a non-faulty client terminates, despite the failures in the system.

Two obstacles in implementing an atomic read/write register are *asynchrony* and *failures*. A *communication round-trip* (or simply round) between two participants A and B, involves a message sent by A to B, then a message sent by B to A. Due to asynchrony, every message sent between two participants experiences an unpredictable *communication delay*. As a result, a communication round-trip involves two communication delays. To obtain the value of the register during a read operation, a reader requires at least one round and thus two communication delays for: (a) delivery of a read message from the reader to at least a single replica host, and (b) delivery of the reply from the replica host to the reader. Similarly, to modify the value of the register during a write operation, a writer requires at least one round and thus two communication delays for: (a) delivery of a write message from the writer to at least a single replica host, and (b) delivery of an acknowledgment from the replica host to the writer. Although the writer may enclose the value to be written in its write message, the write operation cannot terminate before receiving an acknowledgment from the replica host. In fact, this could lead to the termination of the write operation before the replica host receives the write message, either due to delay or due to replica host failure. In any case, atomicity may be violated as a subsequent operation will be unaware of the existence of the write operation. Consequently, both read and write operations require at least *two* communication delays, that is, a *single round* before terminating. We refer to operations that terminate after their first round as *fast*.

Fault-tolerance is not guaranteed if an operation communicates with a single replica host. A crash failure may prevent the delivery of messages to that host, keeping clients waiting for a reply and preventing them from terminating. Additionally, if two operations communicate with different replica hosts, they may observe different replica values, thus atomicity may be violated, as the second operation may return an older value than the one written or read by the first operation. Therefore, a client needs to send messages to a *subset* of replica hosts. To tolerate failures, such a subset should contain more replica hosts than the maximum number of allowed replica host failures. Moreover, to ensure that operations are aware of each other they must obtain information from overlapping subsets of replica hosts.

Communicating with overlapping subsets of replicas may be insufficient to guarantee atomicity. Suppose a write operation communicates with a subset A and a succeeding read operation with a subset $B \neq A$ where $A \cap B \neq \emptyset$. The read operation obtains the value written from the replica hosts in the intersection $A \cap B$. As the read succeeds the write, it returns the value written. Consider, a different scenario where the write operation is delayed and communicates only with the replica hosts in $A \cap B$ before the read communicates with the replica hosts in B. The read operation cannot differentiate the two scenarios and thus returns the value being written. A second read operation may communicate with a subset C, such that $A \cap C \neq \emptyset$, $B \cap C \neq \emptyset$, and $A \cap B \cap C = \emptyset$. Thus, the read is not aware of the delayed write and hence returns an older value, violating atomicity. To ensure that any succeeding operation observes the written value, the first read operation can either: (i) ensure that the written value is propagated to enough replica hosts by waiting for hosts not in A to reply, or (ii) propagate the value to a subset of replica hosts that overlaps with the subset obtained by any subsequent operation. As hosts not in A may crash, waiting for more replies may prevent the read operation from

terminating. So it remains for the read operation to perform another round to propagate the written value. As a result, atomic register implementations may contain operations that may experience four communication delays before terminating.

In general the efficiency of atomic read/write register implementations is measured in terms of the latency of read and write operations. The latency of an operation is affected by two factors: (a) *communication*, and (b) *computation*. In this thesis we focus on the operation latency caused by communication delays and study the number of rounds needed for each read and write operation.

1.2 Background

A distributed storage algorithm is characterized by the number of writer and reader clients it supports e.g., single writer multiple reader (SWMR) and multiple writer multiple reader (MWMR), and the type of participant failures it tolerates (e.g., crash failures, arbitrary failures, etc.).

A seminal work by Attiya, Bar-Noy, and Dolev [9] gives an algorithm for a SWMR atomic register implementation in the asynchronous, crash-prone, message-passing environment. In their solution, the register is replicated among a set $\mathcal{S}$ of replica hosts. Each read or write operation is guaranteed to terminate as long as less than $\frac{|\mathcal{S}|}{2}$ replica hosts crash. Each value written to the register is associated with a natural number, called *timestamp*, that is used by the read operations to determine the latest value of the register. The writer issues the timestamps. At each write operation the writer increments its local timestamp and conveys the new timestamp-value pair to a majority of the replica hosts in a single communication round. The read protocol requires two rounds to complete; in the first round it discovers the maximum timestamp-value

pair, and in the second round it propagates this pair to a majority of replica hosts. Although the value of the read is established in the first round, skipping the second round may lead to violations of atomicity when the read is concurrent with a write. Subsequently, a folklore belief developed that "reads must write" in multi reader atomic register implementations.

Lynch and Shvartsman in [68] generalized the majority-based solution of Attiya et al. [9] and gave a *quorum-based* atomic register implementation for the MWMR environment. In their context, a *quorum system* is a collection of sets of replica hosts, known as *quorums*, with pairwise intersection. To support multiple writers, they introduced a *two* round write protocol, while they preserved the two round read protocol of [9]. The first round of the write protocol was used to discover the latest value of the register. Yet, this was unnecessary in the single writer environment, since the sole writer imposed a total ordering on the write operations. To improve the longevity of the service, this work was the first to suggest and implement the *reconfiguration* of the quorum system. To ensure safety of reconfigurations, the protocol prevented the invocation of read or write operations during reconfigurations.

Englert and Shvartsman in [34] overcame the problem of [68] by restarting any read or write operations that detect an in-progress reconfiguration in the system. Building on this finding Lynch and Shvartsman in [66] provided the first algorithm that implemented a MWMR atomic register in a fully *dynamic* setting. Their solution preserved atomicity while allowing participants to join and fail by crashing. The authors allow multiple reconfigurers circumventing the failure of the single reconfigurer used in [34]. Both read and write operations needed to perform at least two round protocols before completing.

Dolev et al. [28] extended the work presented in [66] and introduced a MWMR read/write atomic register in partitionable ad-hoc mobile networks. The authors in [28] assume that ad-hoc mobile nodes usually populate distinct geographic locations, they called *focal points*. Each focal point is implemented as a virtual node that participates in the service. Individual mobile nodes invoke read and write operations on the atomic register, via the focal points they resided in. Interestingly, this work was the first to introduce single round reads when it was confirmed that the latest value was propagated to at least a single quorum.

A work in 2004 by Dutta et al. [30] was the first to introduce implementations of atomic read/write registers in the SWMR environment where all operations required a *single* round to complete. Such operations are called *fast*. This finding refuted the folklore belief that "reads must write". The same paper proved that such an efficient behavior is possible only when the number of readers is inversely proportional to the number of replica hosts in the system. So, while the first part of the work presents the possibility of practical atomic read/write register implementations, the proved constraint raises major questions on the scalability of the service. In addition, the authors show that fast implementations cannot exist in the MWMR environment.

The work in [30] demonstrates that it is possible to obtain fast atomic register implementations in systems where processes may fail by crashing. It is yet unknown if such a performance may be achieved when participants may suffer arbitrary failures, such as *Byzantine* failures [64]. Martin, Alvisi, and Dahlin in [73] were the first to implement atomic semantics assuming byzantine replica hosts and without the use of replica host authentication. Their work applies diffusion techniques to propagate and discover an *acceptable* value written on the atomic register. As a result, the communication cost of their approach is high. Guerraoui and Vukolić

in [53] developed an algorithm that allowed fast operations despite the existence of byzantine replica hosts. Their approach relies on eventual-synchrony. A read or write operation could terminate in a single round only if “enough” replica hosts replied within a predefined time interval. If the specific number of replies could not be collected within this interval, an operation had to proceed to a second and potentially third round to complete.

1.3 Thesis Contributions

This thesis aims to answer the following general question:

What is the operation latency of algorithms that implement atomic read/write register abstraction in an unconstrained, fail-prone, message-passing, asynchronous distributed system?

Our work focuses on systems with static participation that allow participants to crash. We study the operation latency, in terms of the number of communication rounds required by each read and write operation. Both SWMR and MWMR settings are considered. We establish that the communication delay of each operation is affected by: (1) the number of reader and writer participants, and (2) the subset of replica hosts that each client communicates with per read/write operation. We developed four algorithms that implement atomic read/write registers. Two of them are designed for the SWMR setting, and two are designed for the MWMR setting. Each of the algorithms contains operations that need one or two rounds to terminate.

1.3.1 SemiFast Implementations with Unbounded Number of Readers

Motivated by the result of Dutta et al. in [30], the first part of my thesis investigates the possibility and cost of efficient implementations of atomic read/write registers that support arbitrarily many readers.

Our work builds upon the result in [30]. To allow unbounded reader participation, we group the readers into abstract entities, called *Virtual Nodes*. Each virtual node serves as an enclosure for multiple reader participants. Adapting the techniques presented in [30], each virtual node is treated as a separate participating entity, allowed to perform read operations. This allows unbounded number of readers in each virtual node. Such a refinement raises challenges, especially in regards to maintaining consistency among readers of the same virtual node. Based on this idea, we develop our first atomic read/write register implementation, called SF, that requires *some* read operations to perform two rounds to terminate. In particular, at most a *single complete* read operation performs two rounds for each write operation. Writes and any read operation that precedes or succeeds a two-round read, is fast. This discovery leads to the definition of a new class of implementations, called *semifast* implementations.

Next, we ask whether the operation latency of reads and writes is affected by the number of virtual nodes. We show that semifast implementations are possible only if the number of virtual nodes is less than $\frac{|\mathcal{S}|}{f} - 2$, where f out of $|\mathcal{S}|$ replica hosts are allowed to crash. Notice that such a bound does not restrict the number of readers, as long as a single virtual node exists in the system. Moreover, semifast implementations are feasible only if each read operation sends messages to a subset of at least $3f$ replica hosts during the second round.

Then, we ask whether semifast implementations are possible in the MWMR environment. We obtain a negative answer, and show that semifast implementations are impossible in the simplest MWMR environment that contains 2 writers, 2 readers and encounters a single replica host failure.

Lastly, given that semifast implementations do not guarantee that read operations concurrent with a two-round read are fast, we prove that the number of slow reads per write is logarithmic in the number of readers $|\mathcal{R}|$ and does not grow larger than $|\mathcal{R}|$. Empirical experiments we conduct on single processor simulators and planetary scale environments agree with our theoretical findings.

1.3.2 Examining Robustness of (Semi)Fast Implementations

Fast implementations studied in [30] and the first part of this thesis, rely on the knowledge of the number of replica failures. That is also the case for the exploration of fast implementations under byzantine failures as studied in [53]. These works do not describe how the replica host *access strategies* may affect the fastness of the read or write operations. An access strategy specifies the subsets of replica hosts that each operation may communicate with and can be constructed by either: (a) knowing the maximum number of allowed failures[1] , or (b) explicitly specifying the subsets of replica hosts. Quorum systems belong to the latter category of access strategies, and have been widely used by atomic read/write implementations. A quorum system is a set of subsets with pairwise intersection. Here, we seek the conditions that are necessary and sufficient to enable fast read and write operations in systems that assume arbitrarily many

[1]In this case the access strategy requires an operation to communicate with all but f replica hosts, where f the maximum number of allowed failures

participants and general quorum constructions. In other words, we investigate the fastness of quorum-based implementations.

First, we examine the fault-tolerance of fast and semifast quorum-based implementations. Interestingly, we discover that fast and semifast implementations are only possible if a *common intersection* exists among *all* quorums of the quorum system they use. This implies that such constructions are not fault-tolerant since a single failure in the common intersection renders the entire quorum system unusable.

So, we explore trading efficiency for fault-tolerance. We focus on techniques that allow more than a single slow read operations per write, and enable some fast operations in a general and unconstrained quorum-based environment. Our investigation led to the introduction and development of new client side decision tools, called *Quorum Views*. Such tools do not depend on the underlying quorum construction, and this makes them suitable for use with any general quorum system. To establish the latest written value, quorum views examine the distribution of a value in the members of a quorum. Using quorum views, we develop an atomic read/write register implementation for the SWMR environment, called SLIQ. The new algorithm allows all writes to be fast, while reads perform one or two round. In contrast with fast and semifast implementations, SLIQ allows *multiple* complete two-round reads per write operation. This characteristic formed a new distinct class of implementations which we call *Weak-Semifast*. Experimental results indicate that the operation latency of implementation SLIQ is very close to the operation latency of semifast implementation SF in realistic scenarios.

1.3.3 Fast Operations in the MWMR environment

Thus far we considered the impact of unconstrained constructions – in terms of reader participation and replica host access strategy – on the efficiency of atomic read/write register implementations. Next we explore whether fast operations are possible in systems with multiple writers.

Traditional solutions for the MWMR environment (e.g., [34, 66, 68]) demonstrate that two rounds (four communication delays) are sufficient for any read or write operation. More recently, [30] showed that it is impossible to obtain fast atomic register implementations for the MWMR environment. Yet, it is not known whether algorithms that allow *some* fast operations may exist. A partial answer to this question was given by [28] that allow single round read operations in the MWMR environment, whenever a read was *not concurrent* with a write operation. No solution however, enabled single round write operations. Hence, a belief was shaped that *"writes must read"* before writing a new value to the register in a multi writer environment.

We show that it is unnecessary for writers to read, by devising algorithms that implement an atomic read/write register in the MWMR environment and allow both reads and writes to be fast. This is currently the *first* known solution that allows fast write operations. Moreover, our solution overcomes the shortcomings of previous approaches, and allows fast read operations when those are concurrent with a write operation. Our results assume and employ general quorum constructions.

First, we formally define the notion of the *intersection degree* for a quorum system: a quorum system has intersection degree n (also called n-wise quorum system), if every n quorum members of this system have a non-empty intersection. Given this definition we show that if

a MWMR atomic register implementation deploys an *n*-wise quorum system, then it can only allow up to $n-1$ consecutive fast write operations in any execution.

Driven by this finding, we initially adjust and use *Quorum Views* – algorithmic techniques presented in the SWMR model – to enable fast operations. Such tools yield a new algorithm, called CWFR, that allows some fast read operations, but does not allow fast write operations.

In order to enable fast write operations we introduce a new value ordering technique we call *server side ordering* (SSO). As implied by its name, SSO transfers partial responsibility of the ordering of write operations to the replica hosts. Atomicity requires that write operations are totally ordered. However, two replica hosts may receive messages sent within two write operations in reverse order, and may order the writes according to the order each of them received the write messages. As a result, operations that communicate with these two hosts may observe a different ordering for the same write. To establish a single ordering for each write operation we combine the global ordering imposed by the servers with a local ordering established by each writer participant. If "sufficient" number of servers assign the same order to a write operation then the desired total ordering is ensured. Using this technique we obtain a MWMR atomic register implementation, called SFW, with fast read and write operations. This implementation is near optimal in terms of the number of successive fast operations it allows.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 reviews relevant literature. Chapter 3 defines the model of computation, the terminology, and the notation that is used throughout this thesis. Chapter 4 introduces semifast implementations, presents algorithm SF, and provides the theoretical bounds of such implementations. Chapter 5 studies the robustness of fast

and semifast implementations, and introduces quorum views along with algorithm SLIQ which does not depend on any reader or construction constraints. Finally, we study the operation latency of atomic register implementations under the MWMR environment in Chapter 6, where we develop algorithms CWFR and SFW. We conclude in Chapter 7.

Chapter 2

Related Work

This chapter presents the current research in distributed systems regarding implementations of consistent distributed read/write (R/W) storage objects. We begin with an overview of *consistency semantics* in Section 2.1. Then, we provide an overview of the *consensus* problem in Section 2.2 and *group communication services* in Section 2.3 and discuss how they can be used to implement consistent data services. In Section 2.4 we talk about *Quorum Systems*. In Sections 2.5, 2.6, and 2.7 we discuss implementations that establish consistent distributed storage in message-passing, failure prone, and asynchronous environments.

2.1 Consistency Semantics

Lamport in [62], defined three consistency semantics for a R/W register abstraction in the SWMR environment: *safe*, *regular*, and *atomic*.

The *safe register* semantic ensures that if a read operation is not concurrent with a write operation, it returns the last value written on the register. Otherwise, if the read is concurrent with some write, it returns any arbitrary value that is allowed to be written to the register. The

latter property renders this consistency semantic insufficient for a distributed storage system: a read operation that is concurrent with some write may return a value that was *never* written on the register.

A stronger consistency semantic is the *regular register*. As in the safe register, regularity ensures that a read operation returns the latest written value if the read is not concurrent with a write. In the event of read and write concurrency, the read returns either the value written by the last preceding write operation, or the value written by the concurrent write. In any case, regularity guarantees that a read returns a value that is written on the register, and is not older than the value written by the read's last preceding write operation.

Although regularity is sufficient for many applications that exploit distributed storage systems, it does not provide the consistency guarantees of a traditional sequential storage. In particular, it does not ensure that two read operations overlapping the same write operation will return values as if they were performed sequentially. If the two reads do not overlap then regularity allows the succeeding read to return an older value than the one returned by the first read. This is known as new-old read inversion. *Atomic semantics* overcome this problem by ensuring that a read operation does not return an older value than the one returned by a preceding read operation. In addition, it preserves all the properties of the regular register. Thus, atomicity provides the illusion that operations are ordered sequentially.

Herlihy and Wing in [56] introduce *linearizability*, generalizing the notion of atomicity to any type of distributed object. That same paper presented two important properties of linearizability: *locality* and *non-blocking*. These properties distinguish linearizability from correctness conditions like *sequential consistency* by Lamport in [61] and *seriazability* by Papadimitriou

in [78]. An in-depth comparison between sequential consistency and linearizability was conducted by Attiya and Welch in [10]. As defined in [56], a property P of a concurrent system is *local* if the system satisfies P whenever each individual object satisfies P. Thus, locality allows a system to be linearizable as long as every individual object of the system is linearizable. Non-blocking allows processes to complete some operation without waiting for any other operation to complete. *Wait-freedom*, is stronger than non-blocking, and is defined by Herlihy in [55]: any process completes an operation in a finite number of steps regardless of the operation conducted by other processes. While wait-freedom ensures non-blocking on an operation level, weakest non-blocking progress conditions guarantee only that *some* (and not *all*) operation complete in finite number of steps (*lock-freedom*) or require conflicting operations to abort and retry (*obstruction-freedom*). Both non-blocking and locality properties enhance concurrency. Also, locality improves modularity (since every object can be verified independently), and non-blocking favors the use of linearizability in time critical applications.

Subsequent works revisited and redefined the definitions provided in [62, 56] for more specialized distributed systems. Lynch in [65] provided an equivalent definition of atomicity of [62] to describe atomic R/W objects in the MWMR environment. The new definition, totally orders write operations, and partially orders read operations with respect to the write operations. Shao et al. in [86], extended the definition of regularity of [62] to the MWMR environment and presented three possible definitions, they called *MWR1*, *MWR2* and *MWR3*. The weakest definition (*MWR1*) does not impose any ordering on the overlapping write operations. The definition *MWR2* is stronger, requiring that two reads must perceive the same ordering for all the writes that do not strictly succeed them. The last and strongest definition, *MWR3*, requires that two reads by the same reader preserve the order of the overlapping write

operations. If a single writer is used in the system, *MWR1* and *MWR2* are equivalent to the regularity definition given in [62]. *MWR3* is stronger than Lamport's definition for regularity but is weaker than atomicity. Notice however that *MWR3* is equivalent to atomicity in the case of a single reader.

2.2 Consensus

One obvious way to implement safe, regular or atomic registers (see Section 2.1), is to allow processes to reach agreement on a global event ordering. The fundamental problem in distributed computing that examines how a number of independent processing entities can agree on a common value is *consensus* [63]. The consensus problem requires that the following three properties are satisfied:

Agreement: All correct processes decide the same value.

Validity: The value decided was proposed by some process.

Termination: All correct processes reach a decision.

Multiple papers solve consensus for synchronous systems with failures (e.g., [64, 79, 91]). A breakthrough work by Fischer, Lynch, and Paterson in [38], proves that it is *impossible* to achieve agreement in a totally asynchronous system with a single crash failure. This impossibility result does not state that consensus can never be reached: merely that under the model's assumptions, no algorithm can always reach consensus in bounded time. In order to solve consensus, many studies tried to circumvent asynchrony by modifying the system model. Examples include Randomization (e.g., [83]), Failure Detectors (e.g., [20, 69]), Partial-Synchrony (e.g., [7, 31, 26]), and Wormholes (e.g., [24, 77]).

We note that achieving consensus is a more difficult problem than implementing atomic R/W objects. As shown by Herlihy in [55], consensus can not be solved for two or more processes by using atomic R/W registers.

2.3 Group Communication Services

Another way to achieve consistency in a distributed storage is to globally order the operation requests before those are delivered to the replica hosts.

Group communication services (GCS) have been established as effective building blocks for constructing fault-tolerant distributed applications. The basis of a group communication service is a *group membership service*. Each process maintains a unique *view* of the membership of the group. The view includes a list of the processes that are members of the group. Views can change and may become different at different processes. These services enable multiple independent processes to operate collectively as a group, using a service to multicast messages to all members of the group. Birman and Joseph in [15], introduced *virtual synchronous* multicast which provides the stronger reliability guarantee: a message multicast to a group view is delivered to every non-faulty process in that view. If the sender of the message fails during the multicast then virtual synchrony ensures that the message will either be delivered to all remaining processes, or ignored by each of them. Virtual synchrony does not provide any guarantees on the order in which multicasted messages are delivered.

There is a substantial amount of research dealing with specification and implementation of GCSs (e.g., Ricciardi in [85], Neiger in [76], Chandra et al. in [19]). Notable GCS implementations include Isis by Birman and Joseph in [13] and Birman and Van Renesse in [16], Transis by Dolev and Malki in [27], Newtop by Ezhilchelvan et al. in [35], Relacs by Babaoglu et al.

in [11], Horus by Renesse et al. in [84], Ensemble by Hayden in [54] and by Dolev and Schiller in [29]. Birman in [14] also used GCS to provide high levels of availability and consistency to an air-sector control application for the French Air Traffic Control System.

Fekete, Lynch, and Shvartsman in [37], provided one of the first formal specifications for a partitionable view-oriented GCS, called *view-synchrony*. Their specification requires that each processor knows the membership of the group in its current view, and messages sent by any processor in a view must be received (if at all) in the same view. Processors, are not required to know all the views of which they are members. To demonstrate the value of their specification, the authors utilize it to construct a totally ordered-broadcast application. This application is then used by algorithms that implement fault-tolerant atomic memory.

2.4 Quorum Systems

Intersecting collections of sets can be used to achieve synchronization and coordination of concurrent accesses on distributed data objects. A *Quorum System* is a collection of sets known as *quorums*, such that every pair of such sets intersects.

Gifford [47] and Thomas [88] used quorums to achieve mutual exclusion on concurrent file and database access control respectively. Thomas [88], assumed a distributed replicated database, and allowed multiple nodes to request a transaction by sending a corresponding message to one database copy. Then, the host of that database copy had to gather the permission of the majority of the replica hosts before executing the operation. Gifford in [47], proposed a voting scheme to grant permission for distributed file access, by utilizing read and write quorums. To tailor replica reliability and performance, he assigned votes (weights) to each replica host: the fastest the replica host the more votes were assigned. The client had to contact the

replica hosts and collect r (resp. w) votes during a read (resp. write) operation. By letting $r + w$ to be greater than the summation of the votes assigned to the hosts, the protocol ensured that reads and writes would access a common host and hence, each operation would observe the latest copy of the file.

Garcia-Molina and Babara [39], revisited and compared the counting (or vote assignment) strategy presented in [47, 88], with a strategy that explicitly defines a priori the set of intersecting groups (i.e., the quorum system). Their investigation revealed that although the two strategies appear to be similar, they are not equivalent since one may devise quorum systems for which there exist no vote assignment. Following this finding, quorum systems for distributed services adhere to one of the following design principles:

- **Voting:** Quorums are defined by the number of distributed objects collected during an operation.

- **Explicit Quorums:** Quorum formulation is specified before the deployment and use of the quorum system.

The paper also studied the properties of the two strategies. Subsequent works by Peleg and Wool in [81] and Naor and Wool in [75], focused on defining the criteria for measuring the quality of quorum systems:

- **Availability:** Determines the fault tolerance of the quorum system by defining the probability that a quorum contains only correct members.

- **Load:** Determines the replica host load by specifying the frequency that each replica is accessed.

- **Quorum Size:** Smaller quorums may reduce the number of messages involved for a quorum access.

Guided by those criteria, later works evaluated the efficiency of existing quorum systems and devised new, improved constructions of quorum systems. Notable quorum constructions are: Majority Quorum Systems introduced by Thomas in [88] and by Gifford in [47], Matrix Quorum Systems used by Vitanyi and Awerbuch in [90], Crumbling Walls by Peleg and Wool in [81], Byzantine Quorum Systems by Malkhi and Reiter in [70], and Refined Quorum Systems by Guerraoui and Vukolić in [53].

Some works also consider probabilistic quorum constructions. Such constructions rely on a probabilistic quorum discovery. Given a quorum access strategy, each pair of quorums intersect with a non-zero probability with respect to the access strategy. Probabilistic quorums were first presented by Malkhi et al. in [72] and were also used by Abraham and Malkhi in [5] and Konwar et al. in [60].

As efficient tools for collaboration and coordination, quorums attracted the attention of researchers studying implementations of distributed shared memory. Upfal and Wigderson in [89], introduced an atomic emulation of a synchronous R/W shared memory model, where a set of processes shared a set of data items. To allow faster discovery of a single data item and fault-tolerance, the authors suggested its replication among several memory locations. Retrieval (read) or modification (write) of the value of a data item involved the access of the *majority* of the replicas. The authors exploited coordination mechanisms to allow only a single read or write operation per data item at a time. This work was the first to introduce and use $\langle value, timestamp\rangle$ pairs to order the written values, where $timestamp \in \mathbb{N}$.

Vitanyi and Awerbuch in [90] give an implementation of atomic shared memory for the MWMR environment under asynchrony. Their work organized the register replicas in an $n \times n$ matrix construction, where n is the number of client processes. A process p_i is allowed to access the distinct i^{th} row and distinct i^{th} column per write and read operation respectively. This strategy allows reads to be aware of any preceding write due to the intersection of any row with any column of the matrix. To accommodate concurrent write operations, the authors use $\langle value, tag \rangle$ pairs to order the written values. A *tag* is a tuple of the form $\langle timestamp, WID \rangle$, where the $timestamp \in \mathbb{N}$ and *WID* is a writer identifier. Tags are compared lexicographically. Namely, $tag_1 > tag_2$ if either $tag_1.timestamp > tag_2.timestamp$, or $tag_1.timestamp = tag_2.timestamp$ and $tag_1.WID > tag_2.WID$.

2.5 Consistent Memory Under Crash Failures

The work presented by [89] and [90] was designed for the synchronous and failure-free environments. These approaches are inapplicable in the asynchronous, failure-prone, message-passing model.

As discussed by Chockler et al. in [23], implementations in these environments must be wait-free, tolerate various types of failures, and support concurrent accesses to replicated data.

A seminal paper by Attiya et al. [9] first introduced a solution to the problem, by devising an algorithm that implements a SWMR atomic R/W register in the asynchronous message-passing model. Their algorithm overcomes crash failures of any subset of readers, the writer, and up to f out of $2f + 1$ replica hosts. The correctness of the algorithm is based on the use of majorities, a quorum construction established by voting. This work adopts the idea of [89] and uses $\langle$*value,timestamp*$\rangle$ pairs to impose a partial order on read and write operations. A write

operation, involves a single round: the writer has to increment its local timestamp, associate the new timestamp with the value to be written, and send the new pair to the majority $(f + 1)$ of the replica hosts. A read operation requires two rounds: during the first round the reader collects the timestamp-value pairs from a majority of the replica hosts, discovers the maximum timestamp among those, and propagates (in the second round) the maximum timestamp-value pair to the majority of the replica hosts. Although the value of the read is established after the first round, skipping the second round can lead to violations of atomicity when read operations are concurrent with a write operation.

Lynch and Shvartsman in [68] generalized the majority-based approach of [9] to the MWMR environment using quorum systems. To preserve data availability in the presence of failures, the atomic register is replicated among all the service participants. To preserve consistency, they utilize a quorum system (refer to it as *quorum configuration*). This allows read and write operations to terminate a round as soon as the value of the replicated data object (register) was collected from all the members of a single quorum (instead of collecting the majority of the replicas as in [9]). To order the values written, the algorithm utilizes the $\langle$*tag,value*$\rangle$ pairs as those presented by Vitanyi and Awerbuch in [90], and requires every write operation to perform *two* rounds to complete. Read and write operations are implemented symmetrically. In the first round a read (resp. write) obtains the latest $\langle$tag,value$\rangle$ pair from a complete quorum. In the second round, a read propagates the maximum tag-value pair to some complete quorum. A write operation increments the timestamp enclosed in the maximum tag, and generates a new tag including the new timestamp and the writer's identifier. Then, the writer associates the new tag with the value to be written and propagates the tag-value pair to a complete quorum. To enhance longevity of the service the authors in [68] suggest the reconfiguration (replacement)

of quorums. Transition from the old to the new configuration could lead to violations of atomicity as operations can communicate with quorums of either the old or the new configuration during that period. Thus, the authors suggest a blocking mechanism to suspend the read and write operations during the transition.

A follow up work by Englert and Shvartsman in [34] made a valuable observation: taking the union of the new with the old configuration defines a valid quorum system. Based on this observation they allow R/W operations to be active during reconfiguration, by requiring that any operation communicates with both new and old configurations. Both [34] and [68] dedicate a single reconfigurer to propose the next replica configuration. The reconfiguration involves three rounds. During the first round the reconfigurer notifies the readers and writers about the new configuration and collects the latest register information. During the second round it propagates the latest register information in the members of the new configuration. Finally, during the third round the reconfigurer acknowledges the establishment of the new configuration. Read and write operations involve two rounds when they do not discover that a reconfiguration is in progress. Otherwise they may involve multiple rounds to ensure that they are going to reach the latest proposed configuration.

A new implementation of atomic R/W objects for dynamic networks, called RAMBO, was developed by Gilbert, Lynch, and Shvartsman in [48]. The RAMBO approach improves the longevity of implementations in [34, 68], by introducing multiple reconfigurers (and thus circumventing the failure of the single reconfigurer) and a new mechanism to garbage-collect old and obsolete configurations. The new service preserves atomicity while allowing participants to join and fail by crashing. The use of multiple reconfigurers increases the complexity of the

reconfiguration process. To enable the existence of multiple reconfigurers, the service incorporates a consensus algorithm (e.g., Paxos by Lamport in [63]) to allow reconfigurers to agree on a consistent configuration sequence.

A string of refinements followed to improve the efficiency and practicality of that service. Gramoli et al. in [50] reduce the communication cost of the service and locally optimize the liveness of R/W operations. To improve reconfiguration and operation latency, Chockler et al. in [22], propose the incorporation of an optimized consensus protocol, based on Paxos. Aiming to improve the longevity of RAMBO, Georgiou et al. in [41] implement graceful participant departures. They also deploy an incremental gossip protocol that reduce dramatically the message complexity of RAMBO, both with respect to the number of messages and the message size. The same authors in [42] combine multiple instances of the service to compose a complete shared memory emulation. To decrease the communication complexity of the service, Konwar et al. in [59] suggest the departure from the all-to-all gossiping in RAMBO, and propose an indirect communication scheme among the participants. Retargetting [66] to ad-hoc mobile networks, Dolev et al. in [28] formulate the GeoQuorums approach where replicas are maintained by stationary *focal points* that in turn were implemented by mobile nodes. A focal point is active when some mobile nodes exist in it otherwise it is faulty. All the nodes in a focal point maintain the same information about the register replicas. This is established by a reliable atomic broadcast service, called *LBcast*, that reliably delivers any message that is received at a focal point to every mobile node within the focal point. This allows each focal point to act as a single participant in a dynamic atomic register service. To achieve atomicity, focal points are organized in a quorum system. To expedite write operations, the algorithm relies on a global positioning system (GPS) [1] clock to order the written values. A write operation terminates

in a *single* round by associating the value to be written with the time obtained from the GPS service. Joins and fails of focal points are handled similarly to [66].

Most RAMBO refinements preserve the use of consensus to establish reconfigurations. Recall from Section 2.2 that consensus is impossible in the asynchronous setting with a single crash failure [38]. GeoQuorum approach [28] avoids the use of consensus by using a finite set of all possible focal point configurations. Thus, it is sufficient for a mobile node to discover the latest configuration, and contact and propagate the latest register information to all configurations.

To depart from the need for consensus, Gramoli et al. in [49] propose a new approach for self-adaptiveness of the dynamic system. Despite the consensus avoidance, their work relies on failure detection by deploying a heartbeat protocol to detect departed or failed nodes. Moreover, their reconfiguration scheme involves multiple communications rounds, accounting the excessive gossiping protocol for failure detection and propagation of the new configuration.

A recent work by Aguilera et al. [6] showed that atomic register implementations are possible in the dynamic MWMR environment without the use of consensus or failure detection. Their algorithm utilizes views of the system to maintain the latest system participation. When a new node wants to join or depart the service a new view is introduced and propagated in the majority of the processes. Process additions and removals are contributing into the creation of an acyclic graph of views. Each process records the views known to each participant of the service (itself included). To determine the sequence of the configurations, a process starts from its local view and follows the directed acyclic graph to determine the latest view introduced in the system. For this reason, operation termination is ensured only if the number of additions

and removals (and thus reconfigurations) allowed is *finite*. Following this procedure, a read, write or reconfiguration operations take at least four rounds to complete.

2.5.1 Fastness under Crash Failures

Following the development in [9], a folklore belief formed that "atomic reads must write", i.e., a read operation needs to perform a second round. If that second round is avoided then atomicity may be violated: a read operation may return an older value than the one returned by a preceding read operation.

Dolev et al. in [28] introduced single round read operations in the MWMR environment. According to their approach – later used by Chockler et al. in [22] – a read operation could return a value in a single round when it was confirmed that the write phase that propagated that value completed. To assess the status of each write operation the algorithm associated a binary variable, called *confirmed*, with each tag. A participant would set this variable for a tag t in two cases: (i) it completed a write phase and propagated a value associated with t to a full quorum, or (ii) it discovered that t was marked as confirmed by some other participant. A read operation can complete in a single round if the largest discovered tag is marked as confirmed. This can happen iff some write phase that propagated the tag completed.

Despite the improvement achieved in the operation latency in [28, 22], this strategy is unable to overcome the problem presented in [9]: every read operation requires a second round– and thus a "write" – whenever it is concurrent with a write operation. Dutta et al. in [30] are the first to present fast operations that are not affected by read and write concurrency. Assuming the SWMR environment the authors establish that if the number of readers is appropriately

constrained with respect to the number of replicas, then implementations that contain *only* single round reads and writes, called *fast*, are possible. The register is replicated among a set $\mathcal{S}$ of replica hosts (servers), out of which $f < \frac{|\mathcal{S}|}{2}$ (the minority) is allowed to crash. To implement fast writes, the algorithm adopts the write protocol in [9] and involves the use of $\langle$*timestamp, value*$\rangle$ pairs to order the written values. The only difference is that the write operation propagates the written value to $|\mathcal{S}| - f$ servers, instead of a strict majority of $\frac{|\mathcal{S}|}{2} + 1$ required in [9]. The main departure of the new algorithm involves the server and reader implementations. In particular, each server maintains a bookkeeping mechanism to record any reader that inquires its local timestamp-value pair. This information is enclosed in every message sent by the server to any requested operation. To provide up-to date information, a server needs to reset its local bookkeeping information every time a new timestamp-value pair is received. The recorded information is ultimately utilized by the readers to achieve fast read operations. The read protocol requires the reader to send messages to all the servers, and wait for $|\mathcal{S}| - f$ replies. When those replies are received, the reader discovers the maximum timestamp ($maxTs$) among the replies, and collects all the messages that contain that timestamp. Then, a predicate is applied over the bookkeeping information contained in those messages. If the predicate holds, the reader returns the value associated with $maxTs$; otherwise it returns the value associated with the previous timestamp ($maxTS - 1$). Note that the safety of the algorithm in the latter case is preserved because of the single writer and the assumption that a process can invoke a single operation at a time. Thus, the initiation of the write operation with $maxTs$ implies that the write operation with timestamp $maxTs - 1$ has already been completed. On the other hand, if the read operation decides to return $maxTs$ then the validation of the read predicate ensures

the safety of the algorithm. The predicate is based on the following key observation: the number of servers that reply with $maxTs$ to any two subsequent read operations may differ by at most f. The authors show that fast operations are only possible if the number of readers is $R < \frac{|\mathcal{S}|}{f} - 2$. Furthermore, the paper questions the existence of fast implementations in the MWMR environment. It is shown that fast implementations are also impossible in the MWMR environment even assuming two writers, two readers, and a single server crash.

2.6 Consistent Memory Under Byzantine Failures

A more severe and difficult to handle failure model is the one where participants may exhibit arbitrary and malicious behavior. Such failures are known as *Byzantine Failures*. The term Byzantine was first introduced in the context of consensus (see Section 2.2) by Lamport, Shostack, and Pease in their Byzantine Generals problem [64]. In this problem, the generals try to agree on a time to carry out an attack, and worry about the treacherous behavior of some generals. In distributed storage implementations, replica hosts exhibit byzantine behavior when replying with an outdated or incorrect value of their local replica. Some works also consider byzantine readers and writers. In order to tolerate byzantine failures, implementations of consistent memory adopt two different approaches:

- Verifiable: Authentication primitives are embedded to determine the validity of a value propagated by a participant.
- Non-Verifiable: Reliance only on the number of failures in the system and the messages exchanged between the participants.

Verifiable approaches employ digital signatures. Malkhi and Reiter in [71] consider environments where both clients and servers fail arbitrarily. First, the authors assume that the writers are not byzantine. Based on this assumption they develop an algorithm that requires every write operation to digitally sign every written value. The algorithm uses $\langle$*timestamp,value*$\rangle$ pairs to order the written values. Similar to algorithms for the MWMR environment under crash failures, the algorithm involves two rounds for each write and read operation; the first round is a query and the second round is a propagation phase. The digital signature used by the writer serves two purposes: (a) it prevents any byzantine server from forging a non-written value, and (b) it prevents any byzantine reader from writing a forged value during its second round. In the second part of the paper the authors assume that writers may also be byzantine. To prevent the writer from propagating different timestamps for a single value, they incorporate an *echo* protocol in the server site. According to the protocol, the writer performs an extra round to request signed messages ("echoes") from the servers before propagating a value to be written. Those signed messages are then attached to the written value.

Similarly, Cachin and Tessaro in [18] use verification to establish atomicity. In order to achieve verifiability the authors exploit a technique presented by them in [17], called *verifiable dispersal information*. Their new algorithm uses threshold signatures (e.g., Shoup in [87]) and a disperse protocol to prevent the forging of a value. The main idea is that a writer encodes the value it wants to write (using threshold signatures) and produces a vector of value blocks, one for each server. Then, the writer propagates each block and its hash value to the servers. A reader is able to decode the value written as soon as it receives *correct* replies from a number of servers that exceeds the predefine threshold.

While verifiable solutions are able to limit the power of malicious participants, they proved to be computationally esurient. Thus, researchers sought techniques to allow non-verifiable implementations. Due to the severity of byzantine failures, many developments do not manage to achieve atomic semantics, but rather provide regular and safe semantics (e.g., [3, 4, 52, 57, 70]).

Malkhi and Reiter in [70] introduced quorum systems, called *Byzantine Quorum Systems*, to enable non-verifiable consistent memory implementations. Byzantine quorums specify the characteristics that a quorum system must possess in order to ensure data availability and consistency despite byzantine failures. The authors define the class of *masking quorum systems* that ensures: (i) there exist at least one quorum that contains only correct replica hosts, and (ii) every intersection between two quorums contains at least $2f + 1$ replica hosts, where f is the total number of byzantine failures. The second property guarantees that the intersection between every two quorums contains at least $f+1$ correct replica hosts. Given this definition, the authors explore two variations of masking quorum systems. First, the *Dissemination quorum systems* are suited for services that use self-verifying information from correct participants. In these systems, replica values are signed and faulty replica hosts can not undetectably alter the value of the replica. Thus, it suffices for the intersection of every two quorums to contain at least a single non-faulty replica host. The second and stronger variation is the *Opaque quorum systems*. These systems do not rely on verifiability. They differ from the masking quorum systems in that they do not need to know the failure scenarios for which the service is designed for. The participants can detect the correct values only by voting and thus, the intersection of two quorums has to be large enough to suppress the values of both byzantine and out-date

replica hosts. Using the defined quorum systems the authors describe implementations of *safe* and *regular* registers.

Pierce and Alvisi in [82] study non-verifiable atomic semantics under byzantine failures. Although the paper does not provide any algorithmic contributions, it shows that the problem of obtaining atomic semantics can be reduced to that of regular semantics. Consequently, they show that every regular protocol for the byzantine model can produce an atomic protocol, if the first is combined with a *writeback* mechanism (i.e., two round reads).

Martin, Alvisi, and Dahlin in [73] implement MWMR atomic storage on top of non-verifiable byzantine servers. Their algorithm, called Small Byzantine Quorums with Listeners (SBQ-L for short), relies on timestamps to order the written values. The authors show that any protocol that tolerates f byzantine failures and provides safe or stronger semantics requires at least $3f + 1$ servers. The proposed algorithm is optimal in this respect since it uses exactly $3f + 1$ servers. To establish optimality the algorithm is based on two main ideas: (i) a read operation returns a value v only if it is confirmed by at least $2f + 1$ servers, and (ii) it employs the new idea of "listeners" to acquire a confirmed value when a read is concurrent with a write. The write protocol is carried out in two phases: a query and a propagation phase. As in traditional MW implementations the writer determines the new timestamp in the query phase, associates it with the value to be written, and propagates the pair in the second phase. However, since a write may skip f servers, it would be impossible for a read operation to obtain $2f+1$ confirmed values if the total number of servers is $3f+1$. To overcome this problem, the algorithm requires the writer to wait for at least $3f+1$ (and thus all) servers to receive its write request. In the case where a read operation is concurrent with a write, it may not obtain $2f+1$ confirmations of a value during its first round. Thus, the servers maintain a list of the ongoing

read operations, so called the *"Listeners"* list. Whenever a server updates its local information it sends the new information to every reader in its list. When the necessary confirmations are received by a reader, it sends a completion message to all the servers. This is to exclude itself from the server list before returning the confirmed value.

One drawback of the SBQ-L algorithm is that the writer has to increment the maximum timestamp it discovers during its query phase. Note that some of the timestamps received originated potentially from byzantine servers. Considering that a byzantine server may reply with an arbitrarily large timestamp, the adversary may try to exhaust the timestamp value space. Bazzi and Ding in [12] address this problem by introducing *non-skipping* timestamps. Here the writer collects the $f+1$ largest timestamps, and increments the smaller of those. Unfortunately this work trades the optimality of SBQ-L for non-skipping timestamps, since it requires $4f+1$ servers.

2.6.1 Fastness Under Byzantine Failures

Both [12, 73] use diffusion techniques to propagate and discover an acceptable value written on the register replicas. As such, they do not provide any guarantees on the number of rounds required by a read operation. Hence, the communication cost of these approaches is high.

Abraham et al. in [3] study the communication complexity of write operations. The authors introduce a *pre-write* strategy to develop regular register implementations. They suggest a two round write operation: in the first round the writer propagates the value he intended to write, and in the second round it propagates the value to be written. Readers in their system are allowed to fail arbitrarily. Therefore, readers are precluded from changing the value of the

register in any of their read operations. For this reason, every read can take up to $f+1$ rounds to complete, where f the total number of replica host failures. In the same work the authors show that $2f+b+1$ replica hosts are needed for *safe storage* implementations, where b out of f replica hosts failures may be byzantine and the rest may be crashes. To complement their findings, the authors also show that single round write operations exist if and only if more than $2f+2b$ register replicas are used; otherwise two round write operations are necessary. This bound is shown to be tight for both safe and regular semantics.

A follow up paper by Guerraoui and Vukolić in [52] investigates the efficiency of read operations. This work shows two bounds: (a) two rounds are necessary for each read operation to implement *safe storage* when at most $2f+2b$ servers are used, and (b) two rounds are necessary for each read and write operation to implement *regular storage* that uses $2f+b+1$ servers. The algorithms developed in this paper, store each proposed timestamp in a two-dimensional matrix with an entry for every reader and register replica. The matrix records the entire history of the timestamps written on the register. To achieve two round write and read protocols, the paper adopts a technique similar to the one presented in [3], where the writer propagates the value both in its first and second rounds.

Guerraoui, Levy and Vukolić in [51] establish single round read operations for SWMR atomic implementations with byzantine failures. Their system consists of at least $2f+b+1$ servers, where f is the total number of failures out of which b may be byzantine and the rest are crashes. The authors introduced the notion of *"lucky"* operations that characterizes operations that are *synchronous* and *contention-free*. The operations that receive replies from all the servers within some known time interval are called synchronous. The operations that are not concurrent with any write operation are called contention-free. A "lucky" read or write

operation may sometimes complete in a single round, and hence be fast. The authors show that a “lucky” write (resp. read) can be fast in any execution where up to f_w (resp. f_r) servers fail, provided that $f_w + f_r = f - b$. Note that all f_w (resp. f_r) failures can be byzantine given that $f_w < b$ (resp. $f_r < b$).

A recent work by Guerraoui and Vukolić in [53] presents a powerful notion of *Refined Quorum Systems* (RQS), where quorums are classified in three *quorum classes*. The first class contains quorums of large intersection, the second contains quorums of smaller intersection, and the third class corresponds to traditional quorums. The authors specify the necessary and sufficient intersection properties that the members of each quorum class must possess. Then, they use RQSs to develop an efficient Byzantine-resilient SWMR atomic object implementation and a solution to the consensus problem. The SWMR atomic object algorithm relies on timeouts between each round performed by a read or write operation. Initially, multiple rounds are performed to detect a *safe* and *valid* timestamp-value pair. If an operation communicates with a first class quorum by the time they detect the *safe* timestamp (indicating the end of the first round), it is fast. If such a quorum can not be obtained within the timeout interval, then the operation attempts to perform a second communication round with the hope to reach a second class quorum. If such a quorum can not be obtained either, then the operation proceeds to a third round to obtain replies from a third class quorum. Since according to their failure model a single quorum remains non-faulty throughout the execution of the algorithm, the operation will eventually receive the necessary replies in the third round. Thus, an operation may take three rounds to complete once a safe timestamp-value pair is detected.

2.7 Partitionable Networks

Up to this point we surveyed failures that affect individual participants of the service. Network partitions mainly cope with *link* failures that may lead to the division of the underlying network.

Karumanchi, Muralidharan, and Prakash in [58] explore the problem of information dissemination in partitionable ad-hoc networks. The authors replicate the information among dedicated server nodes and use quorums to allow information discovery. To maintain the latest information, they assume loosely synchronized clocks. Loose synchronization allows the write operations to use the writer's local clock to timestamp the written values. The protocol provides a *regular* register implementation.

Amir et al. [8] utilized a group communication service to provide a distributed replicated shared object in the presence of process failures and network partitions. The authors assume virtual synchrony, and thus a message sent by a process in some partition is delivered to every process of that partition unless a process fails. Furthermore, they assume the existence of a primary component. If the network is partitioned, update operations are applied only when they become known to the primary component. Read operations can be performed in network partitions other than the primary component. The algorithm ensures global ordering of the operations and achieves atomic consistency for the shared object.

Chapter 3

Model and Definitions

This chapter presents the model and terminology we use in the sequel. The model of computation is presented in Section 3.1 and definitions of the data types and the terminology we use follows in Section 3.2. Definitions of complexity measures for algorithms that implement atomic storage objects are presented in Section 3.3.

3.1 Model of Computation

We assume a system that consists of a set of *fail-prone, asynchronous processes* with unique identifiers from a set $\mathcal{I}$. Process identifiers are partitioned into three sets:

- a set $\mathcal{W} \subseteq \mathcal{I}$ of writer identifiers
- a set $\mathcal{R} \subseteq \mathcal{I}$ of reader identifiers
- a set $\mathcal{S} \subseteq \mathcal{I}$ of replica host (or server) identifiers

We say "process p" to denote the process associated with an identifier $p \in \mathcal{I}$. Similarly, we use "server s" if $s \in \mathcal{S}$, "reader r" if $r \in \mathcal{R}$ and "writer w" if $w \in \mathcal{W}$. Processes communicate through asynchronous reliable or unreliable communication channels (see Section 3.1.2).

Our goal is to implement a service that emulates an atomic read/write register. Readers (resp. writers) perform read (resp. write) operations on the atomic register. Each server maintains a copy of the replicated register. We use ρ to denote a read operation. A write operation is denoted by ω. If the write operation ω writes value val then we use the notation $\omega(val)$. Any read or write operation is denoted by π. An operation π invoked by a process p can be uniquely identified by a tuple $\langle pid, pc \rangle$, where pid is the id of the invoking process and pc a local operation index from p. In this thesis we assume uniqueness of each operation without explicitly presenting the association of the operation with the process id and the index.

We consider single writer, multiple reader (SWMR) environments, where $|\mathcal{W}| = 1$ and $|\mathcal{R}| \geq 1$, and multiple writer, multiple reader (MWMR) environments, where $|\mathcal{W}| \geq 1$ and $|\mathcal{R}| \geq 1$.

3.1.1 Input/Output Automata and Executions

Algorithms presented in this work are specified in terms of *Input/Output automata* [67, 65]. An algorithm A is a composition of automata A_p, each assigned to some process $p \in \mathcal{I}$.

Each A_i is defined in terms of a set of states $states(A_p)$ and actions $actions(A_p)$. The set $start(A_p) \subseteq states(A_p)$ denotes the set of initial states of A_p. The set $actions(A_p) = in(A_p) \cup out(A_p) \cup int(A_p)$, where the sets $in(A_p)$, $out(A_p)$, and $int(A_p)$ denote the sets of *input*, *output*, and *internal* actions that can be performed by A_p respectively. The *signature* of A_p , $sig(A_p)$, is the triple $\langle in(A_p), out(A_p), int(A_p) \rangle$. The signature $extsig(A_p) =$

$\langle in(A_p), out(A_p), \emptyset\rangle$ represents the *external signature* or *external interface* of A_p. Finally, we have a set of *transitions* $trans(A_p) \subseteq states(A_p) \times actions(A_p) \times states(A_p)$. For each action $\alpha \in actions(A_p)$, this set contains a triple $\langle \sigma, \alpha, \sigma' \rangle$ defining the transition of A_p from state $\sigma \in states(A_p)$ to state $\sigma' \in states(A_p)$ as the result of action $\alpha \in actions(A_p)$. Such a triple is also called a *step* of A_p.

Two component automata A_p and $A_{p'}$, for $p, p' \in \mathcal{I}$, can be *composed* if there exists an action $\alpha \in actions(A_p) \cap actions(A_{p'})$, and the automata are *compatible*:

- $out(A_p) \cap out(A_{p'}) = \emptyset$
- $int(A_p) \cap int(A_{p'}) = \emptyset$

Composition ensures that if A_p performs a step that involves α, so does $A_{p'}$ that has α in its signature. Compatibility ensures that only one automaton in a composition controls the performance of a given output action and if an automaton performs an internal action does not force the other automaton to take a step.

So the *composition* of countable, compatible collection of automata $A = \prod_{p\in\mathcal{I}} A_p$ is defined as: [1]

- $out(A) = \bigcup_{p\in\mathcal{I}} out(A_p)$
- $int(A) = \bigcup_{p\in\mathcal{I}} int(A_p)$
- $in(A) = \bigcup_{p\in\mathcal{I}} in(A_p) - \bigcup_{p\in\mathcal{I}} out(A_p)$
- $sig(A) = \langle out(A), in(A), int(A)\rangle$
- $states(A) = \prod_{p\in\mathcal{I}} states(A_p)$

[1] The $\prod$ in the definition of $states(A)$ and $start(A)$ refers to the ordinary Cartesian Product.

- $start(A) = \prod_{p \in \mathcal{I}} start(A_p)$

- $actions(A) = in(A) \cup out(A) \cup int(A)$

Every state of automaton A is a vector of the states of the component automata A_p, and is denoted by σ. For a state σ of A, let $\sigma[p]$ denote the state of the automaton A_p in σ. Also, let $\sigma[p].var$ to denote the value of variable var of the process automaton A_p in state $\sigma[p]$. The transition set $trans(A)$ is the set of triples $\langle \sigma, \alpha, \sigma' \rangle$ such that, for all $p \in \mathcal{I}$, if $\alpha \in actions(A_p)$ then $\langle \sigma[p], \alpha, \sigma'[p] \rangle \in trans(A_p)$; otherwise $\sigma[p] = \sigma'[p]$. Such a triple is called a step of A.

An *execution fragment* ϕ of A is a finite or an infinite sequence $\sigma_0, \alpha_1, \sigma_1, \alpha_2, \ldots, \alpha_z, \sigma_z, \ldots$ of alternating states and actions, such that every $\sigma_k, \alpha_{k+1}, \sigma_{k+1}$ is a step of A. If an execution fragment begins with an initial state of A then it is called an *execution*. The set of all executions of A is denoted by $execs(A)$. We say that an execution fragment ϕ' of A, *extends* a finite execution fragment ϕ of A if the first state of ϕ' is the last state of ϕ. The *concatenation*, $\phi \circ \phi'$, of ϕ and ϕ' is the result of the extension of ϕ by ϕ' where the duplicate occurrence of the last state of ϕ is eliminated, yielding an execution fragment of A.

Finally, we denote by $\xi|A_p \in execs(A_p)$ the execution of A_p extracted from an execution $\xi \in execs(A)$, when: (i) each pair α_k, σ_k such that $\alpha_k \notin actions(A_p)$ is deleted from ξ, and (ii) every remaining σ_z (i.e., $z \neq k$) is replaced with $\sigma_z[i]$ in ξ.

3.1.2 Communication

We consider the *asynchronous*, *message-passing* environment where processes communicate by exchanging messages. Each channel is modeled by a channel automaton $Channel_{p,p'}$,

for $p, p' \in \mathcal{I}$. Thus, we consider a system automaton A that is the composition of process automata A_p, for $p \in \mathcal{I}$, and channel automata $Channel_{p,p'}$, for $p, p' \in \mathcal{I}$. Each state σ of A is a vector of the state $\sigma[p]$ for each process $p \in \mathcal{I}$, and the state $\sigma[p, p']$ for each channel $Channel_{p,p'}$, for $p, p' \in \mathcal{I}$.

A channel $Channel_{p,p'}$ automaton models the communication between two processes $p, p' \in \mathcal{I}$. The external signature of a $Channel_{p,p'}$ automaton is defined by an input action $\mathsf{send}(m)_{p,p'}$ and an output action $\mathsf{rcv}(m)_{p,p'}$ for some message m in an alphabet M.

In this thesis we develop algorithms that consider *reliable* communication channels.

Definition 3.1.1 (Reliable Channel) A channel between $p, p' \in \mathcal{I}$ is **reliable** in an execution $\phi \in execs(A)$ if for any execution fragment ϕ' of A that extends ϕ all of the following hold:

- $\forall \mathsf{send}(m)_{p,p'}$ event in ϕ, $\exists$ succeeding $\mathsf{rcv}(m)_{p,p'}$ in $\phi \circ \phi'$ (**message delivery**), and
- $\forall \mathsf{rcv}(m)_{p,p'}$ events in ϕ, $\exists$ preceding $\mathsf{send}(m)_{p,p'}$ in ϕ (**message integrity**).

We say that process p *sends* a message m to process p' in an execution ϕ of A, if the event $\mathsf{send}(m)_{p,p'}$ appears in ϕ. Similarly, we say that a process p' *receives* m that was sent from process p in an execution ϕ of A, if the event $\mathsf{rcv}(m)_{p,p'}$ occurs in ϕ. A message m is *delivered* to process p' from process p in ϕ, if m was sent by p and received by p' in ϕ. Finally, a message m is said to be *in-transit* in ϕ, if m was sent by p but not received by p' in ϕ.

Each message can be uniquely identified by a tuble $\langle src, dest, \pi, c\rangle$, where src and $dest$ are the process identifiers of the sender and receiver respectively, π is the operation during which this message is sent, and c a message index for π incremented by the sender. We denote

by $m(\pi, c)_{p,p'}$ the cth message exchanged between processes p to process p' during operation π. We use $m(\pi, c)_{p,p'}.var$ to denote the value of variable var contained in the message $m(\pi, c)_{p,p'}$.

We say that process p *contacts* a subset of processes $\mathcal{G} \subseteq \mathcal{I}$, for an operation π in an execution ϕ, if for every process $p' \in \mathcal{G}$:

(a) p sends the message $m(\pi, c)_{p,p'}$ to p',

(b) p' receives the message $m(\pi, c)_{p,p'}$ sent by p,

(c) p' sends a reply message $m(\pi, c)_{p',p}$ to p, and

(d) p receives the reply $m(\pi, c)_{p',p}$ from p'.

We denote by $cnt(\pi, \mathcal{G})_p$ the occurrence of such contact. If $cnt(\pi, \mathcal{G})_p$ occurs, and additionally no other process $p' \in \mathcal{I} - \mathcal{G}$ receives any message from p within operation π, then we say that p *strictly contacts* $\mathcal{G}$; this is denoted by $scnt(\pi, \mathcal{G})_p$.

3.1.3 Failures

Failures are considered to arrive from an unspecified external entity, the *adversary*. The adversary determines which components of the system fail, what faults they suffer, and at what step on the computation those faults occur. In this work we assume an *omnicient* and *on-line* adversary that has complete knowledge of the computation , and it makes instant and dynamic decisions during the course of computation.

We assume that the automaton A_p of each process i contains an action $\mathsf{fail}_p \in actions(A_p)$, which defines the type of failure that process p may undergo. The adversary decides if and

when a step $\langle \sigma_k, \mathsf{fail}_p, \sigma_{k+1} \rangle$ appears in an execution $\xi \in execs(A)$. A fail_p event may change only the state of process i. That is, if a step $\langle \sigma_k, \mathsf{fail}_p, \sigma_{k+1} \rangle$ appears in ξ, then:

- for every process automaton $A_{p'}$ such that $p \neq p'$, $\sigma_k[p'] = \sigma_{k+1}[p']$,
- for every channel $Channel_{p',p''}$, for $p', p'' \in \mathcal{I}$, $\sigma_k[p', p''] = \sigma_{k+1}[p', p'']$, and
- $\langle \sigma_k[p], \mathsf{fail}_p, \sigma_{k+1}[p] \rangle \in trans(A_p)$.

The algorithms presented in this thesis are designed to tolerate *crash failures*.

Definition 3.1.2 (Crash Failures) For an algorithm A we define the set of executions $\mathcal{F}_C(A)$ to be a subset of $execs(A)$ such that $\forall \xi \in \mathcal{F}_C(A)$, ξ contains zero or one (**crash**) step $\langle \sigma_k, \mathsf{fail_p}, \sigma_{k+1} \rangle$ for some $p \in \mathcal{I}$, and for any step $\langle \sigma_z, \alpha_{z+1}, \sigma_{z+1} \rangle \in \xi$ where $z \geq k+1$, $\sigma_{k+1}[p] = \sigma_z[p] = \sigma_{z+1}[p]$.

A process p *crashes* in an execution $\xi \in \mathcal{F}_C(A)$, if ξ contains a fail step for p. We say that a process p is *faulty* in an execution ξ if p crashes in ξ; otherwise p is *correct*. We allow the adversary to fail any subset of writer and reader processes, with identifiers in $\mathcal{W} \cup \mathcal{R}$, in any execution $\xi \in \mathcal{F}_C(A)$. We limit the power of the adversary to fail only a proper subset of server processes, with identifiers in $\mathcal{S}$, in any execution $\xi \in \mathcal{F}_C(A)$. Let f denote the number of maximum replica host failures allowed. For an implementation A we can define good executions in terms of the maximum number of host failures as follows:

Definition 3.1.3 (f-Good Executions) An execution $\xi \in \mathcal{F}_C(A)$ of an algorithm A is an f-**good execution** if there exists $F \subseteq \mathcal{I}$, $0 \leq |F| \leq f$, such that $\forall p \in F$, there is a step $\langle \sigma_k, \mathsf{fail_p}, \sigma_{k+1} \rangle$ in ξ. The set of all f-good executions of A is denoted by $goodexecs(A, f)$.

3.1.4 Quorum Systems

We focus on quorum systems over the set of server identifiers $\mathcal{S}$. A quorum system is defined as follows:

Definition 3.1.4 (Quorum System) A **quorum system** $\mathbb{Q} \subset 2^{\mathcal{S}}$ is a set of subsets of $\mathcal{S}$, called **quorums**, such that:

- $\forall Q \in \mathbb{Q} : Q \subseteq \mathcal{S}$, and
- $\forall Q, Q' \in \mathbb{Q} : Q \cap Q' \neq \emptyset$

We generalize the definition of quorum systems based on the number of quorums that together have a non-empty intersection. Let $\mathbb{Q}^i$ denote any set of i quorums from $\mathbb{Q}$.

Definition 3.1.5 (n-Wise Quorum Systems) A **quorum system** $\mathbb{Q} \subset 2^{\mathcal{S}}$, is called **n-wise**, for $2 \leq n \leq |\mathbb{Q}|$, if $\forall \mathbb{Q}^n \subseteq \mathbb{Q}$, $\bigcap_{Q \in \mathbb{Q}^n} Q \neq \emptyset$.

A regular quorum system (Definition 3.1.4) is a 2-$wise$ quorum system. We now define the intersection degree of a quorum system:

Definition 3.1.6 (Intersection Degree) A quorum system $\mathbb{Q} \subset 2^{\mathcal{S}}$ has **intersection degree** δ, if $\mathbb{Q}$ is a δ-wise quorum system, but not a $(\delta + 1)$-wise quorum system.

From Definition 3.1.6 if a quorum system $\mathbb{Q}$ has intersection degree $\delta = |\mathbb{Q}|$, then there exists a common intersection among all the quorum sets of $\mathbb{Q}$. A quorum system $\mathbb{Q}$ with intersection degree δ, for $2 \leq \delta \leq |\mathbb{Q}|$, is also a n-wise quorum system for every $n < \delta$.

We now define quorum system failures with respect to the crash failures of the server processes.

Definition 3.1.7 (Faulty Quorum) A quorum $Q \subseteq \mathcal{S}$ is **faulty** in a state σ_z of an execution $\xi \in \mathcal{F}_C(A)$, if ξ contains a crash step $\langle \sigma_{c-1}, \mathsf{fail}_s, \sigma_c \rangle$ such that $s \in Q$ and $z \geq c$.

If a quorum Q is not faulty in a state σ_z of $\xi \in \mathcal{F}_C(A)$, then Q is *correct* in that state.

Definition 3.1.8 (Faulty Quorum System) A quorum system $\mathbb{Q} \subset 2^{\mathcal{S}}$ is **faulty** in a state σ_z of an execution $\xi \in \mathcal{F}_C(A)$, if $\forall Q \in \mathbb{Q}, Q$ is faulty in σ_z of ξ.

We assume that the adversary may fail *all but one* quorum $Q \in \mathbb{Q}$ in any execution $\xi \in \mathcal{F}_C(A)$. The correct quorum Q is not known to any process $p \in \mathcal{I}$. Our failure assumption implies that no R/W operation can wait for more than a single quorum of replicas to reply. If any process $p \in \mathcal{I}$ initiating a read or a write operation waits for additional messages after receiving responses from a complete quorum of replicas, such an operation may not terminate. Implementations that use quorum systems to specify the subsets of servers that each reader and writer may access, are called *quorum-based* implementations. Good executions for a quorum based implementation A are defined as follows:

Definition 3.1.9 ($\mathbb{Q}$-Good Executions) An execution $\xi \in \mathcal{F}_C(A)$ of an algorithm A that uses a quorum system $\mathbb{Q}$, is a **$\mathbb{Q}$-good execution** if there exists $Q \in \mathbb{Q}$, such that $\forall s \in Q$ there does not exist step $\langle \sigma_k, \mathsf{fail_s}, \sigma_{k+1} \rangle$ in ξ. The set of all $\mathbb{Q}$-good executions of A is denoted by $goodexecs(A, \mathbb{Q})$.

For an implementation A, $goodexecs(A)$ denotes the set $goodexecs(A, f)$ if A assumes knowledge of the maximum number of replica host failures, or the set $goodexecs(A, \mathbb{Q})$ if A uses a quorum system $\mathbb{Q}$.

3.2 Consistency and Object Semantics - Atomic Read/Write Registers

Input:	Output:
$\mathsf{read}_{x,p}$, $x \in X$, $p \in \mathcal{R}$	$\mathsf{read\text{-}ack}(v)_{p,x}$, $p \in \mathcal{R}$, $x \in X$, $v \in V_x$
$\mathsf{write}(v)_{x,p}$, $v \in V_x$, $x \in X$, $p \in \mathcal{W}$	$\mathsf{write\text{-}ack}_{p,x}$, $p \in \mathcal{W}$, $x \in X$

Figure 1: External Signature of a Read/Write Atomic Memory Service.

Our goal is to devise algorithms that implement an atomic R/W memory abstraction. Let X be a set of register identifiers. Each register $x \in X$ may be assigned a value v from a set of values V_x, where $\perp \in V_x$ the initial value of x. A *read/write* register $x \in X$, is modeled by an I/O automaton A_x with input actions $in(A_x) = \{\mathsf{read}_{x,p}, \mathsf{write}(v)_{x,p}\}$, and output actions $out(A_x) = \{\mathsf{read\text{-}ack}(v')_{p,x}, \mathsf{write\text{-}ack}_{p,x}\}$, where $v, v' \in V_x$ and $p \in \mathcal{I}$. A *read/write memory* $\mathcal{M}$ is the *composition* of a countable, *compatible* read/write register I/O automata A_x, for $x \in X$.

Let automaton $\mathcal{M}$ implement a R/W memory abstraction. We say that a process r, for identifier $r \in \mathcal{R}$, invokes a read operation on register $x \in X$ in an execution $\xi \in execs(\mathcal{M})$ if a step $\langle \sigma_k, \mathsf{read}_{x,r}, \sigma_{k+1} \rangle$ appears in ξ. Similarly, we say that a process w, for identifier $w \in \mathcal{W}$, invokes a write operation on $x \in X$ in an execution $\xi \in execs(\mathcal{M})$ if a step $\langle \sigma_z, \mathsf{write}(v)_{x,w}, \sigma_{z+1} \rangle$ appears in ξ.

The step $\langle \sigma_k, \mathsf{read}_{x,r}, \sigma_{k+1} \rangle$ or $\langle \sigma_z, \mathsf{write}(v)_{x,w}, \sigma_{z+1} \rangle$ is called *invocation step* of a read or write operation π respectively, and is denoted by $inv(\pi)$. The corresponding $\langle \sigma_{k-1'}, \mathsf{read\text{-}ack}(v)_{r,x}, \sigma_{k'} \rangle$ or $\langle \sigma_{z-1'}, \mathsf{write\text{-}ack}_{w,x}, \sigma_{z'} \rangle$, for $k' \geq k+1$ and $z' \geq z+1$, is the *response step* and is denoted by $res(\pi)$. The states σ_k and σ_z are called *invocation states*

and the states $\sigma_{k'}$ and $\sigma_{z'}$ are called *response states* of read or write operation π. An invocation state of π is denoted by $\sigma_{inv(\pi)}$. Similarly, the response state of π is denoted by $\sigma_{res(\pi)}$. Following is the definition for operation completeness:

Definition 3.2.1 (Operation Completeness) An operation π is **incomplete** in an execution $\xi \in execs(\mathcal{M})$, if ξ contains $inv(\pi)$ but does not contain $res(\pi)$; otherwise we say that π is **complete**.

We assume that the executions of $\mathcal{M}$ are well-formed. Namely, a process does not invoke a new operation until it receives the response for a previously invoked operation in any execution of $\mathcal{M}$. This notion is captured by the following definition.

Definition 3.2.2 (Well-Formedness) An execution $\xi \in execs(\mathcal{M})$ is **well-formed** if for any read or write operation π invoked by a process p, ξ contains a step $inv(\pi)$ and does not contain any step $inv(\pi')$ for any operation π' invoked by p before the step $res(\pi)$ appears in ξ.

In an execution, we say that an operation (read or write) π_1 *precedes* another operation π_2, or π_2 *succeeds* π_1, if the response step for π_1 precedes the invocation step of π_2; this is denoted by $\pi_1 \rightarrow \pi_2$ [62]. Two operations are *concurrent* if neither precedes the other. This can be expressed more formally by the following definition:

Definition 3.2.3 (Precedence Relations) Two operations π_1 and π_2 may have one of the following precedence relations in an execution $\xi \in execs(\mathcal{M})$:

- π_1 **precedes** π_2 ($\pi_1 \rightarrow \pi_2$): $res(\pi_1)$ appears before $inv(\pi_2)$ in ξ
- π_1 **succeeds** π_2 ($\pi_2 \rightarrow \pi_1$): $inv(\pi_1)$ appears after $res(\pi_2)$ in ξ

- π_1 is **concurrent** to π_2 ($\pi_1 \leftrightarrow \pi_2$): neither $\pi_1 \rightarrow \pi_2$, nor $\pi_2 \rightarrow \pi_1$ in ξ.

Correctness of an implementation of an atomic read/write register is defined in terms of the *atomicity* (safety) and *termination* (liveness) properties.

Definition 3.2.4 (Termination) Consider an operation π that is invoked by process p, and $inv(\pi)$ appears in a finite execution $\phi \in execs(\mathcal{M})$. Then there exists a finite execution fragment $\phi' \in execs(\mathcal{M})$ that extends ϕ, such that if p is correct in $\phi \circ \phi' \in execs(\mathcal{M})$ and $\phi \circ \phi' \in goodexecs(\mathcal{M})$, then $res(\pi)$ appears in $\phi \circ \phi'$.

In other words, termination ensures that an operation invoked from a process p is going to terminate as long as p is correct and the system obeys the failure model. Atomicity is defined as follows [65]:

Definition 3.2.5 (Atomicity) Consider the set Π of all complete operations in any well-formed execution. Then for operations in Π there exists an irreflexive partial ordering $\prec$ satisfying the following:

A1. If for operations π_1 and π_2 in Π, $\pi_1 \rightarrow \pi_2$, then it cannot be the case that $\pi_2 \prec \pi_1$.

A2. If $\pi \in \Pi$ is a write operation and $\pi' \in \Pi$ is any operation, then either $\pi \prec \pi'$ or $\pi' \prec \pi$.

A3. The value returned by a read operation is the value written by the last preceding write operation according to $\prec$ (or $\perp$ if there is no such write).

A read/write register $x \in X$ is *atomic*, if it has the external signature $sig(A_x) = \langle in(A_x), out(A_x), \emptyset \rangle$, and in addition it satisfies *well-formedness* (Definition 3.2.2), *termination* (Definition 3.2.4), and *atomicity* (Definition 3.2.5) conditions. Finally, an *atomic*

read/write memory $\mathcal{M}$ is the *composition* of a countable, *compatible* atomic read/write register I/O automata A_x, for $x \in X$. The external signature of an atomic memory abstraction is given in Figure 1.

In the sequel, we focus on the implementation of a single atomic R/W register abstraction and thus, from this point onward we omit the names of the registers.

3.3 Complexity Measures

We measure the *operation latency* of an atomic register implementation in terms of *communication rounds* (or simply rounds). A round is defined as follows [30]:

Definition 3.3.1 (Communication Round) Process p performs a **communication round** during an operation π in an execution $\xi \in execs(A)$ of an algorithm A, if all of the following hold:

CR1. p sends messages for π to a set of processes $Z \subseteq \mathcal{I}$,

CR2. when a message for π is delivered to $q \in Z$, q sends a reply for π to p without waiting for messages from any other process, and

CR3. when p receives "sufficient" replies it terminates the round (either completing π or starting a new round).

Using Definition 3.3.1, we can define fast operations and fast implementations ([30]):

Definition 3.3.2 (Fast Operations) Consider an operation π that is invoked by a process p in an execution $\xi \in execs(A)$, of some implementation A. We say that π is a **fast operation** if it completes when process p performs a *single* communication round between $inv(\pi)$ and $res(\pi)$; otherwise π is **slow**.

Definition 3.3.3 (Fast Implementation) An implementation A is called **fast implementation** if every execution $\xi \in goodexecs(A)$ contains only fast operations.

For quorum-based implementations that use a quorum system $\mathbb{Q}$, communication rounds can be defined over quorums of servers.

Definition 3.3.4 (Quorum-Based Communication Round) A process p performs a **quorum-based communication round**, in an execution $\xi \in execs(A)$ of a quorum-based implementation A during operation π if:

QBR1. p sends messages for π to a set of processes $Z \subseteq \mathcal{I}$,

QBR2. when a message for π is delivered to $q \in Z$, q sends a reply for π to p without waiting for messages from any other process, and

QBR3 When p receives replies from **at least a single quorum** it terminates the round (either completing π or starting a new round).

By Definition 3.3.4, a quorum-based communication round defers from Definition 3.3.1 in the last property, where the servers of at least a single quorum are expected to reply. Now, we can define fastness for quorum-based implementations.

Definition 3.3.5 (Fast Quorum-Based Operations) Consider an operation π that is invoked by a process p in an execution $\xi \in execs(A)$, of some quorum-based implementation A. We say that π is a **fast quorum-based operation** if it completes when process p performs a *single* quorum-based communication round between $inv(\pi)$ and $res(\pi)$; otherwise π is **slow**.

Definition 3.3.6 (Fast Quorum-Based Implementation) An implementation A is called **fast quorum-based implementation** if every execution $\xi \in goodexecs(A, \mathbb{Q})$ contains only fast quorum-based operations.

To obtain or modify the value of the atomic register, a read or write operation requires at least a single communication round. To ensure termination, any process $p \in \mathcal{R} \cup \mathcal{W}$ needs to send messages to all $s \in \mathcal{S}$ and wait for replies from a set $Z \subseteq \mathcal{S}$. By Definitions 3.3.2 and 3.3.5 a (quorum-based) operation invoked by a process p is fast if it completes after two communication delays: (i) a message from p to all $s \in \mathcal{S}$, and (ii) a reply from every $s \in Z$ to p. The set Z may: (a) have cardinality $|\mathcal{S}| - f$, if the maximum number of server failures f is known, or (b) contain a quorum Q in case of a quorum-based implementation. For the rest of the thesis we assume that the messages from the readers and writers to the servers, and the replies from the servers to the readers and writers are delivered. All other messages remain in transit.

Chapter 4

Trading Speed for Reader Participation

As discussed in Chapter 2, *fast* implementations of atomic R/W registers require that the number of reader participants in the service must be restricted with respect to the number of replica hosts [30]. In this chapter we present a new family of atomic R/W register implementations that trade the fastness of *some* operations to allow *unrestricted* number of readers in the service. We call such implementations *semifast*. In the sections that follow, we explain the restrictions that fast implementations impose in the service and formally define semifast implementations. Next, we present a semifast implementation of an atomic R/W register and we analyze its operation latency. Finally, we specify the conditions that need to be satisfied for semifast implementations to be feasible.

4.1 Fast and Semifast Implementations

Dutta et al. [30] were the first to introduce fast implementations of atomic R/W registers. Their algorithm implements an atomic R/W register in the SWMR environment, where *all* read and write operations require a *single* round to complete. Such an efficient behavior however,

comes with a price: (1) the number of readers must be bounded by $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$ and (2) single round (fast) implementations are not possible for the MWMR environment even with two writers, two readers and a single server failure. While single round implementations seem to be restrictive, works like [68] demonstrated that if all operations perform *two rounds* (slow), then we can obtain atomic R/W register implementations that allow multiple writers and unrestricted number of readers. Naturally, the following question arises: How many operations need to be *slow* in order to overcome the limitations imposed by fast implementations?

4.1.1 Semifast Implementations

We partly answer the above question by introducing *semifast* implementations.

A semifast implementation of an atomic R/W register allows fast writes and reads; yet, under certain conditions it allows reads to perform two rounds. Below we formally define semifast implementations. The notation $\mathfrak{R}(\rho)$ [90], used in the definition, denotes the unique write operation that wrote the value returned by a read operation ρ.

Definition 4.1.1 (Semifast Implementation) An implementation A of an atomic object is ***semifast*** if the following are satisfied:

S1. In any execution ξ of A, every *write* operation is fast.

S2. In any execution ξ of A, any complete *read* operation performs one or two communication rounds.

S3. In any execution ξ of A, if ρ_1 is a two-round read operation, then any read operation ρ_2 with $\mathfrak{R}(\rho_1) = \mathfrak{R}(\rho_2)$, such that $\rho_1 \to \rho_2$ or $\rho_2 \to \rho_1$, must be fast.

S4. There exists an execution ξ of A that contains at least one write operation ω and at least one

read operation ρ_1 with $\Re(\rho_1) = \omega$, such that ρ_1 is concurrent with ω and all read operations ρ with $\Re(\rho) = \omega$ (including ρ_1) are fast.

Properties **S1** and **S2** of Definition 4.1.1, explicitly specify the fastness of read and write operations: writes have to terminate after a single round, while reads are allowed to perform at most two rounds. By property **S3** only a *single complete* slow read operation is allowed per write operation. Therefore, if a slow read operation returns a value val then any read operation that returns value val and *precedes* or *succeeds* the slow read must be fast. Finally, property **S4** rules out trivial solutions that allow fast operations only in the absence of read and write concurrency. Hence, **S4** requires that semifast implementations allow read operations to be fast even if those are executed concurrently with a write operation. Such a characteristic will enable executions where all read and write operations are fast.

In the next section, we show that a semifast implementation may allow unrestricted number of readers. Here, a single complete slow read operation is enough to remove the constraint on the number of readers imposed by fast implementations. Later, we show that semifast implementations also have some limitations: (1) implementations that arrange the readers into groups, can be semifast iff the number of groups is $|\mathcal{V}| < \frac{S}{f} - 2$, (2) semifast implementations are possible if $|\mathcal{S}| > 3f$, and (3) semifast implementations are not possible in the MWMR environment.

4.2 Semifast Implementation: Algorithm SF

In this section we present algorithm SF. This algorithm trades the speed of some read operations for allowing unbounded number of reader participants in the service. SF implements

a SWMR semifast atomic R/W register since it satisfies the properties in Definitions 3.2.5 and 4.1.1 for the SWMR setting.

In brief, the algorithm adopts the timestamp-value pair technique to order the values written on the atomic register. To allow unbounded reader participation, SF introduces the notion of *virtual nodes*, abstract entities that enclose a set of reader participants. The constructed entities take the place of individual readers in an adapted form of the fast implementation of [30]. As a result, the new algorithm achieves the same performance as [30] when read requests originate from a single reader per virtual node. Things become challenging when requests originate from multiple readers residing in both the same and different virtual nodes.

4.2.1 Grouping Reader Participants – Virtual Nodes

The notion of *virtual nodes* allows SF to accommodate arbitrarily many readers. Figure 2 illustrates the deployment of virtual nodes on top of the set of reader processes. A virtual node is a set of reader identifiers and has a unique identifier from a set $\mathcal{V}$. Each reader process with identifier $r \in \mathcal{R}$ maintains a local variable that specifies the virtual node that the reader belongs to. Let us denote by ν_r the virtual node assigned to reader r.

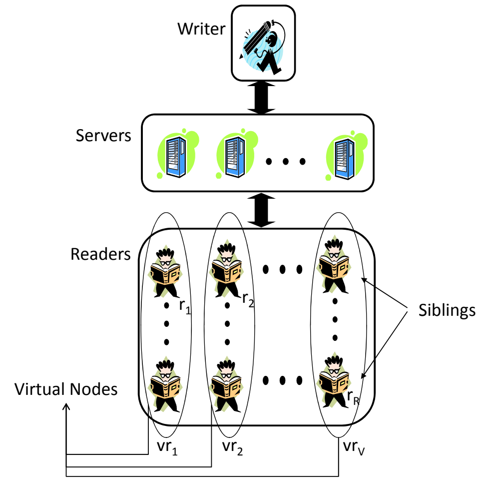


Figure 2: Virtual Nodes.

If two readers r, r' belong to the same virtual node, such that $\nu_r = \nu_{r'}$, then we say that r and r' are *siblings*. Note that it is not necessary for a reader to be aware of its siblings or the members of the

other virtual nodes. This allows local assignment of each individual reader to a virtual node. For instance, the virtual node of a reader $r \in \mathcal{R}$ can be equal to $\nu_r = (r \mod \max(|\mathcal{V}|)) + 1$, assuming that reader identifiers are natural numbers.

In a later section we show that it is not possible to obtain a semifast implementation if the number of virtual nodes is more than $\frac{|\mathcal{S}|}{f} - 2$. In contrast with [30], the restriction on the number of virtual nodes does not affect the number of reader participants: a single virtual node can support unbounded number of readers.

4.2.2 High Level Description of SF

Before we proceed to the formal specification of our algorithm, we provide a high level description. The algorithm uses timestamp-value pairs to order the values written on the register and each writer associates a timestamp with two values $\langle v, vp \rangle$. The variable v is the new value to be written, while the variable vp is the last value written by the writer. The algorithm consists of three protocols, one for the writer, one for the reader, and one for the server.

Writer. The write protocol involves the increment of the timestamp and its propagation, along with the writer's new and previous value, to all the servers. The operation completes once the writer receives $|\mathcal{S}| - f$ replies from the servers.

Reader. The read protocol is more complicated. A reader sends read messages to all the servers and once it receives $|\mathcal{S}| - f$ replies, determines the value to be returned by consulting the validity of a predicate. The predicate depends on (i) the maximum timestamp witnessed within the replies, (ii) the number of servers that replied with that timestamp, and (iii) the number of virtual nodes (members of which) witnessed that timestamp through those servers. The idea behind the predicate is presented in the next section. If the predicate holds then the reader

returns the value v associated with the maximum timestamp it witnessed ($maxTS$); otherwise it returns the previous value vp (associated with the previous timestamp $maxTS - 1$). If the predicate holds with certain conditions, then a reader may perform a second communication round before returning value v.

Server. Each server process maintains an object replica and updates its object's value when it receives a message that contains a timestamp greater than its local timestamp. Additionally, the server records the virtual nodes that requested its atomic object and replies with the information of the atomic object (timestamp,value) along with the recorded set of virtual nodes. If a server receives a message from a the second communication round of a read operation, it stores the timestamp-value pair enclosed in the message in variable *postit*. The postit variable indicates that the server witnessed the intention of a read operation to return the stored pair.

4.2.3 Formal Specification of SF

Here we present the formal specification of algorithm SF. We assume that the number of unique virtual ids is such that $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$. (We show in Section 4.3.1 that semifast implementations are impossible when $|\mathcal{V}| \geq \frac{|\mathcal{S}|}{f} - 2$). The algorithm is composed of four automata: (i) SF_w for $w \in \mathcal{W}$, (ii) SF_r for $r \in \mathcal{R}$, (iii) SF_s for $s \in \mathcal{S}$, and (iv) $Channel_{p,p'}$ for $p, p' \in \mathcal{I}$. The SF_w, SF_r and SF_s automata are given in Figures 3, 4 and 5, and 6 respectively. The $Channel_{p,p'}$ automaton follows the specification of a reliable channel (see Section 3.1.2). Moreover, as discussed in Section 3.3, we assume that only messages from the readers and the writer to the servers, and the replies from the servers to the readers and the writer are delivered. The system automaton of algorithm SF is the composition of automata SF_w, SF_r and SF_s, with channel automata $Channel_{p,s}$ or $Channel_{s,p}$ for $p \in \mathcal{R} \cup \mathcal{W}$ and $s \in \mathcal{S}$.

Automaton $\textsc{Sf}_w$.

The state variables, the signature and the transitions of the $\textsc{Sf}_w$ are given in Figure 3. The state of the $\textsc{Sf}_w$ automaton consists of the following variables:

- $\langle ts, v, vp\rangle \in \mathbb{N} \times V \times V$: writer's local timestamp along with the latest and the previous values written by the writer.
- $wCounter \in \mathbb{N}$: counts the write requests performed by the writer. This is used by the servers to distinguish fresh from stale messages.
- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the writer received all the necessary replies to complete its write operation and is ready to respond to the client.
- $srvAck \subseteq \mathcal{S}$: a set that contains the servers that reply to the write messages as a result of a write request. The set is reinitialized to $\emptyset$ at the response step of every write operation.
- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

The automaton completes a write operation in a single phase. When a $\mathsf{write}(v)_w$ request is received from the environment, the $status$ variable becomes $active$, the previous value vp gets the current value and the new value v is updated with the value requested to be written. The timestamp ts is incremented and is associated with the two values. As long as the $status$ remains active the automaton sends one message to every server process and collects the identifiers of the servers that reply to those messages in the $srvAck$ set. When $|srvAck| \geq |\mathcal{S}| - f$,

Signature:

Input:
$\mathsf{write}(v)_w,\ v \in V$
$\mathsf{rcv}(msg)_{s,w},\ msg \in M,\ s \in \mathcal{S}$
fail_w

Output:
$\mathsf{send}(msg)_{w,s},\ msg \in M,\ s \in \mathcal{S}$
$\mathsf{write\text{-}ack}_w$

Internal:
$\mathsf{write\text{-}fix}_w$

State:

$ts \in \mathbb{N}$, initially 0
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$wCounter \in \mathbb{N}^+$, initially 0
$srvAck \subseteq \mathcal{S}$, initially $\emptyset$
$status \in \{idle, active, done\}$, initially $idle$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{write}(v)_w$
Effect:
 if $\neg failed$ then
  if $status = idle$ then
   $status \leftarrow active$
   $srvAck \leftarrow \emptyset$
   $vp \leftarrow v$
   $(v, ts) \leftarrow (v, ts + 1)$
   $wCounter \leftarrow wCounter + 1$

Input $\mathsf{rcv}(\langle msgT, t, seen, C, postit\rangle)_{s,w}$
Effect:
 if $\neg failed$ then
  if $status = active$ and $wCounter = C$ then
   $srvAck \leftarrow srvAck \cup \{s\}$

Internal $\mathsf{write\text{-}fix}_w$
Precondition:
 $\neg failed$
 $status = active$
 $|srvAck| \geq |\mathcal{S}| - f$
Effect:
 $status \leftarrow done$

Output $\mathsf{send}(\langle msgT, t, C, vid\rangle)_{w,s}$
Precondition:
 $status = active$
 $\neg failed$
 $s \in \mathcal{S}$
 $\langle msgT, t, C, vid\rangle =$
  $\langle \text{WRITE}, \langle ts, v, vp\rangle, wCounter, 0\rangle$
Effect:
 none

Output $\mathsf{write\text{-}ack}_w$
Precondition:
 $status = done$
 $\neg failed$
Effect:
 $status \leftarrow idle$

Input fail_w
Effect:
 $failed \leftarrow true$

Figure 3: SF_w Automaton: Signature, State and Transitions

the precondition of the write-fix action is met and the $status$ of the operation becomes $done$. This, enables the write-ack action and once it occurs the writer responds to the environment and reinitializes $status = idle$ waiting for the next write request.

Automaton SF_r.

The state variables, the signature and the transitions of the SF_r are given in Figures 4 and 5. The state of the SF_r automaton consists of the following variables:

Signature:

Input:
$\mathsf{read}_r,\ r \in \mathcal{R}$
$\mathsf{rcv}(m)_{s,r},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\mathsf{fail}_r,\ r \in \mathcal{R}$

Output:
$\mathsf{send}(m)_{r,s},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\mathsf{read\text{-}ack}(val)_r,\ val \in V,\ r \in \mathcal{R}$

Internal:
$\mathsf{read\text{-}phase1\text{-}fix}_r$
$\mathsf{read\text{-}phase2\text{-}fix}_r$

State:

$vid \in \mathcal{V}$, initially $(r \mod (\frac{|\mathcal{S}|}{f} - 2)) + 1$
$ts \in \mathbb{N}$, initially 0
$maxTS \in \mathbb{N}$, initially 0
$maxPS \in \mathbb{N}$, initially 0
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$retvalue \in V$, initially $\perp$
$rCounter \in \mathbb{N}^+$, initially 0
$phase \in \{1, 2\}$, initially 1
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxTsAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxPsAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxTsSrv \subseteq \mathcal{S}$, initially $\emptyset$
$failed$, a Boolean initially **false**

Figure 4: $\textsc{Sf}_r$ Automaton: Signature and State

- $vid \in \mathcal{V}$: the virtual node to which the reader r belongs.

- $\langle v, vp \rangle \in V \times V$: the value and previous value associated with the maximum timestamp discovered during r's last read operation.

- $maxTS \in \mathbb{N}$, $maxPS \in \mathbb{N}$: the maximum timestamp and postit discovered.

- $\langle ts, retvalue \rangle \in \mathbb{N} \times V$: the timestamp and value returned during the last read operation.

- $rCounter \in \mathbb{N}$: read request counter. Used by the servers to distinguish fresh from stale messages.

- $phase \in \{1, 2\}$: indicates the active communication round of the read operation.

- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the reader decided on the value to be returned and is ready to respond to the client.

Transitions:

Input read_r
Effect:
 if $\neg failed$ then
  if $status = idle$ then
   $status \leftarrow active$
   $rCounter \leftarrow rCounter + 1$

Input $\mathsf{rcv}(\langle msgT, t, seen, C, postit\rangle)_{s,r}$
Effect:
 if $\neg failed$ then
  if $status = active$ and $rCounter = C$ then
   $srvAck \leftarrow srvAck \cup \{(s, \langle msgT, t, seen, C, postit\rangle)\}$

Output $\mathsf{send}(\langle msgT, t, C, vid\rangle)_{r,s}$
Precondition:
 $status = active$
 $\neg failed$
 $[(phase = 1 \wedge \langle msgT, t, C, vid\rangle = \langle \text{READ}, \langle maxTS, v, vp\rangle, rCounter, vid\rangle) \vee$
 $(phase = 2 \wedge \langle msgT, t, C, vid\rangle = \langle \text{INFORM}, \langle maxTS, v, vp\rangle, rCounter, vid\rangle)]$
Effect:
 none

Internal $\mathsf{read\text{-}phase2\text{-}fix}_r$
Precondition:
 $\neg failed$
 $status = active$
 $phase = 2$
 $|srvAck| \geq 2f + 1$
Effect:
 $status \leftarrow done$
 $phase \leftarrow 1$

Output $\mathsf{read\text{-}ack}(val)_r$
Precondition:
 $\neg failed$
 $status = done$
 $val = retvalue$
Effect:
 $status \leftarrow idle$

Input fail_r
Effect:
 $failed \leftarrow true$

Internal $\mathsf{read\text{-}phase1\text{-}fix}_r$
Precondition:
 $\neg failed$
 $status = active$
 $phase = 1$
 $|srvAck| \geq |\mathcal{S}| - f$
Effect:
 $maxTS \leftarrow \{max(m.t.ts) : (s, m) \in srvAck\}$
 $maxPS \leftarrow \{max(m.postit) : (s, m) \in srvAck\}$
 $maxTsAck \leftarrow \{(s, m) : (s, m) \in srvAck$ and $m.t.ts = maxTS\}$
 $maxPsAck \leftarrow \{(s, m) : (s, m) \in srvAck$ and $m.postit = maxPS\}$
 $(v, vp) \leftarrow \{(m.t.v, m.t.vp) : (s, m) \in maxAck\}$
 $maxTsSrv \leftarrow \{s : s \in Q, (s, msg) \in maxAck\}$
 if $\exists \beta \in [1, \ldots, |\mathcal{V}|]$, and $MS \subseteq maxTsAck$ s.t.
  $|MS| \geq |\mathcal{S}| - \beta f$ and $|\bigcap_{(s,m)\in MS} m.seen| \geq \beta$
 then
  $ts \leftarrow maxTS$
  $retvalue \leftarrow v$
  if $|\bigcap_{(s,m)\in MS} m.seen| = \beta$ then
   $phase \leftarrow 2$
   $srvAck \leftarrow \emptyset$
   $rCounter \leftarrow rCounter + 1$
  else
   $status \leftarrow done$
 else
  if $maxPS = maxTS$ then
   $ts \leftarrow maxTS$
   $retvalue \leftarrow v$
   if $|maxPsAck| < t + 1$ then
    $phase \leftarrow 2$
    $srvAck \leftarrow \emptyset$
    $rCounter \leftarrow rCounter + 1$
   else
    $status \leftarrow done$
  else
   $ts \leftarrow maxTS - 1$
   $retvalue \leftarrow vp$
   $status \leftarrow done$

Figure 5: SF_r Automaton: Transitions

- $srvAck \subseteq M \times \mathcal{S}$: a set that contains the servers and their replies to the read operation. The set is reinitialized to $\emptyset$ at the response step of every read operation.

- $maxTsAck \subseteq M \times \mathcal{S}$ and $maxPsAck \subseteq M \times \mathcal{S}$: these sets contain the servers that replied with the maximum timestamp and maximum postit respectively to the last read request. The sets also contain the messages sent by those servers.

- $maxTsSrv \subseteq \mathcal{S}$: The servers that replied with the $maxTS$.

- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

The reader automaton may involve one or two rounds before the response step of a read operation occurs. When the reader automaton receives a $\mathsf{read}()_r$ request from its environment, it sets the $status$ variable to $active$ and increments the read counter. This enables the actions rcv and send. The reader sends a read message to every server (when send occurs) and collects (via the rcv action) the identifiers of the servers and their replies in the $srvAck$ set. As $phase$ is initialized to 1, the send action transmits READ messages to the servers. Once, $|srvAck| \geq |\mathcal{S}| - f$ the action read-phase1-fix is enabled. When this action occurs, the reader discovers the maximum triple $\langle maxTS, v, vp \rangle$ and the maximum postit value $maxPS$ among the received messages. Then, it collects the server acknowledgments that contain $maxTS$ and $maxPS$ in the sets $maxTsAck$ and $maxPsAck$ respectively. To determine the value to be returned the reader consults the validity of a predicate. The predicate depends on: (i) the maximum timestamp witnessed within the replies ($maxTS$), (ii) the number of servers that replied with that timestamp ($|maxTsAck|$), and (iii) the number of virtual nodes (members of which) witnessed that timestamp through those servers. The latter number is derived from the $seen$ set

which is attached in the reply of each server. The predicate, called **SF-RP**, is formally written as follows:

Reader predicate (SF-RP): $\exists\beta \in [1, V+1]$ and $\exists MS \subseteq maxTsAck$, s.t. $(|MS| \geq |\mathcal{S}| - \beta f) \wedge (| \cap_{m \in MS} m.seen| \geq \beta)$

By the read-phase1-fix action, the reader returns the new value v associated with the maximum timestamp if one of the following conditions is satisfied:

(i) **SF-RP** holds, or

(ii) the $maxPS = maxTS$.

In such case, the read operation may perform one or two rounds. The $phase$ variable becomes 2 and the reader proceeds in a second round in the following cases:

(1) **SF-RP** holds with $| \cap_{m \in MS} m.seen| = \beta$, or

(2) **SF-RP** does not hold and $maxPS = maxTS$, but $|maxPsAck| < f + 1$.

When none of the conditions (i) or (ii) hold the read operation returns the previously written value vp in a single round. If one round is enough then the $status$ variable becomes $done$ by the end of read-phase1-fix and the response action read-ack(v) is enabled. If $phase = 2$, the $srvAck = \emptyset$ and the reader sends INFORM messages to all servers. Once, $|srvAck| \geq |\mathcal{S}| - f$ the action read-phase2-fix is enabled. When that action occurs $status$ becomes $done$ and $phase$ is reinitialized to 1. Now, the second round is completed and the reader is ready to respond to the client.

Clearly, in item (2) above, the second round is necessary as a future read operation may not observe $maxPS$ in any of the servers when $|maxPsAck| < f + 1$. Let us now examine why

a second round is needed in case of (1). The predicate **SF-RP** is derived from the following observation:

Observation 4.2.1 For any two read operations ρ_1 and ρ_2 that witness the same $maxTS$ from $maxTsAck_1$ and $maxTsAck_2$ servers respectively (one message per server), the sizes of the sets $|maxTsAck_1|$ and $|maxTsAck_2|$ differ by at most f. That is:

$$||maxTsAck_1| - |maxTsAck_2|| \leq f$$

Consider the following example to visualize the idea behind the predicate. Let ϕ be an execution fragment that contains a complete write operation ω that propagates the triple $\langle maxTS, v, vp\rangle$ to $|\mathcal{S}| - f$ servers. Let extend ϕ by a read operation ρ_1 that discovers $maxTS$ in $|\mathcal{S}| - 2f$ server replies (missing f of the servers that replied to ω). Since $\omega \rightarrow \rho_1$ then ρ_1 has to return v (the value associated with $maxTS$) to preserve atomicity.

Assume now an execution fragment ϕ' that contains an incomplete write ω that propagates the new value with $maxTS$ to $|\mathcal{S}| - 2f$ servers. Let us extend ϕ' by a read ρ_1 from reader r. If ρ_1 discovers $maxTS$ in $|\mathcal{S}| - 2f$ servers – by receiving replies from all the servers that received messages from ω – then it cannot distinguish ϕ from ϕ' and thus has to return v in ϕ' as well. Let ρ_2 be a read operation from r' s.t. $\rho_1 \rightarrow \rho_2$. The read ρ_2 may discover $maxTS$ in $|\mathcal{S}| - 3f$ replies by missing f of the servers that replied to ω. Let us examine the *seen* set of the servers that reply to both ρ_1 and ρ_2. We know that r belongs to the virtual node ν_r and r' belongs to the virtual node $\nu_{r'}$. There are two cases to consider for ν_r and $\nu_{r'}$: (a) either $\nu_r \neq \nu_{r'}$ (b) or $\nu_r = \nu_{r'}$ (r and r' are siblings). Notice that every server adds the virtual node of a reader to its *seen* set before replying to a read operation. Thus, all $|\mathcal{S}| - 2f$ servers that contained $maxTS$ replied with a $seen = \{0, \nu_r\}$ to r because they added the virtual node of the writer (0) and

the virtual node of r before replying for ρ_1. With similar reasoning all $|\mathcal{S}| - 3f$ servers that replied for ρ_2 send a $seen = \{0, \nu_r, \nu_{r'}\}$ to r'. So, if $\nu_r \neq \nu_{r'}$ then the predicate will hold with $\beta = 2$ for r and with $\beta = 3$ for r'. Thus, r' will also return v preserving atomicity. If, however, $\nu_r = \nu_{r'}$ then the predicate will hold for r but will not hold for r' and, thus r' will return an older value (possibly vp) violating atomicity. As a result, a second round is necessary when a read observes $|\cap_{m \in MS}\, m.seen| = \beta$, and this explains item (1) presented above.

Automaton $\textsc{Sf}_s$**.**

Signature:

Input:
$\mathsf{rcv}(m)_{p,s},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \{w\}$
fail_s

Output:
$\mathsf{send}(m)_{s,p},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \{w\}$

State:

$ts \in \mathbb{N}$, initially 0
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$seen \subseteq \mathcal{V} \cup \mathcal{W}$, initially $\emptyset$
$Counter(p) \in \mathbb{N}^+,\ p \in \mathcal{R} \cup \{w\}$, initially 0
$postit \in \mathbb{N}$, initially 0
$msgType \in \{\text{WRITEACK}, \text{READACK}, \text{INFOACK}\}$
$status \in \{idle, active\}$, initially $idle$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{rcv}(\langle msgT, t, C, \nu_p \rangle)_{p,s}$
Effect:
  if $\neg failed$ then
    if $status = idle$ and $C > Counter(p)$ then
      $status \leftarrow active$
      $Counter(p) \leftarrow C$
      if $t.ts > ts$ then
        $(ts, v, vp) \leftarrow (t.ts, t.v, t.vp)$
        $seen \leftarrow \{\nu_p\}$
      else
        $seen \leftarrow seen \cup \nu_p$
      if $msgT = \text{WRITE}$ then
        $msgType \leftarrow \text{WRITEACK}$
      if $msgT = \text{READ}$ then
        $msgType \leftarrow \text{READACK}$
      if $msgT = \text{INFORM}$ then
        if $t.ts > postit$ then
          $postit \leftarrow t.ts$
        $msgType \leftarrow \text{INFOACK}$

Output $\mathsf{send}(\langle msgT, t, seen, C, postit \rangle)_{s,p}$
Precondition:
  $\neg failed$
  $status = active$
  $p \in \mathcal{R} \cup \{w\}$
  $\langle msgT, t, seen, C, postit \rangle = \langle msgType, \langle ts, v, vp \rangle, Counter(p), postit \rangle$
Effect:
  $status \leftarrow idle$

Input fail_s
Effect:
  $failed \leftarrow true$

Figure 6: $\textsc{Sf}_s$ Automaton: Signature, State and Transitions

The servers maintain a passive role. The signature, state and transitions of the SF_s are given in Figure 6. The state of the SF_s contains the following variables:

- $\langle ts, v, vp\rangle \in \mathbb{N} \times V \times V$: the maximum timestamp received by s in a read/write message along with its associated value and previous value. This is the value of the register replica.

- $seen \in \mathcal{V} \cup \mathcal{W}$: a set that contains the virtual identifiers of the processes that inquired the register replica value from server s. If that process is the writer then it records the id of the writer as it does not belong to any virtual node.

- $postit \in \mathbb{N}$: the largest timestamp received by s in a message sent during the second round of a read operation. The value of this variable indicates that some read operation decided to return the value associated with the timestamp equal to postit.

- $Counter(p) \in \mathbb{N}$: this array maintains the latest request index of each client (reader or writer). It is used by s to distinguish fresh from stale messages.

- $status \in \{idle, active\}$: specifies whether the automaton is processing a request received ($status = active$) or it can accept new requests ($status = idle$).

- $msgType \in \{$WRITEACK,READACK,INFOACK$\}$: Type of the acknowledgment depending on the type of the received message.

Upon receiving a read or write message (i.e., the rcv event occurs) of the form $\langle msgType, \langle ts', v', vp'\rangle, vid\rangle$, the server proceeds as follows. First, it compares its local timestamp ts with the timestamp enclosed in the message ts'. If $ts' > ts$ then it updates the value of its local register copy by assigning $\langle ts, v, vp\rangle = \langle ts', v', vp'\rangle$. Then, it changes its $seen$ set

accordingly: if the server updates its local register value then it resets its $seen$ set and includes only the *vid* enclosed in the message; otherwise it appends its $seen$ set with *vid*. Finally, it sets its local variable $postit = ts'$ if the message received is an INFORM message and $postit$ is smaller than ts'. Once the server updates its local variables it sends a reply (when send occurs) to the requesting process. Each reply is of the form $\langle\langle ts, v, vp\rangle, seen, postit\rangle$.

4.2.4 Correctness of SF

In this section we show the correctness of our algorithm. We first prove that SF satisfies the termination and atomicity conditions (Definitions 3.2.4 and 3.2.5). Then we verify that SF is indeed a semifast implementation by showing that it preserves all properties of Definition 4.1.1. The main result of this section is:

Theorem 4.2.2 Algorithm SF implements a semifast atomic SWMR read/write register.

4.2.4.1 SF Implements a Fault-Tolerant Atomic Read/Write Register

In this section we prove that algorithm SF satisfies all properties of Definitions 3.2.4 (*termination*) and 3.2.5 (*atomicity*) and thus correctly implements a fault tolerant atomic read/write register.

Termination

According to our failure model, any subset of readers, the writer, and up to f servers may crash. Any read or write operation in algorithm SF reaches its fixpoint whenever it receives $|\mathcal{S}| - f$ replies from the server processes. Thus, since no operation waits for replies from any

reader or the writer, and as long as less or equal to f server processes crash, then any operation invoked by a correct process eventually terminates. This, satisfies the termination condition.

Atomicity

Now we prove atomicity of the SF implementation. We proceed by showing that atomic consistency is preserved between operations, no matter whether the invoking processes are siblings or not. Operations in SF are ordered with respect to the values they write or return. In particular, an operation π is ordered before an operation π' if the value written or returned by π is "older" than the value written or returned by π'. To establish the order of the values, SF associates each value with a timestamp. Timestamps written and returned can be used to establish the partial order of operations and whether such order satisfies atomicity. In particular, we say that a value val_1 associated with timestamp ts_1 is "older" than a value val_2 associated with timestamp ts_2 if $ts_1 < ts_2$.

Before proceeding to the proofs we introduce the notation we use throughout the rest of the chapter. We adopt the notation presented in Chapter 3 with some additions. We use var_p to refer to the variable var of the automaton A_p. To access the value of a variable var of A_p in a state σ of an execution ξ, we use $\sigma[p].var$ (see Section 3.1.1). We use the notation $\sigma_{fix(\pi)}$, to capture the state right after the occurrence of a read-phase1-fix event if π is a read operation or write-fix event if π is a write operation. Finally, for an operation π, $\sigma_{inv(\pi)}$ and $\sigma_{res(\pi)}$ denote the system state before the invocation and after the response of operation π respectively (as presented in Section 3.2). Therefore, $\sigma_{res(\pi)}[p].ts$ denotes the value of the variable ts of the automaton A_p at the response step of operation π and is the timestamp returned if π is a read operation.

We use $srvAck(\pi)$ to denote the set of servers that reply to the first round of operation π and $srvInf(\pi)$ to denote the set of servers that reply to the second round of π. Thus, $srvAck(\rho)$ contains the servers that receive the READ messages, and $srvInf(\rho)$ contains the servers that receive the INFORM messages from ρ. The value of the maximum timestamp observed during a read operation ρ from a reader r is $\sigma_{fix(\rho)}[r].maxTS$. As a shorthand we use $maxTS_{\rho} = \sigma_{fix(\rho)}[r].maxTS$ to denote the maximum timestamp witnessed by ρ. By $maxTsSrv(\rho)$ we denote the set of servers that reply with $maxTS_{\rho}$. Let, $maxTsMsg(\rho)$ be the set of messages sent by those servers. Also, let $m(\pi, c)_{p,p'}$ to denote the message sent from p to p' during the cth round of operation π. A variable var enclosed in a message is denoted by $m(\pi, c)_{p,p'}.var$ (see Section 3.1.2). Thus, for a read operation ρ invoked by r, $m(\rho, c)_{s,r} \in maxTsMsg(\rho)$ if $m(\rho, c)_{s,r}.ts = maxTS_{\rho}$.

We can express the atomic order of operations on the basis of timestamps:

TS1. For each process p the ts_p variable is non-negative and monotonically nondecreasing.

TS2. If a write operation $\omega(k)$ precedes a read operation ρ from reader r, such that $\omega(k) \rightarrow \rho$ then, $\sigma_{res(\rho)}[r].ts \geq k$.

TS3. if a read ρ returns k $(k \geq 1)$, then $\omega(k)$ either precedes ρ ($\omega(k) \rightarrow \rho$) or is concurrent with ρ ($\omega(k) \leftrightarrow \rho$),

TS4. If ρ and ρ' are two read operations from the readers r and r' respectively, such that $\rho \rightarrow \rho'$, then $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$.

It is not difficult to see that the order of operations imposed by the conditions on the timestampts provides the partial order required by Defintion 3.2.5.

We show that implementation SF preserves the above conditions in any given execution. We begin with a lemma that plays a significant role in the correctness proof. The lemma follows from the fact that no more than f servers may fail and that the communication channels are reliable.

Lemma 4.2.3 Let two readers with ids $\langle r, \nu \rangle$ and $\langle r', \nu \rangle$ be siblings and invoke reads ρ and ρ' respectively, s.t. $\rho \rightarrow \rho'$. Then, for any execution $\xi \in goodexecs(\text{SF}, f)$, $|\,|maxTsSrv(\rho)| - |maxTsSrv(\rho')|\,| \leq f$.

We now proceed to show the first and third atomicity conditions (as given above) as their proof arguments are simpler to present.

Lemma 4.2.4 For any execution $\xi \in goodexecs(\text{SF}, f)$, if a read operation ρ in ξ returns, it returns a non negative integer.

Proof. From algorithm SF the value of the timestamp is incremented by the automaton SF_w. Since the timestamp variable at any automaton is initialized to 0, then any automaton witnesses a timestamp ≥ 0. As any read operation ρ may return $maxTS_\rho$ or $maxTS_\rho - 1$, when $maxTS_\rho \geq 1$ (and thus some write operation is invoked), then $maxTS_\rho \geq 0$. So it remains to examine the case where $maxTS_\rho = 0$.

Consider an execution ξ of SF. The timestamp ts_s variable of the SF_s automaton, for every server $s \in \mathcal{S}$, is initialized to 0 in ξ. Consider now a read operation ρ in ξ that is performed by the reader $\langle r, \nu_r \rangle$ and witnesses $maxTS_\rho = 0$. It follows that ρ receives more than $|\mathcal{S}| - f$ replies with $maxTS_\rho = 0$. Before replying to ρ, each server $s \in srvAck(\rho)$ adds the virtual id, ν_r, of the reader in its $seen$ set. So, every server $s \in srvAck(\rho)$ will reply with

a timestamp $m(\rho,1)_{s,r}.ts = 0$ and a $m(\rho,1)_{s,r}.seen$ set that contains at least the element ν_r. Thus, $|maxTsSrv(\rho)| \geq |\mathcal{S}| - f$ and the predicate will be true for $\beta = 1$. Therefore, ρ will return $maxTS_\rho = 0$ which is not negative. This completes the proof.

□

We now show that the timestamp of any server is monotonically increasing.

Lemma 4.2.5 In any execution $\xi \in goodexecs(\textsc{Sf}, f)$, if $\sigma[s].ts = k$ for a server $s \in \mathcal{S}$, then, given any state σ' such that σ appears before σ' in ξ and $\sigma'[s].ts = y$, we have that $y \geq k$.

Proof. This is ensured by action $\mathsf{rcv}(m)_{p,s}$ of the $\textsc{Sf}_s$ in Figure 6. The timestamp variable ts_s of the server automaton changes only if the received timestamp from process p for some read/write operation π, is $m(\pi,*)_{p,s}.ts \geq ts_s$ when the $\mathsf{rcv}(m(\pi,*)_{p,s})_{p,s}$ event occurs; otherwise it remains unchanged. Thus, if $\sigma[s].ts = k$ at state σ, then for any state σ' that comes after σ in ξ, $\sigma'[s].ts \geq \sigma[s].ts$, and hence $y \geq k$. □

We now show the monotonicity of the postits for any server.

Lemma 4.2.6 In any execution $\xi \in goodexecs(\textsc{Sf}, f)$, if $\sigma[s].postit = k$ for a server $s \in \mathcal{S}$ then, given any sate σ' such that σ appears before σ' in ξ and $\sigma'[s].postit = y$, we have that $y \geq k$.

Proof. This is ensured by action $\mathsf{rcv}(m)_{p,s}$ of Figure 6. □

Given the above lemmas we can prove condition **TS2** by the following lemma:

Lemma 4.2.7 For any execution $\xi \in goodexecs(\textsc{Sf}, f)$, if a read ρ is complete and succeeds some write $\omega(k)$ ($\omega(k) \rightarrow \rho$), then ρ returns ℓ such that $\ell \geq k$.

Proof. Suppose that the writer ω performs a $\omega(k)$ operation and precedes the read ρ operation by reader $\langle r, \nu_r \rangle$ in execution ξ. Let $srvAck(\omega(k))$ be the $|\mathcal{S}| - f$ servers that reply to $\omega(k)$ in the same execution. The read operation ρ may witness timestamp k from $maxTsSrv(\rho) = srvAck(\omega(k)) \cap srvAck(\rho)$, with cardinality $|maxTsSrv(\rho)| \geq |\mathcal{S}| - 2f$, as ρ may miss f servers that replied to the $\omega(k)$. Since $\omega(k) \rightarrow \rho$ the timestamp $m(\rho, 1)_{s,r}.ts$ from each server $s \in maxTsSrv(\rho)$, per Lemma 4.2.5, is greater or equal to k. So, ρ receives a maximum timestamp $maxTS_\rho \geq k$. From the implementation we know that the reader returns either $maxTS_\rho$ or $maxTS_\rho - 1$. We consider two cases:

Case 1: $maxTS_\rho > k$. Since ρ returns either $maxTS_\rho$ or $maxTS_\rho - 1$, it follows that either case it returns a timestamp greater or equal to k.

Case 2: $maxTS_\rho = k$. As we mentioned above each server in $maxTsSrv(\rho)$ replies with a $m(\rho, 1)_{s,r}.ts \geq k$. Since $maxTS_\rho = k$, every server $s \in maxTsSrv(\rho)$ replies with a timestamp $m(\rho, 1)_{s,r}.ts = k$ to ρ. So, for every $s \in maxTsSrv(\rho)$, $m(\rho, 1)_{s,r} \in maxTsMsg(\rho)$. Thus, $|maxTsMsg(\rho)| \geq |\mathcal{S}| - 2f$. But since every server $s \in maxTsSrv(\rho)$ receives a message with $m(\omega(k), 1)_{w,s}.ts = k$ from the writer before receiving any message from ρ, then w is included in the $seen$ set of s. Also, before any $s \in maxTsSrv(\rho)$ responds to ρ, it includes ν_r in its $seen$ set. So, **SF-RP** will hold for $\beta = 2$ and ρ returns $maxTS_\rho = k$. Observe that any read operation returns $maxTS_\rho$, since the writer w has no sibling, and thus the predicate holds for $\beta = 2$ no matter which reader performs the read operation. □

We say that a $postit = k$ is *introduced* in the system by a read operation ρ, if ρ is complete, performs two rounds, and sends message $(\text{INFORM}, k, _, _)$ during its second round. The following lemma shows that if a $postit = k$ is introduced to the system, then there exists a maximum timestamp $maxTS$ in the system such that $maxTS \geq k$.

Lemma 4.2.8 For any execution $\xi \in goodexecs(\text{SF}, f)$, if $\exists s \in \mathcal{S}$ s.t. $\sigma[s].postit = k$ when it receives a message from a read operation ρ at some state σ, then any succeeding read operation ρ' will observe a maximum timestamp $maxTS_{\rho'} \geq k$.

Proof. Consider an execution ξ of SF where the read operation ρ introduces a postit equal to k to the system. It follows that ρ observes as the maximum timestamp in the system $maxTS_{\rho} = k$. Assume that $|maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$ and $|\cap_{m \in maxTsMsg(\rho)} m.seen| = \beta$, and thus ρ performs an informative operation. Since $\beta \in [1, |\mathcal{V}| + 1]$ and $|\mathcal{S}| > (|\mathcal{V}| + 2)f$, we get that $|maxTsMsg(\rho)| > f$. So, if $srvAck(\rho')$ is the set of servers that replies to ρ' ($|srvAck(\rho')| = |\mathcal{S}| - f$), then per Lemma 4.2.5 there is a server, $s \in maxTsSrv(\rho) \cap srvAck(\rho')$ that replies to ρ' with a timestamp $m(\rho', 1)_{s,r'}.ts \geq k$. Since, ρ' detects a maximum timestamp $maxTS_{\rho'} \geq m(\rho', 1)_{s,r'}.ts$, hence $maxTS_{\rho'} \geq k$. □

Lemma 4.2.9 For any execution $\xi \in goodexecs(\text{SF}, f)$ if a read operation ρ receives a $postit = k$ then ρ will return a value $y \geq k$.

Proof. Consider an execution ξ of SF which contains a read operation ρ by a reader $\langle r, \nu_r \rangle$. It follows from Lemma 4.2.8 that if read ρ receives a $postit = k$, then it will detect a maximum timestamp $maxTS_{\rho} \geq k$. Let $maxTS_{\rho} = k$. So, either the predicate will hold or the condition whether $postit_r = maxTS_{\rho}$ will be true and ρ will return $y = maxTS_{\rho}$ in both cases. Thus ρ will return $y = k$. If now $maxTS_{\rho} > k$ then ρ will return $y = maxTS_{\rho}$ if the predicate holds or $y = maxTS_{\rho} - 1$ otherwise. Note that since $postit = k$, it is less than $maxTS_{\rho}$ and so the postit condition does not hold. Either case ρ will return a value $y \geq k$. □

We proceed to the proof of condition **TS3**.

Lemma 4.2.10 In any execution $\xi \in goodexecs(\textsc{Sf}, f)$, if a read operation ρ returns $k \geq 1$, then the write operation $\omega(k)$ either precedes ρ ($\omega(k) \rightarrow \rho$) or is concurrent with ρ.

Proof. Consider an execution ξ of $\textsc{Sf}$. Note that in order for a timestamp k to be introduced in the system during ξ a write operation $\omega(k)$ must be invoked (since only the writer increments the timestamp). We now investigate what happens when a reader returns a timestamp $ts = k$ in ξ. We know that ρ returns, according to the implementation, either $k = maxTS_\rho$ or $k = maxTS_\rho - 1$. The first case is possible if the predicate **SF-RP** holds for the reader or if the reader observes some $postit = maxTS_\rho$. If the predicate holds, ρ detects $maxTS_\rho = k$ in $|maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$ messages. Since $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$ and $\beta \leq |\mathcal{V}| + 1$, then $|maxTsMsg(\rho)| > f$. So, there is at least one server $s \in srvAck(\rho)$ that receives messages from $\omega(k)$ before replying to ρ. If ρ returns k because of a postit, then per Lemma 4.2.8, timestamp k was already introduced in the system. Thus, for both cases $\omega(k)$ is either concurrent or precedes the read operation ρ.

In the case where the reader returns $k = maxTS_\rho - 1$ it follows that the reader detects a maximum timestamp $maxTS_\rho = k + 1$ in the system. Thus, the ω_{k+1} operation has already been initiated by the writer. Hence, $\omega(k)$ operation has already been completed and preceded ρ or was concurrent and completed before ρ completes. $\square$

We next prove that condition **TS4**, is satisfied for read operations invoked from sibling (Lemma 4.2.11) or non-sibling (Lemma 4.2.12) readers. We proceed by investigating all cases of the predicate **SF-RP** used in algorithm $\textsc{Sf}$ in either of the above cases. Note, that the relation between the readers may affect the cardinality of the $seen$ set at the server side. Thus, we analyze each case separately.

Lemma 4.2.11 For any execution $\xi \in goodexecs(\text{SF}, f)$ that contains two read operation ρ and ρ', s.t. ρ and ρ' are invoked by the sibling readers $\langle r, z\rangle$ and $\langle r', z\rangle$ respectively and $\rho \rightarrow \rho'$, then if ρ returns k and ρ' returns y, $y \geq k$.

Proof. We consider an execution ξ of SF. We first investigate the case where $r = r'$. In this case ρ denotes the first read operation of r and ρ' a succeeding read operation from the same reader. Let $\sigma_{res(\rho)}[r].ts = k$ be the value returned from ρ. During the read ρ', r sends a READ message to every server $s \in \mathcal{S}$ with $m(\rho', 1)_{r,s}.ts = maxTS_{\rho} \geq k$. This message is received by every server $s' \in srvAck(r')$ which according to Lemma 4.2.5 replies with a timestamp $m(\rho', 1)_{s,r}.ts \geq maxTS_{\rho} \geq k$. So, $maxTS_{\rho'} \geq k$. If $maxTS_{\rho'} = k$ then $|maxTsMsg(\rho')| = |\mathcal{S}| - f$ and the predicate holds for $\beta = 1$. Thus, $y = maxTS_{\rho'} = k$. Otherwise, if $maxTS_{\rho'} > k$, the y will be equal to $maxTS_{\rho'}$ or $maxTS_{\rho'} - 1$ and thus, in either case $y \geq k$. By a simple induction we can show that this is true for every read operation of r(including ρ') after ρ. For the rest of the proof we assume that $r \neq r'$. We investigate the following two possible cases: (1) ρ returns $k = maxTS_{\rho} - 1$ and (2) ρ returns $k = maxTS_{\rho}$. In all cases we show that $k \leq y$ or that the case is impossible.

Case 1: In this case $k = maxTS_{\rho} - 1$. Therefore, some servers reply to ρ with $maxTS_{\rho} = k + 1$, and hence a write operation $\omega(k+1)$ has started before ρ is completed. So $\omega(k)$ completes before ρ completes and therefore $\omega(k) \rightarrow \rho'$ since $\rho \rightarrow \rho'$. Thus by Lemma 4.2.7 ρ' returns a value $y \geq k$.

Case 2: In this case $k = maxTS_{\rho}$. Hence either there is some $\beta \in [1, |\mathcal{V}| + 1]$ such that $|maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$ and $|\cap_{m \in maxTsMsg(\rho)} m.seen| \geq \beta$ or ρ received a *postit* equal to $maxTS_{\rho}$ from some server. We examine those two possibilities.

Case 2(a): It follows that $k = maxTS_\rho$, and there is some $\beta \in [1, |\mathcal{V}| + 1]$ such that $maxTsMsg(\rho)$ consist at least $|\mathcal{S}| - \beta f$ messages received by ρ with timsetamp k and $|\cap_{m \in maxTsMsg(\rho)} m.seen| \geq \beta$. Since $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$ and $\beta \in [1, |\mathcal{V}| + 1]$, then $|maxTsMsg(\rho)| = |\mathcal{S}| - \beta f > f$. Following we investigate the cases where $|\cap_{m \in maxTsMsg(\rho)} m.seen| = \beta$ and $|\cap_{m \in maxTsMsg(\rho)} m.seen| > \beta$. (1) First lets examine the case where ρ returns $k = maxTS_\rho$ because $|\cap_{m \in maxTsMsg(\rho)} m.seen| = \beta$. According to the implementation, ρ has to inform $|srvInf(\rho)| \geq 2t + 1$ servers about its return value, k. Since ρ precedes ρ', at least $|srvInf(\rho) \cap srvAck(\rho')| \geq f + 1$ servers, that informed by ρ, will reply to ρ'. Any server $s \in srvInf(\rho) \cap srvAck(\rho')$, by Lemma 4.2.8 will reply with a $m(\rho', 1)_{s,r'}.postit \geq k$ to ρ' and with a timestamp $m(\rho', 1)_{s,r'}.ts \geq k$. So ρ' will observe a maximum timestamp $maxTS_{\rho'} \geq k$. According now to Lemma 4.2.9 ρ' will return a value $y \geq k$. (2) The second case arise when ρ returns $k = maxTS_\rho$ because $|\cap_{m \in maxTsMsg(\rho)} m.seen| > \beta$. We can split this case in two subcases regarding the value returned by ρ'. The two possible values that ρ' might return is $y = maxTS_{\rho'}$ or $y = maxTS_{\rho'} - 1$:

(i) Consider the case where $y = maxTS_{\rho'}$. Since ρ returned $k = maxTS_\rho$, as we showed in Lemma 4.2.10 , there is a write operation $\omega(k)$ that precedes or is concurrent with ρ. As stated above $|maxTsMsg(\rho)| > f$ and hence there is a server s such that $s \in maxTsSrv(\rho) \cap srvAck(\rho')$. By Lemma 4.2.5, s sends a timestamp $m(\rho', 1)_{s,r'}.ts \geq k$ to ρ', and hence $maxTS_{\rho'} \geq k$. So $y \geq k$.

(ii) Now, consider the case where ρ' returns $y = maxTS_{\rho'} - 1$. Since $|maxTsSrv(\rho)| > f$, there must be a server $s \in maxTsSrv(\rho) \cap srvAck(\rho')$ and s replies with a timestamp $m(\rho', 1)_{s,r'}.ts \geq k$ to ρ'. So the highest timestamp in $srvAck(\rho')$(i.e. $maxTS_{\rho'} = y + 1$)

will be greater or equal to k. If the inequality is true, namely $y + 1 > k$, then clearly the value returned by ρ' is $y \geq k$. If the equality holds and $y + 1 = x$ then the highest timestamp received by ρ', $maxTS_{\rho'} = y + 1 = k$. Hence, any server $s \in maxTsSrv(\rho) \cap srvAck(\rho')$ replies with a timestamp $m(\rho', 1)_{s,r'}.ts = x = y + 1$ to ρ'. Recall that in this case we assumed that $| \cap_{m \in maxTsMsg(\rho)} m.seen| > \beta$. Also, according to Lemma 4.2.3, $||maxTsMsg(\rho')| - |maxTsMsg(\rho)|| \leq f$ and since $|maxTsSrv(\rho)| = |maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$, it follows that ρ' receives the maximum timestamp $maxTS_{\rho'} = k$ from $|maxTsSrv(\rho')| = |maxTsMsg(\rho')| \geq |\mathcal{S}| - (\beta + 1)f$ servers. Notice, that for any $s \in maxTsSrv(\rho) \cap \mathcal{S}2$, $m(\rho, 1)_{s,r}.ts = m(\rho', 1)_{s,r'}.ts = k$. Since the timestamp is the same and s sent $m(\rho, 1)_{s,r}$ before $m(\rho', 1)_{s,r'}$ then $m(\rho, 1)_{s,r}.seen \subseteq m(\rho', 1)_{s,r'}.seen$. As a result $| \cap_{m \in maxTsMsg(\rho)} m.seen| \leq | \cap_{m \in maxTsMsg(\rho')} m.seen|$. Notice that, since the two readers are siblings, if no *non-sibling* reader receives replies from those servers in between ρ and ρ', then $m(\rho, 1)_{s,r}.seen = m(\rho', 1)_{s,r'}.seen$ and $| \cap_{m \in maxTsMsg(\rho)} m.seen| = | \cap_{m \in maxTsMsg(\rho')} m.seen|$. Either case, $| \cap_{m \in maxTsMsg(\rho')} m.seen| > \beta$ and hence $| \cap_{m \in maxTsMsg(\rho')} m.seen| \geq \beta + 1$. Observe that the predicate **SF-RP** holds for $\beta + 1$ since $|maxTsMsg(\rho')| \geq |\mathcal{S}| - (\beta + 1)f$, and thus ρ' must return $maxTS_{\rho'} = k = y + 1$, contradicting the initial assumption that ρ' returns $y = k - 1$. The same result applies in both cases where $\beta \leq |\mathcal{V}|$ and $\beta = |\mathcal{V}| + 1$ since the $seen$ set remains unchanged.

Case 2(b): Here ρ returns $k = maxTS_{\rho}$ because some postits equal to $maxTS_{\rho}$ received by ρ. We have to consider two cases here. Either (1) ρ receives more than $f + 1$ postits, or (2) ρ receives less than $f + 1$ postits. Both cases imply that, a reader $\langle r'', \nu_{r''} \rangle$ performed a read ρ'', and is about to return or already returned the maximum timestamp (which is equal to $maxTS_{\rho}$) in the system. Furthermore implies that ρ'' initiates an informative phase which is concurrent

or precedes the read operation ρ. By analyzing the cases we obtain the following results:

(1) If ρ receives more than or equal to $f+1$ messages containing a postit with value $postit = maxTS_\rho = k$, then the writer w initiated a $\omega(k)$ operation during or before ρ is completed. Every server $s \in srvInf(\rho'') \cap srvAck(\rho)$ replies to ρ with $m(\rho,1)_{s,r}.postit = maxTS_\rho$. The reader r' receives replies from $|srvAck(\rho')| = |\mathcal{S}| - f$ servers. Since $|srvInf(\rho'') \cap srvAck(\rho)| \geq f+1$, then $|srvAck(\rho') \cap (srvInf(\rho'') \cap srvAck(\rho))| \geq 1$. So the read operation ρ' receives a reply from at least one server $s \in srvInf(\rho'') \cap srvAck(\rho)$. Hence, from Lemma 4.2.6, ρ' receives a $postit_s \geq k$ from s and according to Lemma 4.2.9 will return a value $y \geq postit_s$ and thus $y \geq k$.

(2) Let us now examine the case where ρ receives less than $f+1$ messages containing postits with value equal to $maxTS_\rho$. Let us assume again that $|srvInf(\rho'') \cap srvAck(\rho)| < f+1$ is the set of servers that reply with $postit = maxTS_\rho$ to ρ. Since $|srvAck(\rho')| = |\mathcal{S}| - f$, it is possible that $|(srvInf(\rho'') \cap srvAck(\rho)) \cap srvAck(\rho')| = 0$, So ρ informs $|srvInf(\rho)| \geq 2f+1$ servers with a $postit = maxTS_\rho$ before completing. So there exists a server $s' \in srvAck(\rho') \cap srvInf(\rho)$ that replies to ρ'. By Lemma 4.2.6, s' replies with a $m(\rho',1)_{s',r'}.postit \geq maxTS_\rho$, and by Lemma 4.2.9, ρ' returns a timestamp $y \geq m(\rho',1)_{s',r'}.postit$. Hence ρ' returns a value $y \geq k$. □

We now examine if the timestamps returned by non sibling processes satisfy condition **TS4** presented above.

Lemma 4.2.12 For any execution $\xi \in goodexecs(\text{SF}, f)$ that contains two read operation ρ and ρ', s.t. ρ and ρ' are invoked by non-sibling readers $\langle r, \nu_r\rangle$ and $\langle r', \nu_{r'}\rangle$ respectively and $\rho \rightarrow \rho'$, then if ρ returns k and ρ' returns y, $y \geq k$.

Proof. Consider an execution ξ of SF. We study the case where $r \neq r'$ and $\nu_r \neq \nu_{r'}$ in ξ, and hence the two readers are not siblings. We proceed in cases and show that $y \geq k$ or the case is impossible. We know that ρ may return either $maxTS_\rho$ or $maxTS_\rho - 1$. It can be shown similarly to case (1) of Lemma 4.2.11 that when ρ returns $k = maxTS_\rho - 1$ then ρ' returns $y \geq k$. It remains to investigate the cases where: (1) ρ returns $maxTS_\rho$ because the predicate **SF-RP** does not hold but it receives some $postit = maxTS_\rho$, and (2) ρ returns $maxTS_\rho$ because it receives $|maxTsMsg(\rho)|$ messages that contain the maximum timestamp $maxTS_\rho$ such that there is $\beta \in [1 \dots |\mathcal{V}| + 1]$ and $|maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$ and $| \cap_{m \in maxTsMsg(\rho)} m.seen| \geq \beta$.

Case 1: In this case ρ returns $k = maxTS_\rho$ because some server $s \in srvAck(\rho)$ replies with $m(\rho, 1)_{s,r}.postit = maxTS_\rho$. According to SF some process (sibling or not of r), say r'', performs a read operation ρ'' and is about, or already returned a timestamp equal to $maxTS_\rho$. There are two cases to consider based on the cardinality of $srvInf(\rho'') \cap srvAck(\rho)$: (1) $|srvInf(\rho'') \cap srvAck(\rho)| \geq f + 1$ and (2) $|srvInf(\rho'') \cap srvAck(\rho)| < f + 1$. If (1) is true and r receives $|srvInf(\rho'') \cap srvAck(\rho)| \geq f + 1$, then ρ returns $k = maxTS_\rho$ without performing a second communication round. Since the set of servers that respond to ρ' is $|srvAck(\rho')| = |\mathcal{S}| - f$, it follows that there is at least one server $s \in srvAck(\rho') \cap (srvInf(\rho'') \cap srvAck(\rho))$. According to Lemma 4.2.6, s replies to ρ' with a $m(\rho', 1)_{s,r'}.postit \geq k$. Furthermore by Lemma 4.2.9, ρ' returns a value $y \geq m(\rho', 1)_{s,r'}.postit$. So obviously ρ' returns a value $y \geq k$. On the other hand if (2) is true and ρ receives $|srvInf(\rho'') \cap srvAck(\rho)| < f + 1$ postits, then, before returning, ρ informs every server $s \in srvInf(\rho)$ ($|srvInf(\rho)| \geq 2f + 1$) with a $m(\rho, 2)_{r,s}.postit = maxTS_\rho$.

So there exists a server $s \in srvAck(\rho') \cap srvInf(\rho)$ that replies to ρ'. By Lemma 4.2.6 s, replies with a $m(\rho', 1)_{s,r'}.postit \geq maxTS_\rho$ and by Lemma 4.2.9 it follows that ρ' returns a timestamp $y \geq m(\rho', 1)_{s,r'}.postit$. Hence it follows again that $y \geq k$.

Case 2: This is the case where ρ returns $maxTS_\rho$ because the predicate **SF-RP** holds, namely, there is $\beta \in [1 \ldots |\mathcal{V}| + 1]$ and $|maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$ such that $|\cap_{m \in maxTsMsg(\rho)} m.seen| \geq \beta$. Recall again that since $\beta \in [1 \ldots |\mathcal{V}| + 1]$ and $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$, $|maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f > f$. So, if $maxTsSrv(\rho)$ are the servers that reply with messages in $maxTsMsg(\rho)$, there is at least one server $s \in maxTsSrv(\rho) \cap srvAck(\rho')$. Therefore, s replies to ρ', by Lemma 4.2.5, with a timestamp $m(\rho', 1)_{s,r'}.ts \geq k$. Hence, ρ' observes a maximum timestamp $maxTS_{\rho'} \geq k$. If ρ' observes $maxTS_{\rho'} > k$ then clearly, since ρ' returns either $y = maxTS_{\rho'}$ or $y = maxTS_{\rho'} - 1$, it returns a value $y \geq k$. It remains to investigate the case where the maximum timestamp observed by ρ' is $maxTS_{\rho'} = k$. Since $maxTS_{\rho'} = maxTS_\rho = k$ it follows that any server in $s \in maxTsSrv(\rho) \cap srvAck(\rho')$ replies to ρ' with a timestamp $m(\rho', 1)_{s,r'}.ts = k$. Furthermore, since ρ' might miss up to f servers from $maxTsSrv(\rho)$ and $|maxTsSrv(\rho)| = |maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$, it follows that ρ' receives the maximum timestamp $maxTS_{\rho'} = k$ from $|maxTsSrv(\rho')| = |maxTsMsg(\rho')| \geq |\mathcal{S}| - (\beta + 1)f$ servers. There are two possible return values for ρ'. Either $y = maxTS_{\rho'} = k$ or $y = maxTS_{\rho'} - 1 \Rightarrow y + 1 = k$. So the only case that needs further investigation is when $y + 1 = k$. We consider two possible scenarios: ρ satisfies the **SF-RP** with an (1) $\beta < |\mathcal{V}| + 1$ and (2) $\beta = |\mathcal{V}| + 1$.

Case 2(a): Here ρ satisfies the predicate using $\beta < |\mathcal{V}| + 1$. This implies that $\cap_{m \in maxTsMsg(\rho)} m.seen$ might contain less than $|\mathcal{V}| + 1$ elements and thus not every virtual

identifier is included. So we have to consider two subcases: (1) $\nu_{r'} \notin \cap_{m \in maxTsMsg(\rho)} m.seen$ and (2) $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho)} m.seen$.

(1) Let us first assume that $\nu_{r'} \notin \cap_{m \in maxTsMsg(\rho)} m.seen$. Consider the set of servers $maxTsSrv(\rho) \cap srvAck(\rho')$. Since $|maxTsSrv(\rho)| = |maxTsMsg(\rho)| \geq |\mathcal{S}| - \beta f$ and $|srvAck(\rho')| = |\mathcal{S}| - f$ then $|maxTsSrv(\rho) \cap srvAck(\rho')| \geq |\mathcal{S}| - (\beta + 1)f \geq 1$. Also, since the read ρ precedes ρ', and any server $s \in maxTsSrv(\rho)$ replies with $m(\rho, 1)_{s,r}.ts = maxTS_{\rho} = k$ to ρ, then any server $s' \in maxTsSrv(\rho) \cap srvAck(\rho')$ replies with a timestamp $m(\rho', 1)_{s',r'}.ts \geq k$ to ρ'. So, all the servers in the set $maxTsSrv(\rho) \cap srvAck(\rho')$ reply to ρ' with $m(\rho', 1)_{s',r'}.ts = k = y + 1$. For any server $s' \in maxTsSrv(\rho) \cap srvAck(\rho')$, we know that $m(\rho, 1)_{s',r}.ts = m(\rho', 1)_{s',r'}.ts = k$. Since $m(\rho, 1)_{s',r}$ is sent before $m(\rho', 1)_{s',r'}$, then $m(\rho, 1)_{s',r}.seen \subseteq m(\rho', 1)_{s',r'}.seen$. Thus $\cap_{m \in maxTsMsg(\rho)} m.seen \subseteq \cap_{m \in maxTsMsg(\rho')} m.seen$. Moreover, every server $s \in maxTsSrv(\rho')$ adds $\nu_{r'}$ into its $seen$ set before replying to ρ'. Therefore, $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho')} m.seen$. By assumption, $\nu_{r'} \notin \cap_{m \in maxTsMsg(\rho)} m.seen$, and so it follows that $| \cap_{m \in maxTsMsg(\rho')} m.seen| \geq | \cap_{m \in maxTsMsg(\rho)} m.seen| + 1 \geq \beta + 1$. Since $|maxTsSrv(\rho')| = |maxTsMsg(\rho')|$ and $|maxTsSrv(\rho')| \geq |maxTsSrv(\rho) \cap srvAck(\rho')| \geq |\mathcal{S}| - (\beta + 1)f$, then the predicate **SF-RP** holds for ρ' with $\beta + 1$. Thus ρ' returns $maxTS_{\rho'} = k = y + 1$, contradicting the assumption that it returns $y < k$.

(2) Let us now consider the case where $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho)} m.seen$. So either (i) r' itself or (ii) a sibling of r' performs a read operation before ρ'. Assume that (i) r' itself performs a read, say ρ'', before ρ'. So since $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho)} m.seen$, r' receives a maximum timestamp $maxTS_{\rho''} = maxTS_{\rho}$ during read operation ρ''. Due to well-formedness (Definition 3.2.2) $\rho'' \to \rho'$ and so, during ρ', r' sends a READ message

with $m(\rho', 1)_{r',s}.ts = maxTS_\rho \geq k$ to every server $s \in \mathcal{S}$. This message is received by any server $s \in srvAck(\rho')$ which according to Lemma 4.2.5 replies with a timestamp $m(\rho', 1)_{s,r'}.ts \geq maxTS_\rho \geq k$. If $m(\rho', 1)_{s,r'}.ts = k$ then the set of servers that reply with the maximum timestamp $maxTS_\rho$ to ρ' is $|maxTsSrv(\rho')| = |srvAck(\rho')| \geq |\mathcal{S}| - f$. Since every server $s \in srvAck(\rho')$ before replies to ρ' adds $\nu_{r'}$ to its $seen$ set, then predicate **SF-RP** holds with $\beta = 1$. If now $m(\rho', 1)_{s,r'}.ts > k$, then ρ' returns a value y such that $y = m(\rho', 1)_{s,r'}.ts$ or $y = m(\rho', 1)_{s,r'}.ts - 1$ and thus in any case $y \geq k$. Both cases contradict the assumption that $y + 1 = k$.

In case (ii) $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho)} m.seen$ because a sibling of r' initiates a read operation before ρ'. As we discussed above, $|maxTsSrv(\rho')| \geq |maxTsSrv(\rho) \cap srvAck(\rho')| \geq |\mathcal{S}| - (\beta + 1)f$ and furthermore any server $s \in maxTsSrv(\rho) \cap srvAck(\rho')$ replies to ρ' with a timestamp $m(\rho', 1)_{s,r'}.ts = k = y + 1$. Let $m(\rho, 1)_{s,r}$ and $m(\rho', 1)_{s,r'}$ be the messages of a server $s \in maxTsSrv(\rho) \cap srvAck(\rho')$, in $maxTsMsg(\rho)$ and $maxTsMsg(\rho')$ respectively. We know that $m(\rho, 1)_{s,r}.ts = m(\rho', 1)_{s,r'}.ts$. Since $m(\rho, 1)_{s,r}$ is sent before $m(\rho', 1)_{s,r'}$, then $m(\rho, 1)_{s,r}.seen \subseteq m(\rho', 1)_{s,r'}.seen$. Thus, $\cap_{m \in maxTsMsg(\rho)} m.seen \subseteq \cap_{m \in maxTsMsg(\rho')} m.seen$. Every server that replies to ρ', first adds $\nu_{r'}$ into its $seen$ set and thus $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho')} m.seen$. Since though, $\nu_{r'}$ was already in $\cap_{m \in maxTsMsg(\rho)} m.seen$, it follows that $| \cap_{m \in maxTsMsg(\rho')} m.seen| \geq | \cap_{m \in maxTsMsg(\rho)} m.seen| \geq \beta$. If $| \cap_{m \in maxTsMsg(\rho)} m.seen| > \beta$, then $| \cap_{m \in maxTsMsg(\rho')} m.seen| \geq \beta + 1$. Since $|maxTsSrv(\rho')| = |maxTsMsg(\rho')| \geq |\mathcal{S}| - (\beta + 1)f$, **SF-RP** holds with $\beta + 1$ and ρ' returns $maxTS_{\rho'} = k = y + 1$. If on the other hand $| \cap_{m \in maxTsMsg(\rho)} m.seen| = \beta$ then ρ performs an informative operation before returning, sending the $m(\rho, 2)_{r,s}.postit = k$ to every server s in the set $|srvInf(\rho)| \geq 2f + 1$.

So, there is a server $s \in srvAck(\rho') \cap srvInf(\rho)$ that replies, by Lemma 4.2.6, with a $m(\rho', 1)_{s,r'}.postit \geq k$ to ρ'. So according to Lemma 4.2.9, ρ' returns a value $y \geq m(\rho', 1)_{s,r'}.postit \geq k$. Hence we derive contradiction based on the initial assumption that $k = y + 1$.

Case 2(b): ρ satisfies the predicate with $\beta = |\mathcal{V}| + 1$. Since $|\mathcal{W} \cup \mathcal{V}| = |\mathcal{V}| + 1$ and $|\cap_{m \in maxTsMsg(\rho)} m.seen| \geq \beta = |\mathcal{V}| + 1$, it follows that $\nu_{r'} \in \cap_{m \in maxTsMsg(\rho)} m.seen$. Observe that the set of servers that reply to ρ with messages in $maxTsMsg(\rho)$, $|maxTsSrv(\rho)| \geq |\mathcal{S}| - \beta f > f$. So as shown in the previous case (Case 2(a)) ρ' returns a value $y \geq k$ deriving a contradiction. □

Theorem 4.2.13 Algorithm SF implements an atomic read/write register in the SWMR model.

Proof. Since the writer, any subset of readers and up to f servers might fail by crashing, we ensure termination in any execution of the implementation by letting any reader or writer to wait for messages only from $|\mathcal{S}| - f$ servers during any communication round. All the conditions **TS1** - **TS4** are preserved in any execution ξ by Lemmas 4.2.4, 4.2.10, 4.2.7, 4.2.12. Thus, the order of operations satisfy all the atomicity properties (Definition 3.2.5). Since both termination and atomicity properties are preserved the result follows. □

4.2.4.2 SF is a SemiFast implementation

In this section we show that implementation SF is a semifast implementation, that is, it satisfies all the properties of Definition 4.1.1. We use the same notation as in the previous section. We first show that SF satisfies property **S3** of Definition 4.1.1.

Lemma 4.2.14 For any execution $\xi \in goodexecs(\text{SF}, f)$, if ρ is a two-round read operation, then any read operation ρ' with $\mathfrak{R}(\rho) = \mathfrak{R}(\rho')$, such that $\rho \to \rho'$ or $\rho' \to \rho$, must be fast.

Proof. Since ρ' may precede or succeed ρ we examine the two cases. We proceed by considering an execution ξ of SF that contains both ρ and ρ', and we show that in each case ρ' is fast or the case is not possible. For the rest of the proof we study the timestamps returned by the read operations since every value is associated with a unique timestamp. Let us assume that timestamp $ts = k$ is associated with val_k, written by the unique write operation $\mathfrak{R}(\rho) = \mathfrak{R}(\rho') = \omega(k)$.

Case 1: Starting with the case where $\rho \to \rho'$ there are two subcases to investigate: (a) ρ' observes a maximum timestamp equal to k, and (b) ρ' observes a maximum timestamp $k+1$. Obviously in the second subcase, ρ' is concurrent with $\omega(k+1)$ but $\omega(k+1)$ is not yet completed.

The fast behavior of ρ' in the first subcase follows from the fact that ρ informs every server s in the set $|srvInf(\rho)| \geq 2f+1$ with the timestamp $m(\rho, 2)_{r,s}.ts = k$. So ρ' witnesses $|srvAck(\rho') \cap srvInf(\rho)| \geq f+1$ postits equal to k during its first communication round. Since the maximum timestamp $maxTS_{\rho'}$ observed by ρ' is also equal to k, then ρ', according to Lemma 4.2.9, returns $maxTS_{\rho'}$ no matter the validity of **SF-RP**. Moreover since $|srvAck(\rho') \cap srvInf(\rho)| \geq f+1$ any subsequent read operation witnesses at least one server in $srvInf(\rho)$ and thus ρ' completes without proceeding to a second communication round.

Consider now the second subcase where ρ' observes a maximum timestamp equal to $k+1$. From the implementation we know that a read operation may return either the observed maximum timestamp $maxTS_{\rho'}$ or $maxTS_{\rho'} - 1$. Since ρ' returns k, it implies that a decision

for returning $maxTS_{\rho'} - 1$ is taken by ρ'. According to the implementation, a reader may perform a second communication round only when it decides to return $maxTS_{\rho'}$. In any other case the reader is not required to perform two communication rounds. So ρ' returns $maxTS_{\rho'} - 1$ in one communication round as desired.

Case 2: Consider now the case where $\rho' \to \rho$. Since ρ performs two communication rounds, it returns the maximum timestamp k, that ρ observed during its first communication round. On the other hand ρ' also returns k, by either returning $maxTS_{\rho'}$ or $maxTS_{\rho'} - 1$. So ρ' may observe a maximum timestamp $maxTS_{\rho'} = k$ or $maxTS_{\rho'} = k + 1$.

Let us first investigate the case where $maxTS_{\rho'} = k+1$. Recall that ρ receives replies from $|srvAck(\rho)| = |\mathcal{S}| - f$ servers. Since ρ observes a $maxTS_{\rho} = k$, then if $maxTS_{\rho'} = k + 1$, it means that $k + 1$ is introduced to less than f servers in the system. In order for ρ' to satisfy **SF-RP** there must exist an $maxTsMsg(\rho')$ that contains messages of the servers in $maxTsSrv(\rho')$, such that $|maxTsMsg(\rho')| \geq |\mathcal{S}| - \beta f$ for $\beta \in \{1, \ldots, |\mathcal{V}| + 1\}$ and $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$. Therefore we require that $|maxTsMsg(\rho')| \geq f$. However since $|maxTsSrv(\rho')| \geq |maxTsMsg(\rho')|$ and $|maxTsSrv(\rho')| \leq f$, we have that $|maxTsMsg(\rho')| \leq f$ and thus the predicate does not hold for ρ'. Notice that for each read operation $\rho'' \to \rho$ (including ρ') observing a maximum timestamp $maxTS_{\rho''} = k + 1$, **SF-RP** does not hold and hence no read operation performs a second communication round informing the servers with a $postit = k+1$. So it follows that the second condition whether there are $postit = k + 1$ is false for ρ' as well and thus ρ' returns $maxTS_{\rho'} - 1 = k$. As previously stated, if a ρ' returns $maxTS_{\rho'} - 1$, it does so in one communication round.

It is left to examine the case where ρ' observes a $maxTS_{\rho'} = k$. Remember that ρ performs a second communication round in two cases: (a) the predicate holds with $|\cap_{m \in maxTsMsg(\rho)} m.seen| = \beta$ and (b) it observes "insufficient" postits sent by a concurrent read operation. For simplicity of our analysis we assume that no read operation is concurrent with ρ and that ρ performes a second communication round because case (a) is true. Since ρ' returns $maxTS_{\rho'} = k$ then either (i) the predicate holds for ρ' or (ii) ρ' observed some $postit = k$. Let us examine those subcases and show that in each case ρ' is fast or the case is impossible.

Suppose that the predicate holds for ρ'. So there is an $\beta \in \{1, \ldots, |\mathcal{V}| + 1\}$ and there is $|maxTsMsg(\rho')| \geq |\mathcal{S}| - \beta f$ such that $|\cap_{m \in maxTsMsg(\rho')} m.seen| \geq \beta$. If $|\cap_{m \in maxTsMsg(\rho')} m.seen| = \beta$ then ρ' proceeds to a second communication round informing $|srvInf(\rho')| \geq 2f + 1$ servers about the maximum timestamp is about to return, $maxTS_{\rho'} = k$. Since $\rho' \rightarrow \rho$, then $|srvAck(\rho) \cap srvInf(\rho')| \geq f + 1$, and thus, ρ observes "enough" postists equal to $k = maxTS_{\rho}$ and does not perform a second communication round. This however contradicts our initial assumption, rendering this case impossible for ρ'. Therefore the predicate validity is possible for ρ', only if $|\cap_{m \in maxTsMsg(\rho')} m.seen| > \beta$. This is the case though where ρ' returns in one communication round as desired.

It remains to study the case where ρ' returns $maxTS_{\rho'} = k$ because of some postits equal to k. There are two subcases to consider: (1) ρ' does not observe more than $f + 1$ postits so it performs a second communication round and (2) ρ' observes more than $f + 1$ postits and returns in one communication round. The first subcase results in ρ performing only one communication round as described above contradicting our initial assumption. In the second subcase there is a read operation ρ'' that is concurrent or precedes ρ' and performs two communication

rounds. Since ρ' receives more than $f+1$ postits equal to $maxTS_{\rho'}$, it returns in one communication round. Moreover, since we assumed that no read operation is concurrent with ρ, then ρ'' completes before the invocation of ρ. So ρ'' will inform at least $|srvInf(\rho'')| = 2f+1$ servers with a postit equal to k. Hence $|srvAck(\rho) \cap srvInf(\rho'')| \geq f+1$ and thus ρ returns in one communication round leading to contradiction. $\square$

We now show that SF satisfies the fourth property of Definition 4.1.1. The following proof assumes that all the read operations are concurrent with the write operation and yet are fast.

Lemma 4.2.15 There exists an execution $\xi \in goodexecs(\text{SF}, f)$ that contains at least one write operation $\omega(k)$ and the set of read operations $\Pi = \{\rho : \mathfrak{R}(\rho) = \omega(k)\}$, such that $|\Pi| \geq 1$, $\exists \rho \in \Pi$, ρ is concurrent with $\omega(k)$ and $\forall \rho \in \Pi$, ρ is fast.

Proof. Consider that each read operation $\rho \in \Pi$ returns the timestamp written by $\mathfrak{R}(\rho) = \omega(k)$. Recall that a read operation ρ returns either the maximum timestamp $maxTS_\rho$ or $maxTS_\rho - 1$. So the timestamp k is returned by ρ either when ρ witnesses $maxTS_\rho = k$ or when it witnesses $maxTS_\rho = k+1$. A read operation is fast in the following cases: (1) the predicate **SF-RP** holds and $|\bigcap_{m \in maxTsMsg(\rho)} m.seen| > \beta$, or (2) more than $f+1$ postits equal to $maxTS_\rho$ witnessed, or (3) the operation returns $maxTS_\rho - 1$. A read operation need performs two communication rounds when $|\bigcap_{m \in maxTsMsg(\rho)} m.seen| = \beta$ or when ρ observes less than $f+1$ postits equal to $maxTS_\rho$ in the replies from the servers.

Let us assume, to derive a contradiction, that for any execution ξ of SFthat contains a write operation $\omega(k)$, $\exists \rho \in \Pi$ that returns $\mathfrak{R}(\rho) = \omega(k)$ after performing two communication rounds. Consider the following finite execution fragment that is a prefix of ξ, ϕ_ω. We assume that ϕ_ω contains the write operation $\omega(k)$ performed by the writer w that writes timestamp k.

Moreover, assume that $|srvAck(\omega(k))| = |\mathcal{S}| - \gamma f$ servers received the WRITE messages from $\omega(k)$ in ϕ_ω , where $1 < \gamma \leq |\mathcal{V}| - 1$. Thus the write operation is incomplete.

We extent now ϕ_ω by the finite execution fragment ϕ_1 which contains $\gamma - 2$ read operations $\rho_1, \ldots, \rho_{\gamma-2}$ performed by $\gamma - 2$ readers each of them from different virtual nodes. Let $\langle r_1, \nu_{r_1} \rangle, \ldots, \langle r_{\gamma-2}, \nu_{r_{\gamma-2}} \rangle$ be the identifiers of the readers that invoke the read operations. Furthermore every reader $\langle r_i, \nu_{r_i} \rangle$ receives replies from all the servers that reply to the write operation. Hence each reader $\langle r_i, \nu_{r_i} \rangle$ witnesses a $|maxTsMsg(\rho_i)| = |\mathcal{S}| - \gamma f$ and an $| \cap_{m \in maxTsMsg(\rho_i)} m.seen| \leq \gamma - 1$ and thus $| \cap_{m \in maxTsMsg(\rho_i)} m.seen| < \gamma$. So the predicate **SF-RP** is false for any read ρ_i from $\langle r_i, \nu_{r_i} \rangle$, returning timestamp $maxTS_{\rho_i} - 1$ in one communication round.

We further extend the execution fragment ϕ_1 by execution fragment ϕ_2 that contains two read operations performed by two sibling processes $\langle r, \nu_{\gamma-1} \rangle$ and $\langle r', \nu_{\gamma-1} \rangle$. Observe that those processes are not siblings with any of the previous readers. Let the read operations ρ and ρ' that are performed by the two sibling readers respectively, miss exactly f servers that receive WRITE messages from $\omega(k)$. However, let them miss different f servers. For example, if the servers $s_1, \ldots, s_{2f}$ received WRITE messages, then ρ skips the servers $s_{f+1}, \ldots, s_{2f}$ and ρ' skips the servers $s_1, \ldots, s_f$. Notice now that both readers observe an $| \cap_{m \in maxTsMsg(\rho)} m.seen| = | \cap_{m \in maxTsMsg(\rho')} m.seen| = \gamma$, since they receive messages from servers that also reply to the read operations $\rho_1, \ldots, \rho_{\gamma-2}$. However, both reads ρ and ρ', since they miss f of the servers that receive WRITE messages, they witness an $|maxTsMsg(\rho)| = |maxTsMsg(\rho')| = |\mathcal{S}| - (\gamma + 1)f$. So **SF-RP** is false for them as well and they return $maxTS_{rd} - 1$ (resp. $maxTS_{rd'} - 1$) in one communication round.

Finally we extend ϕ_2 by ϕ_3 that contains two read operations ρ^* by the reader $\langle r^*, \nu_\gamma \rangle$ and ρ^{**} by the reader $\langle r^{**}, \nu_{\gamma+1} \rangle$. Both readers are not siblings of any of the previous readers. We do not make any assumption about the relation of the two reads, that is, they may be concurrent. Let both reads receive messages from all the servers that reply to the writer and thus $|maxTsMsg(\rho^*)| = |maxTsMsg(\rho^{**})| = |\mathcal{S}| - \gamma f$. Recall that any server $s \in maxTsSrv(\rho^*)$ and any server $s' \in maxTsSrv(\rho^{**})$ contain a $seen = \{w, \nu_1, \ldots, \nu_{\gamma-1}\}$, and before replying to ρ^* and ρ^{**}, they add ν_γ and $\nu_{\gamma+1}$ respectively in their $seen$ sets. Suppose that the intersection is $| \cap_{m \in maxTsMsg(\rho^*)} m.seen| = \gamma + 1$ for ρ^* and $| \cap_{m \in maxTsMsg(\rho^{**})} m.seen| = \gamma + 2$ for ρ^{**}, that is, the servers reply to ρ^* before replying to ρ^{**}. Hence **SF-RP** holds by $| \cap_{m \in maxTsMsg(\rho^*)} m.seen| > \gamma$ $| \cap_{m \in maxTsMsg(\rho^{**})} m.seen| > \gamma$ for both reads and thus they return $maxTS_{\rho^*} = maxTS_{\rho^{**}} = k$ in one communication round. Notice that $\rho^*, \rho^{**} \in \Pi$ since $\mathfrak{R}(\rho^*) = \mathfrak{R}(\rho^{**}) = \omega(k)$.

Any subsequent read operation ρ_ℓ by a reader $\langle r_\ell, \nu_{r_\ell} \rangle$ witnesses an $|maxTsMsg(\rho_\ell)| \geq |\mathcal{S}| - (\gamma + 1)f$ and $| \cap_{m \in maxTsMsg(\rho_\ell)} m.seen| \geq \gamma + 2$. So if ρ_ℓ witnesses $maxTS_{\rho_\ell} = k$ then **SF-RP** holds for ρ_ℓ and moreover ρ_ℓ returns $maxTS_{\rho_\ell} = k$ in one communication round. If ρ_ℓ returns k even though it witnesses a maximum timestamp $maxTS_{\rho_\ell} = k + 1$ it is also fast since any read operation that returns $maxTS_{\rho_\ell} - 1$ is fast. So by this construction we showed that there exists an execution ξ of SF containing a write operation $\omega(k)$ and all the read operations $\rho \in \Pi$ such that $\mathfrak{R}(\rho) = \omega(k)$ are fast, contradicting our initial assumption. That completes our proof. □

We now state the main result of this section.

Theorem 4.2.16 Algorithm SF implements a semifast atomic read/write register.

Proof. We need to show that any $\xi \in goodexecs(\text{SF}, f)$ satisfies all the properties of Definition 4.1.1. The properties (1) and (2) of Definition 4.1.1 are trivially satisfied since all the write operations as implemented by SF are fast and every read operation does not require more than two communication rounds to complete. Properties (3) and (4) of the same definition are ensured by Lemmas 4.2.14 and 4.2.15. Thus any ξ of SF satisfies all properties of Definition 4.1.1 and SF is a semifast implementation. □

4.2.5 Quantifying the Number of Slow Reads

Algorithm SF provides guarantees on the fastness of the read operations that precede and succeed a slow read operation. However, guarantees are not given for the read operations concurrent with a slow read. Thus, multiple readers may be slow per write operation. In this section we evaluate the performance of our algorithm in terms of how many reads need to be slow per write operation. We present both theoretical and empirical results:

(i) A probabilistic analysis of algorithm SF, and

(ii) Implementation and simulation of algorithm SF.

4.2.5.1 Probabilistic Analysis

For our probabilistic analysis we assume that for any read operation ρ we have $\Pr[\rho \text{ invoked by some } r \in \nu_r] = \frac{1}{|\mathcal{V}|}$. That is, the read invocations may happen uniformly from the readers of any virtual node. We also assume that readers are uniformly distributed within the virtual nodes. We say that event e happens with high probability (whp) if $\Pr[e] = 1 - |\mathcal{R}|^{-c}$, for $|\mathcal{R}|$ the number of readers and $c > 1$ a constant; we say that event e happens with negligible probability if $\Pr[e] = |\mathcal{R}|^{-c}$.

In summary, we first investigate how the cardinality of set $seen$ of a specific server is affected by the read operations. We show that if a server s receives $\mu|\mathcal{V}|\log|\mathcal{R}|$ read messages without an interleaving write message, the $seen$ set of s contains all the virtual nodes with high probability. Given this result, we then present two read and write contention conditions: (a) *Low contention* if $4f$ servers receive messages from a write operation before receiving any read message, and (b) *High contention* otherwise. We analyze the two conditions separately. We show that $O(\log|\mathcal{R}|)$ and $O(|\mathcal{R}|)$ slow read operations may occur per write operation, with high probability, under low and high contention respectively.

We use the notation presented in Chapter 3 and Section 4.2.4.

The set $seen$ and fast read operations

We seek the number of read operations required, for a single server to record all virtual nodes in its $seen$ set. We first present some definitions that characterize the ordering relation of messages in an execution of the algorithm.

Definition 4.2.17 (Message Ordering) A message $m(\pi,k)_{p,s} \in M$ from process p to server s for an operation π, is **ordered before** a message $m(\pi',z)_{p',s} \in M$ from process p' to server s for an operation π' in an execution $\xi \in goodexecs(\textsc{Sf}, f)$, if action $\mathsf{rcv}(m(\pi,k)_{p,s})_{p,s}$ appears before the action $\mathsf{rcv}(m(\pi',z)_{p',s})_{p',s}$ in $\xi|\textsc{Sf}_s$. Otherwise, $m(\pi,k)_{p,s}$ is **ordered after** $m(\pi',z)_{p',s}$.

Next, we define the notion of *consecutive* read messages. In brief, two read messages received at server s are called consecutive if s did not receive any write messages between them.

Definition 4.2.18 (Consecutive Read Messages) Two read messages $m(\rho,k)_{r,s}, m(\rho',z)_{r',s}$ are **consecutive** in an execution $\xi \in goodexecs(\text{SF}, f)$, if $m(\rho,k)_{r,s}$ is ordered before $m(\rho',z)_{r',s}$, and $\nexists\mathsf{rcv}(m)_{w,s}$ between action $\mathsf{rcv}(m(\rho,k)_{r,s})_{r,s}$ and action $\mathsf{rcv}(m(\rho',z)_{r',s})_{r',s}$ in $\xi|\text{SF}_s$.

In general, we say that a set CM of read messages received by a server s is consecutive, if s does not receive any write message between the first and the last read message of the set CM. Note that, by algorithm SF the absence of a write message implies that the $seen$ set of s is not reset. More formally:

Definition 4.2.19 (Consecutive Read Message Set) A set of ordered read messages $CM \subseteq M$ received at a server s is consecutive in an execution $\xi \in goodexecs(\text{SF}, f)$, if $\forall m(\rho,k)_{r,s}, m(\rho',z)_{r',s} \in CM$, $m(\rho,k)_{r,s}, m(\rho',z)_{r',s}$ are consecutive read messages in $\xi|\text{SF}_s$.

Now we can compute the number of consecutive reads that are required to contact server s so that every virtual node ν_i is included in the $seen$ set of s.

Lemma 4.2.20 For any execution $\xi \in goodexecs(\text{SF}, f)$, there exists constant μ s.t. if server s receives $\mu|\mathcal{V}|\log|\mathcal{R}|$ consecutive read messages and σ the state of the system when s receives the last of those messages, then $\mathcal{V} \subseteq \sigma[s].seen$ whp.

Proof. Recall that we assume that $\Pr[\rho \text{ invoked by some } r \in \nu_i] = \frac{1}{|\mathcal{V}|}$. Let k be the number of reads. From Chernoff Bounds [74] we have

$$\Pr\left[\#\text{ of reads from readers of group } \nu_i \leq (1-\delta)\frac{k}{|\mathcal{V}|}\right] \leq e^{\frac{-\delta^2 \cdot k}{2\cdot|\mathcal{V}|}} \quad (1)$$

where $0 < \delta < 1$. We compute k, s.t. the probability in Equation (1) is negligible.

$$e^{\frac{-\delta^2 \cdot k}{2\cdot|\mathcal{V}|}} = |\mathcal{R}|^{-\gamma} \Rightarrow \frac{-\delta^2 \cdot k}{2\cdot|\mathcal{V}|} = -\gamma \cdot \log|\mathcal{R}| \Rightarrow$$

$$\Rightarrow k = \frac{2\cdot\gamma\cdot|\mathcal{V}|\cdot\log|\mathcal{R}|}{\delta^2} \tag{2}$$

Let $\delta = 0.5$. From Equation (2) we have $k = 8\cdot\gamma\cdot|\mathcal{V}|\cdot\log|\mathcal{R}|$. We set $\mu = 8\cdot\gamma$ and we have that $k = \mu\cdot|\mathcal{V}|\cdot\log|\mathcal{R}|$. If $\mu|\mathcal{V}|\log|\mathcal{R}|$ consecutive reads contact a server s, at least $\frac{\mu\cdot\log|\mathcal{R}|}{2}$ reads will be performed by readers in group ν_i whp, for any group ν_i and thus $\nu_i \in \sigma[s].seen$, since the $seen$ set of s has not been reset. $\square$

Notice that the larger the constant μ in the above lemma, the smaller the probability of having a server that did not receive a read message from a reader from every virtual node.

Lemma 4.2.21 If in a state σ of some execution $\xi \in goodexecs(\text{SF}, f)$, there exists set $\mathcal{S}' \subseteq \mathcal{S}$ s.t $|\mathcal{S}'| \geq 4f$ and $\forall s \in \mathcal{S}'$ $\sigma[s].ts = y$ (from write $\omega(y)$) and $\sigma[s].seen = \{w\} \cup \mathcal{V}$, then any read operation ρ with $\mathfrak{R}(\rho)$=$\omega(y)$ invoked after σ is fast.

Proof. From predicate **SF-RP** of algorithm SF (see Section 4.2.3) and the fact that $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$ (thus $|\mathcal{V}| \leq \frac{|\mathcal{S}|}{f} - 3$), it follows that if a read operation ρ observes

$$|maxTsMsg(\rho)| \geq |\mathcal{S}| - |\mathcal{V}|f \geq (|\mathcal{V}|+3)f - |\mathcal{V}|f \geq 3f$$

then $\beta \leq |\mathcal{V}|$. Suppose ρ observes $|maxTsMsg(\rho)| \geq 3f$ and every server s with message in $maxTsMsg(\rho)$ replied to r with $m(\rho,1)_{s,r}.seen = \{w\} \cup \mathcal{V}$. Then, **SF-RP**holds for ρ for:

$$\left|\bigcap_{m \in maxTsMsg(\rho)} m.seen\right| = |\mathcal{V}| + 1 > \beta.$$

So, for the read operation ρ with $\mathfrak{R}(\rho) = \omega(y)$ invoked after σ , there are two cases for the maximum timestamp $maxTS_\rho$ observed by ρ: (a) $maxTS_\rho = y$, and (b) $maxTS_\rho \geq$

$y+1$. For case (a), since up to f servers may fail, ρ observed an $|maxTsMsg(\rho)| \geq 3f$ with $\left|\bigcap_{m \in maxTsMsg(\rho)} m.seen\right| = |\mathcal{V}|+1$. Thus the predicate holds for $\beta \leq |\mathcal{V}|$ and ρ is fast. For case (b) since $\Re(\rho) = \omega(y)$ from algorithm SF, ρ is fast. $\square$

Note that if less than $4f$ servers contain $seen = \{w\} \cup \mathcal{V}$, then a read operation ρ may observe $|maxTsMsg(\rho)| = 3f$ or $|maxTsMsg(\rho)| = 2f$. If ρ observes $|maxTsMsg(\rho)| = 2f$ messages with $seen = \{w\} \cup \mathcal{V}$, **SF-RP** for ρ holds for $|\bigcap_{m \in maxTsMsg(\rho)} m.seen| = |\mathcal{V}|+1 = \beta$ and ρ is slow.

From Lemma 4.2.21 it follows that predicate **SF-RP** naturally yields two separate cases: (a) $4f$ servers or more contain the maximum timestamp, and (b) less than $4f$ servers contain the maximum timestamp. In both cases we can use Lemma 4.2.20 to bound the number of read operations needed until $seen = \{w\} \cup \mathcal{V}$ for all the servers with the maximum timestamp. To obtain the worst case scenario we assume that there are $O(|\mathcal{R}|)$ reads concurrent with the first slow read operation.

We now define formally the execution conditions that capture the idea behind the aforementioned cases.

Definition 4.2.22 (Succesive Operations) We say that two operations π, π' are **successive** in an execution ξ, if π and π' are invoked by the same process p, and p does not invoke any operation π'' between $res(\pi)$ and $inv(\pi')$.

Next, we define the set of events that occur between two successive operations invoked by a process p.

Definition 4.2.23 (Idle Set) For any execution ξ and for any pair of successive operations π, π' invoked by a process p, we define $idle(\pi, \pi')$ to be the set of all events that appear in ξ and succeed $res(\pi)$ and precede $inv(\pi')$.

Given the above definition we now define the contention conditions that affect our analysis. These conditions characterize cases (a) and (b).

Definition 4.2.24 (Contention) Let ρ, ρ' be any pair of successive read operations invoked by a reader r. We say that an execution fragment ϕ has **low contention** if for every set $idle(\rho, \rho')$, $\exists inv(\omega) \in idle(\rho, \rho')$ for some write operation ω, and $\exists \mathcal{S}' \subseteq \mathcal{S}$, s.t. $|\mathcal{S}'| \geq 4f$ and $\forall s \in \mathcal{S}'$, $\mathsf{rcv}(m(\omega, 1)_{w,s})_{w,s} \in idle(\rho, \rho')$. Otherwise we say that ϕ has **high contention**.

Slow reads under low contention

We consider the case of low contention where a set of at least $4f$ servers receive messages from a write operation ω, before the first slow read operation ρ, with $\Re(\rho)$=ω. For an implementation to be semifast, any read operation ρ' that precedes or succeeds ρ, with $\Re(\rho')$= $\Re(\rho)$=ω, is fast. We now bound the number of slow read operations.

Theorem 4.2.25 If in an execution $\xi \in goodexecs(\text{SF}, f)$, $\exists$ state σ and a set of servers $\mathcal{S}' \subseteq \mathcal{S}$, such that

- $|\mathcal{S}'| \geq 4f$,
- $\forall s \in \mathcal{S}'$, $\sigma[s].ts = m(\omega, 1)_{w,s}.ts$ and $w \in \sigma[s].seen$ as a result of a write operation ω, and
- $\nexists\, inv(\rho)$ before σ in ξ for any read ρ such that $\Re(\rho) = \omega$,

then there exists constant μ, s.t. whp at most $\mu \cdot |\mathcal{S}| \cdot |\mathcal{V}| \cdot \log |\mathcal{R}|$ reads ρ', such that $\Re(\rho') = \omega$, can be slow.

Proof. For each $s \in \mathcal{S}'$, we examine two cases:

Case 1**:** After s receives a write message from ω, s receives a set $CM = \{\rho_1, \ldots, \rho_\ell\}$ of consecutive read messages, s.t. $|CM| = \mu|\mathcal{V}| \log |\mathcal{R}|$ and $m(\rho_\ell, k)_{s,r}.ts = m(\omega, 1)_{w,s}.ts$, where $m(\rho_\ell, k)_{s,r}$ is ordered after every other message in CM. From Lemma 4.2.20 any read ρ' with message $m(\rho', z)_{s,r'}$ received after $m(\rho_\ell, k)_{s,r}$, observes $seen = \{w\} \cup \mathcal{V}$ from s if $m(\rho', z)_{s,r'}.ts = m(\omega, 1)_{w,s}.ts$.

Case 2**:** After s receives a write message from ω, s receives message $m(\pi, 1)_{p,s}$ from p for an operation π with $m(\pi, 1)_{p,s}.ts = y > m(\omega, 1)_{w,s}.ts$, before it receives $\mu|\mathcal{V}| \log |\mathcal{R}|$ read messages. It follows that a write operation ω' that propagates timestamp y, has been invoked. Any read ρ' that receives y, will either return $y - 1$ or y. From the construction of SF, if ρ' returns $y-1$, ρ' will be fast. Thus if a read ρ' contacts server s after $m(\pi, 1)_{p,s}$ and $\Re(\rho') = \omega$, then ρ' is fast and $m(\omega, 1)_{w,s}.ts = y - 1$.

From the above cases, we have a total of $\mu \cdot |\mathcal{S}'| \cdot |\mathcal{V}| \cdot \log |\mathcal{R}| \leq \mu \cdot |\mathcal{S}| \cdot |\mathcal{V}| \cdot \log |\mathcal{R}|$ read messages for the servers in $\mathcal{S}'$. Let Π be the set of the read operations that correspond to these read messages. Clearly $|\Pi| \leq \mu \cdot |\mathcal{S}| \cdot |\mathcal{V}| \cdot \log |\mathcal{R}|$ and any read operation in Π can be slow.

For any read ρ' invoked after ρ, s.t. $\rho' \notin \Pi$ we have the following cases:

Case (i)**:** Read ρ' receives at least $3f$ replies with $maxTS_{\rho'} = m(\omega, 1)_{w,s}.ts$ and observes from Case 1 $\cap_{m \in maxTsMsg(\rho')} m.seen = \{w\} \cup \mathcal{V}$. As discussed in Lemma 4.2.21 $\beta \leq |\mathcal{V}|$ and thus by **SF-RP**, ρ' is fast and $\Re(\rho') = \omega$.

Case (ii): Read ρ' receives $maxTS_{\rho'} > m(\omega,1)_{w,s}.ts$. From algorithm SF and the discussion in case 2, if $\Re(\rho') = \omega$, then ρ' is fast. $\square$

Theorem 4.2.25 proves that under low contention, a write can be followed by at most $O(\log|\mathcal{R}|)$ slow reads whp.

Slow reads under high contention

Here we deal with the high contention case, where a set of less than $4f$ servers receive messages from a write operation ω, before the first slow read operation operation ρ is invoked, with $\Re(\rho)=\omega$. We examine the case where $|\mathcal{V}| = \frac{|\mathcal{S}|}{f} - 3$, which is the maximum number of virtual nodes allowed by algorithm SF. Recall that if a read ρ receives replies from $2f$ servers with $maxTS_\rho$ and $seen = \{w\} \cup \mathcal{V}$, then ρ is slow since $maxTsMsg(\rho) = |\mathcal{S}| - (|\mathcal{V}|+1)f = |\mathcal{S}| - \left(\frac{|\mathcal{S}|}{f} - 2\right) f = 2f$. In contrast with the low contention case, we show that in the high contention case, the system reaches a state where all reads that receive the $maxTS$ from less than $3f$ servers are slow.

Note that according to SF, any read operation that succeeds a slow read and returns the same value is fast. Thus, any reader can perform at most one slow read that returns the value and timestamp of the same write ω. This gives a bound of at most $|\mathcal{R}|$ slow reads per write operation. We next prove that under high contention, $\mu \cdot 4f \cdot |\mathcal{V}| \cdot \log|\mathcal{R}|$ reads may lead the system to a state where all reads concurrent with the first slow read can be slow if they receive replies from less than $3f$ updated servers.

Theorem 4.2.26 If in an execution $\xi \in goodexecs(\text{SF}, f)$, $\exists$ state σ and a set of servers $\mathcal{S}' \subseteq \mathcal{S}$, such that

- $|\mathcal{S}'| < 4f$,
- $\forall s \in \mathcal{S}'$, $\sigma[s].ts = m(\omega, 1)_{w,s}.ts$ and $w \in \sigma[s].seen$ as a result of a write operation ω,
- $\forall s \in \mathcal{S}'$ receives a set $|CM| \leq \mu \cdot 4f \cdot |\mathcal{V}| \cdot \log |\mathcal{R}|$ of consecutive read messages, and
- $\forall s' \in \mathcal{S} - \mathcal{S}'$, $\sigma[s'].ts < m(\omega, 1)_{w,s}.ts$,

then any read ρ' invoked after σ will be slow if $\mathfrak{R}(\rho') = \omega$ and $|maxTsMsg(\rho')| < 3f$.

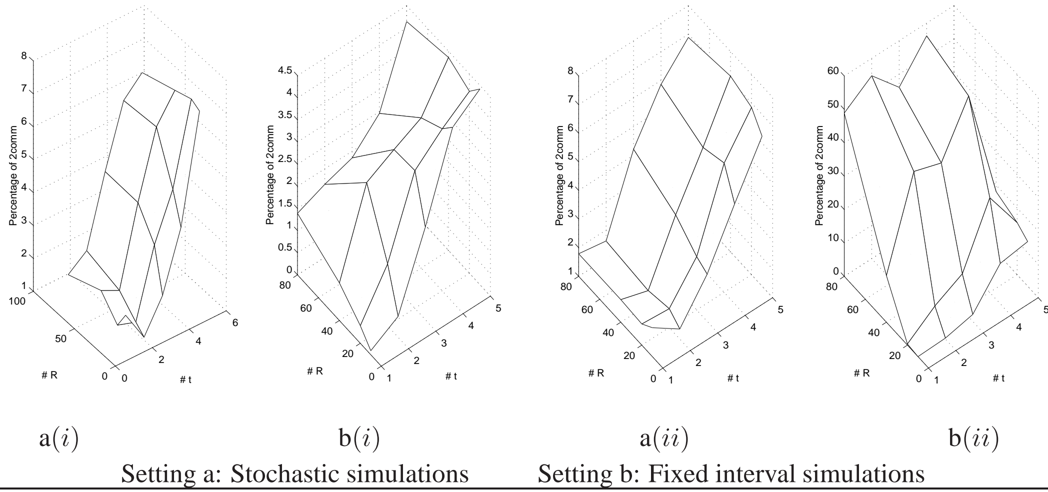
Proof. For any server $s \in \mathcal{S}'$ if $\mu \cdot |\mathcal{V}| \cdot \log |\mathcal{R}|$ consecutive read messages are received by s after σ, where μ is taken from Lemma 4.2.20, then whp the $seen$ set of s becomes $\{w\} \cup \mathcal{V}$ after the last message is received.

If we consider such reads for all servers in $\mathcal{S}'$, we have a total of $\mu \cdot |\mathcal{S}'| \cdot |\mathcal{V}| \cdot \log |\mathcal{R}| < \mu \cdot 4f \cdot |\mathcal{V}| \cdot \log |\mathcal{R}|$ read messages. After these read messages, the system reaches a state σ' where $\forall s \in \mathcal{S}'$. $\sigma'[s].ts = m(\omega, 1)_{w,s}.ts$ and $\sigma'[s].seen = \{w\} \cup \mathcal{V}$. As discussed in Section 4.2.5.1, any read operation that contacts less than $3f$ servers with the maximum timestamp, is slow. This is possible, since up to f servers may fail. □

From Theorem 4.2.26, observe that under high contention an execution relatively fast can reach a state (after $\Omega(\log |\mathcal{R}|)$ reads) that could lead to $O(|\mathcal{R}|)$ slow reads.

4.2.5.2 Empirical Evaluation of SF

To evaluate our implementation in practice, we simulated algorithm SF using the NS2 network simulator ([2]). Our test environment consisted of one writer, a variable set of reader and server processes. We used bidirectional links between the communicating nodes, 1Mb bandwidth, a latency of $10ms$, and a DropTail queue. To model asynchrony, the processes

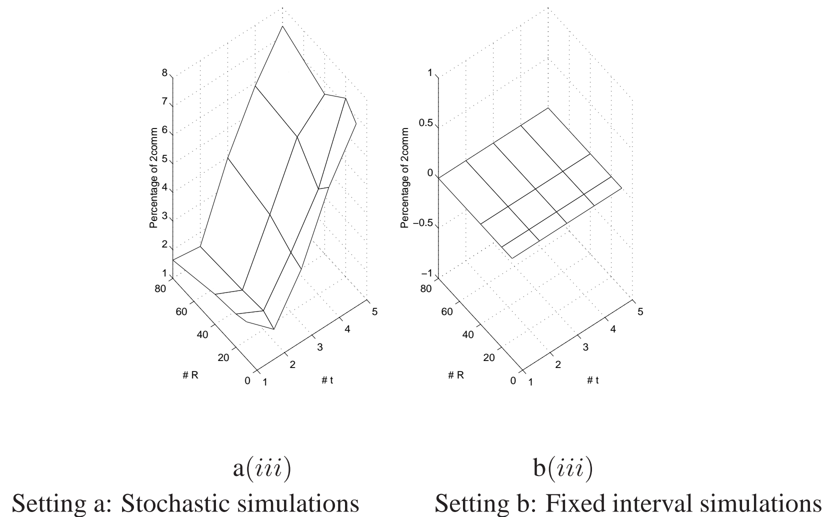


a(i) b(i) a(ii) b(ii)

Setting a: Stochastic simulations Setting b: Fixed interval simulations

Figure 7: Scenarios (i) $rInt = 2.3s$, $wInt = 4.3s$, and (ii) $rInt = 4.3s$, $wInt = 4.3s$.

send messages after a random delay between 0 and 0.3 sec. According to our setting, only the messages from the invoking processes to the servers, and the replies from the servers to the processes are delivered (no messages are exchanged among the servers or the invoking processes).



a(iii) b(iii)

Setting a: Stochastic simulations Setting b: Fixed interval simulations

Figure 8: Scenario (iii) $rInt = 6.3s$, $wInt = 4.3s$.

We measured the percentage of two round read operations as a function of the number of readers and the number of faulty servers. To measure the effectiveness of our approach we manipulated three different environmental parameters:

(1) **Number of Readers:** Varying the number of readers allowed the evaluation of the scalability of SF.

(2) **Read and Write Frequency:** This parameter controls the frequency in which reads and writes are invoked. It comprise one of the most critical components since it defines the traffic load and the concurrency scenarios between R/Woperations.

(3) **Number of Replica Host Failures:** This component examines the robustness of the algorithms and the performance degradation during multiple failure scenarios. Although any participant may fail in the system, more interesting scenarios are generated if we allow the readers and writers to stay alive, and only permit server failures throughout the execution of the simulation. Such rule will demonstrate the performance of the algorithm in the maximum traffic generation (by read and write operations); at the same time we will examine the algorithm robustness on replica (server) failures.

Our simulations include 20 servers ($|\mathcal{S}|$ = 20). To guarantee liveness we need to constrain the maximum number of server failures f so that $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$ or $|\mathcal{V}| \leq \frac{|\mathcal{S}|}{f} - 3$. Thus, $f \leq \frac{|\mathcal{S}|}{|\mathcal{V}|+3}$. In order to maintain at least one group ($|\mathcal{V}| = 1$), f must not exceed $\frac{|\mathcal{S}|}{4}$, or 5 failures. Thus, in our simulations we allow up to 5 servers to fail at arbitrary times. We vary the number of reader processes between 10 and 80. We use the positive time parameters $rInt$ and $wInt$ (both greater than 1 sec) to model the time intervals between any two successive read operations and

any two successive write operations respectively. For our expirements, we considered three frequency simulation scenarios:

- (i) $rInt < wInt$: Frequent reads and infrequent writes,
- (ii) $rInt = wInt$: Evenly spaced reads and writes,
- (iii) $rInt > wInt$: Infrequent reads and frequent writes.

Each of the simulation scenarios (i), (ii), and (iii) was considered in two settings:

a. **Stochastic simulations**, where the invocation of the next read or write operation is chosen randomly within certain bounds determined by the frequency of each operation.

b. **Fixed simulations**, where each read and write operation is invoked in a fixed time with respect to its frequency.

Stochastic scenarios better resemble realistic conditions while fixed scenarios represent frequent and bursty conditions.

We now describe the simulation results for each of the two settings.

Setting a: Stochastic simulations. Here we consider a class of executions where each read (resp. write) operation from an invoking process is scheduled at random time between 1 sec and $rInt$ (resp. $wInt$) after the last read (resp. write) operation. Introducing randomness in the operation invocation intervals renders a more realistic scenario where processes are interacting with the atomic object independently. Note that under this setting, for the three scenarios (i), (ii), and (iii), the comparisons between $rInt$ and $wInt$ are satisfied stochastically. We present the results for a single value of $wInt = 4.3$ sec for write operations. For scenario (i) we use $rInt = 2.3$ sec, for scenario (ii) we use $rInt = 2.3$ sec, and for scenario (iii) we use

$rInt = 6.3$ sec. The results are given in Figures 7 and 8, setting a. We observe that the results in this setting are similar, with the percentage of two-round reads is mainly affected by the number of faulty servers. In all cases the percentage of two-round reads is under 7.5%.

Setting b: Fixed interval simulations. In this setting the intervals between two read (or two write) operations are fixed at the beginning of the simulation. All readers use the same interval rInt, and the writer the interval wInt. This family of simulations represent conditions where operations can be frequent and bursty. Figure 7b(i), illustrates the case of $rInt < wInt$, where $rInt = 2.3sec$. Here a read (write) operation is invoked by every reader (resp. writer) in the system every $rInt = 2.3sec$ (resp. $wInt = 4.3sec$). Because of asynchrony not every read operation completes before the invocation of the write operation and thus we observe that only 4.5% of the reads perform two communication rounds. Figure 7b(ii), illustrates the scenario where $rInt = wInt$. This is the most bursty scenario since all operations, read or write, are invoked at the same time, specifically the operations are invoked every $rInt = wInt = 4.3sec$. Although the conditions in this case are highly bursty (and unlikely to occur in practice), we observe that only up to 60%of the read operations perform two communication rounds. Figure 8b(iii), illustrates the scenario where $wInt < rInt$. In particular a read operation is invoked every $rInt = 6.3sec$ by each reader and a write operation every $wInt = 4.3sec$. Given the modeled channel latency and delays, notice that there is no concurrency between the read and write operations in this scenario. So all the servers reply to any read operation with the latest timestamp and thus no read operation needs to perform a second communication round.

Both scenarios shared a common trend: by increasing the number of readers and the number of faulty servers, the performance of the algorithm degraded. These findings agree with the theoretical results presented in Section 4.2.5.1.

4.3 Limitations of Semifast Read/Write Register Implementations

In this Section we present some restrictions that a semifast implementation imposes on our system. First, we show that in algorithms that assume grouping mechanisms like SF, a bound on the number of groups (or in our case virtual nodes) is necessary. Next, we specify the number of servers that the second round of a read operation must communicate with to ensure atomicity. Then, we investigate whether semifast implementations can be developed for environments that support multiple writers and multiple readers. We show that such implementations are impossible.

4.3.1 Constraints on the Virtual Nodes and Second Round Communication

As it is shown in [30], no fast implementations exist if the number of readers $\mathcal{R}$ in the system is such that $|\mathcal{R}| \geq \frac{|\mathcal{S}|}{f} - 2$. Our approach to semifast solutions is to trade fast implementation for increased number of readers, while enabling some (many) reads to be fast. Here we show that semifast implementations are possible if and only if the number of virtual identifiers (virtual nodes) in the system is less than $\frac{|\mathcal{S}|}{f} - 2$. We show that the bound on the virtual identifiers is tight for algorithms that: (1) consider each node acting individually in the system (as in [30]), and (2) consider weak grouping of the readers such that no reader is required to maintain knowledge of the membership of its own or any other group. Throughout the section we assume that the messages from the clients to the servers and from the servers to the clients are delivered (see Section 3.3).

Definitions and notation.

We consider a system with $|\mathcal{V}|$ node groups (virtual nodes), such that $f \geq \frac{|\mathcal{S}|}{(|\mathcal{V}|+2)}$ (to derive contradiction). We partition the set of servers $\mathcal{S}$ into $|\mathcal{V}|+2$ subsets, called *blocks*, each denoted by B_i for $1 \leq i \leq |\mathcal{V}|+2$, where each block contains no more than f servers.

We say that an *incomplete operation* π *skips* a set of blocks BS in a finite execution fragment, where $BS \subseteq \{B_1, \ldots, B_{|\mathcal{V}|+2}\}$, if : (1) no server in BS receives any READ or WRITE message from π, (2) all other servers receive messages and reply to π, and (3) those replies are in transit. A *complete operation* π that is fast is said to *skip* a block B_i in a finite execution fragment, where $B_i \in \{B_1, \ldots, B_{|\mathcal{V}|+2}\}$ if: (1) no server in B_i receives a READ or WRITE messages from π, (2) all other servers receive the messages from π and reply, and (3) all replies are received by the process performing π. We say that an incomplete operation π that performs a second communication round *informs* a set of blocks BSI in a finite execution fragment, where $BSI \subseteq \{B_1, \ldots, B_{|\mathcal{V}|+2}\}$ if: (1) all servers in BSI receive the INFORM message from π and reply, (2) those replies are in transit, and (3) no servers in any block $B_j \notin BSI$ receive any INFORM messages from π. A complete operation π that performs a second communication round *informs* a set of blocks BSI in an finite execution fragment, $BSI \subseteq \{B_1, \ldots, B_{|\mathcal{V}|+2}\}$ if: (1) all servers in BSI receive the INFORM messages from π and reply, (2) no servers in any block $B_j \notin BSI$ receive any INFORM messages from π, and (3) those replies are received by the process performing π. A complete operation π is said to be *skip-free* in an execution fragment if for every block B_i in the set $\{B_1, \ldots, B_{|\mathcal{V}|+2}\}$, all the servers in B_i receive the messages from π and reply to them.

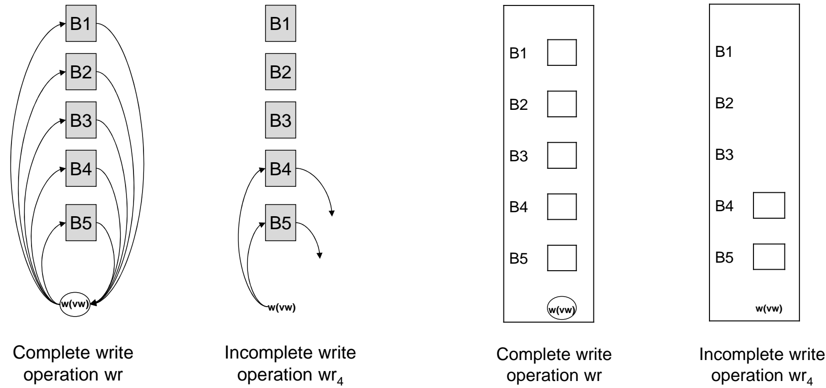


Figure 9: Left: Physical communication between w and the servers in $\phi(wr)$ and $\phi(wr_4)$. Right: Same communication using block diagrams.

Block Diagrams.

To facilitate the understanding of the proofs that follow, we provide schematic representations using block diagrams (e.g., Figure 9). We divide the diagram into columns each of them representing an operation (possibly incomplete) π, and at the bottom of each column we place an identifier of the invoking process in the form (r, ν_r), where r the actual id and ν_r the virtual id of the invoking process. Each column contains a set of rectangles. For an operation π if the ith row of the column contains a rectangle it means that the servers in block B_i received a READ, INFORM or WRITE messages from π and replied to those messages. In other words we draw a rectangle in the ith row of an operation π if π does not skip or informs the block B_i. If a rectangle is colored white, it means that block B_i received only a READ or WRITE messages from π. A two-color rectangle (black and white) in the ith row of an operation π declares that the servers in block B_i received INFORM messages from π. If the operation identifier in a column is in a circle it means the operation is complete. Otherwise the operation has not yet

completed. If the operation identifier is in a rectangle means that the operation is invoking the informative phase and has not yet received the required replies.

We now show that $|\mathcal{V}|$ cannot be greater or equal than $\frac{|\mathcal{S}|}{f} - 2$. The idea behind the proof is to derive contradiction by assuming that semifast implementations exist for $|\mathcal{V}| \geq \frac{|\mathcal{S}|}{f} - 2$. We construct executions that violate atomicity and properties of the semifast definition. In particular we first assume an execution ξ which contains a skip-free write operation. We construct executions that can be extended to ξ, that contain fast read operations. We show that in execution extensions where the value of the write operation is propagated to less than f servers, some fast read operations return the value written, but others return an older value (since they may skip the servers with the maximum timestamp). We emphasize that the first part of the proof can use the proof of Proposition 1 in [30] as a black box with the assumption of the skip-free write operation and the association of a distinct group id to each reader used. However we choose to present the proof here in its entirety for completeness. In the second part of the proof we present executions that violate atomicity even in the presence of a slow read operation.

Lemma 4.3.1 No semifast implementation exists if the number of node groups $|\mathcal{V}|$ in the system is $\geq \frac{|\mathcal{S}|}{f} - 2$.

Proof. We proceed along the lines of Proposition 1 of [30]. We construct an execution of a semifast implementation A that violates atomicity. Namely we show that there exists an execution for A where some read returns 1 (the defined new value) and some subsequent read returns an older value, and in particular the initial value $\perp$. We consider two cases: (1) $|\mathcal{V}| > \frac{|\mathcal{S}|}{f} - 2$ and (2) $|\mathcal{V}| = \frac{|\mathcal{S}|}{f} - 2$. In the first case we show impossibility of the fast behavior if

$|\mathcal{V}| > \frac{|S|}{f} - 2$, thus violating property 4 of Definition 4.1.1. In case (2) we show that there exists an execution where atomicity is violated even in the presence of a two-round read operation. This violates property 3 of Definition 4.1.1.

Case 1: Since $|\mathcal{V}| > \frac{|S}{f} - 2$, it suffices to show that we derive contradiction in the case where $|\mathcal{V}| \geq \frac{|S}{f} - 1$. So we can partition the set of servers into $|\mathcal{V}| + 1$ blocks $\{B_1, \ldots, B_{|\mathcal{V}|+1}\}$ where each block contains $\leq f$ servers. We provide the constructions we use for the needs of this proof in the write and read operation paragraphs and then we present an execution scenario based on those constructions that violates atomicity.

Write Operations. Let $\phi(wr)$ be an execution fragment in which operation $\omega(1)$ is completed by w. Let the operation be *skip-free*; this is the best case for a write operation and thus our lower bound applies to all other possible cases. We define a series of finite execution fragments which can be extended to $\phi(wr)$. We say that in the finite execution fragment $\phi(wr_{|\mathcal{V}|+2})$ the writer w invokes $\omega(1)$, but all the WRITE messages are in transit. Then, for $1 \leq i \leq |\mathcal{V}| + 1$, we say that $\phi(wr_i)$ is the finite execution fragment that contains an incomplete $\omega(1)$ operation that skips the set of blocks $\{B_j | 1 \leq j \leq i - 1\}$. Observe that: (1) the finite execution fragments $\phi(wr_i)$ and $\phi(wr_{i+1})$ differ only on block B_i, (2) since in $\phi(wr_1)$ we do not skip any block but all the replies are in transit, then $\phi(wr)$ is an extension of $\phi(wr_1)$ where all those replies are received by w and (3) only w can distinguish $\phi(wr)$ from $\phi(wr_1)$. Figure 9 illustrates the communication between the writer w and the groups of servers in the finite execution fragments $\phi(wr)$ and $\phi(wr_3)$. The figure shows both physical communication and the corresponding block diagram representation.

Read Operations. We now construct finite execution fragments for read operations. We assume that every reader process r_i, belongs in a different virtual identifier ν_{r_i}, denoted by

the pair $\langle r_i, \nu_{r_i} \rangle$. Let $\phi(1)$ be a finite execution fragment that extends $\phi(wr)$ by containing a complete read operation by a reader with id $\langle r_1, \nu_{r_1} \rangle$ that skips B_1. Consider now $\phi'(1)$ that extends $\phi(wr_2)$ by a complete read operation by the reader $\langle r_1, \nu_{r_1} \rangle$ that skips B_1. Notice that reader $\langle r_1, \nu_{r_1} \rangle$ cannot distinguish $\phi(1)$ from $\phi'(1)$ because $\phi(wr)$ and $\phi(wr_2)$ differ at w and block B_1 and read from $\langle r_1, \nu_{r_1} \rangle$ skips block B_1.

We continue in similar manner, starting from $\phi'(1)$, and create execution fragments for the rest of the readers in the system. In particular we define an execution fragment $\phi(i)$, for $2 \leq i \leq |\mathcal{V}|$ to extend $\phi'(i-1)$ by a complete read operation from $\langle r_i, \nu_{r_i} \rangle$ that skips B_i. We then construct finite execution fragment $\phi'(i)$ by deleting from $\phi(i)$ all the rectangles (steps) from the servers in block B_i. In particular, as previously mentioned, execution fragment $\phi'(i)$ extends $\phi(wr_{i+1})$ by appending that with i reads such that for $1 \leq k \leq i$, $\langle r_k, \nu_{r_k} \rangle$ skips the blocks $\{B_j |\ k \leq j \leq i\}$. Observe that since $\langle r_1, \nu_{r_1} \rangle$ cannot distinguish $\phi(1)$ and $\phi'(1)$, it returns 1 in both executions. Furthermore, since $\phi(2)$ extends $\phi'(1)$, by atomicity $\langle r_2, \nu_{r_2} \rangle$ returns 1. So $\langle r_2, \nu_{r_2} \rangle$ returns 1 in $\phi'(2)$ since it cannot distinguish $\phi(2)$ and $\phi'(2)$. By following inductive arguments we conclude that for $\phi'(i)$, reader $\langle r_i, \nu_{r_i} \rangle$ returns 1. Thus, for the execution fragment $\phi'(|\mathcal{V}|)$, $\langle r_{|\mathcal{V}|}, \nu_{r_{|\mathcal{V}|}} \rangle$ returns 1. An illustration of the following execution fragments can be seen in Figure 10.

Finite Execution fragment $\phi(A)$. Here we consider the execution fragment $\phi'(|\mathcal{V}|)$. As defined above, $\phi'(|\mathcal{V}|)$ extends $\phi(wr_{|\mathcal{V}|+1})$ by appending $|\mathcal{V}|$ reads such that for $1 \leq k \leq |\mathcal{V}|$, $\langle r_k, \nu_{r_k} \rangle$'s read skips the blocks $\{B_j |\ k \leq j \leq |\mathcal{V}|\}$. Observe here that all the read operations are incomplete except for the read operation of reader $\langle r_{|\mathcal{V}|}, \nu_{r_{|\mathcal{V}|}} \rangle$. Moreover only the servers in block $B_{|\mathcal{V}|+1}$ receive WRITE messages from the ω_1 operation of w. Also, only $B_{|\mathcal{V}|+1}$ replies to the read operation of the reader $\langle r_1, \nu_{r_1} \rangle$, and those messages are in transit. All other

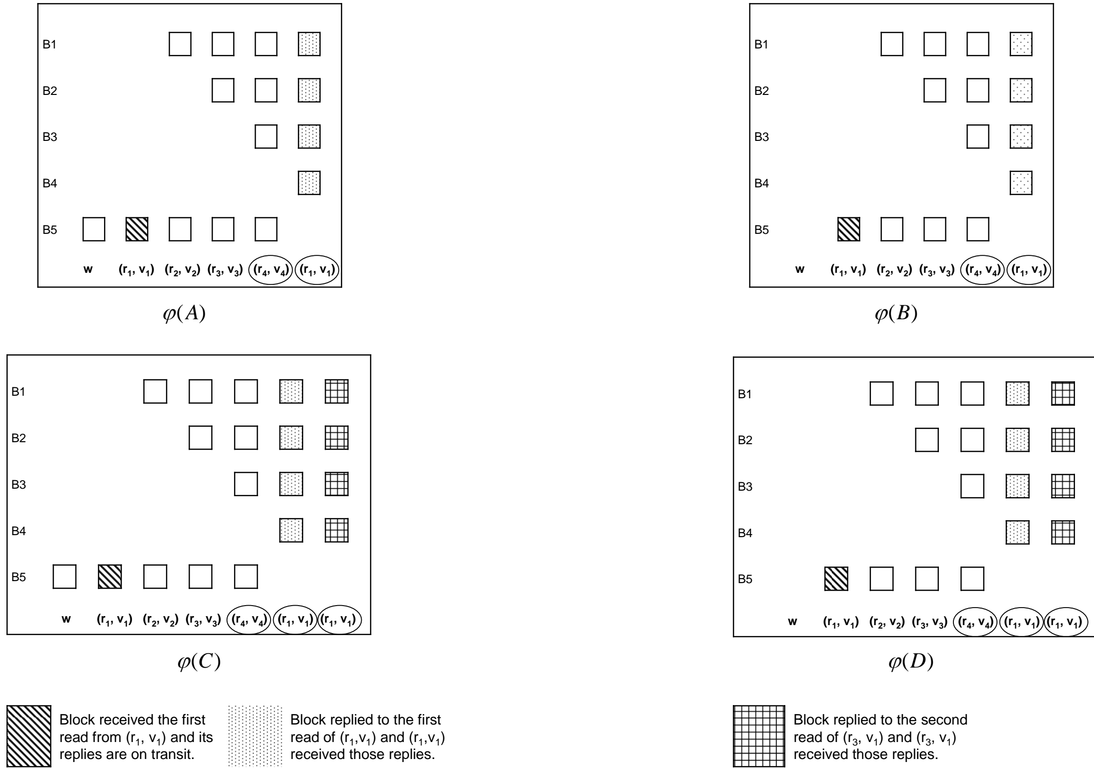


Figure 10: Execution fragments $\phi(A)$, $\phi(B)$, $\phi(C)$, $\phi(D)$.

READ messages of $\langle r_1, \nu_{r_1} \rangle$ are in transit and are not yet received by any other server. Let execution fragment $\phi(A)$ extend $\phi'(|\mathcal{V}|)$ as follows: (1) all the messages send by $\langle r_1, \nu_{r_1} \rangle$ and were in transit, are received by the servers in blocks $B_1, \ldots, B_{|\mathcal{V}|}$, (2) reader $\langle r_1, \nu_{r_1} \rangle$ receives the replies from servers $B_1, \ldots, B_{|\mathcal{V}|}$, and returns from the read operation. Notice that since $B_{|\mathcal{V}|+1}$ contains no more than f servers, it means that reader $\langle r_1, \nu_{r_1} \rangle$ received no less than $|\mathcal{S}| - f$ replies and should not wait for any more replies to return.

Finite Execution fragment $\phi(B)$. We consider as execution fragment $\phi(B)$ with the same communication pattern as $\phi(A)$ but with the difference that the ω_1 operation is not invoked at all. Hence servers in block $B_{|\mathcal{V}|+1}$ do not receive any WRITE messages. Clearly only the

servers in block $B_{|\mathcal{V}|+1}$, the writer and the readers $\langle r_2, \nu_{r_2}\rangle$ to $\langle r_{|\mathcal{V}|}, \nu_{r_{|\mathcal{V}|}}\rangle$ are in position to distinguish $\phi(A)$ from $\phi(B)$. The reader $\langle r_1, \nu_{r_1}\rangle$, since it does not receive any messages from $B_{|\mathcal{V}|+1}$ cannot distinguish $\phi(A)$ from $\phi(B)$. So, since there is no write (ω_*) operation, $\langle r_1, \nu_{r_1}\rangle$ returns $\perp$ in $\phi(B)$ and therefore returns $\perp$ in $\phi(A)$ as well.

Finite Execution fragments $\phi(C)$ and $\phi(D)$. Observe that in $\phi(A)$, reader $\langle r_1, \nu_{r_1}\rangle$ does not violate atomicity even though it returns $\perp$ and $\langle r_{|\mathcal{V}|}, \nu_{r_{|\mathcal{V}|}}\rangle$ returns 1 because the two operations are concurrent. We construct now two more executions: execution fragment $\phi(C)$ and $\phi(D)$ which extend the execution fragments $\phi(A)$ and $\phi(B)$ respectively with a second complete read operation from $\langle r_1, \nu_{r_1}\rangle$ that skips $B_{|\mathcal{V}|+1}$. Since the servers in $B_{|\mathcal{V}|+1}$ are the only ones who can distinguish $\phi(A)$ and $\phi(B)$ and since $\langle r_1, \nu_{r_1}\rangle$'s second read skips $B_{|\mathcal{V}|+1}$ then $\langle r_1, \nu_{r_1}\rangle$ cannot distinguish $\phi(C)$ from $\phi(D)$ either. Since $\phi(C)$ is an extension of $\phi(A)$ it follows that the reader $\langle r_{|\mathcal{V}|}, \nu_{r_{|\mathcal{V}|}}\rangle$ returns 1 in $\phi(C)$. Moreover $\langle r_1, \nu_{r_1}\rangle$ returns $\perp$ since no write (ω_*) operation is invoked in $\phi(D)$. So since $\langle r_1, \nu_{r_1}\rangle$ cannot distinguish $\phi(C)$ from $\phi(D)$, it returns $\perp$ in $\phi(C)$ as well. However, the read operation by $\langle r_1, \nu_{r_1}\rangle$ succeeds the read operation by $\langle r_{|\mathcal{V}|}, \nu_{r_{|\mathcal{V}|}}\rangle$ that returns 1 in $\phi(C)$ and thus *violates atomicity*. This completes the proof of Case (1).

Case 2: The next case that needs investigation is the equality $|\mathcal{V}| = \frac{|\mathcal{S}|}{f} - 2$. Since we are using groups of nodes, it is possible that all the readers will be contained in a single group. Consider this situation for the following proof. As before, since $|\mathcal{V}| = \frac{|\mathcal{S}}{f} - 2$ we can divide the servers into $|\mathcal{V}| + 2$ blocks where each block contains f servers. Since we only assume one virtual node ($|\mathcal{V}| = 1$) then the total number of blocks is 3. We also consider the same construction for the write operation with the difference that the $\omega(1)$ is not skip-free but skips the block $B_{|\mathcal{V}|+2}$. In particular $\phi(wr_i)$ is the execution fragment that contains an incomplete $\omega(1)$ operation and

skips the set of blocks $\{B_{|\mathcal{V}|+2}\} \cup \{B_j | 1 \leq j \leq i-1\}$. As before, $\phi(wr_1)$ is the execution where all the servers $\{B_j | \, 1 \leq j \leq |\mathcal{V}|+1\}$ replied to $\omega(1)$ and all those replies are in transit. So $\phi(wr)$ is the extension of $\phi(wr_1)$ where all those replies are being received by the writer w.

Let now describe a series of finite execution fragments that extend $\phi(wr)$. We say that execution fragment $\phi(e1)$ extends $\phi(wr)$ by a complete read operation from the reader $\langle r_1, \nu_{r_1}\rangle$ that skips block B_1. To preserve atomicity, $\langle r_1, \nu_{r_1}\rangle$ returns 1. Consider now another execution, $\phi'(e1)$, that extends $\phi(wr_2)$ by the same read operation from $\langle r_1, \nu_{r_1}\rangle$ that again skips B_1. Recall that only the writer w and the servers in block B_1 can distinguish $\phi(wr)$ from $\phi(wr_2)$. So since $\langle r_1, \nu_{r_1}\rangle$ skips the servers in the block B_1, it cannot distinguish $\phi(e1)$ from $\phi'(e1)$ and thus returns 1 in $\phi'(e1)$ as well. We now extend $\phi'(e1)$ by execution fragment $\phi(e2)$ as follows: (1) a complete inform(1) operation from $\langle r_1, \nu_{r_1}\rangle$ that skips the servers in the block $B_{|\mathcal{V}|+2}$, and (2) a complete read operation from reader $\langle r_2, \nu_{r_1}\rangle$ that skips block B_1. The read from $\langle r_2, \nu_{r_1}\rangle$ returns 1 to preserve atomicity. Further consider the execution fragment $\phi(e3)$ which is the same with $\phi(e2)$, but with the difference that the inform operation from $\langle r_1, \nu_{r_1}\rangle$ is incomplete and also skips block B_1. Notice that $\phi(e2)$ and $\phi(e3)$ differ at the reader $\langle r_1, \nu_{r_1}\rangle$ and the servers in block B_1 only. Since the reader $\langle r_2, \nu_{r_1}\rangle$ does not receive any messages from B_1, it cannot distinguish the two executions. Therefore $\langle r_2, \nu_{r_1}\rangle$ returns 1 in $\phi(e3)$ as well.

It now remains to investigate two more execution fragments, $\phi(E)$ and $\phi(F)$. Let $\phi(E)$ extend $\phi(e3)$ with a complete read operation by $\langle r_3, \nu_{r_1}\rangle$. This read operation skips block B_2. The read from $\langle r_2, \nu_{r_1}\rangle$ cannot distinguish $\phi(E)$ from $\phi(e3)$ and so it returns 1 in $\phi(E)$ as well. Execution $\phi(F)$ has the same configuration as $\phi(E)$ with the difference that no write (ω_*) or inform(*) operation is invoked by any process. So $\langle r_1, \nu_{r_1}\rangle$, $\langle r_2, \nu_{r_1}\rangle$ and $\langle r_3, \nu_{r_1}\rangle$ return $\perp$ in

$\phi(F)$. However, since $\phi(E)$ and $\phi(F)$ only differ at block B_2, and since $\langle r_3, \nu_{r_1} \rangle$ skips B_2, it cannot differentiate the two executions fragments. Hence, $\langle r_3, \nu_{r_1} \rangle$ returns $\perp$ in $\phi(E)$ as well. Therefore, $\phi(E)$ *violates atomicity* since $\langle r_2, \nu_{r_1} \rangle$ that succeeds $\langle r_3, \nu_{r_1} \rangle$ returns 1 and $\langle r_3, \nu_{r_1} \rangle$ returns an older value, namely $\perp$. This completes the proof. $\square$

Per Lemma 4.3.1 semifast implementation are possible only if $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$. In addition, the following lemma shows that the existence of a semifast implementation also depends on the number of messages sent by a process during its second communication round.

Lemma 4.3.2 There is no semifast implementation of an atomic register if a read operation informs $3f$ or fewer servers during its second communication round.

Proof. Since $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$, we get that $|\mathcal{S}| > f(|\mathcal{V}| + 2)$, and hence in order to maintain at least one reader in the system, $|\mathcal{S}| > 3f$. Suppose by contradiction that there exist a semifast implementation A which requires a complete read operation to send equal to $3f$ INFORM messages during its second communication round. Recall that the reader that performs the informative phase, in order to preserve the termination property, should expect $2f$ replies (since up to f servers might fail). We proceed by showing that there exists an execution of A where a read operation returns 1 and performs a second communication round and a subsequent read operation returns 1 and again needs to perform a second communication round to complete, *violating* the third property of the semifast implementation.

Consider a finite execution fragment $\phi(1)$ where writer w invokes a $\omega(k)$ write operation and writes the value val_k on the atomic register. We extend $\phi(1)$ by a read operation ρ which performs two communication rounds and returns val_k. During the second communication round, ρ sent messages to $3f$ servers. Only $|srvInf(\rho)| = 2f$ servers get INFORM messages

from ρ and replied to those messages. Since f of the servers might be faulty, in order to preserve the termination property, ρ returns after the receipt of those replies. We further extend $\phi(1)$ by a second read operation ρ', which receives messages from $|srvAck(\rho')| = |\mathcal{S}| - f$ servers and misses f of the servers in $srvInf(\rho)$ such that $|srvAck(\rho') \cap srvInf(\rho)| = f$.

We now describe a second finite execution fragment $\phi(2)$ which is similar to $\phi(1)$ but with the difference that ρ is incomplete and only $|srvInf(\rho)| = f$ servers received the INFORM messages from ρ. In this execution, ρ' receives replies from all the servers that have been informed by ρ, namely $|srvAck(\rho') \cap srvInf(\rho)| = f$. Note that ρ' cannot distinguish $\phi(1)$ and $\phi(2)$ in terms of the number of servers informed by ρ. Since ρ' observed that only f servers were informed by ρ in $\phi(2)$ and since ρ might crash before completing, ρ' must perform a second communication round to ensure that any read operation s.t. $\rho' \rightarrow \rho''$ that receives replies from $|srvAck(i)| = |\mathcal{S}| - f$ servers will not observe $|srvAck(\rho'') \cap srvInf(\rho)| = 0$ and thus return an older value violating atomicity. Obviously the fact that ρ' proceeds to a second communication round does not violate the third property of Definition 4.1.1 since ρ and ρ' in $\phi(2)$ are concurrent. Since ρ' cannot distinguish $\phi(1)$ and $\phi(2)$, ρ' must perform a second communication round in $\phi(1)$ as well. However, in $\phi(1)$, $\rho \rightarrow \rho'$ and thus they are not concurrent. So $\phi(1)$ *violates the third property*, contradicting the assumption that there is a semifast implementation A, where any read operation needs to inform $\leq 3f$ servers. $\square$

We now state the main result of this section.

Theorem 4.3.3 No semifast implementation A exists if the number of virtual nodes in the system is $\geq \frac{|\mathcal{S}|}{f} - 2$ and if $3f$ or fewer servers are informed during a second communication round.

Proof. It follows directly from Lemmas 4.3.1 and 4.3.2. □

4.3.2 Impossibility of Semifast Implementations in MWMR environment

In this section we show that is not possible to obtain a semifast implementation of atomic registers in the MWMR setting in the presence of server failures.

4.3.2.1 Preliminaries.

For the MWMR setting we relax the definition of a semifast implementation as presented for the SWMR setting, by allowing read operations to perform more than two communication rounds (i.e., instead of two rounds we allow multiple rounds in Definition 4.1.1).

As presented in Section 4.2.4 operations can be partially ordered with respect to the values they write or return. A MWMR semifast implementation satisfies atomicity (Definition 3.2.5) if any execution satisfies the following conditions:

MW1: if there is a write operation $\omega(k)$ that writes value val_k and a read operation ρ such that $\omega(k) \rightarrow \rho$, and all other writes precede $\omega(k)$ then ρ returns val_k.

MW2: if the response steps of all write operations precede the invocation steps of the read operations ρ and ρ', then ρ and ρ' must return the same value.

MW3: If the response steps of all the write operations precede the invocation step of a read operation ρ then ρ returns a value written by some complete write.

For the reasons discussed in Section 3.3, we assume the communication scheme where a server replies to a READ (or WRITE or INFORM) message without waiting to receive any other READ (or WRITE or INFORM) messages. In this proof we say that an operation performs a *read phase*

during a communication round if it gathers information regarding the value of the object at that round. We say that an operation performs a *write phase* during a communication round if it propagates information regarding the value of the object to any subset of the servers at that round. A read phase of an operation (read or write) does not modify the value of the atomic object. On the other hand a write phase of an operation π behaves as follows according to its type: (1) a new, currently unknown value is written to the register, if π is a write operation (2) only previously known values are written to the register if π is a read operation. Note that by *"value of the atomic object"* we mean the set of parameters that together describe the state of the atomic register. Any operation phase that modifies those parameters (and thus the state of the atomic register) is considered to be a write phase.

We say that a complete operation π *skips* a server s if s does not receive any messages from the process p that invokes π and the process p does not receive any replies from s. All other servers that receive the READ, WRITE or INFORM messages from p reply to these, and p receives those replies. All other messages remain in transit. Since we assume that $f = 1$, any complete operation may skip at most one server. We say that an operation is *skip-free* if it does not skip any server.

Since we consider read operations that might perform multiple communication rounds to complete, we denote by ρ_i^j the jth communication round (phase) of a read operation ρ_i. In order to distinguish between the read and write phases of ρ_i, let ω_i^j denote that the jth phase of the read ρ_i is a write phase. An arbitrary delay may occur between two phases ρ_i^j and ρ_i^{j+1} where other read (write) operations or read (write) phases might be executed. So we define as $sr_i(j-1)$ a set of operation phases (read or write) with the property that any phase

$\rho_*^* \in sr_i(j-1)$, $\rho_*^* \rightarrow \rho_i^j$. A set $sr_i(j-1)$ might be equal to the empty set containing no operations.

Claim 4.3.4 A read operation ρ that succeeds any write operation $\omega(*)$ and write phase ω_*^* from an operation $\pi \neq \rho$, returns the value decided by the read phase that precedes its last write phase.

Proof. The claim follows from the fact that the read operation succeeds all the write operations and from atomicity properties **MW1** and **MW2**. Let assume that reader r performs the read operation ρ which in turn requires n communication rounds to complete. Furthermore let assume that ω^j is the last *write phase* of ρ and for simplicity of analysis we also assume that this is the only write phase of ρ. The result is still valid when multiple write phases are performed by ρ.

Since ρ succeeds all write operations then any read phase ρ^g, for $1 \leq g \leq n$ where n the total number of phases from ρ, will gather the same information about the value of the atomic register. So according to r's local policy and atomicity property **MW3** every read phase that precedes ω^j will decide the same value, say val to be the latest value written on the register. Let ρ^{j-1} be the last read phase operation that precedes ω^j. According to the assumption, a write phase of a read operation propagates the value gathered, to the system. So ω^j propagates value val which was observed by the read phases. Since ω^j performs a write operation on the register then any read phase ρ^ℓ, $j+1 \leq \ell \leq n$, such that $\omega^j \rightarrow \rho^\ell$ must decide val to preserve atomicity property **MW1**. So the last read phase ρ^n of the read operation returns val as well and hence val is the value returned by operation ρ. That completes the proof. □

4.3.2.2 Construction and Main Result.

We now present the construction we use to prove the main result. We show execution constructions assuming that two writers (w and w'), and two readers (r and r') participate in the system. We assume skip-free operations since they comprise the best case scenario and thus a lower bound for these is sufficient. Note here that the constructions of executions with fast read operations are similar to constructions presented in [30]. We use this approach and we present a generalization that contains read operations with single or multiple communication rounds suitable for our exposition. The main idea of the proof exploits executions with certain ordering assumptions which may violate atomicity. In particular we assume executions where the two writers perform concurrent and interleaved write operations. Those write operations are succeeded by a read operation ρ_1 invoked by r, and in turn ρ_1 is succeeded by a read ρ_2 invoked by r'. We analyze all the different cases in terms of communication rounds for ρ_1 and ρ_2. We show that in each case, a single server failure may cause violations of atomicity.

Let us first consider the finite execution fragment ϕ, constructed from the following skip-free, complete operations: (a) operation $\omega(2)$ by w', (b) operation $\omega(1)$ by w, and (c) operation ρ_1 by r. These operations are not concurrent and they are executed in the order $\omega(2) \rightarrow \omega(1) \rightarrow \rho_1$. By property **MW2**, operation ρ_1 returns 1.

We now invert the write operations of the above execution and we obtain execution ϕ', consisting of the following skip-free, complete operations in the following order: (a) operation $\omega(1)$ by w, (b) operation $\omega(2)$ by w', and (c) operation ρ_1 by r. As before, these operations are not concurrent. So in this case, by property **MW2**, operation ρ_1 returns 2.

The generalization ϕ_g of ϕ when the reader r performs n communication rounds is the following, for $1 \leq i \leq n$:

(a) operation $\omega(2)$ by w',

(b) operation $\omega(1)$ by w,

(c) a set of read operations $sr(i-1)$ from reads different than ρ_1, and

(d) a read or a write phase (ρ_1^i or ω_1^i resp.) of the ρ_1 operation from reader r.

Notice that for $n = 1$ and for $sr(0) = \emptyset$ no process can distinguish ϕ_g from ϕ. Clearly at the end of phase ρ_1^n, by property **MW2**, the operation ρ_1 from r returns 1.

Similarly we define the ϕ'_g to be the generalization of ϕ', where the write operations are inversed:

(a) operation $\omega(1)$ by w,

(b) operation $\omega(2)$ by w',

(c) a set of read operations $sr(i-1)$ from reads different than ρ_1, and

(d) a read or a write phase (ρ_1^i or ω_1^i resp.) of the ρ_1 operation from reader r.

In this case by the end of phase ρ_1^n, and by property **MW2**, the ρ_1 operation returns 2.

If we assume now, without loss of generality, that the last communication round ρ_1^n of r in ϕ_g is a write phase, thus ω_1^n, then r should not be able to differentiate ϕ_g from the following execution, for $1 \leq i \leq n-1$:

(a) operation $\omega(2)$ by w',

(b) operation $\omega(1)$ by w,

(c) a set of read operations $sr(i-1)$ from reads different than ρ_1,

(d) a read phase ρ_1^i of the ρ_1 operation from reader r,

(e) a set of read operations $sr(n-1)$ from reads different than ρ_1, and,

(f) operation $\omega(1)$ by ω_1^n.

By operation $\omega(1)$, the reader r tries to disseminate the information gathered from the previous rounds regarding the value of the atomic object. Similarly we can define ϕ_g' with the difference that reader r will perform a $\omega(2)$ operations during its last communication round.

Obviously we have the same setting as in Claim 4.3.4 and so by the same claim the decision for the return value must be made in ρ_1^{n-1}. Notice that the decision of r taken in ρ_1^{n-1} is not affected from the operations in $sr(n-1)$. So we can assume that ϕ_g and ϕ_g' contain only read phases by r. According now to property **MW2**, r will decide 1 by the end of ρ_1^{n-1} in ϕ_g and 2 by the end of ρ_1^{n-1} in ϕ_g'. Since we assume that we only have 2 readers in the system r and r', and assuming that r' does not perform any read operation in either ϕ_g or ϕ_g', then the sets $sr(i-1) = \emptyset$ for $1 \leq i \leq n$ in both executions ϕ_g and ϕ_g'.

Theorem 4.3.5 If the number of writers in the system is $W \geq 2$, the number of readers is $R \geq 2$, and $f \geq 1$ servers may fail, then there is no semifast atomic register implementation.

Proof. It suffices to show that the theorem holds for the basic case where $W = 2$, $R = 2$, and $f = 1$. We assume that there exists a semifast implementation and we derive a contradiction. Let w and w' be the writers, r and r' the readers, and $s_1, \ldots, s_{|\mathcal{S}|}$ the servers participating in the system. We show a series of executions and analyze the different cases of a semifast

implementation where writers are fast and readers perform n communication rounds. We show that in all of these cases atomicity can be violated.

We now define a series of finite execution fragments $\phi(i)$, where $1 \leq i \leq |\mathcal{S}|+1$. We assume that the two write operations from w and w' are concurrent. After the completion of both write operations a ρ_1 read operation, which may involve multiple communication rounds (phases), is invoked by r. For every $\phi(i)$ the set of read operations $sr(0) = \emptyset$ and so the ρ_1 from r is the first read after the completion of the write operations. Define $\phi(1)$ to be similar to ϕ_g. Then we iteratively define $\phi(i+1)$ to be similar to $\phi(i)$ except that server s_i receives the message from w before the message from w'. In other words the arrival order of the write messages are interchanged in s_i. Since the operations from w, w' and each communication round by r are skip-free, they can differentiate between $\phi(i)$ and $\phi(i+1)$. Also, s_i is the only server that can distinguish the two executions since we assume no communication between the servers. Obviously, by our construction, no server can distinguish $\phi(|\mathcal{S}|+1)$ from ϕ'_g since every server received the WRITE messages in the opposite order than in ϕ_g. Thus, r cannot distinguish the two executions either, and so it returns 2 in $\phi(|\mathcal{S}|+1)$ after the completion of its last communication round. Therefore, executions $\phi(|\mathcal{S}|+1)$ and ϕ'_g differ only at w and w'. Since ρ_1 returns 1 in $\phi(1)$, 2 in $\phi(|\mathcal{S}|+1)$ and 1 or 2 in $\phi(i)$ ($2 \leq i \leq |\mathcal{S}|$), there are two executions $\phi(m)$ and $\phi(m+1)$, for $1 \leq m \leq |\mathcal{S}|$, such that the read ρ_1 returns 1 in $\phi(m)$ and 2 in $\phi(m+1)$ at the end of the same communication round.

Consider now an execution fragment $\phi(m)'$ and an execution fragment $\phi(m+1)''$ that extend $\phi(m)$ and $\phi(m+1)$ respectively by a read operation ρ_2 from r' that skips s_m during all its required communication rounds. On the constructed executions we analyze the cases of semifast implementation. Recall that we investigate the case of the semifast implementation where

we allow the readers to perform n communication rounds and write operations are fast (only one communication round). We examine the different possible scenarios during executions $\phi(m)'$ and $\phi(m+1)''$:

(1) both ρ_1 and ρ_2 are fast in both executions,

(2) ρ_2 performs k communication rounds in $\phi(m)'$ and $\phi(m+1)''$ and ρ_1 is fast,

(3) ρ_1 performs n communication rounds in both executions and ρ_2 is fast, and

(4) both ρ_1 and ρ_2 perform n and k communication rounds respectively.

We assume that the processes decide to perform a second communication round according to their local policy.

Case 1: In this case both reads are fast and thus requiring only one communication round to complete. The read operation ρ_2 cannot distinguish the two executions $\phi(m)'$ and $\phi(m+1)''$ since it skips the only server (s_m) that can differentiate them. So the read ρ_2 returns, according to property **MW2**, 1 in $\phi(m)'$ and so it returns 1 in $\phi(m+1)''$ as well. However, ρ_1 cannot distinguish the executions $\phi(m+1)$ and $\phi(m+1)''$, and so, since it returns 2 in $\phi(m+1)$, it returns 2 in $\phi(m+1)''$ as well. Hence, $\phi(m+1)''$ violates property **MW2**.

Case 2: In this case ρ_2 performs k phases in executions $\phi(m)'$ and $\phi(m+1)''$. Since all read phases by ρ_2 skip the server s_m, then none of them is able to distinguish execution $\phi(m)'$ from $\phi(m+1)''$ since s_m is the only server who can differentiate them. Thus, ρ_2 retuns the same value in both executions. Since, according again to **MW2**, ρ_2 returns 1 in $\phi(m)'$ then it returns 1 in $\phi(m+1)''$ as well. Again, ρ_1 cannot distinguish $\phi(m+1)$ from $\phi(m+1)''$ so it returns 2 in $\phi(m+1)''$ as well. Thus, property **MW2** is violated in this case too.

Case 3: This is the case where ρ_1 performs n phases to complete and ρ_2 is fast. Since all the phases by ρ_1 are read phases, skip-free and precede ρ_2, then ρ_1 cannot distinguish execution $\phi(m)'$ from $\phi(m)$ and $\phi(m+1)''$ from $\phi(m+1)$. Therefore, ρ_1 returns 1 in $\phi(m)'$ and 2 in $\phi(m+1)''$. On the other hand, ρ_2 returns (according to property **MW2**) 1 during $\phi(m)'$. Since all n phases of r are read phases in both executions $\phi(m)'$ and $\phi(m+1)''$, then no server, writer or r' can distinguish each phase and they only differ at r. So, only s_m differentiates $\phi(m)'$ from $\phi(m+1)''$. Since though ρ_2 skips s_m, it cannot distinguish $\phi(m)'$ from $\phi(m+1)''$. Thus, it returns 1 in $\phi(m+1)''$ as well violating property **MW2**.

Case 4: Similarly to case 3, ρ_1 returns 1 during $\phi(m)'$ and 2 during $\phi(m+1)''$. With the same reasoning as in case 3 and since all phases of ρ_2 skip the server s_m, no communication round of ρ_2 can distinguish $\phi(m)'$ from $\phi(m+1)''$. So, ρ_2 returns 1 in both executions violating property **MW2**. This completes the proof. □

Chapter 5

Trading Speed for Fault-Tolerance

In Chapter 4 we showed that by not restricting the number of reader participants does not preclude fast operations in an atomic R/W register implementation. It is interesting to know how the replica host access strategies affects the fastness of the operations in the system. In the sections that follow we provide an answer to this question. First, we revisit the assumptions made on replica organization by (semi)fast implementations. Then, we examine whether (semi)fast implementations are feasible assuming replicas are organized in a *general* quorum construction. We show that a common intersection among the quorums of the quorum system is necessary. Such intersection implies that a single replica failure may collapse the underlying quorum system. To increase fault-tolerance, we introduce a new family of implementations, we call *weak-semifast*. We present a new weak-semifast algorithm that implements an atomic, SWMR register and trades the speed of some operations for fault-tolerance of the service. We prove the correctness of the proposed algorithm and we obtain empirical measurements of its operation latency.

5.1 Revisiting Replica Organization of (Semi)Fast Implementations

Operations in (semi)fast implementations of atomic R/W registers, as introduced in [30] and Chapter 4, relied on voting techniques for accessing overlaping subsets of replica hosts. For this reason, the participants of both algorithms needed to quantify and know in advance the maximum number of failures they could tolerate. The authors in [30] claimed that their algorithm tolerated up to $f < \frac{|\mathcal{S}|}{2}$ server failures. By the constraint on the number of readers however, we observe that:

$$|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2 \Rightarrow f < \frac{|\mathcal{S}|}{|\mathcal{R}| + 2}$$

Thus, in order for their algorithm to accommodate at least two reader participants, the number of server failures had to be bounded by $f < \frac{|\mathcal{S}|}{4}$. In general, the fault-tolerance of the algorithm in [30] was degrading proportionally to the number of reader participants in the system. A better fault-tolerance was achieved in the semifast implementations of Chapter 4 since unbounded number of readers were supported in just a *single* virtual node. Therefore, even if the number of virtual nodes was bounded by $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$, the algorithm could tolerate $f < \frac{|\mathcal{S}|}{3}$ server failures regardless of the number of readers in the system.

Despite this improvement, none of the two approaches achieved the optimal resilience obtained by Attiya et al. [9] that tolerated $f < \frac{|\mathcal{S}|}{2}$ failures. This fact demonstrates a possible relation between the failure pattern and thus, replica host organization, with operation fastness. Given that the approach of [9] was readily generalized from voting and majorities to quorum systems (e.g., [66, 68]), and given that every voting scheme yields a quorum system [25], one may ask: *What is the fault-tolerance of fast implementations deploying a general quorum-based framework?*

The work in [53] introduced the properties that a quorum construction must possess to enable fast R/W operations in SWMR atomic register implementations. However, the techniques they presented relied on synchronization assumptions and operation timeouts. So [53], as well as [30] and Chapter 4 of this thesis, neglected to examine the specific properties of a *general quorum construction* that may enable fast operations in atomic R/W register implementations. Answering this question may lead to more complex quorum system constructions that may allow fast read and write operations in completely asynchronous and unconstrained environments.

5.2 On the Fault-Tolerance of (Semi)Fast Implementations

In this section we investigate whether it is possible to obtain fast or semifast quorum-based implementations of atomic read/write register. We focus in examining the fault-tolerance of such implementations when we assume a general quorum construction and we allow unbounded reader participants.

Below we discuss our results regarding quorum-based fast and semifast implementations that respect our failure model and the observations we made in Section 3.1.4. We assume, w.l.o.g., that every quorum-based atomic register implementation utilizes a mechanism to establish when a value v was written “later” than a value v' (or v' is “older” than v).

5.2.1 Fast Implementations are Not Fault Tolerant

We now state the quorum property that is both necessary and sufficient to obtain fast quorum-based implementations.

The first lemma shows that if there is a common intersection between the quorums of the underlying quorum system then we can obtain quorum-based fast implementations.

Lemma 5.2.1 If for a quorum system $\mathbb{Q}$ it holds that $\bigcap_{Q\in\mathbb{Q}} Q \neq \emptyset$, then any quorum-based implementation of an atomic register A that deploys $\mathbb{Q}$ can be fast.

Proof. The fact that the common intersection is sufficient for fast implementations follows from a trivial implementation: each read/write operation contacts (only) the servers in the common intersection and returns the latest value observed in the first communication round. Notice here that according to our failure model, at least a single quorum is correct and thus all the servers of the common intersection must remain alive during the execution. Atomicity is not violated since every read/write operation will gather all the servers in the common intersection and furthermore all operations complete in a single communication round. □

Next we show that we cannot obtain fast quorum-based implementations if the underlying quorum system does not have a common intersection.

Lemma 5.2.2 A quorum-based implementation of an atomic register A that deploys a quorum system $|\mathbb{Q}| = n$ and supports $|\mathcal{R}| \geq n$ cannot be fast if $\mathbb{Q}$ satisfies: $\bigcap\limits_{Q\in\mathbb{Q}} Q = \emptyset$.

Proof. Let $|\mathbb{Q}| = n$ and let Q_i, for $1 \leq i \leq n$, be the identifiers of the quorums in $\mathbb{Q}$. We are going to proof by induction on the size of the quorum system $\mathbb{Q}$ that we cannot obtain a fast quorum-based implementation if the underlying quorum system $\mathbb{Q}$ does not have a common intersection.

Induction Basis: By the definition of a quorum system, for any two quorums $Q_i, Q_j \in \mathbb{Q}$: $Q_i \cap Q_j \neq \emptyset$. Thus, our inductive step examines whether all operations of a quorum-based implementation when $|\mathcal{R}| = |\mathbb{Q}| = 3$ and $Q_1 \cap Q_2 \cap Q_3 = \emptyset$ can be fast.

Consider the following executions. Let ξ_0 be a finite execution of a fast quorum-based implementation A. Let $\langle\sigma, \mathsf{write}(v)_w, \sigma'\rangle$ be the last step that appears in ξ_0, for some value v. Thus, the last step of ξ_0 is an invocation of a write operation $\omega(v)$. Let ξ_0' be a finite execution fragment that starts with the last state σ' of ξ_0 and ends with a state where $scnt(\omega(v), Q_1 \cap Q_2)_w$. That is, all the servers in $Q_1 \cap Q_2$ receive messages from $\omega(v)$. Let us assume w.l.o.g. that only the send and receive events from the writer to the servers appear in ξ_0'. The concatenation of ξ_0 and ξ_0' yields the execution ξ_1. Since $Q_1 \cap Q_2$ then ξ_0' is not empty and hence, ξ_0 is not the same as ξ_1. Similarly we assume that ξ_1' is a finite execution fragment that starts with the last state of ξ_1 and ends with a state where $scnt(\omega(v), Q_1)_w$. Let ξ_2 be equal to the concatenation of ξ_1 and ξ_1'. Furthermore let $\langle\sigma'', \mathsf{write\text{-}ack}_w, \sigma'''\rangle$ be the last step of ξ_2 where σ'' the last step of ξ_1'. In other words the write operation completes by the end of ξ_2.

Consider now the extension of executions ξ_1 and ξ_2 by a set of read operations. In particular let execution ξ_2 be extended by an execution fragment that contains the following operations:

(1) a complete read operation ρ_1 from r_1 that $scnt(\rho_1, Q_1)_{r_1}$, and

(2) a complete read operation ρ_2 from r_2 that $scnt(\rho_2, Q_2)_{r_2}$.

We call the new execution $\Delta(\xi_2)$. Let $\rho_1 \to \rho_2$. Since $\omega(v)$ is completed in ξ_2 then $\omega(v) \to \rho_1 \to \rho_2$ in $\Delta(\xi_2)$. Clearly by the definition of atomicity $\omega(v) \prec \rho_0$ and $\omega(v) \prec \rho_1$ and thus, both ρ_1 and ρ_2 have to return the value v written by $\omega(v)$. Notice that ρ_1 observes this value in all the servers of Q_1, whereas ρ_2 observes this value in the servers of the intersection $Q_1 \cap Q_2$.

We now obtain execution $\Delta(\xi_1)$ by extending ξ_1 with an execution fragment that contains the following operations:

(1) a complete read operation ρ_1 from r_1 that $scnt(\rho_1, Q_1)_{r_1}$,

(2) a complete read operation ρ_2 from r_2 that $scnt(\rho_2, Q_2)_{r_2}$, and

(3) a complete read operation ρ_3 from r_3 that $scnt(\rho_3, Q_3)_{r_3}$.

Let $\rho_1 \rightarrow \rho_2 \rightarrow \rho_3$. Recall that ξ_2 is the extension of ξ_1 that includes send and receive events for any server $s \in Q_1 - (Q_1 \cap Q_2)$. Since the write operation is fast, any server in $Q_1 \cap Q_2$ receive a single write message from the write operation. Thus, the state of every server $s \in Q_1 \cap Q_2$ is the same by the end of both ξ_2 and ξ_1. This is the same for every server $s \in Q_2 - (Q_1 \cap Q_2)$ that does not receive any message from the write operation. Hence, any server in Q_2 has the same state by the end of both ξ_2 and ξ_1.

Let us now examine how the state of the servers changes after the invocation of the first read operation. Since, r_1 does not receive any messages from any process in ξ_2 and ξ_1 then the state of r_1 is the same by the end of both executions. Thus, the same events from the invocation to the end of the first communication round (including response since its fast) of ρ_1 appear in both $\Delta(\xi_2)$ and $\Delta(\xi_1)$. So any server in $Q_1 \cap Q_2$ receive the same messages for ρ_1 in both executions. Thus, the state of all the servers in Q_2 at the response step of ρ_1 is also the same in $\Delta(\xi_2)$ and $\Delta(\xi_1)$. Hence ρ_2, since it strictly contacts Q_2, cannot distinguish $\Delta(\xi_1)$ from $\Delta(\xi_2)$. Thus, since it returns v in $\Delta(\xi_2)$, it returns v in $\Delta(\xi_1)$ as well. In order to preserve atomicity ρ_3 has to return v in $\Delta(\xi_1)$ as well, since $\rho_2 \rightarrow \rho_3$. According to our assumption $Q_1 \cap Q_2 \cap Q_3 = \emptyset$. Thus no server in Q_3 received any messages from $\omega(v)$ in $\Delta(\xi_1)$ but some of them received messages from ρ_1 and ρ_2.

Lastly, consider the execution ξ which is similar to ξ_0 but it ends before the invocation of $\omega(v)$. In other words, while ξ_0 ends with the step $\langle \sigma, \mathsf{write}(v)_w, \sigma' \rangle$, ξ ends with state σ. Observe that since no messages were delivered to the servers in neither ξ nor ξ_0, then the state of every server $s \in \mathcal{S}$ is $\sigma[s] = \sigma'[s]$. Let $\Delta(\xi)$ be the concatenation of ξ with the execution

fragment that contains the operations ρ_1, ρ_2 and ρ_3 as in $\Delta(\xi_1)$. Since, $Q_1 \cap Q_2 \cap Q_3 = \emptyset$ then ξ_0' contains no receive or send events for any server $s \in Q_3$. So, the state of every server $s \in Q_3$ at the end of ξ_1 is the same as the state of s at the end of ξ_0 and subsequently the same as the state of s at the end of ξ. Thus, with similar arguments as before, we can conclude that the state of every server $s \in Q_3$ is the same in both $\Delta(\xi_1)$ and $\Delta(\xi)$ at the response step of ρ_2. Since ρ_3 strictly contacts Q_3, it cannot distinguish the two executions. Hence, since it returned v in $\Delta(\xi_1)$ then it returns v in $\Delta(\xi)$ as well. But according to our construction $\omega(v)$ is not invoked in ξ. So atomicity is violated because the read returns a value that was not written. That contradicts our initial assumption that we can obtain fast quorum-based atomic register implementations when $Q_1 \cap Q_2 \cap Q_3 = \emptyset$.

Inductive Hypothesis: For our induction hypothesis we assume that we cannot obtain fast quorum based implementations when $|\mathbb{Q}| = |\mathcal{R}| = n - 1$ and $\bigcap_{Q \in \mathbb{Q}} Q = \emptyset$. From Lemma 5.2.1 it follows that if $\mathbb{Q}$ satisfies $\bigcap_{Q \in \mathbb{Q}} Q \neq \emptyset$, then any algorithm that deploys $\mathbb{Q}$ can be fast.

Inductive Step: For our induction step we examine the case where $|\mathbb{Q}| = |\mathcal{R}| = n$. By our induction hypothesis we know that we cannot obtain fast implementations if $|\mathbb{Q}| = n - 1$ and $\bigcap_{i=1}^{n-1} Q_i = \emptyset$; otherwise, if $\bigcap_{i=1}^{n-1} Q_i \neq \emptyset$ and by Lemma 5.2.1, any algorithm can be fast. So we assume that we can obtain fast quorum-based implementations when $\bigcap_{i=1}^{n-1} Q_i \neq \emptyset$ and $(\bigcap_{i=1}^{n-1} Q_i) \cap Q_n = \bigcap_{i=1}^{n} Q_i = \emptyset$, for $Q_i \in \mathbb{Q}$.

We consider a generalization of the execution presented in the basic step. In particular, we start from ξ_0 that ends with the step $\langle \sigma, \mathsf{write}(v)_w, \sigma' \rangle$. Let now ξ_0' be the finite execution fragment that starts with the last state σ' of ξ_0 and ends with a state where $scnt(\omega(v), \bigcap_{i=1}^{n-1} Q_i)_w$. That is, all the servers in $\bigcap_{i=1}^{n-1} Q_i$ receive messages from $\omega(v)$. As before, only the send and

receive events from the writer to the servers appear in ξ_0'. The concatenation of ξ_0 and ξ_0' yields the execution ξ_1. If no servers exists in that intersection then ξ_0' contains no events but only the state σ' and thus ξ_1 is equal to ξ_0. Similarly we construct a series of finite executions each of which extend ξ_0. We say that execution ξ_k, for $1 \leq k \leq n-1$, is the result of the concatenation of execution ξ_{k-1} and the execution fragment ξ_{k-1}', where ξ_{k-1}' starts with the last state of ξ_{k-1} and ends with a state where $scnt(\omega(v), \bigcap_{i=1}^{n-k} Q_i)_w$. Let, ξ_{n-1} end with a step $\langle \sigma'', \textsf{write-ack}_w, \sigma''' \rangle$ and σ'' is a state where $scnt(\omega(v), Q_1)_w$. Therefore, the write operation $\omega(v)$ completes by the end of ξ_{n-1}. Note also that the write in ξ_{n-2} strictly contacts the servers in $Q_1 \cap Q_2$, the write in ξ_{n-3} the servers in $Q_1 \cap Q_2 \cap Q_3$ and so on.

Let us extend each execution ξ_k, for $1 \leq k \leq n-1$, by an execution fragment which contains a set of complete read operations ρ_x, for $1 \leq x \leq n-k+1$, such that $\rho_x \rightarrow \rho_{x+1}$ and $scnt(\rho_x, Q_x)_{r_x}$, yielding execution $\Delta(\xi_k)$. Notice that every execution $\Delta(\xi_k)$ is similar to $\Delta(\xi_{k-1})$ except from the fact that only a subset of servers that received write messages for operation $\omega(v)$ in $\Delta(\xi_k)$ receives write messages in $\Delta(\xi_{k-1})$. Furthermore $\Delta(\xi_{k-1})$ contains an additional read operation ρ_{n-k+2} that strictly contacts the quorum Q_{n-k+2}.

Let us examine what is the return value of the last read operation of an execution $\Delta(\xi_k)$, for $1 \leq k \leq n-1$. For execution $\Delta(\xi_{n-1})$ atomicity is preserved if both read operations ρ_1 and ρ_2 return the value v written by $\omega(v)$, since the write operation is completed and precedes both operations. In execution $\Delta(\xi_{n-2})$ the write operation $\omega(v)$ is incomplete and $scnt(\omega(v), Q_1 \cap Q_2)_w$. Since ρ_1 is fast then every server $s \in Q_1 \cap Q_2$ and subsequent any server $s \in Q_2$ reach the same state in both $\Delta(\xi_{n-1})$ and $\Delta(\xi_{n-2})$. Thus, since $scnt(\rho_2, Q_2)_{r_2}$, the read operation ρ_2 cannot distinguish $\Delta(\xi_{n-2})$ from $\Delta(\xi_{n-1})$ and thus, returns v in $\Delta(\xi_{n-2})$ as well. Atomicity is preserved if the last read operation in $\Delta(\xi_{n-2})$, ρ_3, returns v as well. With a simple induction

we can show that the last two read operations ρ_{n-k} and ρ_{n-k+1} of any execution $\Delta(\xi_k)$, for $1 \leq k \leq n-1$, return the value v written by $\omega(v)$. From this it follows that read operation ρ_{n-1} cannot distinguish executions ξ_2 from ξ_1 thus, returns v in both executions. Hence, atomicity is preserved if read operation ρ_n returns v in $\Delta(\xi_1)$ as well.

Consider now the execution ξ which is the same as ξ_0 with the difference that it ends before the invocation step of the write operation. In other words, if $\langle \sigma, \mathsf{write}(v)_w, \sigma' \rangle$ is the last step of ξ_0, then the last state of ξ is σ. We extend ξ by an execution fragment that contains a set of complete read operations ρ_x, for $1 \leq x \leq n$, such that $\rho_x \rightarrow \rho_{x+1}$ and $scnt(\rho_x, Q_x)_{r_x}$, yielding execution $\Delta(\xi)$. So $\Delta(\xi)$ is similar to $\Delta(\xi_1)$ with the only difference that the write operation $scnt(\omega(v), \bigcap_{i=1}^{n-1} Q_i)_w$ in $\Delta(\xi_1)$ before the invocation of any read operation. By our assumption, $\bigcap_{i=1}^{n-1} Q_i \neq \emptyset$. So it follows that the state of any server $s \in \bigcap_{i=1}^{n-1} Q_i$ in $\Delta(\xi)$ is different from the state of s in $\Delta(\xi_1)$ since s received messages from the write operation in $\Delta(\xi_1)$. Thus, it follows that any read operation ρ_i, for $1 \leq i \leq n-1$, can distinguish the two executions since $scnt(\rho_i, Q_i)_{r_i}$. So it remains to examine read operation ρ_n.

We know that ρ_n strictly contacts Q_n. Since we assume that $(\bigcap_{i=1}^{n-1} Q_i) \cap Q_n = \emptyset$ then no server $s \in Q_n$ received messages from $\omega(v)$ in neither execution ξ_1 nor ξ. Moreover, since all the read operations are fast, then any server $s \in Q_n$ received the same messages from the first round of any read operation ρ_i, for $1 \leq i \leq n$, in both $\Delta(\xi_1)$ and $\Delta(\xi)$. Thus, any server $s \in Q_n$ reaches the same state in both $\Delta(\xi)$ and $\Delta(\xi_1)$. From this follows that ρ_n cannot distinguish $\Delta(\xi_1)$ from $\Delta(\xi)$. Therefore, since ρ_n returns v in $\Delta(\xi_1)$, it returns v in $\Delta(\xi)$ as well. This however, violates atomicity since $\Delta(\xi)$ does not contain the invocation and any messages from the write operation $\omega(v)$. Hence, this contradicts our initial assumption and thus ρ_n can return v and be fast only if $\bigcap_{i=1}^{n} Q_i \neq \emptyset$.

□

The main results follows from the Lemmas 5.2.2 and 5.2.1.

Theorem 5.2.3 A quorum-based implementation of a SWMR atomic read/write register A that deploys a quorum system $|\mathbb{Q}| = n$ and supports $|\mathcal{R}| \geq n$ readers can be fast iff $\mathbb{Q}$ satisfies: $\bigcap_{Q\in\mathbb{Q}} Q \neq \emptyset$.

In other words, by Theorem 5.2.3, unconstrained fast implementations of atomic register are possible if and only if all the quorums of the underlying quorum system have a *common* intersection. According to our failure model, a quorum is faulty if one of its members is faulty. Hence, if any node $s \in \bigcap_{Q\in\mathbb{Q}} Q$ fails, then all quorums become faulty since $\forall Q \in \mathbb{Q}, s \in Q$, and the whole quorum system $\mathbb{Q}$ fails. Therefore, such quorum construction suffers from a single point of failure and as a result it is not fault-tolerant. In turn, any implementation that relies on such quorum construction is not fault-tolerant either. So we derive the following observation:

Observation 5.2.4 A fast quorum-based implementation of atomic register is not fault-tolerant.

5.2.2 SemiFast Implementations are Not Fault Tolerant

Given that fast implementations are not possible if common intersection property is not satisfied by the quorum system, the natural question arises whether fault-tolerant *semifast* implementations can be obtained. We show that fault-tolerant semifast implementations are also impossible in the absence of a common intersection among the quorums used by the implementations. We use the properties of Definition 4.1.1.

The following lemma proves that if a read operation obtains the latest value from all servers of a quorum intersection alone, it cannot be fast. The following lemma applies to all quorum-based implementations that use a quorum system without a common intersection.

Lemma 5.2.5 Let $\mathbb{Q}$ be a quorum system without a common intersection that is used by an implementation A. A read operation ρ_1 by reader r_1 that $scnt(\rho_1, Q')_{r_1}$, for $Q' \in \mathbb{Q}$, cannot be fast and return a value v if $\exists \Phi \subset \mathbb{Q} - \{Q'\}$, such that $Z = Q' \cap \left(\bigcap_{Q \in \Phi} Q\right) \neq \emptyset$, and $\forall s \in Z, s.val = v$, and $\forall s' \in Q' - Z$, $s'.val = v'$ for some v' older than v.

Proof. Since $\mathbb{Q}$ has no common intersection, then for any $Q' \in \mathbb{Q}$ it follows that:

$$Q' \cap \left(\bigcap_{Q \in \mathbb{Q} - \{Q'\}} Q\right) = \emptyset$$

Since for any $Q', Q'' \in \mathbb{Q}$, $Q' \cap Q'' \neq \emptyset$, there must exist two non empty sets of quorums $\Phi_r, \Phi_\ell \subset \mathbb{Q}$ such that $\Phi_r = \mathbb{Q} - (\{Q'\} \cup \Phi_\ell)$ and:

$$Q' \cap \left(\bigcap_{Q \in \Phi_r} Q\right) \neq \emptyset$$

Pick the largest set Φ_r that satisfies the above property. Then $\forall Q''' \in \Phi_\ell$ the following is true:

$$Q' \cap \left(\bigcap_{Q \in \Phi_r} Q\right) \cap Q''' = \emptyset \tag{3}$$

Consider now execution ξ that contains an incomplete write operation $\omega(v)$ that $scnt(\omega(v), \bigcap_{Q \in \Phi_r \cup \{Q'\}} Q)_w$. We extend ξ by a read operation ρ_1 from reader r_1 that $scnt(\rho_1, Q')_{r_1}$. Every server $s \in \bigcap_{Q \in \Phi_r \cup \{Q'\}} Q$ received the messages from the write operation, and sets its value to $s.val = v$. Furthermore since we assume a single writer and operation $\omega(v)$ is incomplete, v is the latest value written in the system. Assume to derive contradiction that ρ_1 is fast and returns v.

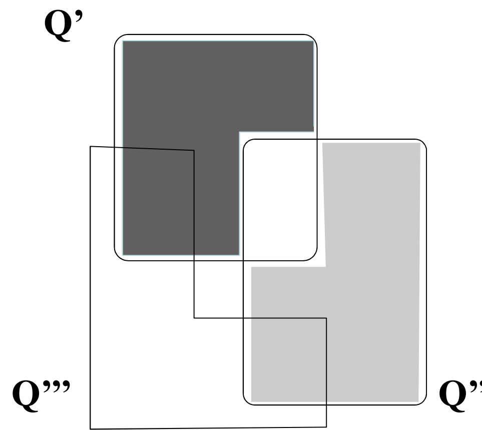


Figure 11: Intersections of three quorums Q', Q'', Q'''.

Let us extend ξ by a second read operation ρ_2 by a reader r_2 that $scnt(\rho_2, Q''')_{r_2}, Q''' \in \Phi_\ell$. Since ρ_1 is fast and it does not perform second communication round, then only the servers in $\bigcap_{Q\in\Phi_r\cup\{Q'\}} Q$ maintain the value v. By equation (3) it follows that ρ_2 observes and returns an older value v' violating atomicity. So, ρ_1 cannot be fast returning v. This completes our proof. $\square$

Using the above lemma we derive the following result.

Theorem 5.2.6 No quorum-based semifast implementation is possible with a quorum system $\mathbb{Q}$ such that $\bigcap_{Q\in\mathbb{Q}} Q = \emptyset$.

Proof. The proof builds upon execution constructions that exploit quorum system $\mathbb{Q}$ that contain triples of quorums (similar to the one presented in Figure 11) without a common intersection. Assume that we can obtain a semifast quorum-based implementation exploiting such

quorum system. Let the quorums $Q', Q'', Q''' \in \mathbb{Q}$ be quorums that have no common intersection, i.e., $Q' \cap Q'' \cap Q''' = \emptyset$.

Let, ρ_i^k denote the kth communication round of the read operation ρ_i. Consider execution ξ_1 that contains the following operations:

1. a complete write operation $\omega(v)$ that $scnt(\omega(v), Q'')_w$ succeeded by,

2. a read operation ρ_1 from r_1 that $scnt(\rho_1^1, Q')_{r_1}$.

Hence, $\omega(v) \rightarrow \rho_1$. In order to preserve atomicity ρ_1 has to return 1 in ξ_1 and according to Lemma 5.2.5 has to perform a second communication round before it completes.

Consider now an execution ξ_1' which is similar to ξ_1 but the write operation is incomplete. In particular ξ_1' consists of the following operations:

1. an incomplete write operation $\omega(1)$ that $scnt(\omega(v), Q' \cap Q'')_w$ succeeded by,

2. a read operation ρ_1 from r_1 that $scnt(\rho_1^1, Q')_{r_1}$.

Here, ρ_1 is concurrent with $\omega(v)$ but is invoked after the write operation $scnt(\omega(v), Q' \cap Q'')_w$. Notice that ρ_1 cannot distinguish between executions ξ_1 and ξ_1'. Thus, it returns v and performs a second communication round before completing in ξ_1' as well.

Let ξ_1' be extended by the second communication round of ρ_1 and a second read operation ρ_2 thus containing the following operations:

1. an incomplete write operation $\omega(v)$ that $scnt(\omega(v), Q' \cap Q'')_w$ succeeded by,

2. a complete read operation ρ_1 from r_1 that $scnt(\rho_1^1, Q')_{r_1}$ and $scnt(\rho_1^2, Q')_{r_1}$ during its first and second communication rounds respectively succeeded by,

3. a complete read operation ρ_2 from r_2 that $scnt(\rho_2^1, Q'')_{r_2}$

Here, both ρ_1 and ρ_2 are concurrent with the write but they are invoked after the write operation $scnt(\omega(v), Q' \cap Q'')_w$. However, the reads are not concurrent with $\rho_1 \rightarrow \rho_2$. Observe that in order to satisfy property **S3** of Definition 4.1.1, ρ_2 has to be *fast* since it succeeds a slow read. Moreover in order to preserve atomicity (and since ρ_1 returns v), ρ_2 must return v as well.

Finally consider an execution ξ_2 which is similar to ξ_1' with the difference that the second communication of the read operation ρ_1 is not yet completed. So the operations contained in ξ_2 are the following:

1. an incomplete write operation $\omega(v)$ that $scnt(\omega(v), Q' \cap Q'')_w$ succeeded by,
2. an incomplete read operation ρ_1 from r_1 that $scnt(\rho_1^1, Q')_{r_1}$ and $scnt(\rho_1^2, Q' \cap Q'')_{r_1}$ during its first and second communication rounds respectively succeeded by,
3. a complete read operation ρ_2 from r_2 that $scnt(\rho_2^1, Q'')_{r_2}$

Here, all operations are concurrent between each other. Both reads however, are invoked after the write $scnt(\omega(v), Q' \cap Q'')_w$, and ρ_2 is invoked after $scnt(\rho_1^2, Q' \cap Q'')_{r_1}$. Since ρ_2 receive replies from the members of the quorum Q'', as in execution ξ_1' observes that the servers in $Q'' \cap Q'$ received messages from $\omega(v)$, and from the first and the second communication round of ρ_1. Thus, it cannot distinguish the executions ξ_1' from ξ_2. Since ρ_2 is fast and returns v in ξ_1', then is fast and returns v in ξ_2 as well.

Finally we extend ξ_2 by a third read operation ρ_3 from r_3 which $scnt(\rho_3^1, Q''')_{r_3}$. Since only the servers in $Q' \cap Q''$ received value v, and since $Q' \cap Q'' \cap Q''' = \emptyset$, then ρ_3 observes and returns an older value v'. However this violates atomicity. Thus, ρ_2 has to be slow in ξ_2

in order to inform enough servers before completing. This action of ρ_2 will preserve atomicity and also does not violate property **S3** of Definition 4.1.1 in ξ_2. Since however, ρ_2 does not distinguish between ξ_2 and ξ_1', then if it is slow in ξ_2 it must be slow in ξ_1' as well. But this will violate property **S3** of Definition 4.1.1 in ξ_1' since there will be a slow read (ρ_2) succeeding a completed slow read (ρ_1) and both return the same value. That contradict our initial assumption and completes our proof. □

We similar reasoning as in Section 5.2.1 we derive the following observation:

Observation 5.2.7 Semifast quorum-based implementations of atomic register are not fault-tolerant.

5.2.3 Common Intersection in Fast and Semifast Implementations

As presented in Sections 5.2.1 and 5.2.2, a common intersection between all the quorums of a given quorum system is necessary in order to obtain fast or semifast implementations. Our findings raise the following question: *Was a common intersection necessary for the fast and semifast approaches proposed in [30] and Section 4.2 of this thesis?*

We construct a simple example which will help us visualize the intersection requirements of a fast implementation as proposed by [30]. Assume the following setting under [30]: a set of five servers with identifiers $\mathcal{S} = \{1, 2, 3, 4, 5\}$ one of which may fail by crashing ($f = 1$). According to [30] fast implementations are possible only if readers are constrained under $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$. Therefore, this setting supports no more than $|\mathcal{R}| = 2$ readers and one writer. Let us assume that readers have identifiers $\mathcal{R} = \{r, r'\}$ and the writer has identifier w.

The algorithm presented in [30], relied on the number of replies received for the completion of each read/write operation. In particular, any read/write operation was expecting $|\mathcal{S}| - f$

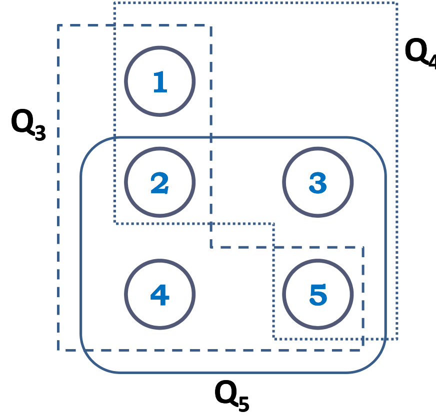


Figure 12: Graphical representation of quorums Q_3, Q_4 and Q_5.

servers to reply before completing. Thus, in our setting any operation had to wait for replies from one of the following sets: $Q_1 = \{1, 2, 3, 4\}, Q_2 = \{1, 3, 4, 5\}$ $Q_3 = \{1, 2, 4, 5\}, Q_4 = \{1, 2, 3, 5\}$, $Q_5 = \{2, 3, 4, 5\}$. Every two sets have a non-empty intersection and hence, those sets compose a quorum system $\mathbb{Q} = \{Q_1, Q_2, Q_3, Q_4, Q_5\}$ where each $Q_i \in \mathbb{Q} : |Q_i| = |\mathcal{S}| - f$. Figure 12 depicts the sets Q_3, Q_4 and Q_5.

Let us now consider an execution of this algorithm that contains a write operation ω from w, followed by a read operation ρ from r and a read operation ρ' from r'. Since the system supports two readers and one writer, in the worst case each participant receives replies from a different quorum for each operation. Let w.l.o.g., w write a value in Q_5 ($scnt(\omega, Q_5)_w$), r to $scnt(\rho, Q_3)_r$ and r' to $scnt(\rho', Q_4)_{r'}$. Observe that the intersection $Q_3 \cap Q_4 \cap Q_5 = \{2, 5\}$ and contains $f + 1$ (two) servers. Since both r and r' observed the value written by w, any subsequent read operation (from either r or r') will also be aware of the value written by ω. As a result, any subsequent read will return an equal or newer value and violation of atomicity is avoided. (Similar arguments can be made for the semifast algorithm presented in Section 4.2.) Hence, the restriction on the number of readers allows the concentration of the common

intersection between a subset of quorum sets, which serves as a “hot spot” to ensure consistency between the operations. This observation yields the following remark:

Remark 5.2.8 Fast or Semifast quorum-based implementations do not require a common intersection in the quorum system they deploy if either:

- we relax the failure model and operations can wait to receive replies from more than a single quorum, or
- we impose restrictions on the participation and on the construction of the quorum system.

Such restrictions however, will negatively affect the performance of the quorum system and will introduce strong assumptions for its maintenance, making eventually the use of quorums impractical. Thus, in this work we avoid making such assumptions and we prefer to trade operation latency for higher fault-tolerance and applicability.

5.3 Weak Semifast Implementations

Recall that fast implementations [30] require every read and write operation to complete in a single round. Semifast implementations (see Chapter 4) on the other hand, allow a single complete read operation to be slow per write operation. Since neither fast nor semifast implementations are fault-tolerant, one may ask whether we can relax some of their requirements and allow at least some operations to be fast in an unconstrained, in terms of quorum construction and participation, environment. As demonstrated by [22, 28], single round reads are possible in quorum-based implementations in the MWMR environment, whenever it is confirmed that a read operation is not concurrent with a write. Such strategy however, did not overcome the observation of [9] that reads concurrent with a write must perform a second round.

We show that one may obtain *weak-semifast* implementations in these settings defined as follows:

Definition 5.3.1 (Weak-Semifast Implementations) An implementation of an atomic R/W register is called **weak-semifast** if it satisfies properties **S1**, **S2** and **S4** (but not **S3**) of Definition 4.1.1.

From the fact that a weak-semifast implementation does not satisfy **S3** of Definition 4.1.1, it follows that it allows more than a single complete read operations to be slow for each write operation. On the other hand, such implementations need to satisfy property **S4** and thus, should be capable to yield executions that contain read and write concurrency and all operations are fast.

5.4 Weak-Semifast Implementation: Algorithm SLIQ

In previous sections we established that no fault-tolerant fast or semifast quorum-based implementations are possible. We therefore now consider *weak-semifast* implementations. We develop a client-side decision tool, called *Quorum Views*, and we devise an algorithm, called SLIQ, for atomic registers. In SLIQ, writes are fast and read operations may perform one or two rounds. We deviate from the restrictive common intersection presented in Section 5.2 and we allow our implementation to use an *arbitrary* quorum system. Our algorithm deploys $\langle$timestamp,value$\rangle$ pairs to order read and write operations. We first present the idea behind the quorum views. In Section 5.4.2 we provide a compact description of the algorithm followed by its formal specification in Section 5.4.3. Finally, we show that the algorithm is correct and satisfies all the properties of the weak-semifast implementations in Section 5.4.4.

5.4.1 Examining Value Distribution – Quorum Views

To facilitate the creation of weak-semifast implementations, we introduced a new client-side decision tool, called *Quorum Views*. A quorum view refers to the distribution of a register value as it is witnessed by a read operation during a communication round. Our approach inherits the $\langle timestamp, value\rangle$ pair to impose partial ordering on the written values. As each value is associated with a unique timestamp, we define quorum views in terms of the timestamp distribution instead of the actual written value. Let $maxTS$ denote the maximum timestamp that a read discovers during some round. Also, let $m(\rho, c)_{s,r}.ts$ denote the timestamp that server s sends during the cth round of the read operation ρ to the invoking reader r. Given this notation, quorum views are defined as follows:

Definition 5.4.1 (Quorum Views) Any read operation ρ that receives replies from all the members of a quorum $Q \in \mathbb{Q}$ in some round, witness one of the following **quorum views**:

QV1. $\forall s \in Q : m(\rho, c)_{s,r}.ts = maxTS$,

QV2. $\forall Q' \in \mathbb{Q}, Q \neq Q', \exists A \subseteq Q \cap Q'$, s.t. $A \neq \emptyset$ and $\forall s \in A : m(\rho, c)_{s,r}.ts < maxTS$, and

QV3. $\exists s' \in Q : m(\rho, c)_{s',r}.ts < maxTS$ and $\exists Q' \in \mathbb{Q}, Q \neq Q'$ and $\forall s \in Q \cap Q' : m(\rho, c)_{s,r}.ts = maxTS$

Under the assumption that servers always maintain the largest timestamp they receive, these three types of quorum views may reveal the state of the write operation (complete or incomplete) which tries to write the value associated to $maxTS$. Figure 13 illustrates those quorum views assuming that the read operation ρ, receives replies from the servers in Q. The dark

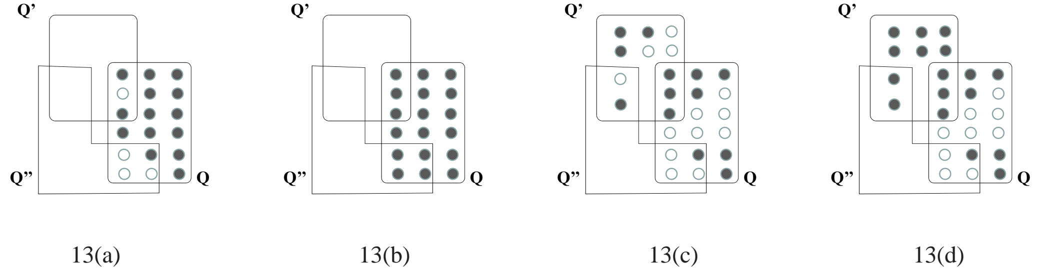


Figure 13: (a) $\mathbf{QV1}$, (b) $\mathbf{QV2}$, (c) $\mathbf{QV3}$ with incomplete write, (d) $\mathbf{QV3}$ with complete write.

nodes maintain the maximum timestamp of the system and white nodes or "empty" quorums maintain an older timestamp.

Recall that by our failure model a single quorum of servers is guaranteed to be non-faulty. Thus, any R/W operation is guaranteed to terminate as long as it waits for the servers of a single quorum to reply.

By the first quorum view, $\mathbf{QV1}$ (see Figure 13(a)), the read operation ρ obtains the maximum timestamp-value pair from all servers of quorum Q. This implies the possible completion of the write operation ω that propagates the value associated with $maxTS$: (1) The writer invoked ω to write $\langle maxTS, v\rangle$ pair, (2) $\forall s \in Q$ received the message for ω and updated its local register replica, and (3) $\forall s \in Q$ possibly replied to ω. Since the write operation cannot wait for more than a single quorum to reply, then ω completes when those replies are delivered to the writer. Thus, we say that $\mathbf{QV1}$ implies a potentially *complete write* operation.

By definition, every two quorums $Q, Q' \in \mathbb{Q}, Q \cap Q' \neq \emptyset$. Therefore, if there exists a quorum Q such that $\forall s \in Q$, $m(\rho, c)_{s,r}.ts = maxTS$, then it follows that $\forall Q' \in \mathbb{Q}$, $s' \in Q' \cap Q$ replies with $m(\rho, c)_{s',r}.ts = maxTS$ to ρ. That is, all servers in any intersection of Q must reply with $maxTS$ to a read operation ρ. From this observation, $\mathbf{QV2}$ reveals an *incomplete write* operation. Recall that, by $\mathbf{QV2}$, ρ witnesses a subset of servers that maintain

a timestamp older than $maxTS$ in each intersection of Q (see Figure 13(b)). This implies that the write operation (which propagates $maxTS$) has not yet received replies from any full quorum and thus, has not yet completed.

Finally, $\mathbf{QV3}$, provides insufficient information regarding the state of the write operation. Specifically, if an operation receives replies from a quorum Q (that contains some servers with timestamp less than $maxTS$) and witnesses some intersection $Q \cap Q'$ that contains $maxTS$ in all of its servers, then a write operation might: (i) have been completed and contacted Q' (see Figure 13(d)) or (ii) be incomplete and contacted a subset of servers B such that $Q \cap Q' \subseteq B$ and $\forall Q'' \in \mathbb{Q}, Q'' \not\subseteq B$ (see Figure 13(c)).

5.4.2 High Level Description of SLIQ

Using quorum views, we developed the first algorithm that allows fast operations and does not depend on any service participation and quorum construction constraints. In particular, the algorithm allows: (i) more than a single slow reads per write operation and (ii) read operations to be fast even in cases they are concurrent with a write operation. Below we provide a brief description of the protocol of each participant of the service. This algorithm utilizes timestamp-value pairs where the value is a tuple $\langle v, vp \rangle$ that contains both the new value to be written v and the previous value written vp.

Writer. The write protocol involves the propagation of a write message to all the servers. Once the writer receives replies from a full quorum it increments its timestamp and the operation completes.

Readers. The read protocol requires that a reader propagates a read message to all the servers. Once the reader receives replies from a full quorum, it examines the distribution of the maximum timestamp-value pair ($\langle maxTS, v, vp \rangle$) within that quorum. This distribution characterizes a quorum view.

If the view is either $\mathbf{QV1}$ or $\mathbf{QV2}$ then the reader terminates in the first communication round. If $\mathbf{QV1}$ is observed, then the write operation that propagates $\langle maxTS, v, vp \rangle$ is potentially completed and thus, the read operation returns v. If $\mathbf{QV2}$ is observed, then the write operation that propagates $\langle maxTS, v, vp \rangle$ is not yet completed. Since we have a single well-formed writer, the detection of $maxTS$ implies the completion of the write operation that propagated the previous value vp (associated with $maxTS - 1$). Thus, in case the reader observes $\mathbf{QV2}$ it returns the value vp in a single round.

If $\mathbf{QV3}$ is observed, then the reader cannot determine the status of the write operation and thus, proceeds to a second communication. During this round, the reader sends $\langle maxTS, v, vp \rangle$ to all servers. Once the reader gets replies from a full quorum, the read operation completes and returns v.

Servers. The servers maintain a passive role; they just receive read and write messages, update their replica value if the timestamp included in a message is higher than their local timestamp, and reply to those messages.

5.4.3 Formal Specification of SLIQ

We now present the formal specification of SLIQ using Input/Output Automata [67] notation. Our implementation includes four automata: (i) automaton SLIQ_w that handles the write operations for the writer process w, (ii) automaton SLIQ_r that handles the reading for each

$r \in \mathcal{R}$, (iii) automaton SLIQ_s that handles the read and write requests on the atomic register for each $s \in \mathcal{S}$, and (iv) $Channel_{p,p'}$ that establish the reliable asynchronous process-to-process communication channels (see Section 3.1.2).

Automaton SLIQ_w.

The state variables, the signature and the transitions of the SLIQ_w can be depicted in Figure 14. The state of the SLIQ_w automaton includes the following variables:

- $\langle ts, v, vp \rangle \in \mathbb{N} \times V \times V$: writer's local timestamp along with the latest and the previous value written by the writer.
- $wCounter \in \mathbb{N}$: the number of write requests performed by the writer. Is used by the servers to distinguish fresh from stale messages.
- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the writer received all the necessary replies to complete its write operation and is ready to respond to the client.
- $srvAck \subseteq \mathcal{S}$: a set that contains the servers that reply to the write messages as a result of a write request. The set is reinitialized to $\emptyset$ at the response step of every write operation.
- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

Signature:

Input:
$\mathsf{write}(val)_w,\ v \in V$
$\mathsf{rcv}(m)_{s,w},\ m \in M,\ s \in \mathcal{S}$
fail_w

Output:
$\mathsf{send}(m)_{w,s},\ m \in M,\ s \in \mathcal{S}$
$\mathsf{write\text{-}ack}_w$

Internal:
$\mathsf{write\text{-}fix}_w$

State:

$ts \in \mathbb{N}$, initially 0
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$wCounter \in \mathbb{N}^+$, initially 0
$srvAck \subseteq \mathcal{S}$, initially $\emptyset$
$status \in \{idle, active, done\}$, initially $idle$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{write}(val)_w$
Effect:
if $\neg failed$ then
if $status = idle$ then
$status \leftarrow active$
$srvAck \leftarrow \emptyset$
$vp \leftarrow v$
$(v, ts) \leftarrow (val, ts + 1)$
$wCounter \leftarrow wCounter + 1$

Input $\mathsf{rcv}(\langle msgT, t, C\rangle)_{s,w}$
Effect:
if $\neg failed$ then
if $status = active$ and $wCounter = C$ then
$srvAck \leftarrow srvAck \cup \{s\}$

Internal $\mathsf{write\text{-}fix}_w$
Precondition:
$\neg failed$
$status = active$
$\exists Q \in \mathbb{Q} : Q \subseteq srvAck$
Effect:
$status \leftarrow done$

Output $\mathsf{send}(\langle msgT, t, C\rangle)_{w,s}$
Precondition:
$status = active$
$\neg failed$
$s \in \mathcal{S}$
$\langle msgT, t, C\rangle = \langle \text{WRITE}, \langle ts, v, vp\rangle, wCounter\rangle$
Effect:
none

Output $\mathsf{write\text{-}ack}_w$
Precondition:
$status = done$
$\neg failed$
Effect:
$status \leftarrow idle$

Input fail_w
Effect:
$failed \leftarrow true$

Figure 14: SLIQ_w Automaton: Signature, State and Transitions

The automaton completes a write operation in a single phase. When a $\mathsf{write}(val)_w$ request is received from the automaton's environment, the $status$ variable becomes $active$, the previous value vp gets the current value, the variable v gets the requested value val to be written, and ts is incremented. As long as the $status$ remains active the automaton sends messages to all server processes and collects the identifiers of the servers that reply to those messages in the $srvAck$ set. The operation is done when the process receives replies from the members of

a full quorum, i.e., $\exists Q \in \mathbb{Q} : Q \subseteq srvAck$. The $status$ of the automaton becomes $idle$ when the writer responds to the environment and the $\mathsf{write\text{-}ack}_w$ event occurs.

Automaton SLIQ_r**.**

The state variables, the signature and the transitions of the SLIQ_r can be depicted in Figure 15. The state of the SLIQ_r automaton includes the following variables:

- $\langle ts, v, vp \rangle \in \mathbb{N} \times V \times V$: the maximum timestamp discovered during r's last read operation along with its associated value and previous value.
- $maxTS \in \mathbb{N}$, $maxPS \in \mathbb{N}$, and $retvalue \in V$: the maximum timestamp and postit discovered, and the value that was returned during the last read operation.
- $rCounter \in \mathbb{N}$: read request counter. Used by the servers to distinguish fresh from stale messages.
- $phase \in \{1, 2\}$: indicates the active communication round of the read operation.
- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the reader decided on the value to be returned and is ready to respond to the client.
- $srvAck \subseteq M \times \mathcal{S}$: a set that contains the servers and their replies to the read operation. The set is reinitialized to $\emptyset$ at the response step of every read operation.

- $maxTsAck \subseteq M \times \mathcal{S}$ and $maxPsAck \subseteq M \times \mathcal{S}$: these sets contain the servers that replied with the maximum timestamp and maximum postit respectively to the last read request. The sets also contain the messages sent by those servers.
- $maxTsSrv \subseteq \mathcal{S}$: The servers that replied with the $maxTS$.
- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

Any read operation requires one or two phases to complete (fast or slow). The decision on the number of communication rounds is based on the quorum views that the reader obtains during its first communication round.

A read operation is invoked when the SLIQ$_r$ automaton receives a read_r request from its environment. The status of the automaton becomes $active$. As long as the reader is active, the automaton sends messages to each server $s \in \mathcal{S}$ to obtain the value of the register replicas. The $\mathsf{rcv}(m)_{s,r}$ action is triggered when a reply from a server s is received. The reader collects the identifiers of servers that replied to the current operation and their messages, by adding a pair (s, m) in the $srvAck$ set. When the set $srvAck$ contains the members of at least a single quorum Q of the quorum system $\mathbb{Q}$, the set of messages is filtered to find the messages that contain the maximum timestamp. Those messages are placed in $maxTsAck$ set. The servers that belong into the collected quorum and have messages in $maxTsAck$ they are placed separately in the $maxTsSrv$ set.

From the newly formed sets the reader extracts the information regarding the quorum view of Q. Based on the quorum view the reader decides to complete the operation or proceed to a second communication round. In particular, the reader is fast and completes in one round trip

Signature:

Input:
$\text{read}_r,\ r \in \mathcal{R}$
$\text{rcv}(m)_{s,r},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\text{fail}_r,\ r \in \mathcal{R}$

Output:
$\text{send-read}(m)_{r,s},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\text{send-info}(m)_{r,s},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\text{read-ack}(val)_r,\ val \in \mathcal{V},\ r \in \mathcal{R}$

Internal:
read-phase1-fix_r
read-phase2-fix_r

State:

$ts \in \mathbb{N}$, initially 0
$maxTS \in \mathbb{N}$, initially 0
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$retvalue \in V$, initially $\perp$
$rCounter \in \mathbb{N}^+$, initially 0
$phase \in \{1, 2\}$, initially 1
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxTsSrv \subseteq \mathcal{S}$, initially $\emptyset$
$failed$, a Boolean initially **false**

Transitions:

Input read_r
Effect:
if $\neg failed$ then
  if $status = idle$ then
    $status \leftarrow active$
    $rCounter \leftarrow rCounter + 1$

Input $\text{rcv}(\langle msgT, t, C\rangle)_{s,r}$
Effect:
if $\neg failed$ then
  if $status = active$ and $rCounter = C$ then
    $srvAck \leftarrow srvAck \cup \{(s, \langle msgT, t, C\rangle)\}$

Output $\text{send}(\langle msgT, t, C\rangle)_{r,s}$
Precondition:
$status = active$
$\neg failed$
$\big[(phase = 1 \wedge \langle msgT, t, C\rangle = \langle \text{READ}, \langle maxTS, v, vp\rangle, rCounter\rangle)\vee$
$(phase = 2 \wedge \langle msgT, t, C\rangle = \langle \text{INFORM}, \langle maxTS, v, vp\rangle, rCounter\rangle)\big]$
Effect:
none

Output $\text{read-ack}(val)_r$
Precondition:
$\neg failed$
$status = done$
$val = retvalue$
Effect:
$status \leftarrow idle$

Input fail_r
Effect:
$failed \leftarrow true$

Internal read-phase1-fix_r
Precondition:
$\neg failed$
$status = active$
$phase = 1$
$\exists Q \in \mathbb{Q} : Q \subseteq srvAck$
Effect:
$maxTS \leftarrow \{max(m.t.ts) : (s, m) \in srvAck \wedge s \in Q\}$
$maxAck \leftarrow \{(s, m) : (s, m) \in srvAck\ \wedge\ m.t.ts = maxTS\}$
$(v, vp) \leftarrow \{(m.t.v, m.t.vp) : (s, m) \in maxAck\}$
$maxTsSrv \leftarrow \{s : s \in Q, (s, m) \in maxAck\}$
if $Q \subseteq maxTsSrv$ then
  $ts \leftarrow maxTS$
  $retvalue \leftarrow v$
  $status \leftarrow done$
else
  if $\exists Q' \in \mathbb{Q}, Q' \neq Q$ s.t. $Q \cap Q' \subseteq maxTsSrv$ then
    $ts \leftarrow maxTS$
    $retvalue \leftarrow v$
    $phase \leftarrow 2$
    $srvAck \leftarrow \emptyset$
    $rCounter \leftarrow rCounter + 1$
  else
    $ts \leftarrow maxTS - 1$
    $retvalue \leftarrow vp$
    $status \leftarrow done$

Internal read-phase2-fix_r
Precondition:
$\neg failed$
$status = active$
$phase = 2$
$\exists Q \in \mathbb{Q} : Q \subseteq srvAck$
Effect:
$status \leftarrow done$
$phase \leftarrow 1$

Figure 15: SLIQ_r Automaton: Signature, State and Transitions

when it observes a $\mathbf{QV1}$ or $\mathbf{QV2}$. As described in Section 5.4.1, $\mathbf{QV1}$ denotes a complete write operation and $\mathbf{QV2}$ an incomplete write operation. So the read operation returns the value v associated with $maxTS$ if $\mathbf{QV1}$ is observed; otherwise, if $\mathbf{QV2}$ is retrieved, it returns the value vp which was associated with $maxTS - 1$. In the case where $\mathbf{QV3}$ is witnessed the read operation proceeds to a second round and after its completion returns $maxTS$. As shown in the algorithm the decision of the fast or slow behavior is determined in the internal action read-phase1-fix. If a second communication round is not necessary the read operation completes and sets the $status$ variable to $done$. Otherwise the phase number increases declaring that a second communication round is necessary and the operation is terminated when the precondition of read-phase2-fix is reached.

Automaton SLIQ_s**.**

The server automaton has relatively simple actions. The signature, state and transitions of the SLIQ_s can be depicted in Figure 16. The state of the SLIQ_s contains the following variables:

- $\langle ts, v, vp \rangle \in \mathbb{N} \times V \times V$: the maximum timestamp reported to s by an invoked operation along with its associated value and previous value. This is the value of the register replica.

- $Counter(p) \in \mathbb{N}$: this array maintains the latest request index of each client (reader or writer). It helps s to distinguish fresh from stale messages.

- $status \in \{idle, active\}$: specifies whether the automaton is processing a request received ($status = active$) or it can accept new requests ($status = idle$).

Signature:

Input:
$\mathsf{rcv}(m)_{p,s},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \mathcal{W}$
fail_s

Output:
$\mathsf{send}(m)_{s,p},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \mathcal{W}$

State:

$ts \in \mathbb{N}$, initially 0
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$Counter(p_i) \in \mathbb{N}^+,\ p_i \in \mathcal{R} \cup \{w\}$, initially 0
$msgType \in \{$WRITEACK,READACK,INFOACK$\}$
$status \in \{idle, active\}$, initially $idle$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{rcv}(\langle msgT, t, C\rangle)_{p,s}$
Effect:
if $\neg failed$ then
  if $status = idle$ and $C > Counter(p)$ then
    $status \leftarrow active$
    $Counter(p) \leftarrow C$
    if $t.ts > ts$ then
      $(ts, v, vp) \leftarrow$
        $(t.ts, t.v, t.vp)$
    if $msgT =$ WRITE then
      $msgType \leftarrow$ WRITEACK
    if $msgT =$ READ then
      $msgType \leftarrow$ READACK
    if $msgT =$ INFORM then
      $msgType \leftarrow$ INFOACK

Output $\mathsf{send}(\langle msgT, t, C\rangle)_{s,p}$
Precondition:
  $\neg failed$
  $status = active$
  $p \in \mathcal{R} \cup \{w\}$
  $\langle msgT, t, C\rangle =$
    $\langle msgType, \langle ts, v, vp\rangle, Counter(p)\rangle$
Effect:
  $status \leftarrow idle$

Input fail_s
Effect:
  $failed \leftarrow true$

Figure 16: SLIQ$_s$ Automaton: Signature, State and Transitions

- $msgType \in \{$WRITEACK,READACK,INFOACK$\}$: Type of the acknowledgment depending on the type of the received message.
- $failed \in \{true, false\}$: indicates whether the server associated with the automaton has failed.

Each server replies to a message without waiting to receive any other messages from any process. Thus, the status of the server automaton determines whether the server is busy processing a message ($status = active$) or if it is able to accept new messages ($status = idle$). When a new message arrives, the $\mathsf{rcv}(m)_{p,s}$ event is responsible to process the incoming message. If the $status$ is equal to idle and this is a fresh message from process p then the $status$

becomes active. The $Counter(p)$ for the specific process becomes equal to the counter included in the message. If the timestamp included in the received message is greater than its local timestamp. the server updates its timestamp and value variables to be equal to the ones included in the received message. The type of the received message specifies the type of the acknowledgment.

While the server is active, the $\mathsf{send}(m)_{s,p}$ event may be triggered. When this event occurs, the server s sends its local replica value, to a process p. The execution of the action results in modifying the $status$ variable to $idle$ and thus setting the server enable to receive new messages.

5.4.4 Correctness of SLIQ

To prove the correctness of our algorithm we need to show that it satisfies both Definition 3.2.4 (*termination*) and Definition 3.2.5 (*atomicity*).

Termination

According to our algorithm an operation terminates whenever a write-ack or read-ack event appears in our execution. Moreover, by the assumed failure model the adversary may fail all but one quorums in our quorum system. Recall, that every correct process p terminates once it receives replies from a single full quorum Q. Thus, it is easy to see that every correct process terminates if the assumed failure model is satisfied.

Atomicity

We proceed to show that algorithm SLIQ satisfies all the properties presented in Definition 3.2.5. We adopt the notation presented in Chapter 3. For completeness we restate the notation here as well. We use var_p to refer to the variable var of the automaton A_p. To access the value of a variable var of A_p in a state σ of an execution ξ, we use $\sigma[p].var$ (see Section 3.1.1). Also, let $m(\pi, c)_{p,p'}$ to denote the message sent from p to p' during the cth round of operation π. Any variable var enclosed in a message is denoted by $m(\pi, c)_{p,p'}.var$ (see Section 3.1.2). We refer to a step $\langle\sigma, \text{read-phase1-fix}_r, \sigma'\rangle$ as the *read-fix step* of a read operation ρ invoked by reader r. Similarly we refer to a step $\langle\sigma, \text{write-fix}_w, \sigma'\rangle$ as the *write-fix step* of a write operation ω invoked by w. We use the notation $\sigma_{fix(\pi)}$, to capture the final state of a read or write fix step (i.e., σ' in the previous examples) for an operation π. Finally, for an operation π, $\sigma_{inv(\pi)}$ and $\sigma_{res(\pi)}$ denote the system state before the invocation and after the response of operation π respectively (as presented in Section 3.2). The timestamp $\sigma_{res(\pi)}[p].ts$ denotes the value of the variable ts of the automaton A_p at the response step of operation π. This is the timestamp returned if π is a read operation.

Given this notation, the value of the maximum timestamp observed during a read operation ρ from a reader r is $\sigma_{fix(\rho)}[r].maxTS$. As a shorthand we use $maxTS_\rho = \sigma_{fix(\rho)}[r].maxTS$ to denote the maximum timestamp witnessed by ρ.

We adopt the definition of Atomicity as presented in Section 4.2.4 to express the atomicity properties using timestamps in the SWMR environment. The first lemma ensures that any process in the system maintains only positive and monotonically increasing timestamps. Hence, once some process p sets its $\sigma[p].ts$ variable to a value k at a state σ of an execution ξ, then it

cannot be the case that p sets its timestamp to a value $\ell \leq k$ at a state σ' such that σ' appears after σ in ξ.

Lemma 5.4.2 In any execution $\xi \in goodexecs(\text{SLIQ}, \mathbb{Q})$, $\sigma[p].ts \leq \sigma'[p].ts$ for some process $p \in \mathcal{I}$, if σ appears before σ' in ξ.

Proof. It is easy to see the monotonic increment of the timestamps in all the processes.

Writer: For every write operation ω from the sole writer w, it holds that $\sigma_{res(\omega)}[w].ts = \sigma_{inv(\omega)}[w].ts + 1$ as ts_w is modified only in the $\mathsf{write}(val)_w$ event of the SLIQ_w (see Figure 14). Thus, ts_w is incremented monotonically.

Server: A server process s modifies the value of its variable during a $\mathsf{rcv}(m)_{p,s}$ event, if the timestamp enclosed in the received message is greater than the local ts_s variable of SLIQ_s. Thus, when the $\mathsf{rcv}(m)_{p,s}$ event happens for an operation π the server replies in the $\mathsf{send}(m')_{s,p}$ event with a timestamp $m'.ts \geq m.ts$.

Reader: A reader process r modifies its ts_r variable in the $\mathsf{read\text{-}phase1\text{-}fix}_r$ event of a read operation ρ and can either take the value of the maximum timestamp it witnesses, $\sigma_{res(\rho)}[r].ts = maxTS_\rho$, or $\sigma_{res(\rho)}[r].ts = maxTS_\rho - 1$. To prove incremental monotonicity of the ts_r variable of the automaton SLIQ_r we need to show that $\sigma_{res(\rho)}[r].ts \geq \sigma_{inv(\rho)}[r].ts$. In other words we need to show that the timestamp decided by the read operation is geater or equal to the ts_r variable at the invocation of ρ. There exists 3 cases to investigate: (1) $\sigma_{inv(\rho)}[r].ts < maxTS_\rho$, (2) $\sigma_{inv(\rho)}[r].ts = maxTS_\rho$ and, (3) $\sigma_{inv(\rho)}[r].ts > maxTS_\rho$.

Consider the first case, where $\sigma_{inv(\rho)}[r].ts < maxTS_\rho$. Since $\sigma_{res(\rho)}[r].ts$ equals $maxTS_\rho$ or $maxTS_\rho - 1$, then in both cases $\sigma_{res(\rho)}[r].ts \geq \sigma_{inv(\rho)}[r].ts$.

In the algorithm every message sent by the $\mathsf{send}(m)_{r,s}$ event of ρ, includes a timestamp $m(\rho,1)_{r,s}.ts = \sigma_{inv(\rho)}[r].maxTS$. Let $\rho' \rightarrow \rho$ be the last read operation invoked by r before ρ. By the $\mathsf{read\text{-}phase1\text{-}fix}_r$ of the SLIQ_r, it holds that $\sigma_{inv(\rho)}[r].maxTS = maxTS_{\rho'}$. Also, $\sigma_{inv(\rho)}[r].ts = \sigma_{fix(\rho')}[r].ts$ since $\mathsf{read\text{-}phase1\text{-}fix}_r$ is the last event that modifies ts_r in ρ'. Since, $\sigma_{fix(\rho')}[r].ts = maxTS_{\rho'}$ or $\sigma_{fix(\rho')}[r].ts = maxTS_{\rho'} - 1$, it follows that $\sigma_{inv(\rho)}[r].ts \leq \sigma_{inv(\rho)}[r].maxTS$. As shown earlier, any server s that receives a message from r for ρ, replies with a timestamp $m(\rho,1)_{s,r}.ts \geq m(\rho,1)_{r,s}.ts$ and thus $m(\rho,1)_{s,r}.ts \geq \sigma_{inv(\rho)}[r].maxTS$. Thus, the third case cannot arise since every message received by any read operation contains a timestamp greater or equal to the $\sigma_{inv(\rho)}[r].maxTS \geq \sigma_{inv(\rho)}[r].ts$ variable.

So it remains to examine the second case where $\sigma_{inv(\rho)}[r].ts = maxTS_{\rho}$. Observe that the case is possible only if $\sigma_{inv(\rho)}[r].ts = \sigma_{inv(\rho)}[r].maxTS$ in the $\mathsf{send}(m)_{r,s}$ event of ρ. Thus, r sends a message $m(\rho,1)_{r,s}.ts = \sigma_{inv(\rho)}[r].ts$ to every server $s \in \mathcal{S}$. Since $maxTS_{\rho} = \sigma_{inv(\rho)}[r].ts$, every s replies to ρ with $m(\rho,1)_{s,r}.ts = \sigma_{inv(\rho)}[r].ts$. Thus, all the members of the quorum from which ρ receives messages, reply with $maxTS_{\rho}$ and thus the read operation observes the quorum view $\mathbf{QV1}$. According to that view the read operation decides $\sigma_{res(\rho)}[r].ts = maxTS_{\rho}$ and therefore $\sigma_{res(\rho)}[r].ts = \sigma_{inv(\rho)}[r].ts$. This completes the proof. □

The following lemma shows that every read operation returns a timestamp greater or equal to the timestamp written by its last preceding write operation.

Lemma 5.4.3 In any execution $\xi \in goodexecs(\text{SLIQ}, \mathbb{Q})$, if the read_r event of a read operation ρ from reader r succeeds the $\mathsf{write\text{-}fix}_w$ event of a write operation ω in ξ then, $\sigma_{res(\rho)}[r].ts \geq \sigma_{res(\omega)}[w].ts$.

Proof. A write operation ω proceeds to a response step and the $\mathsf{write\text{-}ack}_w$ event only if the writer receives replies from the members of a complete quorum. Let assume that every server in the quorum $Q_i \in \mathbb{Q}$ receives the messages for the write operation ω. According to Lemma 5.4.2 the ts_s variable of every server automaton SLIQ_s, for $s \in Q_i$, will be greater or equal to $\sigma_{res(\omega)}[w].ts$. It follows that any message $m(\pi, 1)_{s,p}$ sent by any server $s \in Q_i$ to any succeeding operation π from p, contains a timestamp $m(\pi, 1)_{s,p}.ts \geq \sigma_{res(\omega)}[w].ts$.

Suppose now that the read operation ρ from r receives replies from the members of a quorum $Q_j \in \mathbb{Q}$, not necessarily different from Q_i. By Lemma 5.4.2 every server $s' \in Q_j \cap Q_i$ replies to ρ with a timestamp $m(\rho, 1)_{s',r}.ts \geq m(\omega, 1)_{s',w}.ts \geq \sigma_{res(\omega)}[w].ts$. It follows that, in the $\mathsf{read\text{-}phase1\text{-}fix}_r$ event of ρ, the maximum timestamp witnessed is $maxTS_\rho \geq m(\rho, 1)_{s',r}.ts \geq \sigma_{res(\omega)}[w].ts$. According to the same event of ρ, $\sigma_{res(\rho)}[r].ts = maxTS_\rho$ or $\sigma_{res(\rho)}[r].ts = maxTS_\rho - 1$. If $maxTS_\rho > \sigma_{res(\omega)}[w].ts$ then $\sigma_{res(\rho)}[r].ts \geq \sigma_{res(\omega)}[w].ts$.

Let us now assume that $maxTS_\rho = \sigma_{res(\omega)}[w].ts$. Since, every server $s' \in Q_i \cap Q_j$ replies to ρ with $m(\rho, 1)_{s',r}.ts \leq maxTS_\rho$ and $m(\rho, 1)_{s',r}.ts \geq m(\omega, 1)_{s',w}.ts \geq \sigma_{res(\omega)}[w].ts$, then $m(\rho, 1)_{s',r}.ts = maxTS_\rho = \sigma_{res(\omega)}[w].ts$. So during the $\mathsf{read\text{-}phase1\text{-}fix}_r$ event of ρ, there are two cases to examine: (1) $Q_i = Q_j$ and (2) $Q_i \neq Q_j$. If case 1 is true then $Q_i \cap Q_j = Q_i$ and thus the read operation observes the quorum view **QV1** and returns $\sigma_{res(\rho)}[r].ts = maxTS_\rho$ in one communication round. If case 2 is valid and $Q_i \neq Q_j$, then the read operation observes the quorum view **QV3** since all the members of the intersection $Q_i \cap Q_j$ reply with

the maximum timestamp. In this case the read also returns the maximum timestamp after it performs a second communication round. Thus in both cases $\sigma_{res(\rho)}[r].ts = maxTS_\rho = \sigma_{res(\omega)}[w].ts$ and that completes our proof. $\square$

The final lemma examines the consistency between two read operations. We show that a read operation always returns a greater or equal timestamp than the one returned by its preceding read operations.

Lemma 5.4.4 In any execution $\xi \in goodexecs(\text{SLIQ}, \mathbb{Q})$, if ρ and ρ' are two read operations from the readers r and r' respectively, such that $\rho \rightarrow \rho'$ in ξ, then $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(r)}[\rho].ts$.

Proof. Since $\rho \rightarrow \rho'$ in ξ, then the read-ack$(val)_r$ event of ρ occurs before the read$'_r$ event of ρ'. Let us consider that both read operations are invoked from the same reader $r = r'$. It follows from Lemma 5.4.2 that $\sigma_{res(\rho)}[r].ts \leq \sigma_{res(\rho')}[r].ts$ because the ts_r variable is incrementing monotonically. So it remains to investigate what happens when the two read operations are invoked by two different processes, r and r' respectively. Suppose that every server s in a quorum Q_i receives the messages of operation ρ with an event rcv$(m)_{r,s}$, and replies with a timestamp $m(\rho, 1)_{s,r}.ts$ with an event send$(m)_{s,r}$ to r. Notice that every server replies, by Lemma 5.4.2, with $m(\rho, 1)_{s',r}.ts \geq \sigma_{inv(\rho)}[r].maxTS$. Let the members of the quorum Q_j (not necessarily different than Q_i) receive messages and reply to ρ'. Again for every $s' \in Q_j$, $m(\rho', 1)_{s',r'}.ts \geq \sigma_{inv(\rho')}[r'].maxTS$. We know that the timestamp of the read operation ρ after the read-phase1-fix$_r$ event of ρ may take the value $\sigma_{res(\rho)}[r].ts = maxTS_\rho$ or $\sigma_{res(\rho)}[r].ts = maxTS_\rho - 1$. We examine those two cases separately and for each case we show that $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$.

Case 1: Consider the case where $\sigma_{res(\rho)}[r].ts = maxTS_{\rho} - 1$. Since some server $s \in Q_i$ replies with a value $m(\rho, 1)_{s,r}.ts = maxTS_{\rho}$ and since we assume a single writer then it follows that a write operation ω' with a timestamp $\sigma_{res(\omega')}[w].ts = maxTS_{\rho}$ is invoked by the writer. So the write-fix$_w$ event of the write operation ω, whose $\sigma_{res(\omega)}[w].ts = \sigma_{res(\rho)}[r].ts = maxTS_{\rho} - 1$, occurs before the read-phase1-fix$_r$ event of ρ. Since the read$'_r$ event of ρ' occurs after the read-phase1-fix$_r$ of ρ, then it also occurs after the write-fix$_w$ event of ω. Hence, by Lemma 4.2.7, $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\omega)}[w].ts$ and thus $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$.

Case 2: Here we examine the case where $\sigma_{res(\rho)}[r].ts = maxTS_{\rho}$. We know by definition that in any quorum construction $Q_j \cap Q_i \neq \emptyset$. Moreover, by Lemma 5.4.2 any server $s \in Q_j \cap Q_i$, s replies with a timestamp $m(\rho, 1)_{s,r}.ts$ for ρ and with a timestamp $m(\rho', 1)_{s,r'}.ts \geq m(\rho, 1)_{s,r}.ts$ for ρ'. So the maximum timestamp witnessed by ρ' is

$$maxTS_{\rho'} \geq m(\rho', 1)_{s,r'}.ts \geq m(\rho, 1)_{s,r}.ts, \forall s \in Q_j \cap Q_i \tag{4}$$

Since $\sigma_{res(\rho)}[r].ts = maxTS_{\rho}$ it means that ρ either observes a quorum view $\mathbf{QV1}$ or a quorum view $\mathbf{QV3}$. Let us examine the two cases separately.

Case 2a: In this case ρ witnessed a $\mathbf{QV1}$. Therefore it must be the case that $\forall s \in Q_i$, s replies with $m(\rho, 1)_{s,r}.ts = maxTS_{\rho} = \sigma_{res(\rho)}[r].ts$. Thus $\forall s \in Q_i \cap Q_j$, s replies with a timestamp $m(\rho', 1)_{s,r'}.ts \geq m(\rho, 1)_{s,r}.ts$ to ρ', and hence, ρ' witnesses a maximum timestamp

$$maxTS_{\rho'} \geq maxTS_{\rho} \Rightarrow maxTS_{\rho'} \geq \sigma_{res(\rho)}[r].ts \tag{5}$$

Recall that ρ' returns either $\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'}$ or $\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'} - 1$. If $maxTS_{\rho'} > \sigma_{res(\rho)}[r].ts$ then it follows that $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$. If $maxTS_{\rho'} = \sigma_{res(\rho)}[r].ts$ then r' witnesses a $\mathbf{QV3}$ since there exists at least one intersection $(Q_i \cap Q_j)$ such

that $\forall s \in Q_i \cap Q_j, m(\rho', 1)_{s,r'}.ts = maxTS_{\rho'}$. Hence in this case

$$\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'} \Rightarrow \sigma_{res(\rho')}[r'].ts = \sigma_{res(\rho)}[r].ts$$

By this we show that, if ρ witnesses a $\mathbf{QV1}$, then $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$.

Case 2b: This is the case where $\sigma_{res(\rho)}[r].ts = maxTS_{\rho}$, because r witnessed a quorum view $\mathbf{QV3}$. Hence it follows that $\exists Q_z \in \mathbb{Q}$ s.t. $\forall s \in Q_z \cap Q_i, m(\rho, 1)_{s,r}.ts = maxTS_{\rho}$. In this case ρ proceeds in phase 2 before completing. Since $\rho \to \rho'$ then $inv(\rho')$ happens after the read-ack$_r$ in ξ. That means that $inv(\rho')$ happens after the read-phase2-fix$_r$ action of ρ as well. However ρ proceeds to phase 2 only after the read-phase1-fix$_r$. From the latter action we get that $\sigma_{fix(\rho)}[r].maxTS = maxTS_{\rho}$. Once in phase 2, ρ sends inform messages with its $\sigma_{fix(\rho)}[r].maxTS = maxTS_{\rho}$ to a complete quorum, say Q_k. By Lemma 5.4.2, every server $s' \in Q_k$ replies with a timestamp

$$m(\rho, 2)_{s',r}.ts \geq maxTS_{\rho} \tag{6}$$

There are two subcases to consider: (i) $Q_k = Q_j$ and (ii) $Q_j \neq Q_k$.

Case 2b(i): Assume that $Q_k = Q_j$. Then $\forall s \in Q_j$, s replies to ρ' with a timestamp $m(\rho', 1)_{s,r'}.ts \geq m(\rho, 2)_{s,r}.ts$ (by Lemma 5.4.2). Therefore it follows that

$$maxTS_{\rho'} \geq m(\rho, 2)_{s,r}.ts \Rightarrow maxTS_{\rho'} \geq maxTS_{\rho} \tag{7}$$

from equation (6). If $maxTS_{\rho'} > maxTS_{\rho}$ then $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$ since $\sigma_{res(\rho)}[r].ts = maxTS_{\rho}$ and $\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'}$ or $\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'} - 1$. If $maxTS_{\rho'} = maxTS_{\rho}$ then $\forall s \in Q_j, m(\rho', 1)_{s,r'}.ts = maxTS_{\rho'}$ and thus ρ' observes $\mathbf{QV1}$ and returns $\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'} = maxTS_{\rho} = \sigma_{res(\rho)}[r].ts$.

Case 2b(ii): It remains to examine what happens if $Q_k \neq Q_j$. As in case 2b(i), $\forall s \in Q_k \cap Q_j$, s replies to ρ' with $m(\rho', 1)_{s,r'}.ts \geq m(\rho, 1)_{s,r}.ts$. It follows from equations (6) and (7) that $maxTS_{\rho'} \geq maxTS_{\rho}$. As shown in case 2b(i), if $maxTS_{\rho'} > maxTS_{\rho}$ then $\sigma_{res(\rho')}[r'].ts \geq \sigma_{res(\rho)}[r].ts$.

If now $maxTS_{\rho'} = maxTS_{\rho}$, then ρ' observes every server $s \in Q_k \cap Q_j$ to reply with $m(\rho', 1)_{s,r'}.ts = maxTS_{\rho'}$. But this is exactly the definition of $\mathbf{QV3}$. So, ρ' proceeds to a second communication round (phase 2) and returns a timestamp $\sigma_{res(\rho')}[r'].ts = maxTS_{\rho'} = maxTS_{\rho} = \sigma_{res(\rho)}[r].ts$. □

Using the above lemmas we can obtain the main result of this section:

Theorem 5.4.5 Algorithm SLIQ implements a weak-semifast SWMR atomic read/write register.

Proof. The atomicity requirement of the theorem follows from Lemmas 5.4.2-5.4.4. We now argue that SLIQ belongs in the class of weak-semifast implementations. By the construction of SLIQ, writes are fast and reads require at most two communication rounds satisfying properties **S1, S2** of Definition 4.1.1. To see that SLIQ also satisfies **S4** assume the following execution: (i) a write operation ω sends messages to all the servers, (ii) the servers in a quorum Q_i receive the messages and reply to ω, and (iii) any read that returns $\sigma_{res(\omega)}[w].ts$ strictly contacts Q_i after the servers in Q_i replied to ω. Observe that every such read operation witnesses $\mathbf{QV1}$ if $\sigma_{res(\omega)}[w].ts$ is the maximum, and thus is fast. If a bigger timestamp $\sigma_{res(\omega')}[w].ts$ is observed then a read returns $\sigma_{res(\omega)}[w].ts$ only if $\sigma_{res(\omega)}[w].ts = \sigma_{res(\omega')}[w].ts - 1$. In such a case the read returns the previous timestamp and by construction this is done in a single round. Therefore, all the reads that return $\sigma_{res(\omega)}[w].ts$ are fast and property **S4** of Definition 4.1.1 is

satisfied. Note that the messages from the servers of Q_i for ω may be in-transit and thus all the reads may be concurrent with the write operation. This completes the proof. $\square$

5.4.5 Empirical Evaluation of SLIQ

To practically evaluate our findings, we simulate our algorithm using the the NS-2 network simulator. We use the same test environment as in Section 4.2.5.2 in order to be able to extract meaningful comparison results between the performance of the two algorithms. In particular, our test environment consists of one writer, a variable set of reader and server processes. We use bidirectional links between the communicating nodes, 1Mb bandwidth, a latency of $10ms$, and a DropTail queue. To model asynchrony, the processes send messages after a random delay between 0 and 0.3 sec. According to our setting, only the messages from the invoking processes to the servers, and the replies from the servers to the processes are delivered (no messages are exchanged among the servers or the invoking processes).

We evaluate our approach over three types of quorum systems: majorities ($\mathbb{Q}_m$), matrix quorums ($\mathbb{Q}_x$), and crumbling walls ($\mathbb{Q}_c$). (A description of these quorum systems can be found in [81].) In this section we present some of the plots we obtained exploiting crumbling walls; all plots depicting the results of all the experiments we have conducted appear in the Appendix A. The quorum system is generated apriori and is distributed to each participant node via an external service (out of the scope of this work). No dynamic quorums are assumed, so the configuration of the quorum system remains the same throughout the execution of the simulation. We model server failures by choosing the non-faulty quorum and allowing any server that is not a member of that quorum to fail by crashing. Note that the non-faulty quorum

is not known to any of the participants. The positive time parameter $cInt$ is used, to model the failure frequency or reliability of every server s.

We use the positive time parameters $rInt$ and $wInt$ (both greater than 1 sec) to model the time intervals between any two successive read operations and any two successive write operations respectively. We consider three simulation scenarios corresponding to the following parameters: (i) $rInt < wInt$: this models frequent reads and infrequent writes, (ii) $rInt = wInt$: this models evenly spaced reads and writes, (iii) $rInt > wInt$: this models infrequent reads and frequent writes.

Furthermore for each one of the above scenarios we consider two settings:

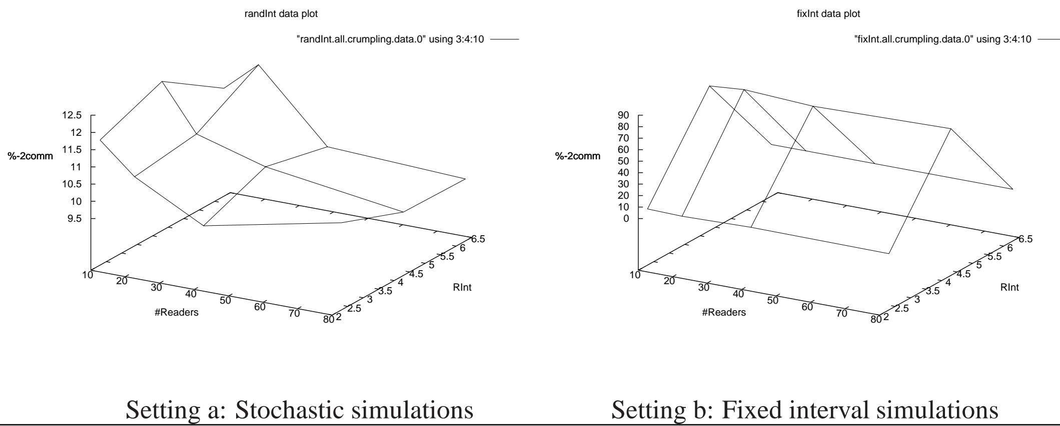
(a) *Stochastic setting*: the read/write intervals vary randomly within $[0 \ldots rInt]$ and $[0 \ldots wInt]$ respectively.

(b) *Fixed setting*: the read/write intervals are fixed to the value of $rInt$ and $wInt$ respectively.

We can summarize our simulations testbed for each class of quorums and for the settings presented above, as follows:

(1) **Simple Runs:** ($\mathbb{Q}_c,\ \mathbb{Q}_x,\ \mathbb{Q}_m$) $|\mathcal{S}| = 25$ ($\mathbb{Q}_c,\ \mathbb{Q}_x$) or $|\mathcal{S}| = 10$ ($\mathbb{Q}_m$), $cInt = 0$ (failure check for every reply) and $|\mathcal{R}| \in [10, 20, 40, 80]$. Here we want to demonstrate the performance of the algorithm under similar environments (quorum,failures) but with different read load.

(2) **Quorum Diversity Runs:** ($\mathbb{Q}_c,\ \mathbb{Q}_x$) $|\mathcal{S}| \in [11, 25, 49]$ ($\mathbb{Q}_c$) and $|\mathcal{S}| \in [11, 25, 49]$ ($\mathbb{Q}_x$), $cInt = 0$ and $|\mathcal{R}| \in [10, 20, 40, 80]$. These runs demonstrate the performance of the



Setting a: Stochastic simulations Setting b: Fixed interval simulations

Figure 17: Simple runs using Crumbling Walls

algorithm in different quorum systems with varying quorum membership. Each quorum is tested in variable read load.

(3) **Failure Diversity Runs:** **(**$\mathbb{Q}_c,\ \mathbb{Q}_x$**)** $|\mathcal{S}| = 25$, $cInt \in [10 \dots 50]$ with steps of 10 and $|\mathcal{R}| \in [10, 20, 40, 80]$. These runs test the durability of the algorithm to failures. Notice that the smaller the crash interval the faster we diverge to the non-faulty quorum. As the crash interval becomes bigger, less servers fail and thus more quorums "survive" in the quorum system. For this class of runs we test both the cases when the servers get the crash interval randomly from $[0 \dots cInt]$ and $[10 \dots 10 + cInt]$.

Figure 17 illustrates the results obtained when we assume simple runs and exploiting crumbling walls quorum. The Z axis presents the percentage of the read operations that perform two communication rounds, the X axis corresponds to the number of reader participants and the Y axis represents time and in particular the $rInt$ interval. In the stochastic environment (Figure 17.a) we observe that the percentage of slow reads drops as the number of readers increases,

regardless of the value of $rInt$. This behavior can be explained from the fact that the concurrency between the operations is minimized and thus the maximum timestamp is propagated (by both the writer and the readers) to enough servers that favor the fast behavior. Since the convergence point is similar regardless the number of readers, then increasing the readers, increases the number of fast reads and decreases the percentage of slow reads. Similar behavior is observed in the fixed interval environment (Figure 17.b) whenever there is no strict concurrency between the reads and the writes. The worst case is observed at the point where all operations are invoked concurrently.

We conduct similar experiments for the rest of the cases and our results appear in the Appendix. Our results (including the ones given in the Appendix A) suggest that in realistic cases (i.e. stochastic settings), the percentage of two communication round reads does not exceed 13%. The only case that requires more than 85% of the reads to be slow is the worst case scenario were the read and write intervals are fixed to the same value. Notice however that this scenario is unlikely to appear in practical settings. Comparing our results with the ones obtained in Section 4.2.5.2, one can observe that the difference in the random scenarios does not exceed 6%.

Chapter 6

Trade-offs for Multiple Writers

So far we have examined the fastness of implementations that allow unbounded number of readers and utilize general quorum constructions. In this chapter we investigate the existence of implementations with executions that contain fast operations when multiple writers participate in the service. First, we present the unique characteristics of multiple-writer (MW) over single-writer (SW) environments (Section 6.1). Next, we provide generic limitations that the MW environment imposes on any R/W atomic register implementation (Section 6.2). Then, we present two separate algorithms that allow fast read and/or write operations in the MWMR environment (Sections 6.3 and 6.4). We first generalize the idea of *Quorum Views* (see Section 5.4.1), that allows the introduction of fast read operations in the MWMR environment. To improve write operation latency, we then introduced a new technique, called *Server Side Ordering*, that allows both read and write operations to complete in a single round. Our algorithms are the *first* to allow single round read and write operations in the MWMR environment without making any assumptions on synchrony or the precedence relation of read/write operations.

6.1 Introducing Fastness in MWMR model

One of the main challenges in atomic R/W register implementations is to provide a *total* ordering among the write operations.

In algorithms designed for the SWMR setting the sole writer may order locally the write operations. This enables the introduction of single round write operations: the writer determines locally the order of a write operation, and propagates that ordering with the new value to the replica hosts.

The existence of multiple writers prevents writers from ordering a write operation locally. To overcome this problem, implementations presented in the MWMR setting [6, 22, 28, 36, 49, 34, 68, 66] suggested that every write operation should "learn" about the latest value written (and thus latest write operation), before propagating a new value to the register. For this purpose, a two round write protocol was proposed where the first round queries the replica hosts for the latest value of the register. Following these works, a belief was shaped that "writes must read" before writing a new value to the register in multi writer environments.

Dutta et al. [30] explored the possibility of *fast* implementations in the MWMR environment. They showed that such implementations are impossible assuming two readers, two writers and a single server failure exist in the system. In Chapter 4 we showed that MWMR are not possible in *semifast* implementations either.

Consequently, MWMR implementations are impossible if *all* the write operations are fast or at most a single complete slow read per write is allowed. Considering that traditional algorithms for the MWMR environment require two round write operations, devising algorithms that allow *any* fast write operations are interesting in their own right.

This stage of the thesis presents the *first* algorithms that allow fast write and/or read operations in the MWMR environment. We show that one of the algorithms developed is near optimal with respect to the number of fast operations allowed in a MWMR environment. We proceed by presenting the implications and inherent limitations that the use of multiple writers imposes in the system. Then we present algorithm CWFR that adopts and generalizes the idea of quorum views in the multiple writer environment. The algorithm is the first to allow some fast read operations when those are invoked concurrently with write operations. The drawback of CWFR is the adoption of the two round write operations. To overcome this problem we introduce a new technique, called *Server Side Ordering*, and we develop algorithm SFW. The new algorithm is the first to allow *both* fast write and read operations.

6.2 Inherent Limitations of the MWMR Environment

In this Section we investigate the implications and restrictions that the MW setting may impose on any execution of a R/W atomic register implementation. We study general n-wise quorum constructions and we rely on the following definitions on any two R/W operations:

Definition 6.2.1 (Consecutive Operations) Two operations π, π' are **consecutive** in an execution ξ if: (i) they are invoked from processes p and p', s.t. $p \neq p'$, (ii) they complete in ξ, and (iii) $\pi \rightarrow \pi'$ or $\pi' \rightarrow \pi$ (they are not concurrent).

Definition 6.2.2 (Quorum Shifting Operations) Two operations π and π' that contact quorums $Q', Q'' \in \mathbb{Q}$ respectively, are called **quorum shifting** if π and π' are consecutive and $Q' \neq Q''$.

Definition 6.2.3 (Quorum Shifting Set) A set of operations Π is called **quorum shifting** if $\forall \pi, \pi' \in \Pi$, π and π' are quorum shifting operations.

Recall that we seek write operations that complete in a single round. It is thus necessary for a write operation to propagate and write its indented value during its first and only round. Since we assume server failures, the writer may complete before communicating with all servers. Moreover, by well-formedness (Definition 3.2.2), each process invokes a single operation at a time. Thus, in the SW setting the invocation of a write operation from the sole writer implies the completion of any previous write operation. This is not the case for the MW setting as multiple writers may invoke write operations concurrently. Following this observation we show that a read operation may retrieve the latest written value (and thus write operation) only from the servers that receive messages from *all* the write operations that preceded that read. This is captured by the following lemma:

Lemma 6.2.4 Let ξ be an execution of an atomic read/write register implementation A, and Π be a set of consecutive write operations in ξ. If ρ a read operation in ξ s.t. $\omega \rightarrow \rho$ for every $\omega \in \Pi$, then ρ receives the latest value val if it communicates with a a server s s.t. $\mathsf{rcv}(m)_{w,s}$ appears in ξ for all write operations $\omega \in \Pi$.

Proof. The proof follows from the fact that a server is not aware of a written value val unless: 1) it receives messages from the writer that propagates value val, or 2) it receives messages from a process that already observed value val in the system. Moreover a server may infer the latest value at time t in any execution if: 1) it receives messages from all the write operations invoked by time $t' < t$ (and thus contains all the values written), or 2) it received a message

that contained the value history at time $t' < t$ and received messages from all write operations thereafter.

It is easy to see that a server may not be aware of the latest value even if Π is a set of consecutive write operations. Assume, to derive contradiction that a server may return the latest value to a read operation even if it does not receive messages from all the write operations. Let us consider a server s at time t of an execution ξ. Suppose that s received all the messages from every write operation invoked by time $t' < t$. Also, suppose w.l.o.g. that the latest value that s received was val. Since the write operations are consecutive, then val is the latest written value in the system. We extend now ξ by a write operation ω that writes value val'. Let the resulting execution be ξ'. Assume that s does not receive messages from ω in ξ'. Thus, s cannot distinguish ξ from ξ'. Hence, it replies with a latest value val to any read operation. Since however the write operations are consecutive (and thus, totally ordered), the latest value in the system is val'. Thus, contradiction. $\square$

Given this finding and an *n-wise* quorum system we show that it is *possible* to obtain *safe register* implementations if any execution contains $n-1$ consecutive, quorum shifting fast write operations. We use $\omega(val)$ to denote the write operation that writes value val. Also, recall that a *safe register* constitutes the weakest consistency guarantee and is defined [62] as property **SR1**: *Any read operation that is not concurrent to any write operation returns the value written by the last preceding write operation.*

Lemma 6.2.5 Any execution fragment ϕ of a safe register implementation that uses an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$, contains at most $n-1$ consecutive, quorum shifting, fast write operations for any number of writers $|\mathcal{W}| \geq 2$.

Proof. Let $\mathbb{Q}$ be some *n-wise* quorum system, where $2 \leq n < |\mathbb{Q}|$ its intersection degree. We provide a series of execution constructions that depend on the intersection degree n. If $n = 2$ then ξ_0 is the execution that consists of a single ($n - 1 = 1$) complete fast write operation $\omega(val_1)$ invoked by w_1 which $scnt(\omega(val_1), Q_1)_{w_1}$. If $n = 3$ then we extend ξ_0 by a complete fast write operation, $\omega(val_2)$, from w_2 with $scnt(\omega(val_2), Q_2)_{w_2}$, to obtain execution ξ_1.

In general, if $n = i+2$, we construct execution ξ_i by extending execution ξ_{i-1} with a complete fast write operation, $\omega(i+1)$, from $w_{(i \mod 2)+1}$ with $scnt(\omega(i+1), Q_{i+1})_{w_{(i \mod 2)+1}}$. By this construction any execution ξ_i contains $i + 1$ (or $n - 1$) consecutive, quorum shifting fast write operations.

We proceed by induction on the intersection degree n, to show that extending any of the above executions with a read operation ρ from reader r preserves property **SR1**. In other words the read operation is able to discern the latest write operation and return its value.

Induction base: We assume that $n = 2$ and hence, pairwise intersection between the quorums of $\mathbb{Q}$. In this case we extend execution ξ_0 by a read operation ρ from r to obtain the following execution ξ_0':

a) a complete fast write operation $\omega(val_1)$ by w_1 which $scnt(\omega(val_1), Q_1)_{w_1}$, and

b) a complete read operation ρ by r with $scnt(\rho, Q_j)_r$.

It is easy to see that the read operation ρ, for any $Q_j \in \mathbb{Q}$, observes the value val_1 written by $\omega(val_1)$ in $Q_1 \cap Q_j$ ($\neq \emptyset$). Since $\omega(val_1)$ is the only write operation then ρ will return the value written by $\omega(val_1)$ and preserve property **SR1**.

Inductive hypothesis: Assume that $n = k+2$ and that extending execution construction ξ_k with a read operation ρ preserves property **SR1**. It follows that ρ returns the value written by the last proceeding write operation which in ξ_k is $\omega(val_{k+1})$ that $scnt(\omega(val_{k+1}), Q_{k+1})_{w_{(k \mod 2)+1}}$.

Induction step: We now investigate the case where $\mathbb{Q}$ is a $(k+3)$-*wise* quorum system. We need to verify if execution ξ_{k+1} preserves property **SR1**. Recall that ξ_{k+1} is constructed by extending ξ_k with a fast complete write operation $\omega(val_{k+2})$. We further extend ξ_k by a read operation ρ by r to obtain ξ'_{k+1}. The last three operations of ξ'_{k+1} are the following:

a) a complete fast write operation $\omega(val_{k+1})$ by $w_{(k \mod 2)+1}$ that $scnt(\omega(val_{k+1}), Q_{k+1})_{w_{(k \mod 2)+1}}$

b) a complete fast write operation $\omega(val_{k+2})$ by $w_{(k+1 \mod 2)+1}$ that $scnt(\omega(val_{k+2}), Q_{k+2})_{w_{(k+1 \mod 2)+1}}$, and

c) a complete read operation ρ by r that $scnt(\rho, Q_j)_r$.

By the inductive hypothesis we know that the execution fragment of ξ_k preserves property **SR1**. Furthermore any $k+3$-*wise* quorum system is also a $k+2$-*wise* quorum system. So, it follows that if ξ_k is extended by a read operation then that read operation returns, by induction hypothesis, val_{k+1} or val_{k+2}. If val_{k+1} is returned then it follows that ρ cannot distinguish ξ'_{k+1} from ξ'_k and hence does not observe val_{k+2} and violates property **SR1**. Since $\mathbb{Q}$ is also a $k+2$-*wise* system then it must be true that under $\mathbb{Q}$, $\bigcap_{i=1}^{k+2} Q_i \neq \emptyset$. Hence, the two writers w_1 and w_2 and the servers in $\bigcap_{i=1}^{k+2} Q_i \neq \emptyset$ can distinguish between ξ'_k and ξ'_{k+1} since those are the only servers that receive messages from all write operations. From the quorum construction however we know that $\mathbb{Q}$ has an intersection degree of $k+3$ and thus $\bigcap_{i=1}^{k+3} Q_i \neq \emptyset$. So, for any quorum Q_j that replies to ρ it must hold that $\left(\bigcap_{i=1}^{k+2} Q_i\right) \cap Q_j \neq \emptyset$. Thus, $\exists s \in \left(\bigcap_{i=1}^{k+2} Q_i\right) \cap Q_j$ such that s receives messages from every write operation and s replies to ρ. Hence, ρ also distinguishes ξ'_{k+1} from ξ'_k and returns val_{k+2} preserving property **SR1**. $\square$

We now show that safe register implementations are *not possible* if we extend any execution that contains $n-1$ consecutive writes, with one more consecutive, quorum shifting write

operation. It suffices to assume a very basic system consisting of two writers w_1 and w_2, and one reader r. Thus, our results hold for at least two writers.

Theorem 6.2.6 No execution fragment ϕ of a safe register implementation that uses an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$, can contain more than $n-1$ consecutive, quorum shifting, fast write operations for any number of writers $|\mathcal{W}| \geq 2$.

Proof. Let $\mathbb{Q}$ be an *n-wise* quorum system, where $2 \leq n < |\mathbb{Q}|$ it intersection degree. From Lemma 6.2.5 we obtain that an implementation exploiting an *n-wise* quorum system may contain $n-1$ consecutive, quorum shifting fast write operations and still preserve property **SR1**. Thesis of this proof follows from the contradiction, where we assume that an implementation can include n consecutive fast writes and still satisfy property **SR1**.

Let $\mathbb{Q}$ be an $(k+2)$-*wise* system and let ξ_k be an execution of the safe register implementation that exploits $\mathbb{Q}$. Suppose the execution follows the construction in Lemma 6.2.5. Thus, ξ_k contains $k+1$ consecutive, quorum shifting, fast writes. Moreover by the induction we know that ξ_k satisfies **SR1** if extended by a read operation. Let us now extend ξ_k with a write $\omega(val_{k+2})$ from writer $w_{(k+1 \mod 2)+1}$ which $scnt(\omega(val_{k+2}), Q_{k+2})_{w_{(k+1 \mod 2)+1}}$, and a read operation ρ from r with $scnt(\rho, Q_j)_r$. Notice that since $n < |\mathbb{Q}|$ then $k+2 < |\mathbb{Q}|$ and thus there exists a quorum $Q \in \mathbb{Q}$ such that $\left(\bigcap_{i=1}^{k+2} Q_i\right) \cap Q = \emptyset$. Let $Q_j \in \mathbb{Q}$ be such quorum and w.l.o.g let us assume that $Q_j = Q_{k+3}$. We denote the obtained execution by $\Delta(\xi_k)$. Below we can see the last three operations in the execution sequence of $\Delta(\xi_k)$:

a) a complete fast write operation $\omega(val_{k+1})$ by $w_{(k \mod 2)+1}$ which $scnt(\omega_{val_{k+1}}, Q_{k+1})_{w_{(k \mod 2)+1}}$,

b) a complete fast write operation $\omega(val_{k+2})$ by $w_{(k+1 \mod 2)+1}$ which $scnt(\omega(val_{k+2}), Q_{k+2})_{w_{(k+1 \mod 2)+1}}$,

c) and a complete read operation ρ by r with $scnt(\rho, Q_{k+3})_r$.

Notice that by the above construction reader r has to return val_{k+2} to preserve property **SR1**. Since we assumed a $(k+2)$-wise quorum then, according to Lemma 6.2.4, ρ observes the value val_{k+1} as the latest written value from the servers in $\left(\bigcap_{i=1}^{k} Q_i\right) \cap Q_{k+1} \cap Q_{k+3}$ and the value val_{k+2} as the latest written value from the servers in $\left(\bigcap_{i=1}^{k} Q_i\right) \cap Q_{k+2} \cap Q_{k+3}$. We should note here that the servers in both sets receive messages from all write operations $\{\omega(val_1), \ldots, \omega(val_k)\}$. The servers in the first set however receive messages from $\omega(val_{k+1})$ but not from $\omega(val_{k+2})$ and vice versa.

Consider now the execution fragment $\Delta(\xi_k')$ where the two write operations are switched. More precisely we obtain ξ_k' by extending ξ_{k-1} with the write operation $\omega(val_{k+2})$ by $w_{(k+2 \mod 2)+1}$. Then, we obtain $\Delta(\xi_k')$ by extending ξ_k' with the write operation $\omega(val_{k+1})$ by $w_{(k+1 \mod 2)+1}$, and the read operation ρ from r. In more detail, the last three operations that appear, and the quorums they contact are as follows:

a) a complete fast write operation $\omega(val_{k+2})$ by $w_{(k+1 \mod 2)+1}$ which $scnt(\omega(val_{k+2}), Q_k + 2)_{w(k+1 \mod 2)+1}$,

b) a complete fast write operation $\omega(val_{k+1})$ by $w_{(k \mod 2)+1}$ which $scnt(\omega(val_{k+1}), Q_{k+1})_{w_{(k \mod 2)+1}}$,

c) and a complete read operation ρ by ρ with $scnt(\rho, Q_{k+3})_\rho$.

Observe that executions $\Delta(\xi_k)$ and $\Delta(\xi_k')$ differ only at the writers and the servers in $\bigcap_{i=1}^{k+2} Q_i$. Any other server and the reader cannot distinguish between the two executions. In particular, the reader does not receive any messages from any server in $\bigcap_{i=1}^{k+2} Q_i$, since $\left(\bigcap_{i=1}^{k+2} Q_i\right) \cap$

$Q_{k+3} = \emptyset$. Moreover, the reader observes the values val_{k+1} and val_{k+2} as the latest values from the servers in $\left(\bigcap_{i=1}^{k} Q_i\right) \cap Q_{k+1} \cap Q_{k+3}$ and $\left(\bigcap_{i=1}^{k} Q_i\right) \cap Q_{k+2} \cap Q_{k+3}$ respectively. Since those are the same servers that replied with the same values to ρ in $\Delta(\xi_k)$ then r cannot distinguish $\Delta(\xi'_k)$ from $\Delta(\xi_k)$ and thus, has to return val_{k+2} in $\Delta(\xi'_k)$ as well. This however violates property **SR1** since in $\Delta(\xi'_k)$ the two write operations are consecutive and the latest completed write operation is $\omega(val_{k+1})$. Hence the read operation had to return val_{k+1} in $\Delta(\xi'_k)$ to preserve property **SR1**, contradicting our findings. $\square$

Note that Theorem 6.2.6 also applies to both regular and atomic R/W register implementations, as safety needs to be satisfied by both regular and atomic semantics [62]. The theorem is exempt in two cases: (i) a single writer exists in the system, and (ii) there is a common intersection among all the quorums in the quorum system. In the first case the sole writer imposes the ordering of the written values and in the second case that ordering is imposed by the servers of the common intersection that are accessed by every operation.

An immediate implication derived from Theorem 6.2.6 is the impossibility of having more than $n - 1$ *concurrent* fast write operations. Since no communication between the writers is assumed and since achieving agreement on the set of concurrent writes is impossible (as shown in [38]), that led us to the following corollary:

Corollary 6.2.7 No MWMR implementation of a safe register, that exploits an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$ and contains only fast writes is possible, if $|\mathcal{W}| > n - 1$.

Moreover assuming that readers also may alter the value of the register replica, and thus write, then the following theorem holds:

Theorem 6.2.8 No MWMR implementation of a safe register, that exploits an n-wise quorum system $\mathbb{Q}$ s.t. $2 \leq n < |\mathbb{Q}|$ and contains only fast operations is possible, if $|\mathcal{W} \cup \mathcal{R}| > n - 1$.

This theorem shows that fast implementations are possible only if the number of reader and writer participants is bounded with respect to the intersection degree of the quorum system that the algorithm uses. If readers do not modify the value of the register then the theorem applies on the number of writer participants. To our knowledge, Theorem 6.2.8 is the first to provide a general result on the relation of fast implementations and the construction of the underlying quorum system.

We demonstrate that the result holds for algorithms that use voting techniques to construct the underlying quorum system. Let us assume a model identical to the one used in [30]. By their algorithm each operation was waiting for $|\mathcal{S}| - f$ replica hosts to reply. Such voting strategy implied a quorum system that contains quorums of size $|\mathcal{S}| - f$, and in extend implied an $(\frac{|\mathcal{S}|}{f} - 1)$-wise quorum system as depicted by the following lemma:

Lemma 6.2.9 The intersection degree of a quorum system $\mathbb{Q}$ where $\forall Q \in \mathbb{Q}$, $|Q| = |\mathcal{S}| - f$ is equal to $\frac{|\mathcal{S}|}{f} - 1$.

Proof. Since any $Q \in \mathbb{Q}$, $|Q| = |\mathcal{S}| - f$ then for $Q, Q' \in \mathbb{Q}$ it follows that $|Q \cap Q'| \geq |\mathcal{S}| - 2f$. For three quorums $Q, Q', Q'' \in \mathbb{Q}$ it then follows that $|Q \cap Q' \cap Q''| \geq |\mathcal{S}| - 3f$. Generalizing for n quorums we get:

$$\left| \bigcap_{i=0}^{n} Q_i \right| \geq |\mathcal{S}| - nf$$

Since we want to find the biggest n such that the intersection is not empty, then it should be the case that

$$\bigcap_{i=0}^{n} Q_i \neq \emptyset \Rightarrow |\bigcap_{i=0}^{n} Q_i| > 0$$

So, it follows that in the worst case $|\mathcal{S}| - nf > 0$ and thus $n \geq \frac{|\mathcal{S}|}{f} - 1$. And that completes the proof. □

Note that by Lemma 6.2.9 and Theorem 6.2.8, the system in [30] could only accommodate:

$$|\mathcal{W} \cup \mathcal{R}| \leq (\frac{|\mathcal{S}|}{f} - 1) - 1 \Rightarrow 1 + |\mathcal{W} \cup \mathcal{R}| \leq \frac{|\mathcal{S}|}{f} - 2$$

Since only a single writer exist in their system, then it follows that $|\mathcal{R}| + 1 \leq (\frac{S}{f} - 2)$ and hence, $|\mathcal{R}| < (\frac{S}{f} - 2)$ which is the bound derived in [30]. This leads us to the following remark.

Remark 6.2.10 Fast implementations, such as the one presented in [30], follow our proved restrictions on the number of participants in the service.

6.3 Enabling Fast Read Operations - Algorithm CwFr

We explored the possibility to introduce fast operations in the MWMR environment by exploiting techniques presented in the SWMR environment. The developments of [28, 22], made an effort to introduce fast read operations in the MWMR environment, but their techniques did not convince that such fast behavior is possible under read and write concurrency.

In this section we introduce a new algorithm, we call CwFr, which enables fast read operations by adopting the general idea of Quorum Views (Section 5.4.1). The algorithm employs two techniques:

(i) the classic query and propagate technique (two round) for write operations, and

(ii) analysis of Quorum Views for potentially fast (single round) read operations.

Read operations can be fast in CwFr even when they are invoked concurrently with one or multiple write operations. This distinguishes CwFr from previous approaches. To impose a

total ordering on the written values, CWFR exploits $\langle tag, value\rangle$ pairs as also used in prior papers (e.g., [22, 28, 66]). A tag is a tuple of the form $\langle ts, w\rangle \in \mathbb{N} \times \mathcal{W}$, where ts is the timestamp and w is a writer identifier. Two tags are ordered lexicographically, first by the timestamp, and then by the writer identifier.

6.3.1 Incorporating Prior Techniques – Quorum Views

To comply with the ordering scheme of CWFR we revised the definition of quorum views as presented in Section 5.4.1, to examine tags instead of timestamps. The revised definition is the following:

Definition 6.3.1 Let process p, receive replies from every server s in some quorum $Q \in \mathbb{Q}$ for a read or write operation π. Let a reply from s include a tag $m(\pi, c)_{s,p}.tag$ and let $maxTag = \max_{s \in Q}(m(\pi, c)_{s,p}.tag)$. We say that p observes one of the following **quorum views** for Q:

$\mathbf{QV1}$: $\forall s \in Q : m(\pi, c)_{s,p}.tag = maxTag$,

$\mathbf{QV2}$: $\forall Q' \in \mathbb{Q} :\ Q \neq Q' \wedge \exists A \subseteq Q \cap Q'$, s.t. $A \neq \emptyset$ and $\forall s \in A : m(\pi, c)_{s,p}.tag < maxTag$,

$\mathbf{QV3}$: $\exists s' \in Q : m(\pi, c)_{s',p}.tag < maxTag$ and $\exists Q' \in \mathbb{Q}$ s.t. $Q \neq Q' \wedge \forall s \in Q \cap Q' :$ $m(\pi, c)_{s,p}.tag = maxTag$

With similar reasoning as presented in Section 5.4.1, $\mathbf{QV1}$ implies the potential completion of the write operation that wrote a value associated with $maxTag$. $\mathbf{QV2}$ imposes its non-completion and $\mathbf{QV3}$ does not reveal any information about the write completion.

6.3.2 High Level Description of CwFr

The original quorum views algorithm as presented in Section 5.4 relies on the fact that a single writer is participating in the system. If a quorum view is able to predict the non-completeness of the latest write operation, is immediately understood that – by well-formedness of the single writer – any previous write operation is already completed. Multiple writer participants in the system prohibit such assumption: different values (or tags) may be written concurrently. Hence, the discovery of a write operation that propagates some tag does not imply the completion of the write operations that propagate a smaller tag. So, algorithm CwFr incorporates an iterative examination of quorum views that not only predicts the completion status of a write operation, but also detects the last potentially completed write operation. Below we provide a high level description of our algorithm and present the main idea behind our technique.

Writers. The write protocol has two rounds. During the first round the writer discovers the maximum tag among the servers: it sends read messages to all servers and waits for replies from all the members of a single quorum. It then discovers the maximum tag among the replies and generates a new tag in which it encloses the incremented timestamp of the maximum tag, and the writer's identifier. In the second round, the writer associates the value to be written with the new tag, it propagates the pair to a complete quorum, and completes the write.

Readers. The read protocol is more involved. When a reader invokes a read operation, it sends a read message to all servers and waits for some quorum to reply. Once a quorum replies, the reader determines the $maxTag$. Then the reader analyzes the distribution of the tag within the responding quorum Q in an attempt to determine the latest, potentially complete, write

operation. Detecting that the tag distribution satisfies $\mathbf{QV1}$ and $\mathbf{QV3}$ is straightforward. When $\mathbf{QV1}$ is detected, the read completes and the value associated with the discovered $maxTag$ is returned. In the case of $\mathbf{QV3}$ the reader continues into the second round, advertising the latest tag ($maxTag$) and its associated value. When a full quorum replies to the second round, the read returns the value associated with $maxTag$.

Detection of $\mathbf{QV2}$ involves discovery of the latest potentially completed write operation. This is done iteratively by (locally) removing the servers from Q that replied with the largest tags. After each iteration the reader determines the next largest tag in the remaining server set, and re-examines the quorum views on the distribution of the tag in the remaining servers. This process eventually leads to either $\mathbf{QV1}$ or $\mathbf{QV3}$ being observed. If $\mathbf{QV1}$ is observed, then the read completes in a single round by returning the value associated with the maximum tag among the servers that *remain* in Q. If $\mathbf{QV3}$ is observed, then the reader proceeds to the second round as above, and upon completion it returns the value associated with the maximum tag $maxTag$ discovered among the original respondents in Q.

Servers. The servers play a passive role. They receive read or write requests, update their object replica accordingly, and reply to the process that invoked the request. Upon receipt of any message, the server compares its local tag with the tag included in the message. If the tag of the message is higher than its local tag, the server adopts the higher tag along with its corresponding value. Once this is done the server replies to the invoking process.

6.3.3 Formal Specification of CwFr

We now present the formal specification of CwFr using Input/Output Automata [67] notation. Our implementation includes four automata: (i) automaton CwFr_w that handles the

write operations for each writer $w \in \mathcal{W}$, (ii) automaton CwFr_r that handles the reading for each $r \in \mathcal{R}$, (iii) automaton CwFr_s that handles the read and write requests on the atomic register for each $s \in \mathcal{S}$, and (iv) $Channel_{p,p'}$ that establish the reliable asynchronous process-to-process communication channels (see Section 3.1.2).

Automaton CwFr_w.

The state variables, the signature and the transitions of the CwFr_w can be depicted in Figure 18. The state of the CwFr_w automaton includes the following variables:

- $\langle\langle ts, wid\rangle, v\rangle \in \mathbb{N} \times \mathcal{W} \times V$: writer's local tag along with the latest value written by the writer. The tag is composed of a timestamp and the identifier of the writer.
- $vp \in V$: this variable is used to hold the previous value written.
- $maxTS \in \mathbb{N}$: the maximum timestamp discovered during the last write operation.
- $wCounter \in \mathbb{N}$: the number of write requests performed by the writer. Is used by the servers to distinguish fresh from stale messages.
- $phase \in \{1, 2\}$: indicates the active communication round of the write operation.
- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the writer received all the necessary replies to complete its write operation and is ready to respond to the client.
- $srvAck \subseteq \mathcal{S}$: a set that contains the servers that reply to the write messages as a result of a write request. The set is reinitialized to $\emptyset$ at the response step of every write operation.

Signature:

Input:
$\mathsf{write}(val)_w,\ val \in V, w \in \mathcal{W}$
$\mathsf{rcv}(m)_{s,w},\ m \in M,\ s \in \mathcal{S}, w \in \mathcal{W}$
$\mathsf{fail}_w, w \in \mathcal{W}$

Output:
$\mathsf{send}(m)_{w,s},\ m \in M,\ s \in \mathcal{S}, w \in \mathcal{W}$
$\mathsf{write\text{-}ack}_w, w \in \mathcal{W}$

Internal:
$\mathsf{write\text{-}phase1\text{-}fix}_w, w \in \mathcal{W}$
$\mathsf{write\text{-}phase2\text{-}fix}_w, w \in \mathcal{W}$

State:

$tag = \langle ts, w\rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, w\}$
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$maxTS \in \mathbb{N}$, initially 0
$wCounter \in \mathbb{N}^+$, initially 0
$phase \in \{1, 2\}$, initially 1
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{write}(val)_w$
Effect:
if $\neg failed$ then
if $status = idle$ then
$status \leftarrow active$
$srvAck \leftarrow \emptyset$
$phase \leftarrow 1$
$vp \leftarrow v$
$v \leftarrow val$
$wCounter \leftarrow wCounter + 1$

Input $\mathsf{rcv}(\langle msgT, t, C\rangle)_{s,w}$
Effect:
if $\neg failed$ then
if $status = active$ and $wCounter = C$ then
if $(phase = 1 \wedge msgT =$ READ-ACK $)\vee$
$(phase = 2 \wedge msgT =$ WRITE-ACK $)$ then
$srvAck \leftarrow srvAck \cup \{s, \langle msgT, t, C\rangle\}$

Output $\mathsf{send}(\langle msgT, t, C\rangle)_{w,s}$
Precondition:
$status = active$
$\neg failed$
$\big[(phase = 1 \wedge \langle msgT, t, C\rangle =$
$\langle$READ$, \langle tag, vp\rangle, wCounter\rangle)\vee$
$(phase = 2 \wedge \langle msgT, t, C\rangle =$
$\langle$WRITE$, \langle tag, v\rangle, wCounter\rangle)\big]$
Effect:
none

Output $\mathsf{write\text{-}ack}_w$
Precondition:
$status = done$
$\neg failed$
Effect:
$status \leftarrow idle$

Internal $\mathsf{write\text{-}phase1\text{-}fix}_w$
Precondition:
$\neg failed$
$status = active$
$phase = 1$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
Effect:
$maxTS \leftarrow \max_{s \in Q \wedge (s,m) \in srvAck}(m.t.tag.ts)$
$tag = \langle maxTs + 1, w\rangle$
$phase \leftarrow 2$
$srvAck \leftarrow \emptyset$
$wCounter \leftarrow wCounter + 1$

Internal $\mathsf{write\text{-}phase2\text{-}fix}_w$
Precondition:
$\neg failed$
$status = active$
$phase = 2$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
Effect:
$status \leftarrow done$

Input fail_w
Effect:
$failed \leftarrow true$

Figure 18: CWFR$_w$ Automaton: Signature, State and Transitions

- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

The automaton completes a write operation in two phases. A write operation ω is invoked when the write$(val)_w$ request is received from the automaton's environment. The $status$ variable becomes $active$, the previous value vp gets the current value and the variable v gets the requested value val to be written. As long as the $status = active$ and $phase = 1$ the automaton sends messages to all server processes and collects the identifiers of the servers that reply to those messages in the $srvAck$ set. To avoid adding any delayed message from a previous phase, the writer examines the type of the acknowledgment and the message counter. The action write-phase1-fix occurs when the replies from the members of a full quorum are received by the writer, i.e., $\exists Q \in \mathbb{Q} : Q \subseteq srvAck$. In the same action the writer discovers the maximum timestamp $maxTS$ among the replies and generates the new tag. In particular, it assigns $tag = \langle maxTS + 1, w \rangle$. Once the new tag is generated, the writer changes the $phase$ variable to 2, to indicate the start of its second round, and reinitializes the $srvAck$ to accept the replies to its new round. When a full quorum replies to w, the $status$ of the automaton becomes $done$. This change, and assuming that the writer does not fail, enables the write-ack$_w$. Finally, when the action write-ack$_w$ occurs, the writer responds to the environment and the $status$ variable becomes $idle$.

Automaton CWFR$_\rho$**.**

The state variables, the signature and the transitions of the CWFR$_r$ can be depicted in Figures 19 and 20. The state of the CWFR$_r$ automaton includes the following variables:

- $\langle\langle ts, wid\rangle, v\rangle \in \mathbb{N} \times \mathcal{W} \times V$: the maximum tag (timestamp and writer identifier pair) discovered during r's last read operation along with its associated value.

- $maxTag \in \mathbb{N} \times \mathcal{W}$, and $retvalue \in V$: the maximum tag discovered and the value that was returned during the last read operation.

- $rCounter \in \mathbb{N}$: read request counter. Used by the servers to distinguish fresh from stale messages.

- $phase \in \{1, 2\}$: indicates the active communication round of the read operation.

- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the reader decided on the value to be returned and is ready to respond to the client.

- $srvAck \subseteq M \times \mathcal{S}$: a set that contains the servers and their replies to the read operation. The set is reinitialized to $\emptyset$ at the response step of every read operation.

- $maxAck \subseteq M \times \mathcal{S}$: this set contains the messages (and the servers senders) that contained the maximum tag during r's last read request.

- $maxTagSrv \subseteq \mathcal{S}$: The servers that replied with the $maxTag$.

- $replyQ \subseteq \mathcal{S}$: The quorum of servers that replied to r during the last read operation.

- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

Signature:

Input:
$\mathsf{read}_r,\ r \in \mathcal{R}$
$\mathsf{rcv}(m)_{s,r},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\mathsf{fail}_r,\ r \in \mathcal{R}$

Output:
$\mathsf{send}(m)_{r,s},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\mathsf{read\text{-}ack}(val)_r,\ val \in V,\ r \in \mathcal{R}$

Internal:
$\mathsf{read\text{-}phase1\text{-}fix}_r$
$\mathsf{read\text{-}phase2\text{-}fix}_r$

State:

$tag = \langle ts, wid \rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, \min(\mathcal{W})\}$
$maxTag = \langle ts, wid \rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, \min(\mathcal{W})\}$
$v \in V$, initially $\perp$
$retvalue \in V$, initially $\perp$
$phase \in \{1, 2\}$, initially 1
$rCounter \in \mathbb{N}^{+}$, initially 0
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$maxTagSrv \subseteq \mathcal{S}$, initially $\emptyset$
$replyQ \subseteq \mathcal{S}$, initially $\emptyset$
$failed$, a Boolean initially **false**

Figure 19: CWFR$_r$ Automaton: Signature and State

Any read operation requires one or two phases to complete (fast or slow). The decision on the number of communication rounds is based on the quorum views that the reader establishes during its first communication round.

The reader r invokes a read operation when the CWFR$_r$ automaton receives a read_r request from its environment. The $status$ of the automaton becomes $active$ and the reader sends messages to each server $s \in \mathcal{S}$ to obtain the value of the atomic register. The $\mathsf{rcv}(m)_{s,r}$ action is triggered when reader r receives a reply from server s. The reader collects the identifiers of the servers and their replies by adding a pair (s, m) in the $srvAck$ set. When r receives messages from a single quorum Q it detects the the maximum tag ($maxTag$) among the messages received from the servers in Q. Those messages are placed in $maxTagAck$ set. The servers that belong into the collected quorum and have messages in $maxTagAck$, are placed separately in the $maxTagSrv$ set. Lastly, the $replyQ$ variable becomes equal to the quorum Q and the value v becomes equal to the value assigned to $maxTag$.

From the newly formed sets the reader iteratively analyzes the distribution of the maximum tag on the members of $replyQ$, in an attempt to determine the latest write operation that has

Transitions:

Input read_r
Effect:
if $\neg failed$ then
if $status = idle$ then
$status \leftarrow active$
$rCounter \leftarrow rCounter + 1$

Input $\mathsf{rcv}(\langle msgT, t, C\rangle)_{s,r}$
Effect:
if $\neg failed$ then
if $status = active$ and $rCounter = C$ then
$srvAck \leftarrow srvAck \cup \{(s, \langle msgT, t, C\rangle)\}$

Output $\mathsf{send}(\langle msgT, t, C\rangle)_{r,s}$
Precondition:
$status = active$
$\neg failed$
$\big[(phase = 1 \wedge \langle msgT, t, C\rangle = \langle \text{READ}, \langle maxTag, v\rangle, rCounter\rangle) \vee$
$(phase = 2 \wedge \langle msgT, t, C\rangle = \langle \text{INFORM}, \langle maxTag, v\rangle, rCounter\rangle)\big]$
Effect:
none

Output $\mathsf{read\text{-}ack}(val)_r$
Precondition:
$\neg failed$
$status = done$
$val_{=} retvalue$
Effect:
$replyQ \leftarrow \emptyset$
$srvAck \leftarrow \emptyset$
$status \leftarrow idle$

Internal $\mathsf{read\text{-}phase2\text{-}fix}_r$
Precondition:
$\neg failed$
$status = active$
$phase = 2$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
Effect:
$status \leftarrow done$
$phase \leftarrow 1$

Internal $\mathsf{read\text{-}phase1\text{-}fix}_r$
Precondition:
$\neg failed$
$status = active$
$phase = 1$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
Effect:
$replyQ \leftarrow Q$
$maxTag \leftarrow \max_{s \in replyQ \wedge (s,m) \in srvAck}(m.t.tag)$
$maxAck \leftarrow \{(s, m) : (s, m) \in srvAck \wedge m.t.tag = maxTag\}$
$maxTagSrv \leftarrow \{s : s \in replyQ \wedge (s, m) \in maxAck\}$
$v \leftarrow \{m.t.val : (s, m) \in maxAck\}$

Internal $\mathsf{read\text{-}qview\text{-}eval}_r$
Precondition:
$\neg failed$
$replyQ \neq \emptyset$
Effect:
$tag \leftarrow \max_{s \in replyQ \wedge (s,m) \in srvAck}(m.t.tag)$
$maxAck \leftarrow \{(s, m) : (s, m) \in srvAck \wedge m.t.tag = maxTag\}$
$maxTagSrv \leftarrow \{s : s \in replyQ \wedge (s, m) \in maxAck\}$
$retvalue \leftarrow \{m.t.val : (s, m) \in maxAck\}$
if $replyQ = maxTagSrv$ then
$status \leftarrow done$
else
if $\exists Q' \in \mathbb{Q}, Q' \neq replyQ$ s.t. $replyQ \cap Q' \subseteq maxTagSrv$ then
$tag \leftarrow maxTag$
$retvalue \leftarrow v$
$phase \leftarrow 2$
$srvAck \leftarrow \emptyset$
$rCounter \leftarrow rCounter + 1$
else
$replyQ \leftarrow replyQ - \{s : s \in maxTagSrv\}$

Input fail_r
Effect:
$failed \leftarrow true$

Figure 20: CWFR$_r$ Automaton: Transitions

potentially completed. This is done by the read-qview-eval$_r$ action. In particular, the iterative approach works as follows. Let $maxTag_\ell$ denote the maximum tag in $replyQ$ at iteration ℓ, with $maxTag_0 = maxTag$. Also let $replyQ_\ell$ be the set of servers that the read operation examines during iteration ℓ, with $replyQ_0 = replyQ = Q$. During every iteration ℓ, the reader r proceeds as follows (locally) depending on the quorum view it observes during ρ in $replyQ_\ell$:

Set $\ell = 0$ and $replyQ_0 = replyQ$

Repeat until return:

$\mathbf{QV1}$: Return the value associated with $maxTag_\ell = \max_{s\in replyQ_\ell}(m(\rho)_{s,r}.t.tag)$

$\mathbf{QV3}$: Proceed to a second round, and propagate messages that contain $maxTag_0 = maxTag$ to all servers. Once the read-phase2-fix$_r$ event occurs, return the value associated with $maxTag$.

$\mathbf{QV2}$: Set $replyQ_{\ell+1} = replyQ_\ell - \{s : (s \in replyQ_\ell) \wedge (m(\rho)_{s,r}.t.tag = maxTag_\ell)\}$ and proceed to iteration $\ell + 1$.

Let us discuss the idea behind our proposed technique. Observe that under our failure model, any write operation can expect a response from at least one full quorum. Moreover a write ω distributes its tag tag_ω to some quorum, say Q', before completing.Thus when a read operation ρ, s.t. $\omega \rightarrow \rho$, receive replies from some quorum Q, then it will observe one of the following tag distributions: (a) if $Q = Q'$, then $\forall s \in Q, m(\rho)_{s,r} = tag_\omega$ ($\mathbf{QV1}$), or (b) if $Q \neq Q'$, then $\forall s \in Q \cap Q', m(\rho)_{s,r} = tag_\omega$ ($\mathbf{QV3}$). Hence, if ρ observes a distribution as in $\mathbf{QV1}$ then it follows that a write operation completed and received replies from the same

quorum that replied to ρ. Alternatively, if only an intersection contains a uniform tag (i.e., the case of $\mathbf{QV3}$) then there is a possibility that some write completed in an intersecting quorum (in this example Q'). The read operation is fast in $\mathbf{QV1}$ since it is determinable that the write potentially completed. The read proceeds to the second round in $\mathbf{QV3}$, since the completion of the write is indeterminable and it is necessary to ensure that any subsequent operation observes that tag. If none of the previous quorum views hold (and thus $\mathbf{QV2}$ holds), then it must be the case that the write that yielded the maximum tag is not yet completed. Hence we try to discover the latest potentially completed write by removing all the servers with the highest tag from Q and repeating the analysis. If at some iteration, $\mathbf{QV1}$ holds on the remaining tag values, then a potentially completed write – that was overwritten by greater values in the rest of the servers – is discovered and that tag is returned (in a single round). If no iteration is interrupted because of $\mathbf{QV1}$, then eventually $\mathbf{QV3}$ is observed in the worst case, when a single server will remain in some intersection of Q. Since a second round cannot be avoided in this case, we take the opportunity to propagate the largest tag observed in Q. At the end of the second round that tag is written to at least a single complete quorum and thus the reader can safely return it.

Automaton $\textsc{CwFr}_s$**.**

The server automaton has relatively simple actions. The signature, state and transitions of the $\textsc{CwFr}_s$ can be depicted in Figure 21. The state of the $\textsc{CwFr}_s$ contains the following variables:

- $\langle\langle ts, wid\rangle, v\rangle \in \mathbb{N} \times \mathcal{W} \times V$: the maximum tag (timestamp, writer identifier pair) reported to s along with its associated value. This is the value of the register replica of s.

Signature:

Input:
$\mathsf{rcv}(m)_{p,s},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \mathcal{W}$
fail_s

Output:
$\mathsf{send}(m)_{s,p},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \mathcal{W}$

State:

$tag = \langle ts, wid \rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, \min(\mathcal{W})\}$
$v \in V$, initially $\perp$
$Counter(p) \in \mathbb{N}^{+}$, $p \in \mathcal{R} \cup \mathcal{W}$, initially 0
$msgType \in \{$WRITEACK,READACK,INFOACK$\}$
$status \in \{idle, active\}$, initially $idle$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{rcv}(\langle msgT, t, C \rangle)_{p,s}$
Effect:
if $\neg failed$ then
if $status = idle$ and $C > Counter(p)$ then
$status \leftarrow active$
$Counter(p) \leftarrow C$
if $tag < t.tag$ then
$(tag.ts, tag.wid, v) \leftarrow (t.tag.ts, t.tag.wid, t.val)$

Output $\mathsf{send}(\langle msgT, t, C \rangle)_{s,p}$
Precondition:
$\neg failed$
$status = active$
$p \in \mathcal{R} \cup \mathcal{W}$
$\langle msgT, t, C \rangle =$
$\langle msgType, \langle tag, v \rangle, Counter(p) \rangle$
Effect:
$status \leftarrow idle$

Input fail_s
Effect:
$failed \leftarrow true$

Figure 21: CWFR$_s$ Automaton: Signature, State and Transitions

- $Counter(p) \in \mathbb{N}$: this array maintains the latest request index of each client (reader or writer). It helps s to distinguish fresh from stale messages.
- $status \in \{idle, active\}$: specifies whether the automaton is processing a request received ($status = active$) or it can accept new requests ($status = idle$).
- $msgType \in \{$WRITEACK,READACK,INFOACK$\}$: Type of the acknowledgment depending on the type of the received message.
- $failed \in \{true, false\}$: indicates whether the server associated with the automaton has failed.

Each server replies to a message without waiting to receive any other messages from any process. Thus, the status of the server automaton determines whether the server is busy processing a message ($status = active$) or if it is able to accept new messages ($status = idle$). When a new message arrives, the $\mathsf{rcv}(m)_{p,s}$ event is responsible to process the incoming message. If the $status$ is equal to idle and this is a fresh message from process p then the $status$ becomes active. The $Counter(p)$ for the specific process becomes equal to the counter included in the message. Then the server checks if $m(\pi, *)_{p,s}.t.tag > tag_s$. The comparison is validated if either:

- the timestamp of the received tag is greater than the timestamp in the local tag of the server (i.e., $m(\pi, *)_{p,s}.t.tag.ts > tag_s.ts$), or
- $m(\pi, *)_{p,s}.t.tag.ts = tag_s.ts$ and the writer identifier included in the tag of the received message is greater than the writer identified included in the local tag of the server (i.e., $m(\pi, *)_{p,s}.t.tag.wid > tag_s.wid$).

If any of the above cases hold, the server updates its tag and v variables to be equal to the ones included in the received message. The type of the received message specifies the type of the acknowledgment.

While the server is active, the $\mathsf{send}(m)_{s,p}$ event may be triggered. When this event occurs, the server s sends its local replica value, to the process p. The action results in modifying the $status$ variable to $idle$ and thus setting the server enable to receive new messages.

6.3.4 Correctness of CWFR

We show that algorithm CWFR, satisfies both termination and atomicity properties presented in Definitions 3.2.4 and 3.2.5 respectively.

Termination

Each phase of any read or write operation terminates when the invoking process receives replies from at least a single quorum. According to our failure model, all but one quorums may fail (see Section 3.1.4). Thus, any correct process receives replies from at least the correct quorum. Thus, every operation from a correct process eventually terminates and hence, Definition 3.2.4 is satisfied.

Atomicity

We now show that algorithm CWFR satisfies the properties of Definition 3.2.5. We adopt the notation presented in Chapter 3. In particular, we use var_p to refer to the variable var of the automaton CWFR$_p$. To access the value of a variable var of CWFR$_p$ in a state σ of an execution ξ, we use $\sigma[p].var$ (see Section 3.1.1). Also, let $m(\pi, c)_{p,p'}$ to denote the message sent from p to p' during the cth round of operation π. Any variable var enclosed in a message is denoted by $m(\pi, c)_{p,p'}.var$ (see Section 3.1.2). We refer to a step $\langle\sigma, \text{read-qview-eval}_r, \sigma'\rangle$, where $\sigma'[r].status = done$ or $\sigma'[r].phase = 2$, as the ***read-fix step*** of a read operation ρ invoked by reader r. Similarly we refer to a step $\langle\sigma, \text{write-phase2-fix}_w, \sigma'\rangle$ as the ***write-fix step*** of a write operation ω invoked by w. We use the notation $\sigma_{fix(\pi)}$, to capture the final state of a read or write fix step (i.e., σ' in the previous examples) for an operation π. Finally, for

an operation π, $\sigma_{inv(\pi)}$ and $\sigma_{res(\pi)}$ denote the system state before the invocation and after the response of operation π respectively (as presented in Section 3.2).

Given this notation, the value of the maximum tag observed during a read operation ρ from a reader r is $\sigma_{fix(\rho)}[r].maxTag$. As a shorthand we use $maxTag_{\rho} = \sigma_{fix(\rho)}[r].maxTag$ to denote the maximum tag witnessed by ρ. Similarly, we use $minTag_{\rho}$ to denote the minimum tag witnessed by ρ. For a write operation we use $maxTag_{\omega} = \sigma_{rfix(\omega)}[w].maxTag$ to denote the maximum tag witnessed during the read phase. The state $\sigma_{rfix(\omega)}$ is the state of the system after the write-phase1-fix$_w$ event occurs during operation ω. Note that $\sigma_{res(\pi)}[p].tag$ is the tag returned if π is a read operation. Lastly given tag' and a set of servers Q that replied to some operation π from p, let $(Q)^{>tag'} = \{s : s \in Q \wedge m(\pi)_{s,p}.tag > tag'\}$ be the set of servers in Q that replied with a tag greater than tag'.

Similar to Section 4.2.4, we can express the ordering of read/write operations with respect to the tags they return/write. For the operation ordering to satisfy the atomicity conditions of Definition 3.2.5, the tags written and returned must satisfy the following properties for every finite or infinite execution ξ of CwFr:

TG1. For each process p the tag_p variable is alphanumerically monotonically nondecreasing and it contains a non-negative timestamp.

TG2. If the read$_r$ event of a read operation ρ from reader r succeeds the write-fix step of a write operation ω in ξ then, $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$.

TG3. If ω and ω' are two write operations from the writers w and w' respectively, such that $\omega \to \omega'$ in ξ, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

TG4. If ρ and ρ' are two read operations from the readers r and r' respectively, such that $\rho \to \rho'$ in ξ, then $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.

First we need to ensure that any process in the system maintains only monotonically non-decreasing tags. In other words, if some process p sets its tag_p variable to a value k at a state σ in an execution ξ, then $tag_p \neq \ell$ such that $\ell \leq k$ at a state σ' that appears after σ in ξ.

Lemma 6.3.2 In any execution $\xi \in goodexecs(\text{CwFr}, \mathbb{Q})$, $\sigma'[s].tag \geq \sigma[s].tag$ for any server $s \in \mathcal{S}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ.

Proof. It is easy to see that a server s modifies its tag variable when the step $\langle \sigma, \mathsf{rcv}(m)_{p,s}, \sigma' \rangle$. From that step, $\sigma[s].tag \neq \sigma'[s].tag$ only if s receives a message during $\mathsf{rcv}_{p,s}$ such that $m(\pi,1)_{p,s}.tag > \sigma[s].tag$. This means that either: a) $m(\pi,1)_{p,s}.tag.ts > \sigma[s].tag.ts$ or b) $m(\pi,1)_{p,s}.tag.ts = \sigma[s].tag.ts$ and $m(\pi,1)_{p,s}.tag.wid > \sigma[s].tag.wid$. So, if $\sigma[s].tag \neq \sigma'[s].tag$, then $\sigma[s].tag < \sigma'[s].tag$ and the tag is monotonically incrementing. Furthermore, since the initial tag of the server is set to $\langle 0, \min(wid) \rangle$ and the tag is updated only if $m(\pi,1)_{p,s}.tag.ts \geq \sigma[s].tag.ts$, then for any state σ'', it holds that $\sigma''[s].tag.ts$ is always greater than 0. □

We can also show that a server replies with a higher tag than the one it receives in a requesting message.

Lemma 6.3.3 In any execution $\xi \in goodexecs(\text{CwFr}, \mathbb{Q})$, if a server s receives a message $m(\pi,1)_{p,s}$ from a process p, for operation π, then s replies to p with $m(\pi,1)_{s,p}.tag \geq m(\pi,1)_{p,s}.tag$.

Proof. When the server receives the message from processor p it first compares $m(\pi,1)_{p,s}.tag$ with its local tag tag_s. If $m(\pi,1)_{p,s}.tag > tag_s$ then the server sets $tag_s = m(\pi,1)_{p,s}.tag$.

From this it follows that the tag of the server at the state σ' after $\mathsf{rcv}_{p,s}$ is $\sigma'[s].tag \geq m(\pi,1)_{p,s}.tag$. Since by Lemma 6.3.2 the tag of the server is monotonically nondecreasing, then when the $\mathsf{send}_{s,p}$ event occurs, the server replies to p with a tag $m(\pi,1)_{s,p}.tag \geq \sigma'[s].tag \geq m(\pi,1)_{p,s}.tag$. Hence, the lemma follows. □

The next lemma shows the monotonicity of tags in every writer process.

Lemma 6.3.4 In any execution $\xi \in goodexecs(\text{CwFr}, \mathbb{Q})$, $\sigma'[w].tag \geq \sigma[w].tag$ for any writer $w \in \mathcal{W}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ. Also, for any state σ in ξ, $\sigma[w].tag.ts \geq 0$.

Proof. Each writer process w modifies its local tag during its first communication round. In particular when the $\mathsf{write\text{-}phase1\text{-}fix}_w$ event happens for a write operation ω, then the tag of the writer becomes equal to $tag_w = \langle maxTag_\omega.ts + 1, w\rangle$. So, it suffice to show that $\sigma_{inv(\omega)}[w].maxTS \leq maxTag_\omega$. Suppose that all the servers of a quorum $Q \in \mathbb{Q}$, received messages and replied to w, for ω. Every message sent from w to any server $s \in Q_j$ (when $\mathsf{send}_{w,s}$ occurs), contains a tag $m(\omega,1)_{w,s}.tag = \sigma_{inv(\omega)}[w].maxTS$. By Lemma 6.3.3, any $s \in Q$ replies with a tag $m(\omega,1)_{s,w}.tag \geq m(\omega,1)_{w,s}.tag \geq \sigma_{inv(\omega)}[w].maxTS$. Thus, $\forall s \in Q$, $m(\omega,1)_{s,w}.tag \geq \sigma_{inv(\omega)}[w].maxTS$ and it follows that $m(\omega,1)_{s,w}.tag.ts \geq \sigma_{inv(\omega)}[w].maxTS.tag.ts$. Since $maxTag_\omega.ts = \max(m(\omega,1)_{s,w}.tag.ts)$ then $maxTag_\omega.ts \geq \sigma_{inv(\omega)}[w].maxTS.tag.ts$ and hence, $\sigma_{res(\omega)}[w].tag = \langle maxTag_\omega.ts+1, w\rangle > \sigma_{inv(\omega)}[w].maxTS$. Therefore not only the tag of a writer is nondecreasing but we show explicitly that the writer's tag is monotonically increasing. Furthermore since the writer adopts the maximum tag sent from the servers, and

since by Lemma 6.3.2 the servers tags contain non-negative timestamps, then it follows that the writer contains non-negative timestamps as well. □

The next lemma shows the monotonicity of the tags in every reader.

Lemma 6.3.5 In any execution $\xi \in goodexecs(\text{CWFR}, \mathbb{Q})$, $\sigma'[r].tag \geq \sigma[r].tag$ for any reader $r \in \mathcal{R}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ. Also, for any state σ in ξ, $\sigma[r].tag.ts \geq 0$.

Proof. Notice that the tag variable of a reader is $\sigma_{inv(\rho)}[r].tag \leq \sigma_{inv(\rho)}[r].maxTS$ when the read_r event occurs. So, it suffices to show that $\sigma_{res(\rho)}[r].tag \geq \sigma_{inv(\rho)}[r].maxTS$. With similar arguments to Lemma 6.3.4 it can be shown that for every $s \in Q$ that replies to an operation ρ invoked by r, $m(\rho,1)_{s,r}.tag \geq \sigma_{inv(\rho)}[r].maxTS$. Since $maxTag_\rho = \max(m(\rho,1)_{s,r}.tag)$ and $minTag_\rho = \min(m(\rho,1)_{s,r}.tag)$ then it follows that both $maxTag_\rho, minTag_\rho \geq \sigma_{inv(\rho)}[r].maxTS$. By the algorithm the tag returned by the read operation is $minTag_\rho \leq \sigma_{res(\rho)}[r].tag \leq maxTag_\rho$. Hence, $\sigma_{res(\rho)}[r].tag \geq \sigma_{inv(\rho)}[r].maxTS$. Thus, no matter which of the tags is chosen to be returned at the end of the read operation nondecreasing monotonicity is preserved. Also since by Lemma 6.3.2 all the servers reply with a non negative timestamp, then it follows that r contains non-negative timestamps as well. □

Lemma 6.3.6 For each process $p \in \mathcal{R} \cup \mathcal{W} \cup \mathcal{S}$ the tag variable is monotonically nondecreasing and contains a non-negative timestamp.

Proof. Follows from Lemmas 6.3.2, 6.3.4 and 6.3.5 □

The following lemma states that if a read operation returns a tag $\tau < maxTag$ it must be the case that any pairwise intersection of the replied quorum contains a server s such that $tag_s \leq \tau$.

Lemma 6.3.7 In any execution $\xi \in goodexecs(\text{CwFr}, \mathbb{Q})$, if a read operation ρ from r receives replies from the members of quorum Q and returns a tag $\sigma_{res(\rho)}[r].tag < maxTag_{\rho}$, then $\forall Q' \in \mathbb{Q}, Q' \neq Q$, $(Q \cap Q') - (Q)^{>\sigma_{res(\rho)}[r].tag} \neq \emptyset$.

Proof. By definition the intersection of two quorums $Q, Q' \in \mathbb{Q}$ is not empty. Let us assume to derive contradiction that a read operation ρ may return a tag $\sigma_{res(\rho)}[r].tag < maxTag_{\rho}$ and may exist $(Q \cap Q') - (Q)^{>\sigma_{res(\rho)}[r].tag} = \emptyset$. According to our algorithm, when read-qview-eval event occurs, we first check if either $\mathbf{QV1}$ or $\mathbf{QV3}$ is observed in Q. If neither of those quorum views is observed then we remove all the servers with the current maximum tag from Q and we repeat the check on the remaining servers. It follows that since all the servers $s' \in Q \cap Q'$ were removed from Q then it must be the case that $m(\rho, 1)_{s',r}.tag > \sigma_{res(\rho)}[r].tag$. So there must be a tag $\tau' > \sigma_{res(\rho)}[r].tag$ s.t. $A = (Q \cap Q') - (Q)^{>\tau'} \neq \emptyset$ and all servers $s' \in A$ replied with $m(\rho, 1)_{s',r}.tag = \tau'$. If this happens there are two cases for the reader:

a) $\forall s' \in (Q) - (Q)^{>\tau'}, m(\rho, 1)_{s',r}.tag = \tau'$ and thus $\mathbf{QV1}$ is observed and the reader returns $\sigma_{res(\rho)}[r].tag' = \tau'$, or

b) $\forall s' \in A, m(\rho, 1)_{s',r}.tag = \tau'$ and thus, $\mathbf{QV3}$ is observed and the reader returns $\sigma_{res(\rho)}[r].tag' = maxTag_{\rho}$.

Since $maxTag_{\rho} \geq \tau'$, then in any case the read operation ρ would return a tag $\sigma_{res(\rho)}[r].tag' > \sigma_{res(\rho)}[r].tag$ and that contradicts our assumption. $\square$

Derived from the above lemma, the next lemma states that a read operation returns either the $maxTag$ or the maximum of the smaller tags from the pairwise intersections of the replied quorum.

Lemma 6.3.8 In any execution $\xi \in goodexecs(\text{CWFR}, \mathbb{Q})$, if a read operation ρ from r receives replies from a quorum Q, then $\forall Q' \in \mathbb{Q}, Q' \neq Q$, $\sigma_{res(\rho)}[r].tag \geq \min(m(\rho,1)_{s,r})$ for $s \in Q \cap Q'$.

Proof. This lemma follows directly from Lemma 6.3.7. Let a subset of servers in $Q \cap Q'$ replied to ρ with the minimum tag among all the servers of that intersection, say τ. If the iteration of the read-eval-qview$_r$ event of ρ reaches tag τ then either ρ observes $\mathbf{QV1}$ and returns $\sigma_{res(\rho)}[r].tag = \tau$ or it observes $\mathbf{QV3}$ and returns $\sigma_{res(\rho)}[r].tag = maxTag_\rho \geq \tau$. This is true for all the intersections $Q \cap Q'$, for $Q \neq Q'$. And the lemma follows. □

Next we show that a read returns a higher tag than the one written by a preceding write.

Lemma 6.3.9 If in an execution $\xi \in goodexecs(\text{CWFR}, \mathbb{Q})$, the invocation step of a read operation ρ from reader r succeeds the write-fix step of a write operation ω from w then, $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$.

Proof. Assume w.l.o.g. that the write operation receives messages from two, not necessarily different, quorums Q and Q' during its first and second communication rounds respectively. Furthermore, let us assume that the read operation receives replies from a quorum Q'', not necessarily different from Q or Q', during its first communication round. According to the algorithm the write operation ω detects the maximum tag from Q, increments that and propagates the new tag to Q'. Since $\forall s \in Q, maxTag_\omega \geq m(\omega,1)_{s,w}.tag$ then from the intersection property of a quorum system it follows that $\forall s' \in (Q \cap Q') \cup (Q \cap Q''), \sigma_{res(\omega)}[w].tag > maxTag_\omega \geq m(\omega,1)_{s',w}.tag$. From the fact that w propagates $\sigma_{res(\omega)}[w].tag$ in ω's second communication round and from Lemma 6.3.3 it follows that every $s \in (Q' \cap Q'')$ replies with a tag $m(\omega,2)_{s,w}.tag \geq \sigma_{res(\omega)}[w].tag$ during the second round of ω.

Since the read_r operation succeeds the write-fix step of ω, then from Lemma 6.3.3 the read operation will obtain a tag $m(\rho,1)_{s,r}.tag \geq m(\omega,2)_{s,w}.tag \geq \sigma_{res(\omega)}[w].tag$, from every server $s \in Q' \cap Q''$. So, $\min(m(\rho,1)_{s,r}.tag) \geq \sigma_{res(\omega)}[w].tag$. Thus from Lemma 6.3.8 $\sigma_{res(\rho)}[r].tag \geq m(\rho,1)_{s,r}$ for $s \in Q' \cap Q''$ and hence $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$ completing the proof. $\square$

The next lemma shows that CWFR satisfies **TG3**.

Lemma 6.3.10 In any execution $\xi \in goodexecs(\text{CWFR}, \mathbb{Q})$, if ω and ω' are two write operations from the writers w and w' respectively, such that $\omega \to \omega'$ in ξ, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega')}[w].tag$

Proof. From the precedence relation of the two write operations it follows that the write-fix step of ω occurs before the $\mathsf{write}_{w'}$ event of ω'. Recall that for a write operation ω, $\sigma_{res(\omega)}[w].tag = \langle maxTag_{\omega}.ts{+}1, w\rangle$. So, it suffices to show here that $maxTag_{\omega'} > maxTag_{\omega}$. This however is straightforward from Lemma 6.3.3 and the value propagated during the second communication round of ω. In particular let ω propagate $\sigma_{res(\omega)}[w].tag > maxTag_{\omega}$ to a quorum Q. Notice that every $s \in Q$ replies with $m(\omega,2)_{s,w}.tag \geq \sigma_{res(\omega)}[w].tag$ to the second communication round of ω. Furthermore, let the write operation ω' receive replies from a quorum Q', not necessarily different than Q, during its first communication round. Since the write-fix step of ω occurs before the $\mathsf{write}_{w'}$ event of ω' then, by Lemmas 6.3.2 and 6.3.3, $\forall s' \in Q \cap Q'$ $m(\omega',1)_{s',w'} \geq m(\omega,2)_{s,w} \geq \sigma_{res(\omega)}[w].tag$. Thus, $maxTag_{\omega'} \geq m(\omega',1)_{s',w'} \geq \sigma_{res(\omega)}[w].tag$ and hence, since $\sigma_{res(\omega)}[w].tag = \langle maxTag_{\omega'}.ts + 1, w'\rangle > maxTag_{\omega'}$, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$. $\square$

The following two lemmas show that the tags returned by read operations in CWFR satisfy property **TG4**.

Lemma 6.3.11 In any execution $\xi \in goodexecs(\text{CWFR}, \mathbb{Q})$, if ρ and ρ' are two read operations from the readers r and r' respectively, such that $\rho \rightarrow \rho'$ in ξ, then $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.

Proof. Since $\rho \rightarrow \rho'$ in ξ, then the read-ack$_r$ event of ρ occurs before the read$_{r'}$ event of ρ'. Lets consider that both read operations are invoked from the same reader $r = r'$. It follows from Lemma 6.3.5 that $\sigma_{res(\rho)}[r].tag \leq \sigma_{res(\rho')}[r].tag$ because the tag variable is monotonically non-decrementing. So it remains to investigate what happens when the two read operations are invoked by two different processes, r and r' respectively. Suppose that every server $s \in Q$ receives the messages of operation ρ with an event rcv$(m)_{r,s}$, and replies with a tag $m(\rho,1)_{s,r}.tag$ with an event send$(m)_{s,r}$ to r. Notice that for every server that replies, as mentioned in Lemma 6.3.3, $m(\rho,1)_{s,r}.tag \geq \sigma_{inv(\rho)}[r].maxTS$. Let the members of the quorum Q' (not necessarily different than Q) receive messages and reply to ρ'. Again for every $s' \in Q'$, $m(\rho',1)_{s',r'} \geq \sigma_{inv(\rho')}[r'].maxTS$. We know that the tag of the read operation ρ after the read-qview-eval$_r$ event of ρ may take a value between $maxTag_\rho \geq \sigma_{res(\rho)}[r].tag \geq minTag_\rho$. It suffice to examine the two extreme cases and every intermediate value can be proved similarly. So we have two cases to examine: 1) $\sigma_{res(\rho)}[r].tag = minTag_\rho$, and 2) $\sigma_{res(\rho)}[r].tag = maxTag_\rho$.

Case 1: Consider the case where $\sigma_{res(\rho)}[r].tag = minTag_\rho$, including the case where $minTag_\rho = maxTag_\rho$. This may happen only if the read-qview-eval$_r$ event reaches an iteration with tag $\tau = minTag_\rho$ and observes **QV1**. In other words all servers

$s \in Q - (Q)^{>\tau}$ reply with $m(\rho,1)_{s,r}.tag = minTag_{\rho}$. By Lemma 6.3.7 it follows that $(Q \cap Q') - (Q \cap Q')^{>\tau} \neq \emptyset$ and thus every server $s' \in Q \cap Q'$ replies to ρ with a tag $m(\rho,1)_{s,r}.tag \geq minTag_{\rho}$. By Lemma 6.3.2 it follows that every server $s' \in Q \cap Q$, replies with a tag $m(\rho',1)_{s',r'}.tag \geq m(\rho,1)_{s',r}.tag \geq minTag_{\rho}$. The read operation ρ' may return a value within the interval $minTag_{\rho} \leq \sigma_{res(\rho')}[r'].tag \leq maxTag_{\rho}$. Since for every server $s' \in Q \cap Q'$, $m(\rho',1)_{s',r'}.tag \geq minTag_{\rho} = \sigma_{res(\rho)}[r].tag$ then $maxTag_{\rho'} \geq m(\rho',1)_{s',r'}.tag \geq \sigma_{res(\rho)}[r].tag$. Hence, if $\sigma_{res(\rho)}[r'].tag = maxTag_{\rho'}$ it follows that $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$. On the other hand, if $\sigma_{res(\rho')}[r'].tag = minTag_{\rho'}$ we need to consider two cases: a) $minTag_{\rho'} \geq minTag_{\rho}$ and b) $minTag_{\rho'} < minTag_{\rho}$. If the first case is valid then it follows immediately that $\sigma_{res(\rho')}[r'].tag \geq minTag_{\rho}$ and thus $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$. If case b) is valid then it follows that the iteration reaches a tag equal to $minTag_{\rho'}$. Since however every server $s' \in Q \cap Q'$, replies with $m(\rho',1)_{s',r'}.tag \geq minTag_{\rho}$, then $m(\rho',1)_{s',r'}.tag \geq minTag_{\rho'}$ as well and thus all these servers are removed by iteration where tag is equal to $minTag_{\rho'}$. So it follows that $(Q \cap Q') - (Q')^{>minTag_{\rho'}} = \emptyset$ and that contradicts Lemma 6.3.7. So the case is impossible.

Case 2: Here we examine the case where $\sigma_{res(\rho)}[r].tag = maxTag_{\rho}$. This may happen after the read-qview-eval$_r$ of ρ if either observes a quorum view $\mathbf{QV1}$ or a quorum view $\mathbf{QV3}$. Let us examine the two cases separately.

Case 2a: In this case ρ witnessed a $\mathbf{QV1}$.

Therefore it must be the case that $\forall s \in Q, s$ replied with $m(\rho,1)_{s,r}.tag = maxTag_{\rho} = minTag_{\rho} = \sigma_{res(\rho)}[r].tag$. Thus by Lemma 6.3.2 $\forall s \in Q \cap Q'$, s replies with a tag

$m(\rho', 1)_{s,r'}.tag \geq m(\rho, 1)_{s,r}.tag$ to ρ', and hence, ρ' witnesses a maximum tag

$$maxTag_{\rho'} \geq maxTag_{\rho} \Rightarrow maxTag_{\rho'} \geq \sigma_{res(\rho)}[r].tag \tag{8}$$

Recall that $minTag_{\rho'} \leq \sigma_{res(\rho')}[r'].tag \leq maxTag_{\rho'}$. Clearly if $\sigma_{res(\rho')}[r'].tag = maxTag_{\rho'}$ then $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$. So it remains to examine the case where $\sigma_{res(\rho')}[r'].tag < maxTag_{\rho'}$. By Lemma 6.3.8, $\sigma_{res(\rho')}[r'].tag$ must be greater or equal to the minimum tag of any intersection of Q'. Since $\min(m(\rho', 1)_{s',r'}.tag) \geq \sigma_{res(\rho)}[r].tag$, for every $s' \in Q \cap Q'$, then by that lemma $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.

Case 2b: This is the case where $\sigma_{res(\rho)}[r].tag = maxTag_{\rho}$, because r witnessed a quorum view $\mathbf{QV3}$. In this case ρ proceeds in phase 2 before completing. Since $\rho \to \rho'$ and since ρ' happens after the read-ack$_r$[1] action of ρ, it means that ρ' happens after the read-phase2-fix$_r$ action of ρ as well. However ρ proceeds to phase 2 only after the read-phase1-fix$_r$ and read-qview-eval$_r$ actions. In the latter action ρ fixes the $maxTag$ variable to be equal to the $maxTag_{\rho}$. Once in phase 2, ρ sends inform messages with $maxTag_{\rho}$ to a complete quorum, say Q''. By Lemma 6.3.6, every server $s \in Q''$ replies with a tag

$$m(\rho, 2)_{s,r}.tag \geq maxTag_{\rho} \Rightarrow m(\rho, 2)_{s,r}.tag \geq \sigma_{res(\rho)}[r].tag \tag{9}$$

So ρ' will observe (by Lemma 6.3.2) that at least $\forall s' \in Q' \cap Q''$, $m(\rho', 1)_{s',r'}.tag \geq \sigma_{res(\rho)}[r].tag$. Hence by Lemma 6.3.8 ρ' returns a tag $\sigma_{res(\rho')}[r'].tag \geq \min(m(\rho', 1)_{s',r'}.tag)$ and thus, $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$ and this completes our proof. □

Lastly the following lemma states that if two read operations return two different tags then the values that correspond to these tags are also different.

[1] read-ack$_r$ occurs only if all phases reach a fix point and the $status$ variable becomes equal to $done$

Lemma 6.3.12 In any execution $\xi \in goodexecs(\text{CWFR}, \mathbb{Q})$, if ρ and ρ' two read operations from readers r and r' respectively, such that ρ (resp. ρ') returns the value written by ω (resp. ω'), then if $\sigma_{res(\rho)}[r].tag \neq \sigma_{res(\rho')}[r'].tag$ then ω is different than ω' otherwise they are the same write.

Proof. This lemma is ensured because a unique tag is associated to each written value by the writers. So it cannot be the case that two readers such that $\sigma_{res(\rho)}[r].tag \neq \sigma_{res(\rho')}[r'].tag$ returned the same value. □

Using the above lemmas we can obtain:

Theorem 6.3.13 Algorithm CWFR implements a MWMR atomic read/write register.

Proof. This theorem follows from Lemmas 6.3.6, 6.3.9, 6.3.10, and 6.3.11. Moreover Lemma 6.3.12 shows that each value is associated with a unique tag, and thus operations can be ordered with respect to the tags they write or return. □

6.4 Server Side Ordering - Algorithm SFW

In traditional MWMR atomic register implementations, [68, 34, 66, 36] (including algorithm CWFR) the writer is solely responsible for incrementing the tag that imposes the ordering on the values of the register. With the new technique, and our hybrid approach, this task is now also assigned to the servers, hence the name *Server Side Ordering* (SSO). Figure 22 presents a data flow of the two techniques.

At a first glance, SSO appears to be an intuitive and straightforward approach: servers are responsible to increment the timestamp associated with their local replica whenever they receive a write request. Yet, this technique proved to be extremely challenging. Traditionally,

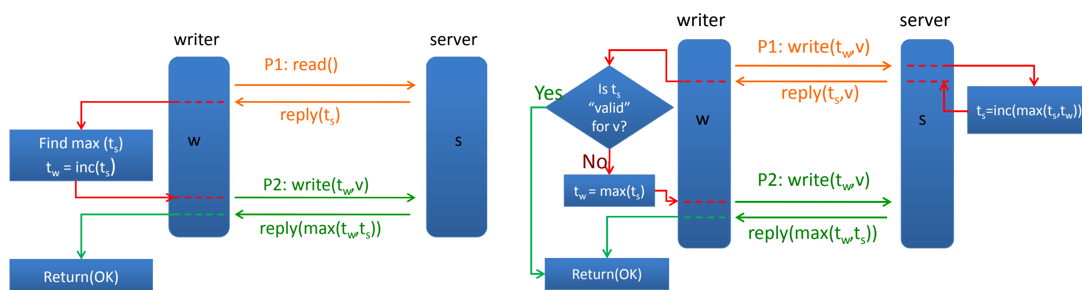


Figure 22: Traditional Writer Side Ordering Vs Server Side Ordering

two phase write operations were querying the register replicas for the latest timestamp, then they were incrementing that timestamp and finally they were assigning the new timestamp to the value to be written. Such methodology established that each individual writer was responsible to decide a *single* and *unique* timestamp to be assigned to a written value. Following this technique a belief was shaped that "writes must read".

The new technique promises to allow the writer avoid the query phase during a write operation. However, allowing the servers to increment the timestamps introduces new complexity to the problem: *multiple* and *different* timestamps may now be assigned to the same write request (and thus the same written value). Since timestamps are used to order the write operations, then multiple timestamps for a single write imply the appearance of the same operation in different points in the execution timeline. Hence the great challenge is to provide clients with enough information so that they decide a unique ordering for each written value to avoid violation of atomicity. For this purpose we combine the server generated timestamps (*global ordering*) with writer generated operation counters (*local ordering*).

In this Section we present algorithm SFW which adopts SSO to allow fast read *and* write operations. To our knowledge, this is the *first* algorithm that allows fast write operations in the MWMR environment.

6.4.1 New Way of Tagging the Values

Algorithm SFW uses $\langle tag, val \rangle$ pairs, to impose ordering on the values written to the register. In contrast to traditional approaches where the tag is a two field tuple, this algorithm requires the tag to be a triple. In particular, the tag is of the form $\langle ts, wid, wc \rangle \in \mathbb{N} \times \mathcal{W} \times \mathbb{N}$. The fields ts and wid are used as in common tags and represent the timestamp and writer identifier respectively. Field wc represents the write operation counter and is used to distinguish between write operations originating from the writer with identifier wid. In other words ts represent the *global* and wc the *local* value orderings, and are incremented by the servers and writers respectively (as required by SSO).

The necessity of the third field in a tag lies on the following observation: if a tag is a tuple of the form $\langle ts, wid \rangle$, then two server processes s and s' may associate two different tags $\langle ts_s, w \rangle$ and $\langle ts_{s'}, w \rangle$ respectively to a single write operation ω. Any operation however that witnesses such tags is not able to distinguish whether the tags refer to a single or different write operations from w. By including the writer's local ordering wc in each tag, the tags will become $\langle ts_s, w, wc \rangle$ and $\langle ts_{s'}, w, wc \rangle$. From the new tags it becomes apparent that the same write operation was assigned two different timestamps. The triples are compared lexicographically. In particular, we say that $tag_1 > tag_2$ if one of the following holds:

1. $tag_1.ts > tag_2.ts$, or

2. $(tag_1.ts = tag_2.ts) \wedge (tag_1.wid > tag_2.wid)$, or

3. $(tag_1.ts = tag_2.ts) \wedge (tag_1.wid = tag_2.wid) \wedge (tag_1.wc > tag_2.wc)$.

Notice that in traditional approaches, where the writer increments the timestamp each writer generates a *unique* timestamp for each of its writes. The writer identifier in that case is included in the tag to distinguish two write operations invoked by *different* writers that generated the *same* timestamp.

6.4.2 High Level Description of SFW

Below we present a high level description of the algorithm SFW. The algorithm deploys quorum systems and relies on the assumption that a single quorum is non-faulty throughout the algorithm's execution. To enable fast operations, algorithm SFW involves two predicates: one for the read protocol and one for the write protocol. Both predicates reveal the latest written value by evaluating the distribution of a tag within the quorum that replies to the read or write operation. The description that follows focus on tags written and returned. The association of values to those tags is straightforward.

Server. We begin with the description of the server as it plays a significant role in the system. In SFW, each server maintains the value of the replica and generates the tags associated with each value. As in the previous algorithms, the server waits for read/write requests. If a read request is received, the server updates its local information ($\langle tag, val \rangle$ pair) if the tag enclosed in the request is greater than the local tag of the server. Also, it marks the enclosed tag as confirmed, since a read/write requests contain the last tag-value pair returned by the invoking

process. In addition to the local information updates, the server generates a new tag when it receives a write request from some writer w. The elements of the new tag are:

(i) the incremented timestamp of the local tag of the server,

(ii) the identifier of the sender of the request (w), and

(iii) the new value enclosed in the request.

The new tag is inserted in a set, called *inprogress*, that stores all the tags generated by the particular server. Only a singe tag per writer is kept in the set. Thus, the server removes all the tags of w from the set before adding the new tag. Once the tag is added the server replies to w with the generated tag.

Writer. The write operation requires one or two rounds. To perform a write operation ω, a writer w sends messages to all of the servers and waits for a quorum of these, Q, to reply. Once w receives replies from all the servers of some quorum Q, w collects all of the tags assigned to ω from the $inprogress$ set of each of those servers. Then it applies a predicate on the collected tags. That predicate checks if any of the collected tags appear in some intersection of Q with at most $\frac{n}{2} - 1$ other quorums, where n the intersection degree of the deployed quorum system. If there exists such a tag τ then the writer adopts τ as the tag of the value it tried to write; otherwise the writer adopts the maximum among the collected tags in the replied quorum. The writer proceeds in a second round to propagate the tag assigned to the written value if:

(a) the predicate holds but the tag is only propagated in an intersection of Q with more than $\frac{n}{2} - 2$ other quorums, or

(b) the predicate does not hold.

In any other case the write operation is fast and completes in a single communication round. In short, the key idea of the predicate depends on the observation that if τ is observed in the intersection defined by the predicate then no other tag in Q will be observed in an equal or bigger intersection by any subsequent operation. Thus, ω will be uniquely associated with τ by any operation since no other tag will satisfy the predicate of any subsequent operation that returns the value written by ω.

Reader. The reader protocol is similar to the writer protocol in the sense that it uses a predicate to decide the latest tag written on the register. When the reader wants to perform a read operation ρ, it sends messages to all the servers and waits for all the servers in some quorum Q to reply. As soon as those replies arrive, the reader discovers the maximum confirmed tag ($maxCT$) among the received messages. In addition it collects all the tags contained in every $inprogress$ set received in a set inP. Then the reader discovers and returns the largest tag $maxTag \in inP$ that:

(i) $maxTag_{>} maxCT$, and

(ii) $maxTag$ satisfies the reader predicate (defined below).

According to the read predicate, ρ must discover $maxTag$ in an intersection between Q and at most $\frac{n}{2} - 2$ other quorums, where n the intersection degree of the quorum system. As we discuss in later sections, this ensures that any subsequent operation will at least observe the $maxTag$ returned by ρ. If there exists no such tag in inP, then $maxCT$ is returned. Notice that a read operation ρ cannot return any tag smaller than $maxCT$, as it denotes a tag already decided by an operation π that precedes or is concurrent with ρ. A read operation is slow and performs a second communication round if one of the following cases hold:

(a) the predicate holds but the tag is propagated in an intersection between Q and exactly $\frac{n}{2} - 2$ other quorums, or

(b) the predicate does not hold and $maxCT$ is not propagated in an intersection between Q and at most $n - 1$ other quorums.

During the second round, the reader propagates the tag decided during the first round to some quorum of servers.

6.4.3 Formal Specification of SFW

In this section we provide the formal description of SFW using Input/Output Automata [67]. The algorithm is composed of four automata: (i) SFW$_w$ automaton for every $w \in \mathcal{W}$, (ii) SFW$_r$ automaton for every $r \in \mathcal{R}$, (iii) SFW$_s$ automaton for every $s \in \mathcal{S}$ to handle the read and write requests on the atomic register, and (iv) $Channel_{p,s}$ and $Channel_{s,p}$ that establish the reliable asynchronous process-to-process communication channels (see Section 3.1.2). Unlike previous algorithms we first present the formal specification of the SFW$_s$ automaton. This will help us present important data types that are used later by the reader and writer automata.

Automaton SFW$_s$.

The state variables, the signature and the transitions of the SFW$_s$ automaton are given in Figure 23. The local state of a server process s, is defined by the following local variables:

- $\langle\langle ts, wid, wc\rangle, v\rangle \in \mathbb{N} \times \mathcal{W} \times \mathbb{N} \times V$: the local tag stored in the server along with its associated value. This is the latest tag-value pair received or generated at server s.

- $\mathit{confirmed}_s \in \mathbb{N} \times \mathcal{W} \times \mathbb{N}$: the largest tag known by s that has been returned by some reader or writer process.

- $\mathit{inprogress}_s \subseteq \mathbb{N} \times \mathcal{W} \times \mathbb{N}$: set which includes all the latest tags assigned by s to write requests. The set includes one tag per writer.

- $Counter(p) \in \mathbb{N}$: this array maintains the latest request index of each client (reader or writer). It helps s to distinguish fresh from stale messages.

- $status \in \{idle, active\}$: specifies whether the automaton is processing a request received ($status = active$) or it can accept new requests ($status = idle$).

- $msgType \in \{\text{WRITEACK},\text{READACK},\text{PROPACK}\}$: Type of the acknowledgment depending on the type of the received message.

- $failed \in \{true, false\}$: indicates whether the server associated with the automaton has failed.

Each server s waits to receive read or write messages originated from some process p that invokeds a read or write operation respectively. Each message received by s contains:

(a) $\mathit{mType} \in \{W, R, RP\}$: the type of the message W:write, R:read, P:propagation,

(b) $t \in \mathbb{N} \times \mathcal{W} \times \mathbb{N}$: the local tag of p,

(c) $val \in V$: the value to be written if p invokes a write operation or the latest value returned by p if p invokes a read operation,

(c) $opCount \in \mathbb{N}$: the sequence number of the operation if p invokes a write or $t.tag.wc$ if p invokes a read, and

---

Signature:

Input:
$\mathsf{rcv}(m)_{p,s},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \mathcal{W}$
fail_s

Output:
$\mathsf{send}(m)_{s,p},\ m \in M,\ s \in \mathcal{S},\ p \in \mathcal{R} \cup \mathcal{W}$

State:

$tag \in \mathbb{N} \times \mathcal{W} \times \mathbb{N}$, initially $\{0, \min(\mathcal{W}), 0\}$
$v \in V$, initially $\perp$
$inprogress \subseteq \mathbb{N} \times \mathcal{W} \times \mathbb{N}^{+} \times V$, initially $\emptyset$
$confirmed \in \mathbb{N} \times \mathcal{W} \times \mathbb{N}^{+} \times V$, initially $\{0, \min(\mathcal{W}), 0\}$
$Counter(p) \in \mathbb{N}^{+}, p \in \mathcal{R} \cup \mathcal{W}$, initially 0
$msgType \in \{$WRITEACK,READACK,PROPACK$\}$
$status \in \{idle, active\}$, initially $idle$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{rcv}(\langle msgT, t, val, opCount, C\rangle)_{p,s}$
Effect:
if $\neg failed$ then
if $status = idle$ and $C > pCount(p)$ then
$status \leftarrow active$
$Counter(p) \leftarrow count$
if $tag < t.tag$ then
$(\langle tag.ts, tag.wid, tag.wc\rangle, v) \leftarrow$
$(\langle t.tag.ts, t.tag.wid, t.tag.wc\rangle, t.val)$
if $msgType = \mathsf{W}$ then
$(tag.ts, tag.wid, tag.wc) \leftarrow$
$(tag.ts + 1, p, opCount)$
$inprogress \leftarrow$
$(inprogress - \langle\langle *, p, *\rangle, *\rangle) \cup \langle tag, val\rangle$
if $confirmed < t.tag$ then
$confirmed \leftarrow t.tag$

Output $\mathsf{send}(\langle msgT, inprog, conf, C\rangle)_{s,p}$
Precondition:
$\neg failed$
$\langle msgT, inprog, conf, C\rangle =$
$\langle msgType, inprogress, confirmed, Counter(p)\rangle$
Effect:
$status = idle$

Input fail_s
Effect:
$failed \leftarrow$ **true**

Figure 23: SFW$_s$ Automaton: Signature, State and Transitions

(d) $C \in \mathbb{N}$: a counter that distinguishes new from stale messages from p.

Upon receipt of any type of message, s updates its local information as needed. In particular, if $t.tag > tag_s$ then s assigns $tag_s = t.tag$ and its value $v = t.val$. Similarly if $t.tag > confirmed_s.tag$, then s sets $confirmed_s = t$. Once those updates are completed, the server replies back to process p if $mType \in \{R, RP\}$. If $mType = W$, s needs to take additional actions to record the value to be written. First, s generates and assigns to its local tag a new tag ($newTag$) that contains the following fields:

- $tag.ts + 1$: the timestamp of the local tag incremented by 1,
- $t.tag.wid$: the identifier of the requesting writer, and
- $opCount$: the sequence number of the write operation at the requesting writer.

Note, that the new tag is greater than both the tag_s and $t.tag$, the tag included in the message. Once the new tag is generated, s updates its $inprogress_s$ set. Constructed to hold one entry per writer, the server removes any entry of a write operation from p from the $inprogress_s$ set, before adding the new tag-value pair in it. As a result, the $inprogress_s$ contains the latest tags assigned by s to any write operations that requested the server's replica value. Thus, in the worst case the $inprogress_s$ contains a single tag for each writer. The importance of using the $inprogress_s$ is twofold:

1. Each write operation witnesses the tag assignments to the value to be written from all servers in the replying quorum (since even a concurrent write will not overwrite the tag of the specific write). Thus, the writer can establish if any of the tags was adopted by enough servers in the replying quorum or if it needs to proceed to a second round.

2. Each read operation obtains full knowledge on the tags reported to each writer, and is able to predict which tag each writer adopts for its latest write operation. By ordering the tags the reader is able to establish the write operation with the largest tag and hence, the latest written value.

The read and write predicates, utilize the above observations to allow fast read and write operations. Once a server s completes all the necessary actions, it acknowledges every message received by sending its $inprogress_s$ set and $confirmed_s$ variable to the requesting process p.

Automaton SFW_w**.**

The state, signature and transitions of the SFW_w automaton are given in Figure 24. The variables that define the state of the SFW_w are the following:

- $\langle\langle ts, w, wc\rangle, v\rangle \in \mathbb{N} \times \{w\} \times \mathbb{N} \times V$: writer's local tag along with the latest value written by the writer. The tag is composed of a timestamp, the identifier of the writer, and the sequence number of the last write operation.

- $vp \in V$: this variable is used to hold the previous value written.

- $wc \in \mathbb{N}$: the sequence number of the last write operation or 0 if the writer did not perfrom any writes.

- $wCounter \in \mathbb{N}$: the number of write requests performed by the writer. Is used by the servers to distinguish fresh from stale messages.

- $phase \in \{W, RP\}$: indicates whether the write operation is in the write or propagation phase (first or second round respectively).

- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the writer received all the necessary replies to complete its write operation and is ready to respond to the client.

- $srvAck \subseteq \mathcal{S} \times M_2$: a set that contains the servers and their replies to the write request. The set is reinitialized to $\emptyset$ at the response step of every write operation.

- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

To uniquely identify all write operations, a writer w maintains a local variable wc that is incremented each time w invokes a write operation. That variable denotes w's local ordering on the write operations it performs. Any other process in the system may identify a

Signature:

Input:
$\mathsf{write}(val)_w,\ val \in V, w \in \mathcal{W}$
$\mathsf{rcv}(m)_{s,w},\ m \in M_1,\ s \in \mathcal{S}, w \in \mathcal{W}$
$\mathsf{fail}_w, w \in \mathcal{W}$

Output:
$\mathsf{send}(m)_{w,s},\ m \in M_2,\ s \in \mathcal{S}, w \in \mathcal{W}$
$\mathsf{write\text{-}ack}_w, w \in \mathcal{W}$

Internal:
$\mathsf{write\text{-}phase1\text{-}fix}_w, w \in \mathcal{W}$
$\mathsf{write\text{-}phase2\text{-}fix}_w, w \in \mathcal{W}$

State:

$tag = \langle ts, w, wc \rangle \in \mathbb{N} \times \{w\} \times \mathbb{N}$, initially $\{0, w, 0\}$
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$wc \in \mathbb{N}^+$, initially 0
$phase \in \{\mathsf{W}, \mathsf{RP}\}$, initially W
$wCounter \in \mathbb{N}^+$, initially 0
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq \mathcal{S} \times M_2$, initially $\emptyset$
$failed$, a Boolean initially **false**

Transitions:

Input $\mathsf{write}(val)_w$
Effect:
if $\neg failed \wedge status = idle$ then
$status \leftarrow active$
$srvAck \leftarrow \emptyset$
$phase \leftarrow \mathsf{W}$
$vp \leftarrow v$
$v \leftarrow val$
$wCounter \leftarrow wCounter + 1$
$wc \leftarrow wc + 1$

Input $\mathsf{rcv}(\langle inprogress, confirmed, C \rangle)_{s,w}$
Effect:
if $\neg failed$ then
if $status = active$ and $wCounter = C$ then
$srvAck \leftarrow$
$srvAck \cup \{\langle s, m \rangle\}$

Output $\mathsf{send}(\langle msgT, t, val, wc, C \rangle)_{w,s}$
Precondition:
$status = active$
$\neg failed$
$\langle msgT, t, val, wc, C \rangle =$
$\langle phase, \langle tag, vp \rangle, v, wc, wCounter \rangle$
Effect:
none

Output $\mathsf{write\text{-}ack}_w$
Precondition:
$status = done$
$\neg failed$
Effect:
$status \leftarrow idle$

Input fail_w
Effect:
$failed \leftarrow$ **true**

Internal $\mathsf{write\text{-}phase1\text{-}fix}_w$
Precondition:
$\neg failed$
$status = active$
$phase = \mathsf{W}$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
Effect:
$T \leftarrow \{\langle ts, w, * \rangle : \langle ts, w, * \rangle \in \bigcup_{(s,m) \in srvAck} m.inprogress\}$
if $\exists\, \tau, MS, \mathbb{Q}^i$:
$\tau \in T$
$\wedge\ MS = \{s : s \in Q \wedge (s, m) \in srvAck \wedge \tau \in m.inprogress\}$
$\wedge\ \mathbb{Q}^i \subseteq \mathbb{Q}$ s.t. $0 \leq i \leq \lfloor \frac{n}{2} - 1 \rfloor\ \wedge\ (\bigcap_{Q' \in (\mathbb{Q}^i \cup \{Q\})} Q') \subseteq MS$
then
$\langle tag.ts, tag.wid, tag.wc \rangle \leftarrow \langle \tau.ts, w, wc \rangle$
if $i \geq \max(0, \frac{n}{2} - 2)$ then
$phase \leftarrow \mathsf{RP}$
$wCounter \leftarrow wCounter + 1$
else
$status \leftarrow done$
else
$\langle tag.ts, tag.wid, tag.wc \rangle \leftarrow \max_{\tau \in T}(\langle \tau.ts, w, wc \rangle)$
$wCounter \leftarrow wCounter + 1$
$phase \leftarrow \mathsf{RP}$
$srvAck \leftarrow \emptyset$

Internal $\mathsf{write\text{-}phase2\text{-}fix}_w$
Precondition:
$status = active$
$\neg failed$
$phase = \mathsf{RP}$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, ., .) \in srvAck\}$
Effect:
$status \leftarrow done$

Figure 24: SFW_w Automaton: Signature, State and Transitions

write operation ω by the tuple $\langle w, wc \rangle$: ω was invoked by writer w, and it was the wc write from w. When the $\mathsf{write}(val)_w$ occurs, then the writer sends messages to any server $s \in \mathcal{S}$ by action $\mathsf{send}(m)_{w,s}$. Each message to s includes the type of the message (W or RP), the last written value along with the tag that the writer associated to that value ($\langle tag, vp \rangle$), the new value (v), the sequence counter (wc), and the counter that helps the detection of delayed messages ($wCounter$). Each time the action $\mathsf{rcv}(m)_{s,w}$ occurs, the writer receives a reply from s. As noted in the server protocol, each server s replies with a message that contains $m(\omega)_{s,w}.inprogress$ set and $m(\omega)_{s,w}.confirmed$ variable. Once the writer receives messages from the servers of a full quorum Q, it collects the tags generated from each $s \in Q$ for ω. A predicate is then applied to every tag in the collection. According to the predicate, we want to know for a tag tag if there is any subset of the servers of the replied quorum s.t. they that generated tag and cover the intersection between the replied quorum and $\frac{n}{2} - 1$ other quorums in $\mathbb{Q}$. If tag satisfies the predicate, the writer's tag_w variable becomes equal to tag. More formally the writer predicate is the following:

> **Writer predicate for a write** ω **(PW):** $\exists\ \tau, \mathbb{Q}^i, MS$ where: $\tau \in \{\langle ., \omega \rangle : \langle ., \omega \rangle \in m(\omega, 1)_{s,w}.inprogress \ \wedge \ s \in Q\}$, $MS = \{s : s \in Q \ \wedge \ \tau \in m(\omega, 1)_{s,w}.inprogress\}$, and $\mathbb{Q}^i \subseteq \mathbb{Q}, 0 \leq i \leq \left\lfloor \frac{n}{2} - 1 \right\rfloor$, s.t. $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q\}} \mathcal{Q}) \subseteq MS$.

In case the predicate is true for $i \geq \max(0, \frac{n}{2} - 2)$, the writer changes its $phase = RP$ variable and reinitializes the $srvAck$ set. This leads the writer to a second round. As soon as the writer receives replies from the servers of a full quorum during its second round, it terminates the operation by setting the $status$ variable to $idle$. The idea behind the predicate is quite intuitive: since we assume an n-wise quorum system, if all of the servers in the intersection of

the *minority* ($< \frac{n}{2}$) of any n quorums observe a particular tag then the intersection of any other minority of quorums has at least one server in common with the first intersection (since we assume that ever n quorums intersect). If the predicate does not hold for any of the collected tags then the writer assigns the maximum of the collected tags to the value to be written, and proceeds to a second round to propagate the particular tag to a full quorum.

Automaton SFW_r**.**

The state, signature and transitions of the SFW_r automaton are given in Figure 25. The variables that define the state of the SFW_r are the following:

- $\langle\langle ts, wid, wc\rangle, v\rangle \in \mathbb{N} \times \{w\} \times \mathbb{N} \times V$: the latest tag-value pair returned by the reader r. The tag is composed of a timestamp, a writer identifier, and the sequence number of the write operation from that writer.

- $rCounter \in \mathbb{N}$: the number of read requests performed by rdr. Is used by the servers to distinguish fresh from stale messages.

- $phase \in \{R, RP\}$: indicates whether the read operation is in the read or propagation phase (first or second round respectively).

- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the writer received all the necessary replies to complete its write operation and is ready to respond to the client.

- $maxCT \in \mathbb{N} \times \mathcal{W} \times \mathbb{N} \times V$: the maximum confirmed tag-value pair discovered during the last read operation from r.

Signature:

Input:
$\mathsf{read}_r,\ r \in \mathcal{R}$
$\mathsf{rcv}(m)_{s,r},\ m \in M_1,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\mathsf{fail}_r,\ r \in \mathcal{R}$

Output:
$\mathsf{send}(m)_{r,s},\ m \in M_2,\ r \in \mathcal{R},\ s \in \mathcal{S}$
$\mathsf{read\text{-}ack}(val)_r,\ val \in V,\ r \in \mathcal{R}$

Internal:
$\mathsf{read\text{-}phase1\text{-}fix}_r,\ r \in \mathcal{R}$
$\mathsf{read\text{-}phase2\text{-}fix}_r,\ r \in \mathcal{R}$

State:

$tag = \langle ts, wid, wc \rangle \in \mathbb{N}\times\mathcal{W}\times\mathbb{N}$, initially $\{0, \min(\mathcal{W}), 0\}$
$v \in V$, initially $\perp$
$phase \in \{\mathsf{R}, \mathsf{RP}\}$, initially R
$rCounter \in \mathbb{N}^+$, initially 0
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq \mathcal{S} \times M_1$, initially $\emptyset$
$maxCT \in \mathbb{N} \times \mathcal{W} \times \mathbb{N} \times V$, initially $\{\langle 0, \min(\mathcal{W}), 0\rangle, \perp\}$
$inPtag \subseteq \mathbb{N} \times \mathcal{W} \times \mathbb{N} \times V$, initially $\emptyset$
$failed$, a Boolean initially **false**

Transitions:

Input read_r
Effect:
if $\neg failed \wedge status = idle$ then
$phase \leftarrow \mathsf{R}$
$status \leftarrow active$
$rCounter \leftarrow rCounter + 1$

Input $\mathsf{rcv}(\langle inprogress, confirmed, C\rangle)_{s,r}$
Effect:
if $\neg failed \wedge status = active$ then
if $rCounter = C$ then
$srvAck \leftarrow$
$srvAck \cup \{\langle s, m\rangle\}$

Output $\mathsf{send}(\langle msgT, t, val, wc, C\rangle)_{r,s}$
Precondition:
$status = active$
$\neg failed$
$\langle msgT, t, val, wc, C\rangle =$
$\langle phase, \langle tag, v\rangle, v, tag.wc, rCounter\rangle)$
Effect:
none

Output $\mathsf{read\text{-}ack}(val)_r$
Precondition:
$\neg failed$
$status = done$
$val = retvalue$
Effect:
$replyQ \leftarrow \emptyset$
$srvAck \leftarrow \emptyset$
$status \leftarrow idle$

Input fail_r
Effect:
$failed \leftarrow true$

Internal $\mathsf{read\text{-}phase1\text{-}fix}_r$
Precondition:
$\neg failed$
$status = active$
$phase = \mathsf{R}$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
Effect:
$maxCT \leftarrow \{\max(m.confirmed) : (s, m) \in srvAck \wedge s \in Q\}$
$inPtag = \{\tau : \tau \in \bigcup_{(s,m)\in srvAck \wedge s\in Q} m.inprogress\}$
if $\exists\ \tau, MS, \mathbb{Q}^j$ **:**
$\tau = \max_{\tau' \in inPtag}(\tau')$ s.t.
$\tau > maxCT$
$\wedge\ MS = \{s : s \in Q \wedge (s, m) \in srvAck \wedge \tau \in m.inprogress\}$
$\wedge\ \mathbb{Q}^j \subseteq \mathbb{Q}$ s.t. $0 \leq j \leq \lfloor \frac{n}{2} - 2 \rfloor\ \wedge (\bigcap_{Q' \in (\mathbb{Q}^j \cup \{Q\})} Q') \subseteq MS$
then
$\langle\langle tag.ts, tag.wid, tag.wc\rangle, v\rangle \leftarrow \langle\langle \tau.ts, \tau.w, \tau.wc\rangle, \tau.val\rangle$
if $j = \max(0, \frac{n}{2} - 2)$ then
$phase \leftarrow \mathsf{RP}$
else
$status \leftarrow done$
else
$MC \leftarrow \{s :$
$s \in Q \wedge (s, m) \in srvAck \wedge m.confirmed = maxCT\}$
$\langle tag.ts, tag.wid, tag.wc\rangle \leftarrow$
$\langle maxCT.ts, maxCT.w, maxCT.wc\rangle$
$v \leftarrow maxCT.val$
if $\exists C : C \subseteq \mathbb{Q} \wedge |C| \leq n - 2 \wedge (\bigcap_{Q' \in C} Q') \cap Q \subseteq MC$ then
$status \leftarrow done$
else
$phase \leftarrow \mathsf{RP}$

Internal $\mathsf{read\text{-}phase2\text{-}fix}_r$
Precondition:
$\neg failed$
$status = active$
$phase = 2$
$\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, ., .) \in srvAck\}$
Effect:
$status \leftarrow done$
$phase \leftarrow 1$

Figure 25: SFW_r Automaton: Signature, State, and Transitions

- $inPtag \subseteq \mathbb{N} \times \mathcal{W} \times \mathbb{N} \times V$: a set that contains all the tags discovered during a read operation from r.

- $srvAck \subseteq \mathcal{S} \times M_2$: a set that contains the servers and their replies to the write request. The set is reinitialized to $\emptyset$ at the response step of every write operation.

- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

The reader protocol is similar to the writer protocol in the sense that it uses a predicate to decide the latest tag written on the register. When the reader performs a read operation ρ, it sends messages to all servers and waits for all servers in some quorum Q to reply. A message is sent when the $\mathsf{send}(m)_{r,s}$ occurs and a message is received when the $\mathsf{rcv}(m)_{s,r}$ action occurs. When those replies are received, the action read-phase1-fix$_r$ is enabled. If that action occurs, the reader discovers the maximum confirmed tag ($maxCT$) among the received message. In addition, it collects all the tags contained in every $m(\rho)_{s,r}.inprogress$ set received in $inPtag$. Then the reader examines the tags in $inPtag$ to find the largest tag, say $maxTag$, that is greater than $maxCT$ and also satisfies the reader predicate. Notice, that there may be larger tags than $maxTag$ in $inPtag$. Although those tags are greater than $maxCT$, they do not satisfy the predicate. The reader predicate for a read operation ρ from r that receives messages from a quorum Q, is the following:

> **Reader predicate for a read ρ (PR):** $\exists\ \tau, \mathbb{Q}^j, MS$, where: $\max(\tau) \in \bigcup_{s\in Q} m(\rho,1)_{s,r}.inprogress$, $MS = \{s : s \in Q \wedge \tau \in m(\rho,1)_{s,r}.inprogress\}$, and $\mathbb{Q}^j \subseteq \mathbb{Q}, 0 \leq j \leq \left\lfloor \frac{n}{2} - 2 \right\rfloor$, s.t. $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q\}} \mathcal{Q}) \subseteq MS$.

The reader predicate shares a similar idea as the writer predicate. To ensure that subsequent operations do not return a smaller tag – and hence an older value – the reader must discover $maxTag$ in an intersection between Q and at most $\frac{n}{2}-2$ other quorums, where n the intersection degree of the quorum system. If there exists no tag in $inPtag$ that is greater than $maxCT$ and satisfies the predicate, then $maxCT$ is returned. Notice that a read operation ρ cannot return any tag smaller than $maxCT$. Recall from the server description that $maxCT$ denotes a tag already decided by an operation π that precedes or is concurrent with ρ.

A read operation is slow and performs a second round if the $phase$ variable becomes equal to RP. This happens in two cases according to read-phase1-fix$_r$:

(a) the predicate holds with $|\mathbb{Q}^j| = \max(0, \frac{n}{2}-2)$, or

(b) $maxCT$ is not propagated in an $(n-1)$-wise intersection.

During the second round, the reader reinitializes the set of replies $srvAck$, propagates the tag decided during the first round of the read operation, and waits to receive replies from the servers of some quorum Q. Once those replies are received the action read-phase2-fix$_r$ may occur, and the $status$ becomes $done$. Once $status = done$ the action read-ack$(val)_r$ returns the value to the environment. Notice that if a second round is not necessary then $status$ becomes equal to $done$ in the read-phase1-fix$_r$ action.

Remark 6.4.1 By close investigation of the predicates of Algorithm SFW, one can see that SFW approaches the bound of Theorem 6.2.6, as it produces executions that contain up to $n/2$ fast consecutive write operations, while maintaining atomic consistency. Obtaining a tighter upper bound is subject of future work.

6.4.4 Correctness of SFW

We proceed to show the correctness of algorithm SFW, that is, to show that the algorithm satisfies the termination and atomicity properties presented in Definitions 3.2.4 and 3.2.5.

Termination

Termination can be shown similar to Section 6.3.4. Each phase of any read or write operation terminates when the invoking process receives replies from at least a single quorum. According to our failure model, all but one quorums may fail (see Section 3.1.4). Thus, any correct process receives replies from at least the correct quorum. Thus, every operation from a correct process eventually terminates and hence, Definition 3.2.4 is satisfied.

Atomicity

For the rest of the section we use part of the notation introduced in Chapter 3 and Section 6.3.4. We denote by $inprogress[\omega].tag$ the tag of the write operation ω in an $inprogress$ set. Analogously, $\sigma[s].inprogress[\omega].tag$ and $m(\pi, c)_{p,p'}.inprogress[\omega].tag$, are used to denote the tag of the write ω in the $inprogress_s$ set at a state σ and in the $m(\pi, c)_{p,p'}.inprogress$ set in the message sent from p to p' during the cth round of operation π. By well-formedness, it follows that a write operation $\omega = \langle w, wc\rangle$ precedes a write operation $\omega' = \langle w, wc'\rangle$, both invoked by the same writer $w \in \mathcal{W}$, if and only if $wc < wc'$. For a read/write operation π invoked from a reader/writer process p, we denote by $\sigma_{inv(\pi)}[p].tag$ the value of the tag variable at the read/write event of π. The tag assigned to a write $\omega = \langle w, wc\rangle$ can be obtained by $\sigma_{res(\omega)}[w].tag = \langle ts, \omega\rangle$, where $ts \in \mathbb{N}$ the timestamp included in the tag. Similarly, the tag

returned by a read operation ρ is denoted by $\sigma_{res(\rho)}[r].tag$. Also, let $m(\pi, c)_{s,p}.confirmed$ and $m(\pi, c)_{s,p}.inprogress$ denote the $confirmed$ variable and $inprogress$ set that s sends to p during the cth round of π. Finally, let $\mathbb{Q}^i \subseteq \mathbb{Q}$ be a set of quorums with cardinality $|\mathbb{Q}^i| = i$ (see Section 3.1.4).

Recall that the two predicates used in the algorithm are the following assuming that an operation received messages from a quorum Q:

Writer predicate for a write ω (PW): $\exists\ \tau, \mathbb{Q}^i, MS$ where: $\tau \in \{\langle *, \omega\rangle : \langle *, \omega\rangle \in m(\omega, 1)_{s,w}.inprogress \ \wedge\ s \in Q\}$, $MS = \{s : s \in Q \ \wedge\ \tau \in m(\omega, 1)_{s,w}.inprogress\}$, and $\mathbb{Q}^i \subseteq \mathbb{Q}, 0 \leq i \leq \lfloor \frac{n}{2} - 1 \rfloor$, s.t. $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q\}} \mathcal{Q}) \subseteq MS$.

Reader predicate for a read ρ (PR): $\exists\ \tau, \mathbb{Q}^j, MS$, where: $\max(\tau) \in \bigcup_{s\in Q} m(\rho, 1)_{s,r}.inprogress$, $MS = \{s : s \in Q \ \wedge\ \tau \in m(\rho, 1)_{s,r}.inprogress\}$, and $\mathbb{Q}^j \subseteq \mathbb{Q}, 0 \leq j \leq \max(0, \lfloor \frac{n}{2} - 2 \rfloor)$, s.t. $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q\}} \mathcal{Q}) \subseteq MS$.

The writer predicate is located in write-phase1-fix of Figure 24 and the reader predicate is located in read-phase1-fix of Figure 25. We adopt the definition of atomicity presented in Section 6.3.4, that expresses the three properties of Definition 3.2.5 based on the tags returned from the read/write operations.

First, we show that each process maintains monotonically nondecreasing tags.

Lemma 6.4.2 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, $\sigma'[s].tag \geq \sigma[s].tag$ for any server $s \in \mathcal{S}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ.

Proof. It is easy to see that a server s modifies its tag_s variable only if the tag in a received message from a process p during an operation π is such that $m(\pi, *)_{p,s}.tag > tag_s$. In addition, if

$m(\pi, 1)_{p,s}.mType = W$ is a write message, $tag_s.ts = \max(tag_s.ts, m(\pi, 1)_{p,s}.tag.ts) + 1$. Since from a state σ to a state σ' only these these modifications can be applied on the server's tag then $\sigma'[s].tag \geq \sigma[s].tag$, and hence the server's tag is monotonically increasing. Furthermore, since the initial tag of the server is set to $\langle 0, \min(wid), 0\rangle$ and since $m(\pi, *)_{p,s}.tag > tag_s$ only if $m(\pi, *)_{p,s}.tag.ts \geq tag_s.ts$, then $tag_s.ts$ is always greater than 0. □

The next lemma shows the monotonicity of the confirmed variable of each server.

Lemma 6.4.3 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, $\sigma'[s].confirmed \geq \sigma[s].confirmed$ for any server $s \in \mathcal{S}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ.

Proof. From the algorithm it follows that the server s modifies the value of its $confirmed_s$ variable only if the tag $m(\pi, *)_{p,s}.tag$ in a message received by s from p for operation π, is such that $m(\pi, *)_{p,s}.tag > confirmed_s$. Thus, the lemma follows. □

The following lemma examines the monotonicity of the tag of each writer as this is kept in the inprogress set of each server.

Lemma 6.4.4 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, $\sigma'[s].inprogress\,[\langle w, wc'\rangle].tag \geq \sigma[s].inprogress\,[\langle w, wc\rangle].tag$ for any server $s \in \mathcal{S}$, any writer $w \in \mathcal{W}$, and any σ, σ' in ξ, such that σ appears before σ' in ξ.

Proof. Notice that the server s maintains just a single record for a writer w in its $inprogress_s$ set. Each time the server s receives a new write $m(\omega, 1)_{w,s}$ message from a write operation $\omega = \langle w, wc\rangle$ from w, it first updates its local tag if $tag_s < m(\omega, 1)_{w,s}.tag$ and then

generates a new tag. The generated tag and, hence the tag inserted in the $inprogress_s$ is $\sigma[s].inprogress[\omega].tag > m(\omega,1)_{w,s}.tag$, since $\sigma[s].inprogress[\omega].tag = \langle tag_s.ts+1, \omega \rangle$. The local tag of the server becomes equal to the generated tag, and thus $\sigma[s].tag > m(\omega,1)_{w,s}.tag$ at state σ. By Lemma 6.4.2 the local tag of the server s is non decreasing. So, if σ'' is the state right before σ' (possibly $\sigma'' = \sigma$) in ξ, it must be true that $\sigma''[s].tag \geq \sigma[s].tag$. Since, s adds the new write operation $\omega' = \langle w, wc' \rangle$ it follows that it generates a tag $\sigma'[s].inprogress[\omega'].tag > \sigma''[s].tag$. Therefore, $\sigma'[s].inprogress[\omega'].tag > \sigma[s].tag$, and hence $\sigma'[s].inprogress[\omega'].tag > \sigma[s].inprogress[\omega].tag$ and the lemma follows. □

Lemma 6.4.5 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a server $s \in \mathcal{S}$ receives a tag $m(\pi,*)_{p,s}.tag$ from a process p for an operation π, then s replies to p with a $m(\pi,*)_{s,p}.confirmed \geq m(\pi,*)_{p,s}.tag$.

Proof. It follows by Lemma 6.4.3 that s upgrades the $confirmed_s$ variable only if the tag enclosed in the message $m(\pi,*)_{p,s}.tag > confirmed_s$. If so, s replies with $m(\pi,*)_{s,p}.confirmed = m(\pi,*)_{p,s}.tag$; otherwise it replies with $m(\pi,*)_{s,p}.confirmed \geq m(\pi,*)_{p,s}.tag$ to operation π. □

The next lemma shows that when a server receives a write message it generates a tag greater than the tag enclosed in the received message and any other tag the server has generated.

Lemma 6.4.6 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a server $s \in \mathcal{S}$ receives a tag $m(\omega,1)_{w,s}.tag$ from a process $w \in \mathcal{W}$, for the first communication round of a write operation ω (i.e. type W), then s replies with an $m(\omega,1)_{s,w}.inprogress$ that contains $m(\omega,1)_{s,w}.inprogress[\omega].tag > m(\omega,1)_{w,s}.tag$ and $m(\omega,1)_{s,w}.inprogress[\omega].tag = \max_{\tau' \in m(\omega)_{s,w}.inprogress}(\tau')$.

Proof. When s receives W message from the write operation ω, it checks if the tag $m(\omega,1)_{w,s}.tag$ is greater than its local tag. If so it updates its tag to be equal to $m(\omega,1)_{w,s}.tag$. Thus, after this update it is true that the tag of s, $tag_s \geq m(\omega,1)_{w,s}.tag$. If W message is received from ω, then s generates a new tag $\tau' = \langle tag_s.ts+1, w, m(\omega,1)_{w,s}.wc\rangle$. Thus, the new tag is greater than the local tag of the server $\tau' > tag_s$ and thus $\tau' > m(\omega,1)_{w,s}.tag$ as well. Then, s replaces any previous operations from w in its $inprogress_s$ set and inserts the new tag. Since $\omega = \langle w, m(\omega,1)_{w,s}.tag\rangle$ then τ' is the unique value of ω in server $inprogress_s$. Thus it follows that $m(\omega,1)_{s,w}.inprogress[\omega].tag = \tau'$ and hence $m(\omega,1)_{s,w}.inprogress[\omega].tag > tag_s$.

For the second part of the proof notice that any tag added in the $inprogress_s$ set of s contains the timestamp of the local tag of s along with the id of the writer and the writer's operation counter. Furthermore, since by Lemma 6.4.2 the tag of a server is monotonically incremented then, when the $\text{rcv}(m)_{w,s}$ event happens, say at state σ, $\sigma[s].tag \geq \max_{\tau\in\sigma[s].inprogress}(\tau)$. Since the new tag entered in the set is $m(\omega,1)_{s,w}.inprogress[\omega].tag = \langle\sigma[s].tag.ts+1,\omega\rangle$ then it follows that $m(\omega,1)_{s,w}.inprogress[\omega].tag > \sigma[s].tag$ and hence $m(\omega,1)_{s,w}.inprogress[\omega].tag > \max_{\tau\in\sigma[s].inprogress}(\tau)$. Therefore $m(\omega,1)_{s,w}.inprogress[\omega].tag$ is the maximum tag in the set. That completes the proof. $\square$

The following lemma shows the uniqueness of each tag in the $inprogress_s$ set of any server $s \in \mathcal{S}$.

Lemma 6.4.7 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a server $s \in \mathcal{S}$ maintains two tags $\tau_1, \tau_2 \in inprogress_s$, such that $\tau_1 = \langle ts_1, w_1, wc_1 \rangle$ and $\tau_2 = \langle ts_2, w_2, wc_2 \rangle$, then $w_1 \neq w_2$ and $ts_1 \neq ts_2$.

Proof. The first part of the lemma, namely that $w_1 \neq w_2$, follows from the fact that the server s adds a new tag for a write operation from w_1 by removing any previous tag in $inprogress_s$ associated with a previous write from w_1. Hence only a single write operation is recorded in the $inprogress_s$ for every writer process $w \in \mathcal{W}$, and thus our claim follows.

Let us assume, for the second part of the lemma, that w.l.o.g server s receives the message from ω_1 before receiving the message from ω_2. Before replying to ω_1, s adds in the $inprogress_s$ set the tag $\tau_1 = \langle tag_s.ts + 1, w_1, wc_1 \rangle$, and sets $tag_s = \tau_1$. The server s repeats the same process for ω_2. Since by Lemma 6.4.2 the local tag tag_s of the server in monotonically non-decreasing, then it follows that $tag_s \geq \tau_1$ when s receives the message from ω_2. Thus, if $\tau_2 = \langle tag_s.ts + 1, w_2, wc_2 \rangle$, then $\tau_2 \geq \langle \tau_1.ts + 1, w_2, wc_2 \rangle$, and hence $\tau_2.ts \geq \tau_1.ts + 1$. So $\tau_2.ts > \tau_1.ts$ and the lemma follows. $\square$

The following lemma proves the monotonicity of the tag variable at any writer.

Lemma 6.4.8 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, $\sigma'[w].tag \geq \sigma[w].tag$ for any writer $w \in \mathcal{W}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ.

Proof. Each writer process w modifies its local tag during its first communication round. When the write-phase1-fix$_w$ event happens for a write operation ω, tag_w becomes equal to either the tag that satisfies the predicate or the maximum tag, both derived from the $m(\omega, 1)_{s,w}.inprogress$ sets found in the reply of every server s to w for ω. So it suffices to show that $\sigma_{inv(\omega)}[w].tag < \min_{s \in Q}(m(\omega, 1)_{s,w}.inprogress[\omega].tag)$, assuming that

all the servers of a quorum $Q \in \mathbb{Q}$, receive messages and reply to w, for ω. Notice that every message sent from w to any server $s \in Q$ (when $\mathsf{send}(m)_{w,s}$ occurs), contains a tag $m(\omega,1)_{w,s}.tag = \sigma_{inv(\omega)}[w].tag$. Since by Lemma 6.4.6, every server $s \in Q$ replies with $m(\omega,1)_{s,w}.inprogress[\omega].tag > m(\omega,1)_{w,s}.tag$ (to any communication round of ω), then $m(\omega,1)_{s,w}.inprogress[\omega].tag > \sigma_{inv(\omega)}[w].tag$ and the claim follows. Furthermore by Lemma 6.4.2 and Lemma 6.4.6 it follows that w contains non-negative timestamps as well. □

Next we show the monotonicity of the tag at each reader process.

Lemma 6.4.9 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, $\sigma'[r].tag \geq \sigma[r].tag$ for any reader $r \in \mathcal{R}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ.

Proof. The tag_r variable at r is modified only if r invokes some read operation ρ and becomes equal to either the maximum tag in the $m(\rho,1)_{s,r}.inprogress$ set that satisfies the read predicate or the maximum $m(\rho,1)_{s,r}.confirmed$ tag obtained from some server s that reply to r for ρ. Notice however that $\sigma_{res(\rho)}[r].tag$ is equal to some $m(\rho,1)_{s,r}.inprogress[\omega].tag$ of some write ω, only if $m(\rho,1)_{s,r}.inprogress[\omega].tag > \max(m(\rho,1)_{s,r}.confirmed)$. So it suffices to show that $\max(m(\rho,1)_{s,r}.confirmed) \geq \sigma_{inv(\rho)}[r].tag$. Assume that all the servers in a quorum $Q \in \mathbb{Q}$ reply to ρ. Since ρ includes its $\sigma_{inv(\rho)}[r].tag$ in every message sent during the event $\mathsf{send}(m)_{r,s}$ to any server $s \in Q$, then by Lemma 6.4.5, s replies with a $m(\rho,1)_{s,r}.confirmed \geq \sigma_{inv(\rho)}[r].tag$. Hence it follows that $\max(m(\rho,1)_{s,r}.confirmed) \geq \sigma_{inv(\rho)}[r].tag$ as well and our claim holds. Also since by Lemma 6.4.2 all the servers reply with a non negative timestamp, then it follows that rdr contains non-negative timestamps as well. □

Lemma 6.4.10 For each process $p \in \mathcal{R} \cup \mathcal{W} \cup \mathcal{S}$, tag_p is monotonically nondecreasing and contains a non-negative timestamp.

Proof. Follows from Lemmas 6.4.2, 6.4.8 and 6.4.9 □

One of the most important lemmas is presented next. The lemma shows that two tags can be found in the servers of an intersection among k quorums only if $k > \frac{n+1}{2}$.

Lemma 6.4.11 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a read/write operation π invoked by p receives replies from a quorum $Q \in \mathbb{Q}$ and observes two tags τ_1 and τ_2 for a write operation $\omega = \langle w, wc \rangle$, s.t. $\tau_1 = \langle ts_1, w, wc \rangle$, $\tau_2 = \langle ts_2, w, wc \rangle$, $ts_1 \neq ts_2$ and τ_1 is propagated in a k-wise intersection, then τ_2 is propagated in at least k-wise intersection as well iff $k > \frac{n+1}{2}$.

Proof. Let $\mathcal{S}_{\tau_1} \subseteq Q$ be a set of servers such that $\forall s \in \mathcal{S}_{\tau_1}$ replies with $m(\pi, 1)_{s,p}.inprogress[\omega].tag = \tau_1$ to π and $\mathcal{S}_{\tau_2} \subseteq Q$ the set of servers such that $\forall s' \in \mathcal{S}_{\tau_2}$ replies with $m(\pi, 1)_{s',p}.inprogress[\omega].tag = \tau_2$ to π. Since both τ_1 and τ_2 are propagated in a k-wise intersection and since every server maintains just a single copy in its $inprogress$ set for ω, then there exists two sets of quorums $\mathbb{Q}_1^k$ and $\mathbb{Q}_2^k$ such that $(\bigcap_{\mathcal{Q} \in \mathbb{Q}_1^k} \mathcal{Q}) \subseteq \mathcal{S}_{\tau_1}$ and $(\bigcap_{\mathcal{Q} \in \mathbb{Q}_2^k} \mathcal{Q}) \subseteq \mathcal{S}_{\tau_2}$ and $(\bigcap_{\mathcal{Q} \in \mathbb{Q}_1^k} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q} \in \mathbb{Q}_2^k} \mathcal{Q}) = \emptyset$. From the fact that $\mathcal{S}_{\tau_1}, \mathcal{S}_{\tau_2} \subseteq Q$, it follows that $Q \in \mathbb{Q}_1^k$ and $Q \in \mathbb{Q}_2^k$. So, $(\bigcap_{\mathcal{Q} \in \mathbb{Q}_1^k} \mathcal{Q}) = (\bigcap_{\mathcal{Q} \in \mathbb{Q}_1^{k-1}} \mathcal{Q}) \cap Q$ and $(\bigcap_{\mathcal{Q} \in \mathbb{Q}_2^k} \mathcal{Q}) = (\bigcap_{\mathcal{Q} \in \mathbb{Q}_2^{k-1}} \mathcal{Q}) \cap Q$, and hence $(\bigcap_{\mathcal{Q} \in \mathbb{Q}_1^k} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q} \in \mathbb{Q}_2^k} \mathcal{Q}) = (\bigcap_{\mathcal{Q} \in \mathbb{Q}_1^{k-1}} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q} \in \mathbb{Q}_2^{k-1}} \mathcal{Q}) \cap Q = \emptyset$. By definition we know that $\mathbb{Q}^i$ is the quorum set that contains i quorums. So the intersection contains at most $k-1+k-1+1 = 2k-1$ quorums. Since we assume an n-wise intersection then the two sets of quorums maintain an empty intersection only if they consist

of more than n quorums. Hence it follows that the intersection is empty if and only if:

$$2k - 1 > n \Leftrightarrow k > \frac{n+1}{2}$$

This completes the proof. $\square$

From the previous lemma we can derive that if the predicate of the writer holds for some tag then this is the only tag that may satisfy the writer's predicate.

Lemma 6.4.12 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if T is the set of tags witnessed by a write operation ω from a writer w, during its first communication round, and $\tau \in T$ a tag that satisfies the writer predicate, then $\nexists \tau' \in T$ such that $\tau' \neq \tau$ satisfies the writer predicate.

Proof. Let us assume to derive contradiction that there exist a pair of tags $\tau, \tau' \in T$ that both satisfy the writer predicate. Furthermore assume that the write operation $cnt(\omega, Q)_w$. According to the predicate a write operation accepts a tag only if $\exists \mathbb{Q}^i \subset \mathbb{Q}$ such that $i \in [0 \ldots \frac{n}{2} - 1]$ and the tag is contained in all the servers $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q\}} \mathcal{Q})$. If the predicate is valid for τ with $i = 0$ then clearly all the servers $s \in Q$ reply with $m(\omega, 1)_{s,w}.inprogress[\omega].tag = \tau$. Thus, the write operation does not observe any server $s' \in Q$ that replies with $m(\omega, 1)_{s',w}.inprogress[\omega].tag = \tau'$ and hence τ' cannot satisfy the predicate, contradicting our assumption.

Note that since we assume n-wise intersections any k-wise intersection ($k < n - 1$) contains an $(k+1)$-wise intersection. So if τ satisfies the predicate with $i < \frac{n}{2} - 1$ it also satisfies it with $i = \frac{n}{2} - 1$. If now τ satisfies the predicate with $|\mathbb{Q}^i| = \frac{n}{2} - 1$ then it follows that there exists an intersection $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q\}} \mathcal{Q})$ such that every $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q\}} \mathcal{Q})$ reply to w with $m(\omega, 1)_{s,w}.inprogress[\omega].tag = \tau$. Since $\mathbb{Q}^i$ is a set of quorums that contains $\frac{n}{2} - 1$ members then $|\mathbb{Q}^i \cup \{Q\}| = \frac{n}{2}$ and thus τ is propagated in an $\frac{n}{2}$-wise intersection. Since $\frac{n}{2}$

is smaller than $\frac{n+1}{2}$ then by Lemma 6.4.11, τ' cannot be propagated in $\frac{n}{2}$-wise intersection and thus can only be propagated in at least $(\frac{n}{2}+1)$-wise intersection. That however means that $\exists \mathbb{Q}^z$ such that $|\mathbb{Q}^z \cup \{Q\}| = \frac{n}{2}+1$, and hence $z = \frac{n}{2}$, and $\forall s' \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q\}} \mathcal{Q})$, $m(\omega,1)_{s',w}.\mathit{inprogress}[\omega].tag = \tau'$. Since the predicate is only satisfied if $z \in [0 \ldots \frac{n}{2}-1]$ then it follows that $\mathbb{Q}^z$ does not satisfy the predicate. That contradicts our assumption and completes our proof. □

Given that a writer will decide on a single tag per write operation we show that each reader associates the same tag as the writer with each written value.

Lemma 6.4.13 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a write operation ω from w witnesses multiple tags and sets $\sigma_{res(\omega)}[w].tag = \tau$, then any read operation ρ from r that returns the value written by ω decides a tag $\sigma_{res(\rho)}[r].tag = \tau = \sigma_{res(\omega)}[w].tag$.

Proof. We proceed in cases and we show that either the read operation returns the value written by ω and $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$ or the case is impossible and thus ρ does not return the value written by ω. Let us assume w.l.o.g. that $cnt(\omega, Q')_w$ and $cnt(\rho, Q)_r$, during their first communication round. There are two cases to consider for the write operation: (1) ω is fast and completes in one communication round, or (2) ω is slow and performs two communication rounds. Let us examine the two cases separately.

Case 1: Here the write operation ω is fast and thus its predicate is valid and completes in a single communication round. Since ω is fast then there is a set $MS = \{s : s \in Q' \wedge \sigma_{res(\omega)}[w].tag = m(\omega,1)_{s,w}.\mathit{inprogress}[\omega].tag\}$ and a set of quorums $\mathbb{Q}^i$ with $0 < i \leq \frac{n}{2}-3$, s.t. $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q'\}} \mathcal{Q}) \subseteq MS$. Every server $s \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q',Q\}} \mathcal{Q})$ replies with a $m(\rho,1)_{s,r}.\mathit{inprogress}[\omega].tag = \sigma_{res(\omega)}[w].tag$ to r for ρ, if s receives messages from

ω before ρ. Otherwise s replies with a tag for an older write operation of w. Since according to the predicate $|\mathbb{Q}^i| \leq \frac{n}{2} - 3$ then the union of the set $|\mathbb{Q}^i \cup \{Q', Q\}| \leq \frac{n}{2} - 1$ (strictly less if $Q = Q'$ or $Q \in \mathbb{Q}^i$). Thus, the intersection $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q', Q\}} \mathcal{Q})$ involves at most $i + 2 \leq \frac{n}{2} - 1$ quorums and hence by Lemma 6.4.11 every tag $\tau' \neq \sigma_{res(\omega)}[w].tag$ is observed by ρ in a k-wise intersection, such that $k > \frac{n}{2} - 1$. Thus, ρ either observes a $\mathbb{Q}^j \subseteq \mathbb{Q}^i \cup Q'$, such that $\forall s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q\}} \mathcal{Q})$ reply with $m(\rho, 1)_{s',r}.\mathit{inprogress}[\omega].tag = \sigma_{res(\omega)}[w].tag$, and hence the predicate is valid for $j \leq i + 1 \leq \frac{n}{2} - 2$ and returns $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$, or since no other tag satisfies its predicate it returns a value of a write $\omega' \neq \omega$. Notice that since the predicate is false for any read operation ρ' preceding or concurrent with ρ then no tag other than $\sigma_{res(\omega)}[w].tag$ is propagated in the confirmed variable of any server associated with the write operation ω. Hence, if ρ returns the value written by ω, then it returns a tag $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$.

Case 2: Here the write operation ω is slow. This may happen in three cases: (a) either the predicate was true with $|\mathbb{Q}^i| = \frac{n}{2} - 2$ or $|\mathbb{Q}^i| = \frac{n}{2} - 1$, or (b) the predicate was false and thus no tag $\tau \in T$ received from a set of servers $MS = \{s : s \in Q' \wedge \tau = m(\omega, 1)_{s,w}.\mathit{inprogress}[\omega].tag\}$ such that $\exists |\mathbb{Q}^i| \leq \frac{n}{2} - 1$ and $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q'\}} \mathcal{Q}) \subseteq MS$.

Case 2a: Here the predicate is true with $|\mathbb{Q}^i| = \frac{n}{2} - 2$ or $|\mathbb{Q}^i| = \frac{n}{2} - 1$. Notice that the read operation ρ may observe the tag $\sigma_{res(\omega)}[w].tag$ in the intersection $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q', Q\}} \mathcal{Q})$. Thus the set $|\mathbb{Q}^j| \leq |\mathbb{Q}^i \cup \{Q'\}|$ which in the first case it would be $j \leq \frac{n}{2} - 1$ and in the second case $j \leq \frac{n}{2}$. We should consider the two cases for $\mathbb{Q}^i$ separately. Notice that since ω does not modify the inprogress set during its second communication round then the read ρ observes the

same values in that set no matter if it succeeds the first or second communication round of ω. So our claims are valid for both cases.

Case 2a(i): Here $\sigma_{res(\omega)}[w].tag$ is propagated in the servers $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q'\}} \mathcal{Q})$, for $i = \frac{n}{2} - 2$. Since the value $\sigma_{res(\omega)}[w].tag$ is sent by any server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q', Q\}} \mathcal{Q})$ to ρ, then there are three possible cases for Q and $\mathbb{Q}^j$:

1) $Q = Q' \Rightarrow j = i = \frac{n}{2} - 2$,

2) $Q \in \mathbb{Q}^i \Rightarrow j = |\mathbb{Q}^i - Q| = \frac{n}{2} - 3$,

3) $Q \notin \mathbb{Q}^i \cup \{Q'\} \Rightarrow j = |\mathbb{Q}^i \cup \{Q'\}| = \frac{n}{2} - 1$

By Lemma 6.4.11 it follows that for any of the above cases, any tag $\tau' \neq \sigma_{res(\omega)}[w].tag$ may be propagated in a k-wise intersection, such that $k > \frac{n}{2} + 1$ and thus $j > \frac{n}{2}$ for such a tag.

In the first two cases the predicate of ρ holds since $j \leq \frac{n}{2} - 2$, and thus ρ returns $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$ for ω. It remains to examine the third case where $j = |\mathbb{Q}^i \cup \{Q'\}| = \frac{n}{2} - 1$. In this case, $|\mathbb{Q}^j \cup \{Q\}| = j + 1 = \frac{n}{2}$, and thus by Lemma 6.4.11, none of the tags assigned to ω will satisfy the predicate. So ρ returns the value of ω if it observers $\max_{s \in Q}(m(\rho, 1)_{s,r}.confirmed) = \langle maxTS, \omega \rangle$, and hence $\langle maxTS, \omega \rangle = \sigma_{res(\omega)}[w].tag$. Let $s \in Q$ be the server that replied to ρ with $m(\rho, 1)_{s,r}.confirmed = \langle maxTS, \omega \rangle$. Server s sets its confirmed tag to $\langle maxTS, \omega \rangle$ if it receives one of the following messages: (a) a W message for a write ω' such that $\omega \to \omega'$, (b) a RP from a second communication round of ω, (c) a RP from the second communication round of a read operation ρ' that returns a tag $\langle maxTS, \omega \rangle$, or (d) a R from a read operation ρ'' that already returned $\langle maxTS, \omega \rangle$. If (a) or (b) is true, and since the writer propagates the tag it returns in any of those messages,

then $\langle maxTS, \omega \rangle = \sigma_{res(\omega)}[w].tag$. Thus, we are down to the case that some reads propagated the tag in the confirmed variable. Since both ρ' and ρ'' should precede or be concurrent to ρ then they are also concurrent or succeed the first communication round of ω. So, either they observed (as ρ) the tag $\sigma_{res(\omega)}[w].tag$ in a set of quorums $|\mathbb{Q}^j| \leq \frac{n}{2} - 1$, or no tag for ω satisfies their predicate. Since, $\sigma_{res(\omega)}[w].tag$ is the only tag that may satisfy their predicate, then both reads must propagate $\langle maxTS, \omega \rangle = \sigma_{res(\omega)}[w].tag$. So, it follows that $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$ in this case as well.

Case 2a(ii): Let us assume in this case that the write operation received $\sigma_{res(\omega)}[w].tag$ from every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q'\}} \mathcal{Q})$, and $i = \frac{n}{2} - 1$. With similar reasoning as in Case 2a(i) we have the following cases for Q and $\mathbb{Q}^j$ for ρ:

1) $Q = Q' \Rightarrow j = i = \frac{n}{2} - 1$,

2) $Q \in \mathbb{Q}^i \Rightarrow j = |\mathbb{Q}^i - Q| = \frac{n}{2} - 2$,

3) $Q \notin \mathbb{Q}^i \cup \{Q'\} \Rightarrow j = |\mathbb{Q}^i \cup \{Q'\}| = \frac{n}{2}$

Observe again that in the first two cases and by Lemma 6.4.11 no other tag $\tau' \neq \sigma_{res(\omega)}[w].tag$ for ω is propagated in less than $\frac{n}{2}$-wise intersection, and thus τ' does not satisfy the predicate for ρ.

If $Q \in \mathbb{Q}^i$, then the predicate is satisfied with $j = \frac{n}{2} - 2$ for ρ and thus it returns $\sigma_{res(\omega)}[w].tag$. Also if $Q = Q'$ and $j = \frac{n}{2} - 1$ then as showed in case 2a(i) ρ also returns $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$.

So it remains to examine the case where $Q \notin \mathbb{Q}^i \cup \{Q'\}$ and $j = \frac{n}{2}$. It follows that $|\mathbb{Q}^j \cup \{Q\}| = j + 1 = \frac{n}{2} + 1$ and $\forall s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q\}} \mathcal{Q})$, $m(\rho, 1)_{s,r}.inprogress[\omega].tag =$

$\sigma_{res(\omega)}[w].tag$. The read operation ρ may decide to return a tag τ' different from $\sigma_{res(\omega)}[w].tag$, if there are servers $s' \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q\}} \mathcal{Q})$ such that $\tau' \in m(\rho,1)_{s',r}.inprogress$ and $z \leq \frac{n}{2} - 2$. Furthermore, it must be true that $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q\}} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q\}} \mathcal{Q}) = \emptyset$ otherwise a server in that intersection would reply either with $m(\rho,1)_{s,r}.inprogress[\omega].tag = \sigma_{res(\omega)}[w].tag$ or $m(\rho,1)_{s,r}.inprogress[\omega].tag = \tau'$. It suffices then to show that the aforementioned intersection is impossible. Since we know that $z = \frac{n}{2} - 2$ and $j = \frac{n}{2}$ then the intersection $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q}\in\mathbb{Q}^z} \mathcal{Q}) \cap Q$ contains $j + z + 1 = \frac{n}{2} + \frac{n}{2} - 2 + 1 = n - 1$ quorums. Since we assumed n-wise quorum system then the intersection $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q}\in\mathbb{Q}^z} \mathcal{Q}) \cap Q \neq \emptyset$. That will be true even if we assume a smaller $\mathbb{Q}^z$. So, no tag in this case satisfies the predicate of ρ, and thus ρ returns the value written by ω only if it observers $\max_{s\in Q}(m(\rho,1)_{s,r}.confirmed) = \langle maxTS, \omega\rangle$. With similar arguments as in Case 2a(i), we can show that no read or the write operation will propagate a tag different than $\sigma_{res(\omega)}[w].tag$ for ω and thus no server replies with a tag $m(\rho,1)_{s,r}.inprogress[\omega].tag \neq \sigma_{res(\omega)}[w].tag$. Thus if ρ returns ω, then $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$ in this case as well.

Case 2b: In this case the predicate does not hold for the write operation ω. So any tag received from ω was observed in a set $|(\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q'\}} \mathcal{Q})|$ such that $i \geq \frac{n}{2}$. Let us split this case in two subcases:(i) $i = \frac{n}{2}$, and (ii) $i > \frac{n}{2}$.

Case 2b(i): Based on the three cases presented in case 2a for Q and $\mathbb{Q}^j$ then ρ may observe one of the following distributions for $\sigma_{res(\omega)}[w].tag$:

1) $Q = Q' \Rightarrow j = i = \frac{n}{2}$,

2) $Q \in \mathbb{Q}^i \Rightarrow j = |\mathbb{Q}^i - Q| = \frac{n}{2} - 1$,

3) $Q \notin \mathbb{Q}^i \cup \{Q'\} \Rightarrow j = |\mathbb{Q}^i \cup \{Q'\}| = \frac{n}{2} + 1$

Observe that neither of the cases satisfies the predicate for ρ. Furthermore in the first two cases ρ observes $\sigma_{res(\omega)}[w].tag$ in the intersection $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q\}} \mathcal{Q})$ and $|\mathbb{Q}^j \cup Q| = j + 1 \leq \frac{n}{2} + 1$. So according to Lemma 6.4.11 no tag $\tau' \neq \sigma_{res(\omega)}[w].tag$ will be propagated in a k-wise intersection, such that $k < \frac{n}{2} + 1$ and hence τ' will not satisfy the predicate for ρ either.

It remains to examine the third case where $\sigma_{res(\omega)}[w].tag$ is received from every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q\}} \mathcal{Q})$, and $|\mathbb{Q}^j \cup Q| = j + 1 = \frac{n}{2} + 2$. We need to examine if there could be set of quorums $\mathbb{Q}^z$ such that $z \leq \frac{n}{2} - 2$ and every server $s' \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^z \cup \{Q\}} \mathcal{Q})$ replies to ρ with a tag $m(\rho, 1)_{s',r}.inprogress[\omega].tag \neq \sigma_{res(\omega)}[w].tag$ for ω. Such a set would satisfy the predicate for ρ and thus ρ would return $m(\rho, 1)_{s',r}.inprogress[\omega].tag$. This is only possible if $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^j} \mathcal{Q}) \cap (\bigcap_{\mathcal{Q} \in \mathbb{Q}^z} \mathcal{Q}) \cap Q = \emptyset$. Since $j = \frac{n}{2} + 1$ and $z = \frac{n}{2} - 2$ then the intersection consists of $j + z + 1 = \frac{n}{2} + 1 + \frac{n}{2} - 2 + 1 = n$ quorums. Since we assume an n-wise quorum system then it follows that the intersection is not empty. Thus, there exist no tag equal to $m(\rho, 1)_{s',r}.inprogress[\omega].tag$ and hence no tag will satisfy the predicate of ρ in this case. So, ρ will return the value written by ω only if it observes a maximum confirmed tag such that $\max_{s \in Q}(m(\rho, 1)_{s,r}.confirmed) = \langle maxTS, \omega \rangle$. But since the predicate will be false for every read operation, then the first to confirm a tag for ω is the writer w in the second communication round of ω. Since though ω returns $\sigma_{res(\omega)}[w].tag$ then it propagates that tag during its second round to a full quorum. Thus, any read operation that returns the value of ω must observe $\max_{s \in Q}(m(\rho, 1)_{s,r}.confirmed) = \langle maxTS, \omega \rangle = \sigma_{res(\omega)}[w].tag$ and hence returns $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$.

Case 2b(ii): Suppose now that the predicate does not hold for the writer because it observed every tag to be distributed in at least some intersection $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q'\}}\mathcal{Q})$, where $i>\frac{n}{2}$. Let us assume that $i=\frac{n}{2}+1$. By that it follows that the read operation ρ would observe the writer tag in $\mathbb{Q}^j$ in one of the following distributions:

1) $Q=Q'\Rightarrow j=i=\frac{n}{2}+1$,

2) $Q\in\mathbb{Q}^i\Rightarrow j=|\mathbb{Q}^i-Q|=\frac{n}{2}$,

3) $Q\notin\mathbb{Q}^i\cup\{Q'\}\Rightarrow j=|\mathbb{Q}^i\cup\{Q'\}|=\frac{n}{2}+2$

Obviously none of the cases satisfy the predicate of ρ. Furthermore, in the first two cases, by Lemma 6.4.11 and as shown in case 2b(i), no tag assigned to ω satisfies the predicate for ρ and so if ρ returns the value written by ω, it returns the value propagated during ω's second round.

Finally we need to explore what happens in the case where $j=\frac{n}{2}+2$. So we should examine if we can devise a tag for ω such that is distributed in some set of quorums $\mathbb{Q}^z$, such that $z\leq\frac{n}{2}-2$ and every server $s'\in(\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q\}}\mathcal{Q})$ replies to ρ with a tag $\tau'=m(\rho,1)_{s',r}.inprogress[\omega].tag$ such that $\tau'\neq\sigma_{res(\omega)}[w].tag$. Such a set would satisfy the predicate for ρ and thus ρ would return τ'. This is only possible if $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j}\mathcal{Q})\cap(\bigcap_{\mathcal{Q}\in\mathbb{Q}^z}\mathcal{Q})\cap Q=\emptyset$. Since $j=\frac{n}{2}+2$ and $z=\frac{n}{2}-2$ then the intersection consists of $j+z+1=\frac{n}{2}+2+\frac{n}{2}-2+1=n+1$ quorums. Thus, such intersection is possible. If however ρ receives τ' from every server $s'\in(\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q\}}\mathcal{Q})$ and since every server $s\in(\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q,Q'\}}\mathcal{Q})$ replies to ω with a tag $m(\omega,1)_{s,w}.inprogress[\omega].tag=\tau'$ before replying to ρ, then there are three possible distributions for the write operation ω for τ' as observed in the replying quorum $\mathbb{Q}^i$:

1) $Q'=Q\Rightarrow i=z=\frac{n}{2}-2$,

2) $Q' \in \mathbb{Q}^z \Rightarrow i = |\mathbb{Q}^z - Q'| = \frac{n}{2} - 3$,

3) $Q' \notin \mathbb{Q}^z \cup \{Q\} \Rightarrow i = |\mathbb{Q}^z \cup \{Q\}| = \frac{n}{2} - 1$

This however shows that in any case the predicate for ω should have been true for the tag τ'. But this contradicts our initial assumption that the predicate for ω is false and hence such a case is impossible. Thus, no read operation observes a different tag τ' that satisfies its predicate and so every read ρ that returns the value of ω must observe $\max_{s \in Q}(m(\rho,1)_{s,r}.confirmed) = \langle maxTS, \omega \rangle = \sigma_{res(\omega)}[w].tag$. As shown in case 2b(i) $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$. $\square$

Since Lemma 6.4.13 shows that every read operation returns the same tag for the same write operation then from this point onwards we can say that different tags represent different write operations. This is presented formally by the following corollary:

Corollary 6.4.14 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if two read operations ρ and ρ' return tags $\sigma_{res(\rho)}[*].tag = \sigma_{res(\omega)}[*].tag$ and $\sigma_{res(\rho')}[*].tag = \sigma_{res(\omega')}[*].tag$ respectively, then either $\sigma_{res(\rho)}[*].tag = \sigma_{res(\rho')}[*].tag$ and $\omega = \omega'$ or $\sigma_{res(\rho)}[*].tag \neq \sigma_{res(\rho')}[*].tag$ and $\omega \neq \omega'$.

Lemma 6.4.15 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a server $s \in \mathcal{S}$ replies with a $m(\omega,1)_{s,w}.inprogress[\omega].tag$ to the write operation ω from w, then s replies to any subsequent message from an operation π from p with $m(\pi)_{s,p}.inprogress$, s.t. $\max_{\tau \in m(\pi)_{s,p}.inprogress}(\tau) \geq m(\omega,1)_{s,w}.inprogress[\omega].tag$ if π is a read and $\max_{\tau \in m(\pi)_{s,p}.inprogress}(\tau) > m(\omega,1)_{s,w}.inprogress[\omega].tag$ if π is a write.

Proof. If π is a write operation from a process w' then by Lemma 6.4.6 s replies to π with $m(\pi,1)_{s,w'}.inprogress[\pi].tag = \max_{\tau \in m(\pi)_{s,w'}.inprogress}(\tau)$.

But before s adds $m(\pi,1)_{s,w'}.inprogress[\pi].tag$ in $inprogress_s$, according again to Lemma 6.4.6, $m(\pi,1)_{s,w'}.inprogress[\pi].tag$ was greater than any tag in that set. Since s also added $m(\omega,1)_{s,w}.inprogress[\omega].tag$ before replying to ω then it follows that $m(\pi,1)_{s,w'}.inprogress[\pi].tag = \max_{\tau \in m(\pi)_{s,w'}.inprogress}(\tau) > m(\omega,1)_{s,w}.inprogress[\omega].tag$.

If π is a read operation from r then by algorithm SFW server s receives either a R or RP message. In none of those cases s updates its $inprogress_s$ set. By Lemma 6.4.6 when server s replied to ω, $m(\omega,1)_{s,w}.inprogress[\omega].tag = \max_{\tau \in m(\omega)_{s,w}.inprogress}(\tau)$. Thus if s did not receive any write message between the message from ω and π then the operation observes $\max_{\tau \in m(\pi)_{s,r}.inprogress}(\tau) = m(\pi,1)_{s,r}.inprogress[\omega].tag$. Otherwise, with the combination of the first part of this proof, π observes $\max_{\tau \in m(\pi)_{s,r}.inprogress}(\tau) > m(\omega,1)_{s,w}.inprogress[\omega].tag$. Hence our claim follows. □

The next lemma shows that the tag returned by a read operation is larger than the maximum confirmed tag received by the read.

Lemma 6.4.16 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if a read operation ρ from r receives a confirmed tag $m(\rho,1)_{s,r}.confirmed$ from a server s, then $\sigma_{res(\rho)}[r].tag \geq m(\rho,1)_{s,r}.confirmed$.

Proof. Let the read operation ρ receive replies from the servers in Q. By the algorithm a read operation returns either the $\max_{s' \in Q}(m(\rho,1)_{s',r}.confirmed)$ or the maximum tag τ that satisfies predicate **PR**. Notice that if $\max_{s' \in Q}(m(\rho,1)_{s',r}.confirmed) \geq \tau$ then the reader does not evaluate the predicate but rather returns $\sigma_{res(\rho)}[r].tag = \max_{s' \in Q}(m(\rho,1)_{s,r}.confirmed)$ in one or two communication rounds. Since ρ returns

either $\sigma_{res(\rho)}[r].tag = \max_{s'\in Q}(m(\rho,1)_{s',r}.confirmed) \geq m(\rho,1)_{s,r}.confirmed$ or $\sigma_{res(\rho)}[r].tag = \tau > \max_{s'\in Q}(m(\rho,1)_{s',r}.confirmed$, then in either case $\sigma_{res(\rho)}[r].tag \geq m(\rho,1)_{s,r}.confirmed$. $\square$

Next we show that SFW satisfies property **TG2**.

Lemma 6.4.17 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if the read_r event of a read operation ρ from reader $r \in \mathcal{R}$ succeeds the write_w event of a write operation ω from $w \in \mathcal{W}$ in an execution ξ then, $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$.

Proof. Let assume that every server in the quorums $Q', Q'' \in \mathbb{Q}$ (not necessarily $Q' \neq Q''$) receives the messages for the first and the second (if any) communication rounds of the write operation ω respectively and reply to those messages. Also let the servers in Q reply to the first communication round of ρ operation, not necessarily different than Q' or Q''. Moreover let $T = \{\langle *, \omega\rangle : s \in Q' \wedge \langle *, \omega\rangle \in m(\omega)_{s,w}.inprogress\}$ be the set of tags witnessed by ω during its first communication round. Notice that either: (1) $\sigma_{res(\omega)}[w].tag = \tau$ such that $\tau \in T$ and its distribution satisfies the predicate **PW**, or (2) $\sigma_{res(\omega)}[w].tag = \max_{s\in Q'}(m(\omega,1)_{s,w}.inprogress[\omega].tag)$, otherwise. We should investigate these two cases separately. The read operation returns a tag $\sigma_{res(\rho)}[r].tag$ equal to either the $\max_{s\in Q}(m(\rho,1)_{s,r}.confirmed)$ or the maximum tag in $\bigcup_{s\in Q} m(\rho)_{s,r}.inprogress$ that satisfies predicate **PR**. Therefore if $\max_{s\in Q}(m(\rho,1)_{s,r}.confirmed) \geq \sigma_{res(\omega)}[w].tag$ or $\exists\tau \in \bigcup_{s\in Q} m(\rho)_{s,r}.inprogress$ s.t. $\tau > \sigma_{res(\omega)}[w].tag$ that satisfies **PR** then ρ returns $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$. Also notice that if w invokes a write operation ω' such that $\omega \to \omega'$ then by Lemmas 6.4.4, 6.4.6 and 6.4.5 it follows that every server receiving messages from ω' will reply to ρ with $m(\rho,1)_{s,r}.confirmed \geq \sigma_{res(\omega)}[w].tag$ since w will include

$\sigma_{res(\omega)}[w].tag$ in its next write operation. Thus ρ returns $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$ in this case as well. So it suffices to examine the case where there is no write ω' s.t. $\omega \to \omega'$ and no $\tau' \in \bigcup_{s \in Q} m(\rho)_{s,r}.inprogress$ s.t. $\tau' \geq \sigma_{res(\omega)}[w].tag$ and τ' satisfies the predicate for the read operation ρ.

Case 1: Observe that by Lemma 6.4.12, $\sigma_{res(\omega)}[w].tag$ is the only tag that satisfies the writer predicate for the write operation $\omega = \langle w, wc \rangle$. In this case we need to consider the following subcases for the set of quorums $\mathbb{Q}^i$ that satisfies the predicate **PW**: (a) $i < \frac{n}{2} - 2$ and thus the write operation is fast, or (b) $i \in [\frac{n}{2} - 2, \frac{n}{2} - 1]$ and thus the write operation is slow. Notice that by Lemma 6.4.15 it follows that every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q', Q\}} \mathcal{Q})$ replies to ρ with $m(\rho)_{s,r}.inprogress$ that contains a tag $\tau \geq \sigma_{res(\omega)}[w].tag$. Since we only examine the cases where no s receives messages from a write ω' from w s.t. $\omega \to \omega'$, thus it must hold that $m(\rho)_{s,r}.inprogress = \sigma_{res(\omega)}[w].tag$.

Case 1a: This is the case where the write operation is fast and hence $i < \frac{n}{2} - 2$ and every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^i \cup \{Q'\}} \mathcal{Q})$ replies with $\sigma_{res(\omega)}[w].tag \in m(\omega)_{s,w}.inprogress$ to ω. The read operation ρ will witness the tag $\sigma_{res(\omega)}[w].tag$ from the servers in $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q\}} \mathcal{Q})$ where Q and $\mathbb{Q}^j$ may be as follows:

1) $Q = Q' \Rightarrow j = i < \frac{n}{2} - 2$,

2) $Q \in \mathbb{Q}^i \Rightarrow j = |\mathbb{Q}^i - Q'| \leq \frac{n}{2} - 3$,

3) $Q \notin \mathbb{Q}^i \cup \{Q'\} \Rightarrow j = |\mathbb{Q}^i \cup \{Q'\}| \leq \frac{n}{2} - 2$

In any case $j \leq \frac{n}{2} - 2$ and thus the predicate **PR** is valid for the ρ for $\sigma_{res(\omega)}[w].tag$. Hence, ρ returns $\sigma_{res(\rho)}[r].tag = \sigma_{res(\omega)}[w].tag$ in one or two communication rounds.

Case 1b: This is the case where $i \in [\frac{n}{2} - 2, \frac{n}{2} - 1]$ and thus the predicate holds for ω, but ω proceeds to a second communication round before completing. During its second round, ω propagates the tag $\sigma_{res(\omega)}[w].tag$ to a complete quorum say Q''. Since $\omega \rightarrow \rho$ and by Lemma 6.4.5, then any server $s \in Q \cap Q''$ replies to ρ with a $m(\rho, 1)_{s,r}.confirmed \geq \sigma_{res(\omega)}[w].tag$. Thus by Lemma 6.4.16 $\sigma_{res(\rho)}[r].tag \geq m(\rho, 1)_{s,r}.confirmed$, and hence $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$.

Case 2: In this case the predicate does not hold for ω. Thus the writer discovers the maximum tag among the ones it receives from the servers and propagates that to a full quorum say Q'', not necessarily different from Q or Q'. It follows that by Lemma 6.4.3 ρ will receive a $m(\rho, 1)_{s,r}.confirmed \geq \sigma_{res(\omega)}[w].tag$ from any server $s \in Q' \cap Q''$. Thus by Lemma 6.4.16 ρ returns $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$ in this case as well. $\square$

The remaining Lemmas shows that SFW satisfies **TG3** and **TG4**.

Lemma 6.4.18 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if ω and ω' are two write operations from the writers w and w' respectively, such that $\omega \rightarrow \omega'$ in ξ, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

Proof. First consider the case where $w = w'$ and thus ω and ω' are two subsequent writes of the same writer. It is easy to see by Lemmas 6.4.8 and 6.4.4 that $\sigma_{res(\omega)}[w].tag > \sigma_{res(\omega')}[w].tag$ since the tag of the writer is monotonically increasing. So for the rest of the proof we focus on the case where ω and ω' are invoked from two different writers $w \neq w'$. Let us assume that every server in the quorums $Q', Q'' \in \mathbb{Q}$ (not necessarily $Q' \neq Q''$) receives the messages for first and the second (if any) communication rounds of the write operation ω respectively and reply to those messages, and let Q be the quorum that replies to the first communication round of ω''s

operation, not necessarily different than Q' or Q''. Notice here that since ω' decides about the tag $\sigma_{res(\omega')}[w'].tag$ in its first communication round, then it suffices to examine ω''s first communication round alone. Moreover let $T_1 = \{\langle *, \omega\rangle : s \in Q' \wedge \langle *, \omega\rangle \in m(\omega)_{s,w}.\mathit{inprogress}\}$ be the set of tags witnessed by ω during its first communication round and T_2 the respective set of tags for ω'. Notice that either: (1) $\sigma_{res(\omega)}[w].tag = \tau$ such that $\tau \in T_1$ and its distribution satisfies the predicate **PW**, or (2) $\sigma_{res(\omega)}[w].tag = \max_{s\in Q'}(m(\omega,1)_{s,w}.\mathit{inprogress}[\omega].tag)$, otherwise. We now study these two cases individually.

Case 1: This is the case where the predicate **PW** holds for ω. Thus according to the predicate there exists some set of quorums $|\mathbb{Q}^i| \leq \frac{n}{2} - 1$ such that $\forall s \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q'\}} \mathcal{Q})$, $m(\omega,1)_{s,w}.\mathit{inprogress}[\omega].tag = \sigma_{res(\omega)}[w].tag$. From the predicate we can see that if $n \leq 4$ then $i \in [0,1]$. So we can split this case into two subcases: (a) $n > 4$, and (b) $n \leq 4$.

Case 1a: Here we assume that $n > 4$ and thus the predicate may be satisfied with $i \leq \frac{n}{2} - 1$ and $\frac{n}{2} - 1 > 0$. From the monotonicity of the servers (Lemma 6.4.2) and from Lemmas 6.4.6 and 6.4.7, it follows that every server $s' \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q',Q\}} \mathcal{Q})$ replies with a $m(\omega',1)_{s',w'}.\mathit{inprogress}[\omega'].tag > m(\omega,1)_{s',w}.\mathit{inprogress}[\omega].tag$ and thus $m(\omega',1)_{s',w'}.\mathit{inprogress}[\omega'].tag > \sigma_{res(\omega)}[w].tag$. There are three subcases for Q: (i) $Q = Q'$, (ii) $Q \in \mathbb{Q}^i$, or (iii) $Q \notin \mathbb{Q}^i \cup \{Q'\}$. If one of the first two cases is true then ω' observes a set of quorums $|\mathbb{Q}^z| \leq \frac{n}{2} - 1$ such that every server $s' \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^z\cup\{Q\}} \mathcal{Q})$ replies with a tag greater than $\sigma_{res(\omega)}[w].tag$. Since $|\mathbb{Q}^z \cup \{Q\}| = z + 1 \geq \frac{n}{2}$, then according to Lemma 6.4.11 no tag $\tau' < \sigma_{res(\omega)}[w].tag$ is propagated in an intersection $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^d\cup\{Q\}} \mathcal{Q})$ such that $d \leq \frac{n}{2}$. Thus, no such tag satisfies predicate **PW** for ω'. It follows that ω' returns a tag $\sigma_{res(\omega')}[w'].tag$ either because $\sigma_{res(\omega')}[w'].tag \in m(\omega')_{s,w'}.\mathit{inprogress}$, and $s \in (\bigcap_{\mathcal{Q}\in C\cup\{Q\}} \mathcal{Q})$ and **PR**

is satisfied or $\sigma_{res(\omega')}[w'].tag = \max_{s\in Q}(m(\omega',1)_{s,w'}.inprogress[\omega'].tag)$. In both cases $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

It remains to investigate the subcase (iii) where $Q \notin \mathbb{Q}^i \cup \{Q'\}$. If $i \leq \frac{n}{2} - 2$ then ω' observes a set of quorums $\mathbb{Q}^z \leq \frac{n}{2} - 1$ and the proof is similar as in cases (i) and (ii). If however $i = \frac{n}{2} - 1$ then before ω completes it performs a second communication round and propagates $\sigma_{res(\omega)}[w].tag$ to a full quorum Q''. But every server $s \in Q''$ that receives this message sets its local tag to $tag_s = \sigma_{res(\omega)}[w].tag$ if $\sigma_{res(\omega)}[w].tag > tag_s$; otherwise they do not update their tag. Thus, every server $s \in Q''$ contains a tag $tag_s \geq \sigma_{res(\omega)}[w].tag$ when ω completes. Since by Lemma 6.4.2 the local tag of a server is monotonically increasing, then by Lemmas 6.4.6 and 6.4.7, every server $s \in Q \cap Q''$ reply with $m(\omega',1)_{s,w'}.inprogress[\omega'].tag > \sigma_{res(\omega)}[w].tag$ to ω'. So, $|\mathbb{Q}^z| = |\{Q''\}| = 1$. Since we assume that $n > 4$ then $z \leq \frac{n}{2} - 1$ and hence as before and by Lemma 6.4.11 there cannot exist tag $\tau' < \sigma_{res(\omega)}[w].tag$ that satisfies the predicate for ω'. Thus in this case $\sigma_{res(\omega')}[w'].tag = m(\omega',1)_{s,w'}.inprogress[\omega'].tag$ for some $s \in Q \cap Q''$ and hence $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

Case 1b: Here $n \leq 4$. In this case it follows that the predicate is valid for ω with $i \in [0,1]$. If the predicate is valid for $i = 0$ then it follows that ω receives $\sigma_{res(\omega)}[w].tag$ from all the servers in Q' while if $i = 1$ it receives that tag from a pairwise intersection. Notice that the predicate for ω holds for $|\mathbb{Q}^i| = 1$ only in the case where $n = 4$ and with $|\mathbb{Q}^i| = 0$ for $n \leq 3$. Thus in any case $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^i\cup\{Q,Q'\}} \mathcal{Q}) \neq \emptyset$. Hence in case the predicate does not hold for ω' and returns the $\max_{s\in Q}(m(\omega',1)_{s,w'}.inprogress[\omega'].tag)$ then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$. So it remains to explore the two cases where the predicate holds for ω'.

If the predicate for ω holds with $|\mathbb{Q}^i| = 0$ then it follows that all the servers $s \in Q \cap Q'$ reply, by Lemmas 6.4.6 and 6.4.7, to ω' with $m(\omega', 1)_{s,w'}.inprogress[\omega'].tag > m(\omega, 1)_{s,w}.inprogress[\omega].tag$ and thus greater than $\sigma_{res(\omega)}[w].tag$. Notice that for ω' the predicate may also hold for a quorum set $|\mathbb{Q}^z| \in [0, 1]$. If the predicate for ω' holds with $z = 0$, then it follows that every server $s \in Q$ replies with $m(\omega', 1)_{s,w'}.inprogress[\omega'].tag = \sigma_{res(\omega')}[w'].tag$. Since every servers $s \in Q \cap Q'$ replies with $m(\omega', 1)_{s,w'}.inprogress[\omega'].tag > \sigma_{res(\omega)}[w].tag$, then it follows that every server $s \in Q$ replies with that same tag, and hence $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$. Otherwise, if $z = 1$, let us assume to derive contradiction that $\mathbb{Q}^z = \{Q'''\}$ for $Q''' \neq Q', Q$, and every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^z \cup \{Q\}} \mathcal{Q}) = Q''' \cap Q$ reply to ω' with a $\tau' < \sigma_{res(\omega)}[w].tag$. Since $\tau' < \sigma_{res(\omega)}[w].tag$, then it must be the case that $(Q''' \cap Q) \cap (Q' \cap Q) = \emptyset$. Since we assume $z = 1$ it follows that $n = 4$ and hence this is impossible. Thus the predicate may only hold in this case for $\mathbb{Q}^z = \{Q'\}$ and for a tag obtained by the servers in $Q' \cap Q$ and hence $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

If the predicate for ω holds with $|\mathbb{Q}^i| = 1$ then ω performs a second communication round propagating $\sigma_{res(\omega)}[w].tag$ to a full quorum, say Q''. Thus every server $s \in Q \cap Q''$ replies by Lemma 6.4.6 with a tag $m(\omega', 1)_{s,w'}.inprogress[\omega'].tag > \sigma_{res(\omega)}[w].tag$. Since a full intersection replies to ω' with $m(\omega', 1)_{s,w'}.inprogress[\omega'].tag > \sigma_{res(\omega)}[w].tag$ then following similar analysis as in the previous case (and by Lemma 6.4.11) we can show that there cannot exist tag $\tau' < \sigma_{res(\omega)}[w].tag$ to satisfy ω''s predicate. Thus ω' retruns $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

Case 2: In this case the predicate does not hold for ω and thus proceeds to a second communication round propagating a tag $\sigma_{res(\omega)}[w].tag = \max s \in Q(m(\omega,1)_{s,w}.inprogress[\omega].tag)$ to a full quorum, say Q''. Since every server $s \in Q \cap Q''$ replies by Lemma 6.4.6 with a tag $m(\omega',1)_{s,w'}.inprogress[\omega'].tag > \sigma_{res(\omega)}[w].tag$ then by Lemma 6.4.11 and following similar analysis as in Case 1b, we can show that there cannot exist tag $\tau' < \sigma_{res(\omega)}[w].tag$ to satisfy ω''s predicate. Thus ω' retruns $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$ in this case as well. □

Lemma 6.4.19 In any execution $\xi \in goodexecs(\text{SFW}, \mathbb{Q})$, if ρ and ρ' are two read operations from the readers r and r' respectively, such that $\rho \to \rho'$ in ξ, then $\sigma_{res(\rho')}[r].tag \geq \sigma_{res(\rho)}[r'].tag$.

Proof. Let us assume w.lo.g. that ρ to $scnt(Q',\rho)_r$ and $scnt(Q'',\rho)_r$ (not necessarily different than Q') during its first and second communication round respectively. Moreover let ρ' to $scnt(Q,\rho')_{r'}$ during its first communication round. Notice here that since a read operation decides on the tag that it returns when read-phase1-fix happens then we only need to investigate the first communication round of ρ'. Let us first consider the case where the two read operations are performed by the same reader, i.e. $r = r'$. In this case r will enclose in every message sent out a tag greater or equal to $\sigma_{res(\rho)}[r].tag$ Thus every server $s \in Q$, by Lemma 6.4.5, replies to ρ' with $m(\rho',1)_{s,r'}.confirmed \geq \sigma_{res(\rho)}[r].tag$. Thus by Lemma 6.4.16 ρ' returns a tag $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.

So it remains to investigate the case where $r \neq r'$. Notice that if ρ proceeds to a second communication round, either because the predicate holds for $|\mathbb{Q}^j| = \frac{n}{2} - 2$ or not enough confirmed tags where received, then ρ propagates $\sigma_{res(\rho)}[r].tag$ in Q'' before completing. By Lemmas 6.4.5 and 6.4.16 it follows that every server $s \in Q'' \cap Q$ replies to ρ' with a

$m(\rho', 1)_{s,r'}.confirmed \geq \sigma_{res(\rho)}[r_1].tag$ and thus ρ' returns $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$ in one or two communication rounds. Thus we left to explore the case where ρ is fast and returns in a single communication round. This may happen in two cases: (1) predicate **PR** holds for ρ with $j \leq \frac{n}{2} - 3$ or (2) ρ returns the maximum confirmed tag which he observed in a k-intersection with $k \leq n - 1$. Let us examine those two cases.

Case 1: In this case ρ returns $\sigma_{res(\rho)}[r].tag$ because it receives that tag from every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q'\}} \mathcal{Q})$, s.t. $j \leq \frac{n}{2} - 3$ and $\sigma_{res(\rho)}[r].tag = m(\rho, 1)_{s,r}.inprogress[\omega].tag$ for some write operation $\omega = \langle w, wc \rangle$ from writer w. Thus by Lemma 6.4.15 every server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q', Q\}} \mathcal{Q})$, replies to ρ' with a $m(\rho', 1)_{s,r'}.inprogress[\omega'].tag \geq \sigma_{res(\rho)}[r].tag$ as the tag for a write ω' from the writer w. So there are two subcases to consider: (a) there exists server $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q', Q\}} \mathcal{Q})$ such that replies with $m(\rho', 1)_{s,r'}.inprogress[\omega'].tag > \sigma_{res(\rho)}[r].tag$, and (b) all servers in $(\bigcap_{\mathcal{Q} \in \mathbb{Q}^j} \mathcal{Q}) \cap Q' \cap Q$ reply with $m(\rho', 1)_{s,r'}.inprogress[\omega'].tag = \sigma_{res(\rho)}[r].tag$.

Case 1a: If there exists $s \in (\bigcap_{\mathcal{Q} \in \mathbb{Q}^j \cup \{Q', Q\}} \mathcal{Q})$ such that $m(\rho', 1)_{s,r'}.inprogress[\omega'].tag > \sigma_{res(\rho)}[r].tag$, then it follows that the $m(\rho', 1)_{s,r'}.inprogress[\omega'].tag > m(\rho, 1)_{s,r}.inprogress[\omega].tag$ and thus writer w performed a write ω' such that $\omega \to \omega'$. But according to the algorithm the message that w sent to s for ω' contains a tag $\tau \geq \sigma_{res(\omega)}[w].tag$. Hence by Lemma 6.4.5 s replies with $m(\omega, 1)_{s,w}.confirmed \geq \tau$ to ω and thus, by monotonicity of the confirmed tag (Lemma 6.4.3), s replies with $m(\rho', 1)_{s,r'}.confirmed \geq \tau \geq \sigma_{res(\omega)}[w].tag$ to ρ' as well. Therefore from Lemma 6.4.16 ρ' returns a tag $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\omega)}[w].tag$ and thus $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.

Case 1b: If all the servers in $s \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q',Q\}} \mathcal{Q})$ reply with $m(\rho',1)_{s,r'}.inprogress[\omega'].tag = \sigma_{res(\rho)}[r].tag$ then there are three different values for Q to consider: (i) $Q = Q'$, (ii) $Q \in \mathbb{Q}^j$, and (iii) $Q \notin \mathbb{Q}^j \cup Q'$. Since $j \leq \frac{n}{2} - 3$ then in all three cases the predicate **PR** holds for ρ' for at least tag $m(\rho',1)_{s,r'}.inprogress[\omega'].tag$ and with a quorum set $|\mathbb{Q}^z| \leq \frac{n}{2} - 2$. Thus ρ' either returns a $m(\rho',1)_{*,r'}.confirmed \geq m(\rho',1)_{s,r'}.inprogress[\omega'].tag$, a tag $m(\rho',1)_{s,r'}.inprogress[*].tag > m(\rho',1)_{s,r'}.inprogress[\omega'].tag$ or $m(\rho',1)_{s,r'}.inprogress[\omega'].tag$. Hence, $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$ in this case as well.

Case 2: In this case ρ is fast and returns in one communication round since he observed a $\sigma_{res(\rho)}[r].tag = m(\rho,1)_{s,r}.confirmed$ tag from every server $s \in (\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q'\}} \mathcal{Q})$ such that $j = n-2$. Since we assume that $\mathbb{Q}$ is an n-wise quorum system then it follows that $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q',Q\}} \mathcal{Q}) \neq \emptyset$ (since $|\mathbb{Q}^j \cup \{Q',Q\}| \leq j+2 = n$), and hence ρ' receives a $m(\rho',1)_{s,r'}.confirmed \geq \sigma_{res(\rho)}[r].tag$ from at least a single server in $(\bigcap_{\mathcal{Q}\in\mathbb{Q}^j\cup\{Q',Q\}} \mathcal{Q})$. Thus by Lemma 6.4.16, ρ' returns $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$ and that completes the proof. □

Theorem 6.4.20 SFW implements a near optimal MWMR atomic read/write register.

Proof. It follows from Lemmas 6.3.6, 6.4.17, 6.4.19, 6.4.18 and 6.4.13. □

Chapter 7

Summary and Future Directions

This dissertation examined the operation latency of atomic read/write memory implementations in failure prone, asynchronous, message passing environments. We researched the existence of such implementations under process crashes in SWMR and MWMR environments. We developed algorithmic solutions that utilize new techniques to provide an efficient solution to this problem. In addition, we studied the implications of the environmental parameters on the operation latency of read and write operations, and examined the conditions that improve the operation latency of those operations. We now provide a summary of the contributions of this thesis and we identify future directions in this research area.

7.1 Summary

This thesis presents results in three topic areas. Based on the results presented in [30], we first examine whether implementations of a SWMR atomic register can support unbounded number of readers while allowing some operations to complete in a single communication round. We define *semifast* implementations that allow writes and all, but a *single complete*

read operation per write, to be fast. By introducing the notion of *Virtual Nodes*, we present the semifast algorithm SF that implements a SWMR atomic register. We show that semifast implementations are possible only if the number of virtual nodes is bounded under $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 2$, and the second round of a read operation communicates with at least $3f + 1$ servers. Furthermore, we investigate whether semifast MWMR atomic register implementations are possible and we obtain a negative answer.

Next, we examine the relation between the latency of read and write operations and the organization of the replica hosts. We show that (semi)fast implementations can be obtained when organizing the replica hosts in arbitrary intersecting sets (i.e., *general quorum systems*), only if there exists a common intersection among those sets. However, this implies that if a replica host in the common intersection fails then no operation will obtain replies from a complete set and thus, no operation will terminate. This violates the termination condition and makes such implementations non-fault-tolerant since they suffer from a *single point of failure*. This finding led to the introduction of *weak-semifast* implementations that allow more than a single slow read per write. To devise a weak-semifast algorithm, we developed a client side decision tool, called *Quorum Views*. The idea of the tool is to examine the distribution of a value in the replies from a specific quorum. Utilizing the tool we obtain algorithm SLIQ that allows some read operations to be fast. SLIQ allows arbitrarily many readers to participate in the service and does not impose any restrictions on the quorum construction of the replica hosts.

The examination of the efficiency of R/W atomic register implementations in the MWMR setting was our third goal. First, we investigate the impact of MWMR setting on the fastness of read and write operations. We discovered that, in a system that deploys an n-wise quorum

system, at most $n-1$ *consecutive* and *quorum shifting* replica modifications can be fast. This includes both read and write operations when they modify the value of the register replicas. Therefore, if both reads and writes modify the register replica value, no more than $|\mathcal{W} \cup \mathcal{R}| > n-1$ clients may participate in a fast implementation. Next, we present two algorithms that enable fast read and write operations in the MWMR setting. First, we generalized the Quorum Views idea to be used in the MWMR setting. The generalized definition was used by algorithm CWFR, to allow slow (or classic) writes and some fast read operations. The algorithm uses quorum views to *iteratively* analyze the value distribution in the replies of a specific quorum, to detect the latest potentially completed write operation. Unlike previous algorithms (i.e., [28, 22]), CWFR allows read operations to be fast even when they are concurrent with write operations and before the value they return is propagated in a complete quorum. Next, to enable fast write operations in the MWMR setting, we proposed the *server side ordering* approach which transfers partial responsibility of the ordering of the values to the servers. Using this technique we managed to obtain algorithm SFW. This algorithm is near optimal in terms of the number of successive fast write operations it allows. In particular, SFW uses an n-wise quorum system and allows $\frac{n}{2}-1$ consecutive, quorum shifting write operations to be fast. This is the *first* atomic register implementation that allows both write and read operations to be fast in the MWMR setting.

7.2 Future Directions

The thesis focused on the study of the operation latency of read/write atomic register implementations in systems that tolerate benign *crash* failures and assume *fix participation* and reliable communication. However, real systems may experience variable participation (*dynamic*

participation), link failures that may lead to the division of the network (*network partitions*), and message alterations and arbitrary process failures (*byzantine failures*). These environmental parameters introduce new challenges in devising efficient, in terms of operation latency, atomic R/W register implementations. We present possible research directions that utilize the results presented in this thesis as the basis for the development of efficient atomic memory implementations in more hostile environments. In particular, Section 7.2.1 deals with environments where system participation may change during the execution of the service and Section 7.2.2 considers systems that cope with byzantine failures. Finally, Section 7.2.3 examines the application of distributed storage in specialized environments.

7.2.1 Dynamism

System participation may change dynamically during the execution of a service if: (i) participants are allowed to join, fail and voluntarily leave the service, and/or (ii) unreliable channels cause network partitions. In this section, we study the implications of dynamic systems on atomic register implementations and propose possible techniques to obtain lower bounds on the operation latency of such implementations.

7.2.1.1 Fast Operations in Dynamic Systems

Dynamic service participation improves the scalability and longevity of the service, as it allows participants to join and fail/leave the service at any point in the execution. As discussed in Chapter 2, solutions designed to handle dynamic participation require high communication demands: participant additions and removals lead eventually to the need to reorganize (reconfigure) the set of replica hosts to include or exclude the new or departed participants.

Algorithms like the ones presented in [34, 68, 66], separate the join/depart and reconfiguration protocols. More recently, [6] combined the two protocols and suggested the tracing of the new system view (service participation) whenever an addition or removal occurred in the system. This categorization boils down to the replica organization that each algorithm utilizes:

(i) Voting: participants need to know the replica hosts, and

(ii) Quorums: participants need to know the replica hosts *and* the replica organization.

Voting techniques eliminate the necessity of having a dedicated entity to decide and propagate the next replica configuration as long as the service participants know the set of replica hosts. However, knowledge of the replica hosts when quorums are used does not imply the knowledge of the next configuration. For this reason, algorithms that use quorums need to introduce a separate service to reorganize the replica hosts into quorums, and propagate the new configuration to the service participants.

We suggest investigation of both directions. On one hand, voting allows reconfiguration-free approaches, but it requires propagation of the set of replica hosts at each node addition or removal. On the other hand quorums will allow inexpensive joins and trade operation-latency during periodic reconfigurations.

Utilizing Voting Strategies.

We will first examine the incorporation of voting strategies to obtain an atomic register implementation. The main research questions we need to examine are:

(a) *How fast a join/remove operation can be?*, and

(b) *How fast a read or write operation can be that is concurrent with a join/removal?*

Note that any read and write operation that detects it is not concurrent with a join/removal may follow the algorithmic approaches proposed for the static environment.

The adoption and combination of techniques presented in both [6] and static environments may help us to answer the above questions. The first goal can be to improve the operation latency of join and departure protocols. Intuitively a join protocol needs at least three rounds: (i) the new participant (joinee) sends join request to an existing service participant (joiner), (ii) the joiner propagates the new system view (known locally and including the joinee) and gathers the latest system view in the same round, and (iii) the joiner sends the join acknowledgment and the system view to the joinee. Merging rounds (i) and (ii) by forcing the joinee to communicate with a set of participants (say a majority) may decrease the latency of the join protocol. Departures on the other hand (assuming that a participant can depart on its own) should include at least a single round similar to round (ii) of the join protocol.

As the second step we need to focus on read and write operations. Each operation may witness (from the received replies) that a join or departure of a participant is in progress. As joins/departures may alter the set of replica hosts, each operation is responsible for discovering the latest replica host membership and communicate with a sufficient number of recent replica hosts. This will guarantee that the operation observes the latest written value. An operation may need to perform multiple rounds to “catch up” with the new joins/departures. The challenge is to reduce the amount of rounds needed for “catching up”. It appears that such procedure is affected by the setting we assume: SWMR or MWMR. By well formedness, only a single write operation (and thus, a single value) may be in progress by the sole writer, in the SWMR setting. Older values have been propagated by a completed write operation. Since the sole writer is the only one who modifies the value of the replica, it may propagate some

"traces/clues" on how many new configurations it encounter along with the value to be written. This could potentially help read operations to discover the latest configuration in fewer rounds. In the MWMR setting multiple writers may perform concurrent write operations. Thus, discovery of the latest replica host configuration becomes even more challenging. Operation latency of such implementations can benefit from: (i) relaxing the failure model (e.g., $f < \frac{|S|}{c}$, for $c > 2$ a constant) and allowing the operations to contact more replica hosts, and/or (ii) restricting the number of participants.

Utilizing Quorum Systems.

The fastness of reconfiguring the replica hosts and propagating the new organization to the service participants is the main concern when using quorums. Also, every read and write operation that is concurrent with a reconfiguration needs to ensure that old and new configuration maintain the latest replica information. Thus, fastness of read/write operations is also affected as an operation may need to perform additional rounds to contact the servers of the latest configuration. Thus, the main challenges we need to address are:

(a) *How fast a quorum system can be reconfigured?*, and

(b) *How fast read/write operation can be during a reconfiguration process?*

Things are simpler when we assume a single reconfigurer. The single reconfigurer imposes a total ordering on the series of configurations (its local ordering). Thus, it may locally obtain the next configuration. To preserve atomicity, a reconfiguration needs to ensure that the latest replica information will be propagated to enough replicas of the new configuration. For this purpose we propose enhancing the role of every reader and writer to assist the reconfigurer

in this task. This will allow a reconfiguration to be faster, but it may require extra rounds from each read or write operation that is concurrent with a reconfiguration. It is essential to expedite the reconfiguration process, since this may allow more reads and writes to be faster. Here techniques from [34, 66], along with those proposed to achieve fast operation in the static environment with quorums may be utilized. Moreover, having access to the order of reconfigurations from the single reconfigurer, may help read/write operations to utilized techniques similar to the ones proposed earlier in this section to predict the latest configuration.

The introduction of multiple reconfigurers improves fault-tolerance but introduces the need of achieving agreement between the reconfigurers on the next configuration to be deployed. This will affect negatively the fastness of a reconfiguration process, since extra rounds will be needed for the agreement protocol. Thus, it is important to diverge from traditional solutions (i.e., [66]) that use consensus to decide on the next configuration. This may affect the flexibility of the system. For instance, the works in [28, 6] assumed a *finite* set of configurations and thus, each read/write was communicating with a single quorum from each possible configuration. So, a challenging task is to design a protocol that will impose a total ordering on the configuration sequence, without utilizing strong primitives like consensus and failure detection. It will be interesting to analyze the latency of such protocols and its effect on the latency of read and write operations. The utilization of some techniques presented for the single reconfigurer may also find application in the multiple reconfigurer setting. Finally, it will be important to examine if restricting the number of participants and the organization of the replica host allows expediting some of the read, write, or reconfiguration operations.

7.2.1.2 Fast Operations in Partitionable Networks

Connectivity failures can split the network into non-communicating groups, called *partitions*. In this section we consider another aspect of dynamic systems that does not depend on the join/departure of network participants, but rather involves channel inability to deliver messages. Let us first provide a formal definition of an unreliable channel similar to the Definition 3.1.1 of reliable channels presented in Section 3.1.2.

Definition 7.2.1 (Unreliable Channel) A channel between $p, p' \in \mathcal{I}$ is **unreliable** in an execution $\phi \in execs(A)$, if for any execution fragment ϕ' of A that extends ϕ one of the following holds:

- $\exists \mathsf{send}(m)_{p,p'}$ event in ϕ and $\nexists$ succeeding $\mathsf{rcv}(m)_{p,p'}$ in $\phi \circ \phi'$ (**message loss**), or
- $\exists \mathsf{rcv}(m)_{p,p'}$ event in ϕ and $\nexists$ preceding $\mathsf{send}(m)_{p,p'}$ in ϕ (**message forging**).

The main results related to information dissemination and consistency in partitionable networks was presented in Section 2.7. Karumanchi et al. [58] focused in implementing a regular register using synchronized clocks to expedite write operations. On the other hand, Dolev et al. [28] demonstrated that it is possible to obtain atomic register implementations in a partitionable ad-hoc mobile network by utilizing a connected focal point infrastructure.

The above results, either focused on weaker consistency semantics or relied on practically expensive broadcasting primitives. So, one may ask: *Is it possible to obtain latency efficient atomic* R/W *register implementations in partitionable networks?*

There is a thin line that divides an environment with dynamic participation – where individual nodes are added and removed – and an environment that allows network partitions –

where groups of nodes are detached. The main difference lies in the perception of the nodes in the group once they are detached from the rest of the network. Individual nodes stop participating in the service because either: (i) they departed voluntarily, or (ii) they do not receive any messages – replies or requests – from any other process in the system, or (iii) they failed. In all three cases the node stops participating in the service. On the other hand when the network is split into some partitions, the nodes within a partition may continue to communicate. Therefore – because of asynchrony and failures – they may assume that they are the only correct processes in the system. So they may take measures – similar to those proposed by [66] – to reorganize the register replica within their partition and recover the execution of the service. Such an approach, however, would affect the consistency of the register replica among the different partitions. So can we preserve consistency among the replicas of the different network partitions?

Consistency between network partitions (or participant groups) was studied in the context of group communication services. Birman and Joseph in [13] suggested the Isis system which designated one of the partitions to be *primary*. Operations are performed on the primary partition while any non-primary partition was “shut-down”. Chandra et al. [19] showed that it is impossible for the participants to agree on a primary partition in an asynchronous message passing system. To overcome this problem Dolev and Malki in [27] introduced Transis which did not rely on a single primary component but rather allowed operations to be performed on all partitions.

Transis seems to be applicable in additive applications. An example given in [27] is a tabulation service. During a merge, lost messages were communicated and the vote tally could be recovered. The order of the messages in such applications is not important as long as we

ensure that all of them are delivered. In contrast, atomic register implementations require operations to be ordered sequentially. Totally-ordered broadcast among all the groups was presented in [37]. This approach allowed the clients to perform an operation in any existing group and held the broadcast service responsible to deliver the message in the same order to all the groups. Atomicity is derived by allowing each participant to apply each ordered request to its local replica copy.

It would be interesting to investigate the number of rounds needed to establish a totally-ordered message broadcast service as suggested in [37]. We also want to examine hybrid approaches that combine techniques presented in the GCSs for consistent message delivery within the groups, and techniques presented to establish atomic consistency among individual processes. Our approaches should utilize methodologies presented in [15, 27, 37] to ensure the reliable delivery of the messages despite participant failures. During merging, reliable delivery of messages will ensure that all the messages exchanged will be propagated to the merged network. Techniques like timestamping (e.g., [9]) can be used to allow participants to order the messages they witness, and impose consistency between the operations in the merged group.

Operation latency needs to be measured over a larger set of operations: (i) Reads and Writes, (ii) Message Broadcasting, and (iii) Group Merging. We may need to study the operation latency of each service individually and then investigate the implications their combination will impose on operation latency.

7.2.2 Byzantine Failures

The thesis focused on systems with crash-prone processes that provide the assurance that every process does not deviate from its program specifications as long as it remains active in

the system. In this section we present possible future directions that consider failures where the processes may exhibit arbitrary behavior. These failures are widely known as *Byzantine Failures* [64]. Our goal will be to study the operation latency of atomic R/W register implementations where the participants may exhibit byzantine behavior. Similar to Definition 3.1.2 for crash failures, byzantine failures can be defined formally as follows:

Definition 7.2.2 (Byzantine Failures) For an algorithm A we define the set of executions $\mathcal{F}_B(A)$ to be a subset of $execs(A)$, such that in any execution $\xi \in \mathcal{F}_B(A)$, for all $p \in \mathcal{I}$, there are zero or more steps $\langle \sigma_k, \mathsf{fail_p}, \sigma_{k+1} \rangle$ that are transitions from a state $\sigma_k[p] \in states(A_p)$ to any arbitrary state $\sigma_{k+1}[p] \in states(A_p)$.

A process p is *byzantine* in an execution $\xi \in \mathcal{F}_B(A)$, if ξ contains a byzantine step for p. A byzantine step $\langle \sigma_{b-1}, \mathsf{fail_p}, \sigma_b \rangle$ of a process p may be the same as a crash step $\langle \sigma_{c-1}, \mathsf{fail_p}, \sigma_c \rangle$ of p, if $\sigma_b[p] = \sigma_c[p]$. If an execution $\xi \in \mathcal{F}_B(A)$ contains only crash steps, then $\xi \in \mathcal{F}_C(A)$ as well and thus, $\mathcal{F}_C(A) \subset \mathcal{F}_B(A)$. Due to the severity of this type of failures, early papers investigated tight lower bounds and introduced algorithms for *safe* and *regular* semantics (e.g., [3, 52, 70]).

Abraham et al. [3] showed a tight lower bound on the communication efficiency of *write* operations. In particular, they showed that in the SWMR model, a write operation needs two rounds when at most $2f + 2b$ register replicas are used; otherwise a single round is sufficient. The work considered f to be the total number of replica host failures, out of which b may be byzantine and the rest may crash. Additionally, this work showed that $2f + b + 1$ register replicas are needed in order to establish a *safe storage* under byzantine failures.

Extending upon this work, Guerraoui and Vucolić [52] studied the operation latency of *read* operations in the SWMR environment under byzantine failures. The authors showed that two rounds for every read operation are necessary for the implementation of *safe storage*, when at most $2f + 2b$ register replicas are used. Interestingly, they showed that when both reads and writes perform two rounds, a *regular* register is possible even under *optimal resilience* where $2f + b + 1$ register replicas are used.

Guerraoui, Levi and Vucolić [51], showed that "lucky" R/W operations of an *atomic* storage implementation may be fast when $2f + b + 1$ register replicas are used. For them, lucky operations are the ones that are not concurrent with any other operation.

Another direction to the solution of the problem was presented by Gerraoui and Vukolić [53]. In this work, the authors introduce a new family of quorum systems, called *Refined Quorum Systems*, that allow some *fast* operations in atomic register implementations under byzantine failures. To achieve fastness, they rely on a synchronization assumption that required each operation to wait for a predefined *timeout interval*.

The results presented by previous works show that it is deifficult to implement *any* consistent semantic under the assumption of byzantine replica hosts. However prior results do not adequately answer the question: *How fast read and write operations of an atomic* R/W *register implementation can be under asynchrony and byzantine failures?*

Byzantine processes may reply with *some* value from the set of values allowed to be written on the register. Adoption of that value by a reader may lead to violations of consistency. To avoid this problem, each correct process needs to collect replies from at least a subset of correct processes. Two strategies are utilized:

1. Detect the byzantine failures, or

2. Collect replies from a set of processes that strictly contains any possible set of byzantine failures

Byzantine failure detection allows correct processes to discard the replies originating from byzantine processes. Authentication is commonly used to facilitate the detection of erroneous behavior. Processes use cryptographic primitives to prevent information forging. However, authentication is computationally expensive. The second strategy collects a set of processes, large enough to contain all byzantine and at least a single correct process. This strategy does not depend on any computational load but it requires knowledge of either of the following parameters: (i) the number of byzantine processes, or (ii) a set of subsets of processes, such that only the processes of a single subset can be byzantine. In the first case a correct process will collect more replies than the number of byzantine processes. On the second case a strict superset of a subset of those sets need to be collected.

To conclude, the regular and safe implementations presented in previous works (e.g., [3, 51, 52, 53, 70]) can provide a basis for the development of atomic register implementations. Allowing operations to perform extra rounds in implementations that guarantee safe and regular semantics, like [52], may lead to atomic consistency. As shown by [3], relaxing fault-tolerance and allowing $2b + 2f + 1$ replica hosts immediately enables single round write operations. Quorums may also be used to improve operation latency, by adopting techniques as the ones presented in [53, 70] or techniques similar to Quorum Views in Section 5.4.1. Obviously direct application of quorum views in an environment that suffers from byzantine failures is impossible: every server in a single intersection may report a faulty value and thus, two different read operations may witness different values in the same set of replica hosts. Finally,

assuming byzantine clients (readers and writers) requires the adoption of authentication techniques (see [73, 70]). Gossip among the servers may allow bypassing the client authentication and preserve value confidence among the servers.

7.2.3 Other Environments

Partially synchronous environments [31] maintain interesting properties and pose a good candidate for a latency-efficient atomic register implementation. Operations may take advantage of the periods of synchrony to collect more information about the replicated register. Such information may allow operations to complete in a single communication round, even when such performance is impossible in the asynchronous model.

Bibliography

[1] Global positioning system (GPS). http://www.gps.gov/.

[2] NS2 network simulator. http://www.isi.edu/nsnam/ns/.

[3] ABRAHAM, I., CHOCKLER, G., KEIDAR, I., AND MALKHI, D. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing 18*, 5 (2006), 387–408.

[4] ABRAHAM, I., CHOCKLER, G., KEIDAR, I., AND MALKHI, D. Wait-free regular storage from byzantine components. *Information Processing Letters 101*, 2 (2007), 60–65.

[5] ABRAHAM, I., AND MALKHI, D. Probabilistic quorums for dynamic systems. *Distributed Computing 18*, 2 (2005), 113–124.

[6] AGUILERA, M. K., KEIDAR, I., MALKHI, D., AND SHRAER, A. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09)* (New York, NY, USA, 2009), ACM, pp. 17–25.

[7] ALISTARH, D., GILBERT, S., GUERRAOUI, R., AND TRAVERS, C. How to solve consensus in the smallest window of synchrony. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 32–46.

[8] AMIR, Y., DOLEV, D., MELLIAR-SMITH, P. M., AND MOSER, L. E. Robust and efficient replication using group communication. Tech. rep., 1994.

[9] ATTIYA, H., BAR-NOY, A., AND DOLEV, D. Sharing memory robustly in message passing systems. *Journal of the ACM 42(1)* (1996), 124–142.

[10] ATTIYA, H., AND WELCH, J. L. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst. 12*, 2 (1994), 91–122.

[11] BABAOGLU, O., DAVOLI, R., GIACHINI, L.-A., AND BAKER, M. G. Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Hawaii International Conference on System Sciences* (Los Alamitos, CA, USA, 1995), IEEE Computer Society, p. 612.

[12] BAZZI, R. A., AND DING, Y. Non-skipping timestamps for byzantine data storage systems. In *Proceedings of 18th International Conference on Distributed Computing (DISC)* (2004), pp. 405–419.

[13] BIRMAN, K., AND JOSEPH, T. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev. 21*, 5 (1987), 123–138.

[14] BIRMAN, K. P. A review of experiences with reliable multicast. *Software: Practice and Experience 29:9* (1999), 741–774.

[15] BIRMAN, K. P., AND JOSEPH, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (1987), 47–76.

[16] BIRMAN, K. P., AND RENESSE, R. V. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.

[17] CACHIN, C., AND TESSARO, S. Asynchronous verifiable information dispersal. In *Distributed Computing, 19th International Conference, DISC2005, Cracow, Poland, September 26-2* (2005), pp. 503–504.

[18] CACHIN, C., AND TESSARO, S. Optimal resilience for erasure-coded byzantine distributed storage. IEEE Computer Society, pp. 115–124.

[19] CHANDRA, T. D., HADZILACOS, V., TOUEG, S., AND CHARRON-BOST, B. On the impossibility of group membership. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1996), ACM, pp. 322–330.

[20] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (1996), 225–267.

[21] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. Raid: high-performance, reliable secondary storage. *ACM Computing Surveys 26*, 2 (1994), 145–185.

[22] CHOCKLER, G., GILBERT, S., GRAMOLI, V., MUSIAL, P. M., AND SHVARTSMAN, A. A. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 100–116.

[23] CHOCKLER, G., KEIDAR, I., GUERRAOUI, R., AND VUKOLIC, M. Reliable distributed storage. *IEEE Computer* (2008).

[24] CORREIA, M., NEVES, N. F., LUNG, L. C., AND VERÍSSIMO, P. Low complexity byzantine-resilient consensus. *Distrib. Comput. 17*, 3 (2005), 237–249.

[25] DAVIDSON, S. B., GARCIA-MOLINA, H., AND SKEEN, D. Consistency in a partitioned network: a survey. *ACM Comput. Surv. 17*, 3 (1985), 341–370.

[26] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (1987), 77–97.

[27] Dolev, D., and Malki, D. The transis approach to high availability cluster communication. *Commun. ACM 39*, 4 (1996), 64–70.

[28] Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceedings of 17th International Symposium on Distributed Computing (DISC)* (2003).

[29] Dolev, S., and Schiller, E. Communication adaptive self-stabilizing group membership service. vol. 14, IEEE Press, pp. 709–720.

[30] Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.

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

[32] Englert, B., Georgiou, C., Musial, P. M., Nicolaou, N., and Shvartsman, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)* (2009), pp. 240–254.

[33] Englert, B., Georgiou, C., Musial, P. M., Nicolaou, N., and Shvartsman, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. Tech. rep., University of Connecticut, 2009.

[34] Englert, B., and Shvartsman, A. A. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)* (2000), pp. 454–463.

[35] Ezhilchelvan, P., Macedo, R., and Shrivastava, S. Newtop: a fault-tolerant group communication protocol. *Distributed Computing Systems, International Conference on* (1995), 0296.

[36] Fan, R., and Lynch, N. Efficient replication of large data objects. In *Distributed algorithms* (Oct 2003), F. E. Fich, Ed., vol. 2848/2003 of *Lecture Notes in Computer Science*, pp. 75–91.

[37] Fekete, A., Lynch, N., and Shvartsman, A. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst. 19*, 2 (2001), 171–216.

[38] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *Journal of ACM 32*, 2 (1985), 374–382.

[39] Garcia-Molina, H., and Barbara, D. How to assign votes in a distributed system. *Journal of the ACM 32*, 4 (1985), 841–860.

[40] Georgiou, C., Kentros, S., Nicolaou, N. C., and Shvartsman, A. A. Analyzing the number of slow reads for semifast atomic read/write register implementations. In *Proceedings Parallel and Distributed Computing and Systems (PDCS09)* (2009), pp. 229–236.

[41] GEORGIOU, C., MUSIAL, P. M., AND SHVARTSMAN, A. A. Long-lived RAMBO: Trading knowledge for communication. *Theoretical Computer Science 383*, 1 (2007), 59–85.

[42] GEORGIOU, C., MUSIAL, P. M., AND SHVARTSMAN, A. A. Developing a consistent domain-oriented distributed object service. *IEEE Transactions of Parallel and Distributed Systems (TPDS) 20*, 11 (2009), 1567–1585. A preliminary version of this work appeared in the proceedings of the 4th IEEE International Symposium on Network Computing and Applications (NCA'05).

[43] GEORGIOU, C., NICOLAOU, N., AND SHVARTSMAN, A. Fault-tolerant semifast implementations for atomic read/write registers. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2006), pp. 281–290.

[44] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.

[45] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 62–79. A preliminary version of this work appeared in the proceedings 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'06).

[46] GIBSON, G. A., AND VAN METER, R. Network attached storage architecture. *Commun. ACM 43*, 11 (2000), 37–45.

[47] GIFFORD, D. K. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles* (1979), pp. 150–162.

[48] GILBERT, S., LYNCH, N. A., AND SHVARTSMAN, A. A. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing 23*, 4 (2010), 225–272.

[49] GRAMOLI, V., ANCEAUME, E., AND VIRGILLITO, A. SQUARE: scalable quorum-based atomic memory with local reconfiguration. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), ACM, pp. 574–579.

[50] GRAMOLI, V., MUSIAL, P. M., AND SHVARTSMAN, A. A. Operation liveness and gossip management in a dynamic distributed atomic data service. In *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems, September 12-14, 2005 Imperial Palace Hotel, Las Vegas, Nevada, US* (2005), pp. 206–211.

[51] GUERRAOUI, R., LEVY, R. R., AND VUKOLIC, M. Lucky read/write access to robust atomic storage. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 125–136.

[52] GUERRAOUI, R., AND VUKOLIĆ, M. How fast can a very robust read be? In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2006), ACM, pp. 248–257.

[53] GUERRAOUI, R., AND VUKOLIĆ, M. Refined quorum systems. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC07)* (New York, NY, USA, 2007), ACM, pp. 119–128.

[54] HAYDEN, M. G. *The ensemble system.* PhD thesis, Ithaca, NY, USA, 1998.

[55] HERLIHY, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[56] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[57] KANJANI, K., LEE, H., AND WELCH, J. L. Byzantine fault-tolerant implementation of a multi-writer regular register. In *Parallel and Distributed Processing Symposium, International* (Los Alamitos, CA, USA, 2009), IEEE Computer Society, pp. 1–8.

[58] KARUMANCHI, G., MURALIDHARAN, S., AND PRAKASH, R. Information dissemination in partitionable mobile ad hoc networks. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 1999), IEEE Computer Society, p. 4.

[59] KONWAR, K. M., MUSIAL, P. M., NICOLAOU, N. C., AND SHVARTSMAN, A. A. Implementing atomic data through indirect learning in dynamic networks. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007), 12 - 14 July 2007, Cambridge, MA, USA* (2007), pp. 223–230.

[60] KONWAR, K. M., MUSIAL, P. M., AND SHVARTSMAN, A. A. Spontaneous, self-sampling quorum systems for ad hoc networks. In *7th International Symposium on Parallel and Distributed Computing (ISPDC 2008), 1-5 July 2008, Krakow, Poland* (2008), pp. 367–374.

[61] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[62] LAMPORT, L. On interprocess communication, part I: Basic formalism. *Distributed Computing 1*, 2 (1986), 77–85.

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

[64] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4* (1982), 382–401.

[65] LYNCH, N. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[66] LYNCH, N., AND SHVARTSMAN, A. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)* (2002), pp. 173–190.

[67] LYNCH, N., AND TUTTLE, M. An introduction to input/output automata. *CWI-Quarterly* (1989), 219–246.

[68] LYNCH, N. A., AND SHVARTSMAN, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

[69] MALKHI, D., AND REITER, M. Unreliable intrusion detection in distributed computations. In *CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations* (Washington, DC, USA, 1997), IEEE Computer Society, p. 116.

[70] MALKHI, D., AND REITER, M. Byzantine quorum systems. *Distributed Computing 11* (1998), 203–213.

[71] MALKHI, D., AND REITER, M. K. Secure and scalable replication in phalanx. In *In Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems* (1998), pp. 51–60.

[72] MALKHI, D., REITER, M. K., WOOL, A., AND WRIGHT, R. N. Probabilistic quorum systems. *Inf. Comput. 170*, 2 (2001), 184–206.

[73] MARTIN, J.-P., ALVISI, L., AND DAHLIN, M. Minimal byzantine storage. In *In Proceedings of the 16th International Symposium on Distributed Computing (DISC* (2002), Springer-Verlag, pp. 311–325.

[74] MITZENMACHER, M., AND UPFAL, E. *Probability and Computing*. Cambridge University Press, 2005.

[75] NAOR, M., AND WOOL, A. The load, capacity, and availability of quorum systems. *SIAM Journal of Computing 27*, 2 (1998), 423–447.

[76] NEIGER, G. A new look at membership services (extended abstract). In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1996), ACM, pp. 331–340.

[77] NEVES, N. F., CORREIA, M., AND VERISSIMO, P. Solving vector consensus with a wormhole. *IEEE Trans. Parallel Distrib. Syst. 16*, 12 (2005), 1120–1131.

[78] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *Journal of ACM 26*, 4 (1979), 631–653.

[79] PARVÉDY, P. R., AND RAYNAL, M. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2004), ACM, pp. 302–310.

[80] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (raid). *SIGMOD Rec. 17*, 3 (1988), 109–116.

[81] PELEG, D., AND WOOL, A. Crumbling walls: A class of high availability quorum systems. In *Proceedings of 14th ACM Symposium on Principles of Distributed Computing (PODC)* (1995), pp. 120–129.

[82] PIERCE, E., AND ALVISI, L. A recipe for atomic semantics for byzantine quorum systems. Tech. rep., University of Texas at Austin, Department of Computer Science, 2000.

[83] RABIN, M. O. Randomized byzantine generals. In *SFCS '83: Proceedings of the 24th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1983), IEEE Computer Society, pp. 403–409.

[84] RENESSE, R. V., BIRMAN, K. P., AND MAFFEIS, S. Horus: A flexible group communications system. *Communications of the ACM 39* (1996), 76–83.

[85] RICCIARDI, A. M. *The group membership problem in asynchronous systems*. PhD thesis, Ithaca, NY, USA, 1993.

[86] SHAO, C., PIERCE, E., AND WELCH, J. L. Multi-writer consistency conditions for shared memory objects. *Distributed Computing* (2003), 106–120.

[87] SHOUP, V. Practical threshold signatures. In *EUROCRYPT* (2000), pp. 207–220.

[88] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (1979), 180–209.

[89] UPFAL, E., AND WIGDERSON, A. How to share memory in a distributed system. *Journal of the ACM 34(1)* (1987), 116–127.

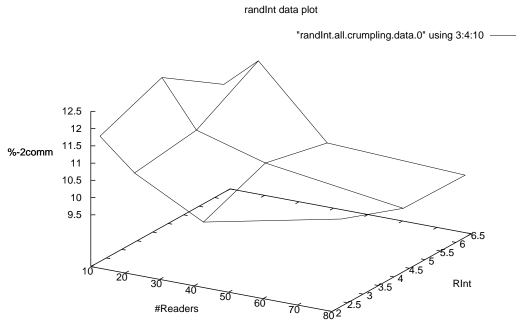
[90] VITANYI, P., AND AWERBUCH, B. Atomic shared register access by asynchronous hardware. In *Proceedings of 27th IEEE Symposium on Foundations of Computer Science (FOCS)* (1986), pp. 233–243.

[91] WANG, X., TEO, Y. M., AND CAO, J. Message and time efficient consensus protocols for synchronous distributed systems. *J. Parallel Distrib. Comput. 68*, 5 (2008), 641–654.

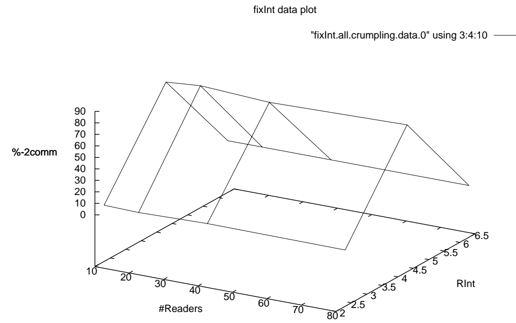
Appendix A

SLIQ Simulation: Plots

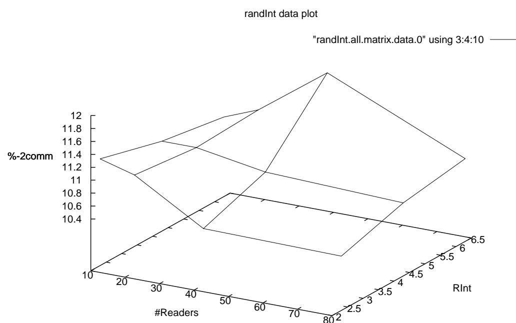
In this section we present all graphs obtained during the simulation of the algorithm in different scenarios. In the figures, $\mathbb{Q}_m$ denotes the use of majority quorums, $\mathbb{Q}_c$ the use of crumbling walls, and $\mathbb{Q}_x$ the use of matrix quorums. Figure 26 demonstrates how the algorithm performs in the simple run scenario, exploiting different quorum systems. In the experiment illustrated by Figures 27 and 28 we consider runs with variable quorum membership. We ran those simulations on the most efficient quorum constructions (i.e., crumbling walls and matrix quorums). Lastly, Figures 29-32 examine the performance of the algorithm under executions with variable failure scenarios.



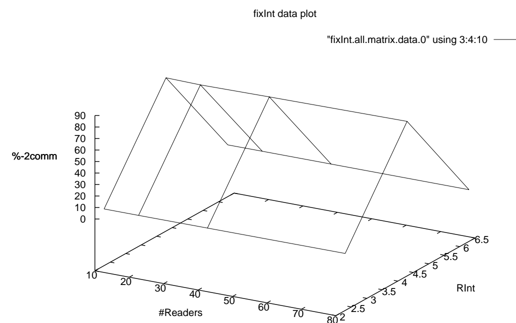
26.a (1) - $\mathbb{Q}_c$



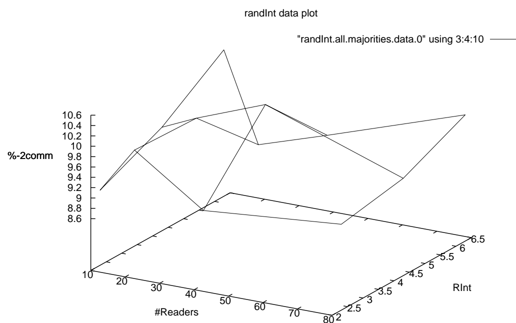
26.b (1) - $\mathbb{Q}_c$



26.a (1) - $\mathbb{Q}_x$

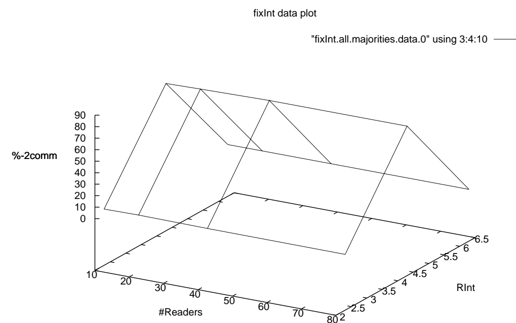


26.b (1) - $\mathbb{Q}_x$



26.a (1) - $\mathbb{Q}_m$

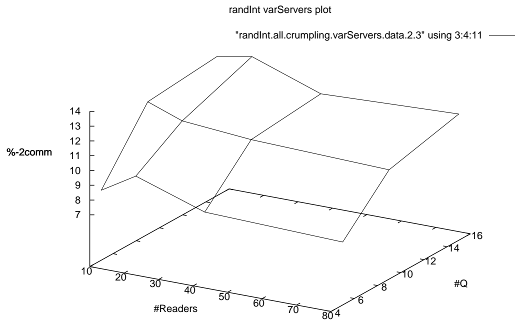
Setting a: Stochastic simulations



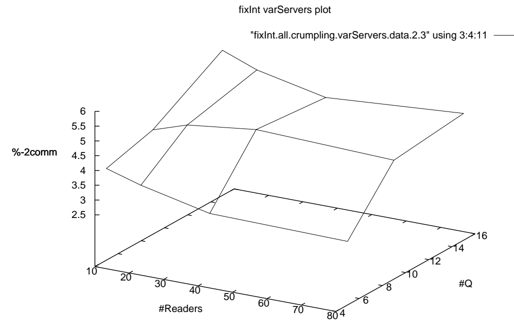
26.b (1) - $\mathbb{Q}_m$

Setting b: Fixed interval simulations

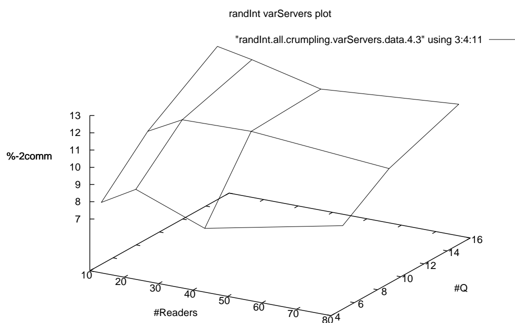
Figure 26: Simple runs



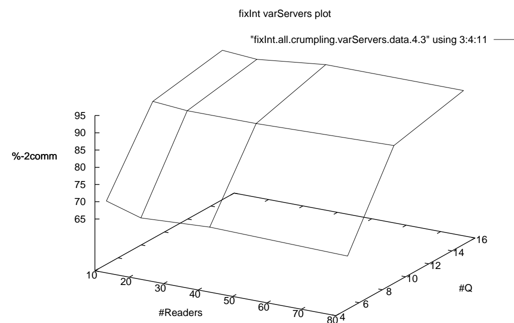
27.a $(2)(i)$ - $\mathbb{Q}_c$



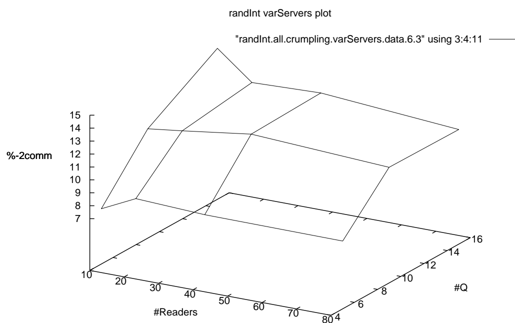
27.b $(2)(i)$ - $\mathbb{Q}_c$



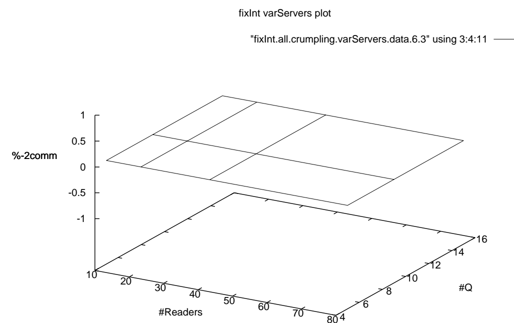
27.a $(2)(ii)$ - $\mathbb{Q}_c$



27.b $(2)(ii)$ - $\mathbb{Q}_c$



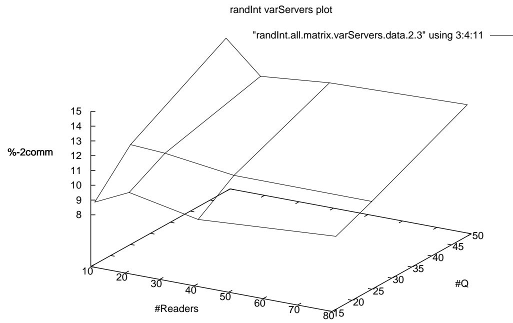
27.a $(2)(iii)$ - $\mathbb{Q}_c$



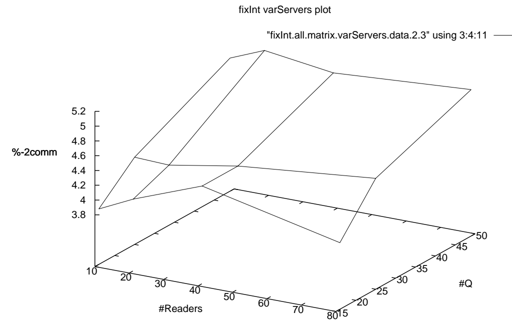
27.b $(2)(iii)$ - $\mathbb{Q}_c$

Setting a: Stochastic simulations Setting b: Fixed interval simulations

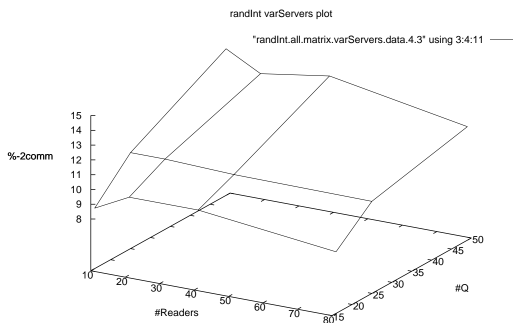
Figure 27: Crumbling Walls - Quorum Diversity Runs



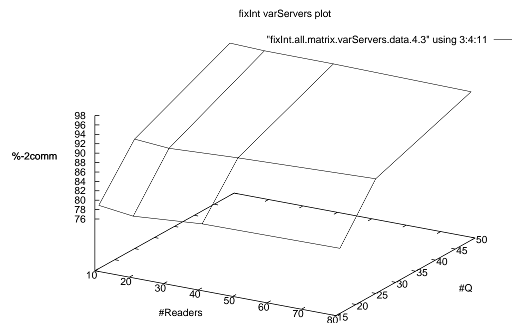
28.a $(2)(i)$ - $\mathbb{Q}_x$



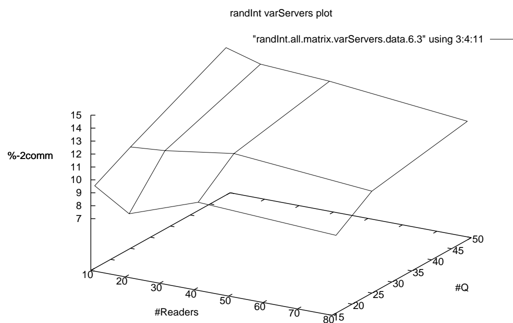
28.b $(2)(i)$ - $\mathbb{Q}_x$



28.a $(2)(ii)$ - $\mathbb{Q}_x$

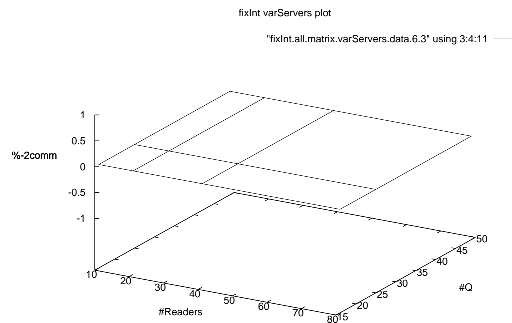


28.b $(2)(ii)$ - $\mathbb{Q}_x$



28.a $(2)(iii)$ - $\mathbb{Q}_x$

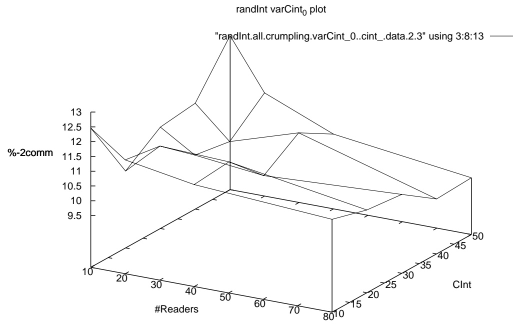
Setting a: Stochastic simulations



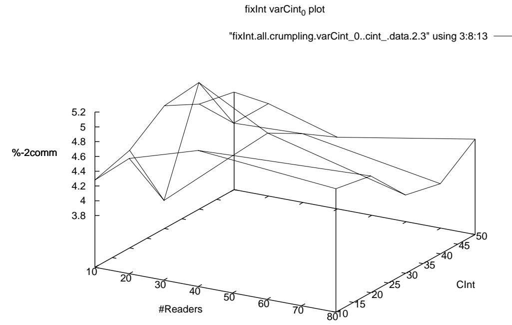
28.b $(2)(iii)$ - $\mathbb{Q}_x$

Setting b: Fixed interval simulations

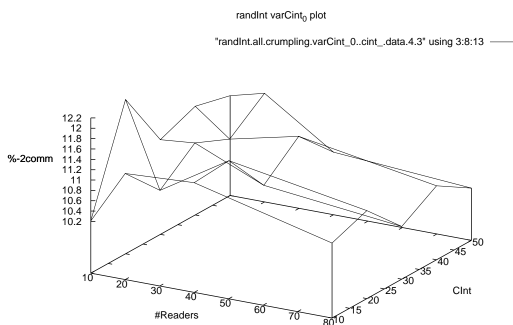
Figure 28: Matrix - Quorum Diversity Runs



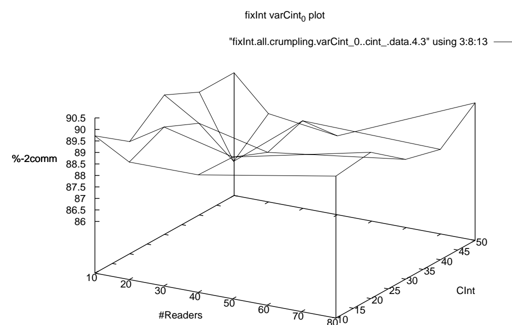
29.a $(3)(i)$ - $\mathbb{Q}_c$



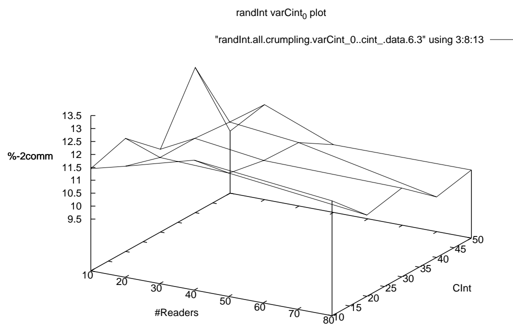
29.b $(3)(i)$ - $\mathbb{Q}_c$



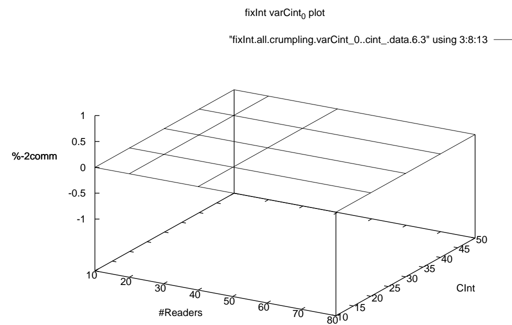
29.a $(3)(ii)$ - $\mathbb{Q}_c$



29.b $(3)(ii)$ - $\mathbb{Q}_c$



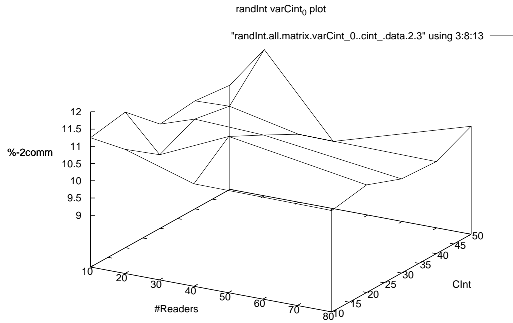
29.a $(3)(iii)$ - $\mathbb{Q}_c$



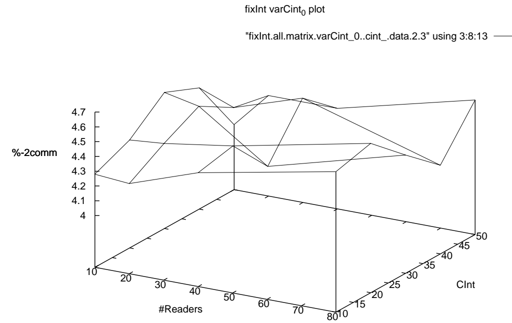
29.b $(3)(iii)$ - $\mathbb{Q}_c$

Setting a: Stochastic simulations — Setting b: Fixed interval simulations

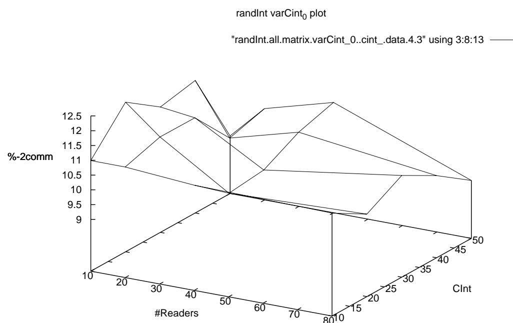
Figure 29: Crumbling Walls - Failure Diversity Runs ($cInt \in [0 \dots 50]$)



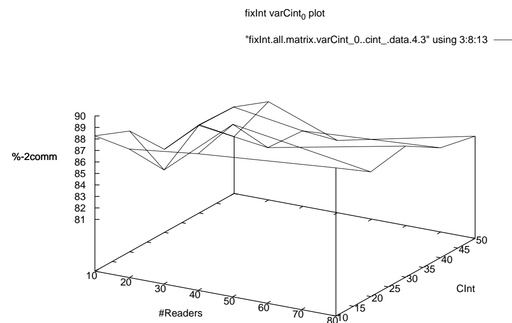
30.a $(3)(i)$ - $\mathbb{Q}_x$



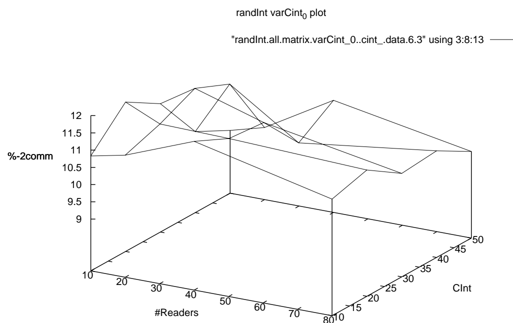
30.b $(3)(i)$ - $\mathbb{Q}_x$



30.a $(3)(ii)$ - $\mathbb{Q}_x$

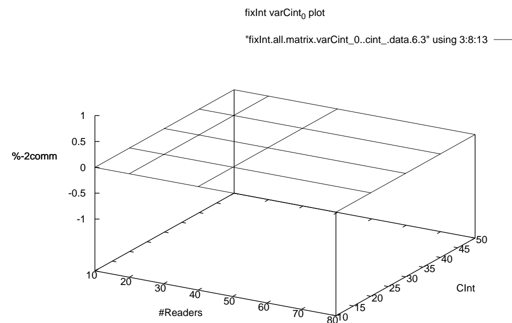


30.b $(3)(ii)$ - $\mathbb{Q}_x$



30.a $(3)(iii)$ - $\mathbb{Q}_x$

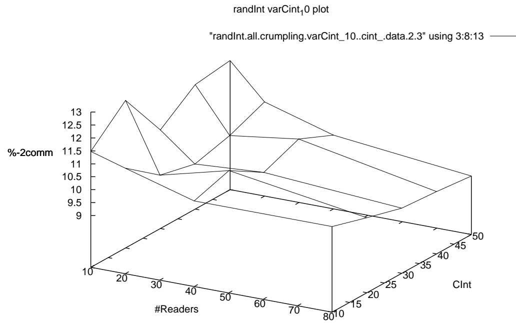
Setting a: Stochastic simulations

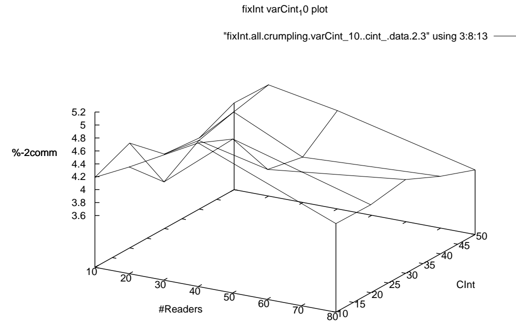


30.b $(3)(iii)$ - $\mathbb{Q}_x$

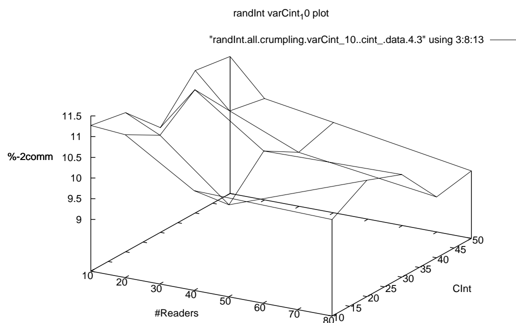
Setting b: Fixed interval simulations

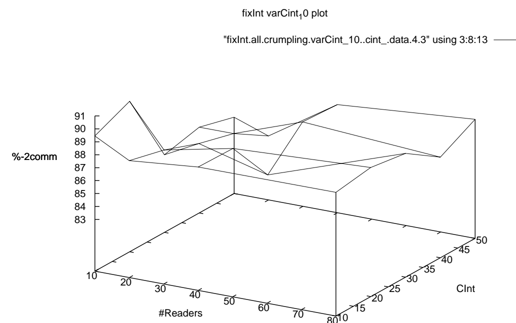
Figure 30: Matrix - Failure Diversity Runs ($cInt \in [0 \ldots 50]$)



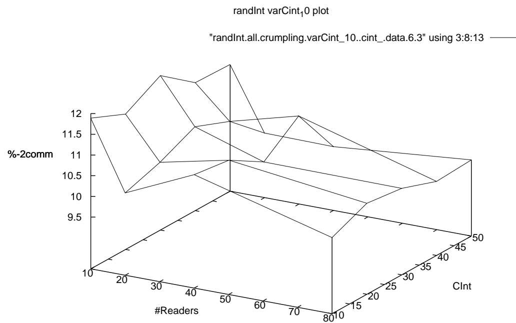


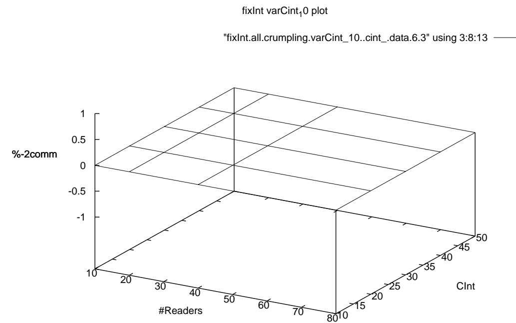
31.a $(3)(i)$ - $\mathbb{Q}_c$ 31.b $(3)(i)$ - $\mathbb{Q}_c$





31.a $(3)(ii)$ - $\mathbb{Q}_c$ 31.b $(3)(ii)$ - $\mathbb{Q}_c$

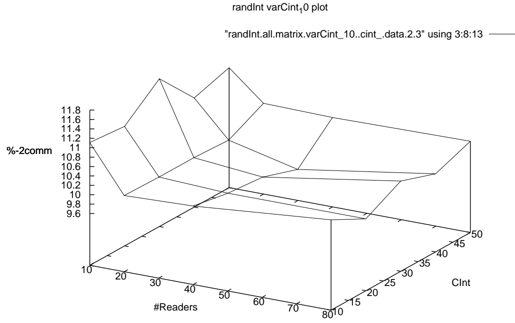




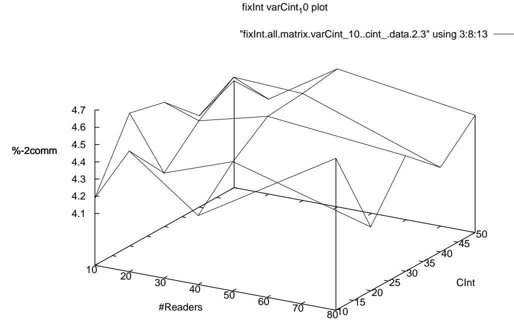
31.a $(3)(iii)$ - $\mathbb{Q}_c$ 31.b $(3)(iii)$ - $\mathbb{Q}_c$

Setting a: Stochastic simulations Setting b: Fixed interval simulations

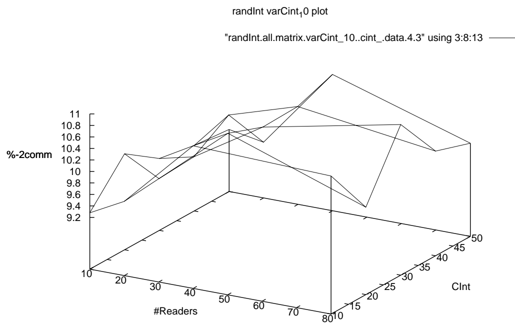
Figure 31: Crumbling Walls - Failure Diversity Runs ($cInt \in [10 \ldots 60]$)



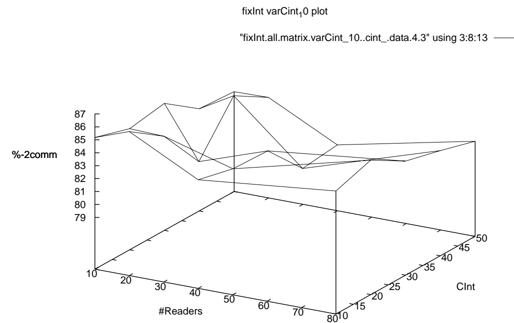
32.a $(3)(i)$ - $\mathbb{Q}_x$



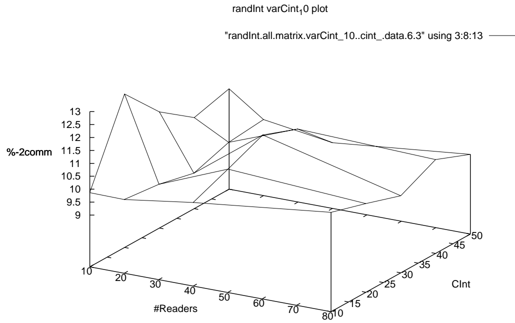
32.b $(3)(i)$ - $\mathbb{Q}_x$



32.a $(3)(ii)$ - $\mathbb{Q}_x$

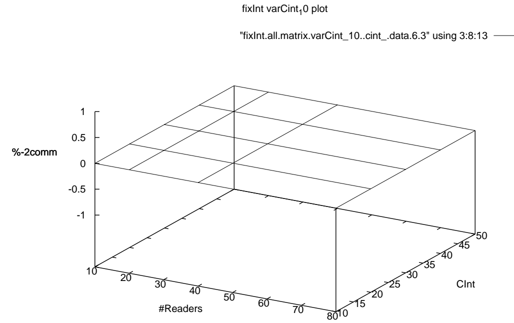


32.b $(3)(ii)$ - $\mathbb{Q}_x$



32.a $(3)(iii)$ - $\mathbb{Q}_x$

Setting a: Stochastic simulations



32.b $(3)(iii)$ - $\mathbb{Q}_x$

Setting b: Fixed interval simulations

Figure 32: Matrix - Failure Diversity Runs ($cInt \in [10 \dots 60]$)