3-29-2019

Memory Access Efficiency in Distributed Atomic Object Implementations

Theophanis Hadjistasi

University of Connecticut - Storrs, theophanis.hadjistasi@uconn.edu

Follow this and additional works at: https://opencommons.uconn.edu/dissertations

Recommended Citation

https://opencommons.uconn.edu/dissertations/2080
Distributed data services use redundancy to ensure data *availability* and *survivability*. Replication can be used to mask failures, however it introduces the problem of *consistency* because read and write operations may access different object replicas, possibly containing obsolete values. *Atomicity* is a venerable notion of consistency, introduced in 1979 by Lamport. Atomicity is the most natural type of consistency because it provides an illusion of equivalence with the serial object type that software designers expect. Atomicity provides strong consistency guarantees, making it more expensive to provide than weaker consistency guarantees.

We deal with the storage of atomic shared readable and writable data in distributed systems that are subject to perturbations in the underlying distributed platforms composed of computers and networks that interconnect them. The perturbations may include permanent crashes of individual computers, transient failures, and delays in the communication medium. The contents of each object are replicated across several *replica servers* and clients invoke read/write operations on the objects. A new approach that exploits *server-to-server* communication is introduced and we consider atomic implementations that utilize it under three assumptions.

First, we consider the single-writer, multiple-reader (SWMR) setting and we devise a solution where operations do not necessarily need to involve complete round-trips between clients and servers, i.e., operations take “one-and-a-half-rounds”. Then, we extend the SWMR solution to yield an algorithm for the multiple-writer, multiple-reader (MWMR) setting. We investigate conditions where reads can terminate in a single round-trip and we show revised algorithms.
Next, we investigate implementations that reduce both communication and computation demands and we present two SWMR algorithms. The first, makes clients to switch to a slow mode (e.g., two round-trips) whenever the system is congested. The second, pushes the responsibility of deciding the communication latency of operations to the servers. This allows the algorithm to utilize one and one-and-a-half-rounds operations, as necessary.

Lastly, we explore how the organization of the replica hosts is related to or affects the efficiency of the operations in the system. We devise algorithms for both SWMR and MWMR settings where read operations can take at most one-and-a-half-rounds, in a system with unconstrained quorum construction and reader participation.

Proposed algorithms trade latency for message complexity and have provable performance and correctness guarantees. To understand how the analytical results are reflected in practical efficiency, empirical studies are performed on the proposed algorithms.
Memory Access Efficiency in Distributed Atomic Object Implementations

Theophanis Hadjistasi

M.S., Computer Science and Engineering, University of Connecticut, 2018
B.S., Computer Science, University of Cyprus, 2014

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
at the
University of Connecticut
2019
APPROVAL PAGE

Doctor of Philosophy Dissertation

Memory Access Efficiency in Distributed Atomic Object Implementations

Presented by
Theophanis Hadjistasi, M.S., B.S.

Major Advisor
Alexander A. Shvartsman

Associate Advisor
Alexander C. Russell

Associate Advisor
Laurent Michel

Associate Advisor
Chryssis Georgiou

University of Connecticut
2019
ACKNOWLEDGEMENTS

This thesis was carried out at the Department of Computer Science and Engineering at the University of Connecticut, in Storrs, from 2014 to 2019. Despite the fact that it is just my name on the cover, completion of this doctoral dissertation was possible only with the support and the encouragement from a great number of individuals. All of them, contributed in their own particular way; ergo, with this last finishing touch I would like to express my sincere gratitude to all of them.

It is a genuine pleasure to express my deep sense of thanks and gratitude to my mentor, philosopher, and committee chair, Dr. Alexander A. Schwarzmann, for his unwavering support, collegiality, and mentorship throughout this project. Dr. Alexander continually and convincingly conveyed a spirit of adventure in regard to research; scholarship; and teaching. His scholarly advice, valuable insights, guidance, clarity of thought, meticulous scrutiny, patience, and encouragement allowed my educational growth and delivered a young scholar.

Dr. Alexander, I would like you to know that during this long journey I have always called you “The Strategos” (Greek: Ο Στρατηγός, meaning – the army leader). You were always in charge, taking a series of earth shattering decisions, which all had a significant impact in my life. I am honored that I served and conducted research under your commands. Thank you.

I would like to extend my thanks to my associate advisors who offered me collegial guidance and support over this memorable journey: Dr. Alexander C. Russell, Dr. Laurent Michel
and Dr. Chryssis Georgiou. Your outstanding ideas, enthusiasm, dedication, and endless passion for this research topic have contributed to significantly improve this thesis. I am grateful that you showed confidence in our work.

A special word of gratitude is due to Dr. Chryssis Georgiou who invested in me. Back in 2013, during my undergraduate studies at the University of Cyprus, you accepted me as your student and with your guidance, encouragement, support, and patience you crafted a young researcher. It was back then when you inspired me to achieve success. Your support and faith in me kept me going forward through this tough process. Thank you for being a bright friend.

My thankfulness is also to Dr. Nicolas Nicolaou for being an exceptional collaborator, mentor and most importantly a brilliant friend. His patience, guidance, dedication and continuous feedback have broaden my knowledge. Our endless discussions taught me how to deliver meaningful and important research. Thank you for helping develop our ideas and for never stopped challenging me.

Without hesitation, I would like to express my thanks to all the staff of the Computer Science and Engineering Department at the University of Connecticut. In particular, my special thanks to our administrative services specialist Rebecca Randazzo, administrative coordinator Lara Chiaverini, program assistant Joy Billion, and computer lab coordinator Howard Ellis. Thank you for your dedication, support, guidance, and your patience with my requests.

A very special thanks goes to the engineers and the research personnel of the UCONN VoTER Center, Della Farney and Matthew Desmarais. Both for being exceptional collaborators and great friends. Our discussions along with their extreme patience and continuous feedback, helped me to improve my coding skills and taught me to deliver meaningful code.
During my graduate studies at UCONN I had the opportunity to collaborate on research projects with Dr. Antonio Fernández Anta, Dr. Alexandru Popa and Dr. Roberto De Prisco. I am thankful for the knowledge and experience I obtained through these wonderful collaborations.

I would also like to express my gratitude to my best friends that are located all over the world, Pantelis Pavlou, Misagh Kordi, Ioannis Papavasileiou, Aaron Palmer, Nectarios Antoniou, Mike Georgiou, Charalambos Tylliros, Constantinos Nicolaides, and Pavlos Kyprianou for all the valuable discussions on literally anything. Additionally, I am extremely thankful to Dr. Seda Davtyan and Tigran Antonyan for supporting my transition to the United States and for their continued support during my studies at the University of Connecticut. You all gave me the strength, meaning, and endurance to overcome the most difficult parts of this journey.

Last but not the least, I would like to extend my thanks and emphasize that I am indebted to my family. My parents, Aristarchos Hadjistasi and Paraskevi Tyllirou Hadjistasi, for sacrificing their personal lives for me. My brother Chris Hadjistasi, and my sister Anna Christofi, for always believing in me and supporting all my decisions. Thank you all for your dedication, enthusiasm, encouragement and for pushing me farther than I ever thought I could go. Thank you for inspiring me to follow my dreams. I dedicate this dissertation to all the four of you.
CREDITS

This dissertation incorporates research results appearing in the following publications:

[41, 42] This is a joint work with N. Nicolaou and A. A. Schwarzmann that appears in the Proceedings of NEtworked sYStems, *NETYS 2017*, [42]. A brief announcement version [41] appears in the Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2016. This work is included in Chapter 4.

[7] This is a joint work with A. Fernandez Anta, and N. Nicolaou that appears in the 20th International Conference on Principles of Distributed Systems, *OPODIS 2016*. This work corresponds to Chapter 5.

[27] This paper is a joint work with C. Georgiou, N. Nicolaou and A. A. Schwarzmann that appears in the Proceedings of NEtworked sYStems, *NETYS 2018*. This work is included in Chapter 6.

[43] This is a joint work with A. A. Schwarzmann that appears in the Colloquium on Automata, Languages, and Programming, *ICALP 2018*. Parts of Chapters 1, 3 and 7 correspond to this work.
# TABLE OF CONTENTS

## Chapter 1: Introduction

1.1 Motivation ......................................................... 1  
1.2 Shared Storage: A Landscape ................................. 3  
1.3 Thesis Contributions ........................................... 8  
  1.3.1 Efficient Survivable Distributed Storage Implementations ... 9  
  1.3.2 Tractable Low-Delay Atomic Memory ...................... 10  
  1.3.3 Quorum Systems vs Atomic Implementations ............... 12  
1.4 Thesis Organization ............................................ 13

## Chapter 2: Model of Computation

2.1 Distributed System ............................................. 15  
2.2 Consistency: Atomic Object Semantics ..................... 20  
2.3 Efficiency, Rounds and Message Exchanges ................ 24  
2.4 Fastness ....................................................... 26  
2.5 Notation ....................................................... 27

## Chapter 3: Consistent Distributed Memory Services

3.1 Consistency .................................................... 28  
3.2 Atomic Memory Under Crash Failures in Static Settings .... 30  
  3.2.1 The SWMR Setting ....................................... 31  
  3.2.2 The MWMR Setting ...................................... 36
3.3 Atomic Memory Under Crash Failures in Dynamic Settings ............... 39
  3.3.1 Consensus ......................................................... 41
  3.3.2 The RAMBO Framework ........................................... 42
  3.3.3 Dynastore .......................................................... 43
  3.3.4 Group Communication Services ................................. 45

Chapter 4: Efficient Survivable Distributed Storage Implementations 47
  4.1 OHRAM: One-and-a-Half Round Atomic Memory .................... 48
  4.2 The SWMR Setting ................................................... 49
    4.2.1 Description of SWMR Algorithm OHSAM ...................... 49
    4.2.2 Correctness of OHSAM ........................................... 53
    4.2.3 Performance of OHSAM .......................................... 59
  4.3 The MWMR Setting ................................................... 61
    4.3.1 Description of MWMR Algorithm OHMAM ...................... 62
    4.3.2 Correctness of OHMAM ........................................... 64
    4.3.3 Performance of OHMAM .......................................... 74
  4.4 Reducing the Latency of Read Operations ............................ 75
    4.4.1 Obtaining Algorithms OHSAM' and OHMAM' ................... 75
    4.4.2 Correctness of OHSAM' .......................................... 77
    4.4.3 Correctness of OHMAM' .......................................... 82
    4.4.4 Performance of the Revised Algorithms ....................... 89
  4.5 Experimental Evaluation ............................................ 89
Chapter 5: Tractable Low-Delay Atomic Memory

5.1 Switching between One and Two Rounds ............................................. 97
  5.1.1 Description of SWMR Algorithm ccHYBRID ........................................ 99
  5.1.2 Correctness of ccHYBRID .............................................................. 105
  5.1.3 Performance of ccHYBRID ............................................................. 117

5.2 Switching between One and One-and-a-Half Rounds ......................... 120
  5.2.1 Description of SWMR Algorithm OHFAST ....................................... 121
  5.2.2 Correctness of OHFAST .............................................................. 127
  5.2.3 Performance of OHFAST ............................................................. 138

5.3 Experimental Evaluation ................................................................. 140

Chapter 6: Quorum-based Atomic Implementations ................................. 148

6.1 Incorporating Prior Techniques ......................................................... 150

6.2 The SWMR Setting ............................................................................. 153
  6.2.1 Description of SWMR Algorithm ERATO ......................................... 153
  6.2.2 Correctness of ERATO .................................................................. 158
  6.2.3 Performance of ERATO ............................................................... 169

6.3 The MWMR Setting ............................................................................ 171
  6.3.1 Description of MWMR Algorithm ERATO-MW ................................ 171
  6.3.2 Correctness of ERATO-MW ........................................................... 176
  6.3.3 Performance of ERATO-MW .......................................................... 189

6.4 Experimental Evaluation ................................................................. 190
## Chapter 7: Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Future Directions</td>
<td>196</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Extensions of the Current Work</td>
<td>196</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Theoretical Bounds</td>
<td>201</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Dynamic Systems</td>
<td>203</td>
</tr>
<tr>
<td>7.2</td>
<td>Closing Remarks</td>
<td>208</td>
</tr>
</tbody>
</table>

**Bibliography**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>209</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This dissertation investigates latency-efficient algorithms for consistent and fault-tolerant distributed storage. The overall objective is to devise new algorithmic solutions that lead to efficient survivable distributed storage implementations with provable performance and correctness guarantees. We commence by presenting the motivation for this work in Section 1.1 and then we describe the general landscape for implementing consistent shared memory services in Section 1.2. Next, we present the research contributions of this dissertation in Section 1.3. Lastly, we give the overall structure of this thesis in Section 1.4.

1.1 Motivation

Reading, 'Riting, and 'Rithmetic, the three R’s underlying much of human intellectual activity, not surprisingly, also stand as a venerable foundation of modern computing technology. Indeed, both the Turing machine and von Neumann machine models operate by reading, writing, and computing, and all practical uniprocessor implementations are based on performing activities structured in terms of the three R’s. With the advance of networking technology,
communication became an additional major systemic activity. However, at a high level of abstraction, it is apparently still more natural to think in terms of reading, writing, and computing. While it is hard to imagine distributed systems—such as those implementing the World-Wide Web—without communication, we often imagine browser-based applications that operate by retrieving (i.e., reading) data, performing computation, and storing (i.e., writing) the results.

In this dissertation, we deal with the storage of shared readable and writable data in distributed systems that are subject to perturbations in the underlying distributed platforms composed of computers and networks that interconnect them. In particular, the perturbations may include permanent failures (or crashes) of individual computers, transient failures, and delays in the communication medium. The objective of our work is to provide efficient emulations of atomic read/write object implementations in distributed systems. Communication latency is a factor that typically dominates the performance of message-passing systems, consequently the efficiency of algorithms implementing atomic objects is measured in terms of the number of communication exchanges involved in each read and write operation. Atomicity is a venerable notion of consistency, introduced in 1979 by Lamport [48]. To this day atomicity remains the most natural type of consistency because it provides an illusion of equivalence with the serial object type that software designers expect.

With the advent of Cloud services, distributed storage services are bound to continue attracting attention. It is widely known that high performance memory systems with superior fault-tolerance nowadays play a significant role in the construction of sophisticated distributed applications. Distributed storage services continue to attract awareness but still, there are important challenges that researchers have to overcome and address in order to increase the “quality” of consistent storage systems.
1.2 Shared Storage: A Landscape

Shared storage services are located at the core of most information-age systems. Shared memory systems surveyed in this work provide objects that support two different access operations. That is, a read, that obtains the current value of the object, and a write that replaces the old value of the object with a new one. To be useful, such objects need to be resilient to failures and perturbations in the underlying computing medium, and must be consistent in that there are guarantees regarding relationships between previously written values and the values read by subsequent read operations. Such resilient and consistent object are also called registers. In this dissertation we focus on read/write objects, however for objects with more complicated semantics, such as transactions or read-modify-write operations, there exist common implementation challenges that any distributed storage system faces and needs to resolve. Imagine a storage system that is implemented as a central server. The server accepts client requests to perform operations on its data objects and returns responses. Conceptually, this approach is simple, however, two major problems can already be observed. The first is that the central server is a performance bottleneck. The second is that the server is a single point of failure. The quality of service in such an implementation degrades rapidly as the number of clients grows, and the service becomes unavailable if the server crashes (imagine how inadequate a web news service would be were it implemented as a central server).

Thus, the system must be available. This means it must provide its services despite failures within the scope of its specification, for example, the system must be able to mask certain server and communication failures. The system must also support multiple concurrent accesses
without imposing unreasonable degradation in performance. The only way to guarantee availability is through redundancy, that is, by using multiple servers and by replicating the contents of objects among these servers. Moreover, the replication must be done at geographically distributed and distinct network locations, where the disconnection or failures of certain subsets of data servers can be masked by the system. It is also critically important for a storage system to ensure data longevity. A storage system may be able to tolerate failures of some servers, but over a long period it is conceivable that all servers (or some of them) may need to be replaced. This, due to planned upgrades and because no servers are infallible. The storage system must provide seamless run time migration of data: one cannot stop the world and reconfigure the system in response to failures and changing environment.

Replication introduces the challenge of ensuring consistency. How does the system record new values so that consequently it can find and return the latest value of a replicated object? This problem was not present with a central server implementation: the server always contains the latest value. In a replicated implementation, a trivial solution would be in each operation to consult all replicas servers in search of the latest value, however, this is not fault-tolerant (as it assumes all replicas are accessible) and expensive. In any case, none of the implementation issues should be a concern for the clients of the distributed memory service. What the clients should expect to see is the illusion of a single-copy object that serializes all accesses so that each read operation returns the value of the preceding write operation, and that this value is at least as recent as that returned by any preceding read. More generally, the behavior of the object, as observed externally, must be consistent with the abstract sequential data type of the object, and in developing applications that use such objects the clients must be able to rely on the abstract data type of the object. This notion of consistency is formalized as atomicity.
for read/write objects, and equivalently, as linearizability [45] that extends atomicity to arbitrary data types. While there is no argument that atomicity is the most convenient notion of consistency, we note that weaker notions have also been proposed and implemented, motivated primarily by efficiency considerations. Atomicity provides strong guarantees, making it more expensive to provide than weaker consistency guarantees [11]. We take the view that it is nevertheless important to provide simple and intuitive, be it more expensive, atomic consistency. Barbara Liskov, a Turing Prize laureate, in a keynote address (at [52]) remarked that atomicity is not cheap, however, if we do not guarantee it, this creates headaches for developers.

Contemporary storage systems may also provide more complex data access primitives implementing atomic read-modify-write operations. Such access primitives are much stronger than separate read and write primitives we consider in this work. Implementing such operations is expensive, and at its core requires atomic updates that in practice are implemented by reducing parts of the system to a single-writer model (e.g., Microsoft’s Azure [14]), by depending on clock synchronization hardware (e.g., Google’s Spanner [18]), or by relying on complex mechanisms for resolving event ordering such as vector clocks (e.g., Amazon’s Dynamo [20]).

In a typical implementation of a distributed memory, efficiency is assessed in terms of operation latency and message complexity. Latency of an operation is determined by computation time and the communication delays. Computation time accounts for all local computation within an operation. Delays are measured in terms of communication exchanges. The protocol implementing each operation involves a collection of sends (or broadcasts) of typed messages and the corresponding receives. As defined in Section 2.3, a communication exchange within an execution of an operation is the set of sends and matching receives for the specific message type. Traditional implementations in the style of ABD [10] are structured in terms of rounds,
each consisting of two exchanges, the first, a broadcast, is initiated by the process executing an operation, and the second, a convergecast, consists of responses to the initiator. The number of messages that a process expects during a convergecast depends on the implementation. 

*Message complexity* measures the total number of messages exchanged.

A distributed shared register implementation is characterized by the number of writer and reader processes that it allows to participate in the system, e.g., the Single Writer, Multiple Reader (SWMR) setting and the Multiple Writer, Multiple Reader (MWMR) setting. Additionally, the implementation is also categorized by the type of participant failures that it tolerates, e.g., byzantine-failures, crash-failures, etc.

A seminal work of Attiya, Bar-Noy, and Dolev [10] provides an algorithm, colloquially referred to as ABD, that implements SWMR atomic objects in message-passing crash-prone asynchronous environments. The Dijkstra Prize was awarded to this work in 2011. In ABD, replication helps achieve fault-tolerance and availability, and the implementation replicates objects at servers, and it tolerates \( f \) replica servers crashes, provided a majority of replicas do not fail. 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. Read and write operations are ordered using logical timestamps associated with each written value. Timestamps totally order write operations, and therefore determine the values that read operations return. All operations terminate provided a majority of replicas do not crash. Write operations involve a single communication round-trip consisting of *two* communication exchanges. The writer broadcasts its request to all replica servers during the first exchange and terminates once it collects acknowledgments from some majority of servers
in the second exchange. Each read operation takes two rounds involving in four communication exchanges. The reader broadcasts a read request to all replica servers in the first exchange, collects acknowledgments from some majority of servers in the second exchange, and it discovers the maximum timestamp. In order to ensure that any subsequent read will return a value associated with a timestamp at least as high as the discovered maximum, the reader propagates the value associated with the maximum timestamp to at least a majority of servers before completion. The correctness of this implementation, that is, atomicity, relies on the fact that any two majorities have a non-empty intersection.

Subsequently, Lynch and Schwarzmann [55] presented an extension of algorithm ABD for the MWMR setting. We refer to the static version of their MWMR implementation as ABD-MW. Notice that, the simplest approach to order written values in the MWMR setting is to use pairs consisting of a timestamp and writer’s id. Such a pair is termed as a tag and they are ordered lexicographically in establishing an order on the operations. In contrast with ABD, where the sole writer generates new timestamps without any communication, the writers in ABD-MW start a write operation by performing an additional round in which the replica servers are queried for their latest tags. Once tags are received from a majority of servers, the writer increments the timestamp of the highest detected timestamp to produce its new tag. The second round is performed as in ABD. Thus a write operation for ABD-MW takes four exchanges in comparison with the two exchanges in ABD. The read protocol is identical to the four-exchange protocol in ABD, the only difference being that tags are used instead of timestamps. This algorithm can also be used with quorum systems [55, 67] instead of majorities. A quorum is a collection of servers. Additionally, given a set of servers, a quorum system is a collection of subsets of servers with non-empty pair-wise intersections. Majorities are a special case of
quorums. The failure model for the quorum based solution is that any pattern of crashes is tolerated, provided that the servers in at least one quorum do not crash.

Consequent research focused on the traditional client-to-server communication pattern and studied conditions under which operations can terminate fast without violating the property of atomicity, resulting in “multi-speed” implementations consisting of either one or two rounds per operation. In this dissertation we introduce server-to-server communication in the system and we present a new family of atomic read/write shared register implementations where operations do not necessarily require complete communication rounds to terminate, i.e., operations are able to complete in at most one-and-a-half-rounds. We elaborate on the inherent limitations that such a technique may impose on the distributed system.

1.3 Thesis Contributions

The objective of this dissertation is to investigate algorithmic improvement to emulations of atomic read/write shared objects in message-passing systems, and to explore solutions to existing open research questions in this area. In particular, this thesis focuses on the gap between one-round and two-round algorithms that implement an atomic register abstraction and aims to answer the following general question:

What improvements can be obtained in implementations of atomic read/write registers in distributed systems with asynchronous, message-passing, crash-prone processors by exploring different patterns of communication among the participants.

We assume systems with static participation that allow participants to crash in both the SWMR and the MWMR setting. We study the operation latency, in terms of the number of
communication rounds (or exchanges) required by each operation. Six algorithmic solutions are developed that implement atomic read/write registers. Four of them are designed for the SWMR setting, and two are devised for the MWMR setting. Each of the algorithms contains operations that take one or one-and-a-half or two rounds for completion. In order to discern algorithms efficiency, empirical studies on the proposed algorithms are performed. A summary for each contribution of this thesis is given in the sections that follow.

1.3.1 Efficient Survivable Distributed Storage Implementations

Motivated by the observation of Dutta et al. in [22], suggesting that atomic memory may be implemented (using a max/min technique) so that each read and write operation completes in “one-and-a-half rounds”, the first part of the dissertation elaborates on the inherent limitations that such a technique may impose on the distributed system. In particular, we investigate the possibility and the cost of efficient implementations of atomic read/write registers where read and write operations can take one-a-half-rounds, i.e., complete in three communication exchanges.

We present a new SWMR algorithm for atomic objects in the asynchronous message-passing model with processor crashes, named OHSAM. Write operations take two communication exchanges and are similar to the write operations of ABD [10]. For read operations we introduce server-to-server communication resulting in operations that take three communication exchanges. Read operations utilize the following communication pattern: (1) the requesting reader sends a message to the participating servers, (2) the servers share this information between them, and (3) once this is “sufficiently” done, servers reply to the reader. A
key idea of the algorithm is that the reader returns the value that is associated with the \textit{minimum} timestamp (cf. the observation in [22]).

Then, we extend the SWMR algorithm to yield an implementation for the MWMR setting, called OHMAM. In the new algorithm the write operations are more complicated, taking \textit{four} communication exchanges (cf. [55]). The read protocol is identical to the one that the SWMR algorithm OHSAM uses and read operations complete as before in \textit{three} exchanges.

Next, we investigate the conditions under which the read operations in the presented algorithms can terminate in less than three communication exchanges and we present a revised SWMR algorithm and a revised MWMR algorithm. We name the revised algorithms OHSAM$'$ and OHMAM$'$. In both algorithms, read operations complete in either \textit{two} or \textit{three} communication exchanges. The revised versions of each algorithm are presented for pedagogical reasons: for ease of understanding and reasoning about the algorithms.

Lastly, using NS3 [4] we simulate our algorithms to observe how the analytical results of the proposed algorithms are reflected in practical efficiency. Not surprisingly, the simulation results suggest that in practical settings, such as data centers with well-connected servers, the communication overhead is not prohibitive.

\subsection{1.3.2 Tractable Low-Delay Atomic Memory}

Communication cost is the most commonly used metric in assessing the efficiency of operations in distributed algorithms for message-passing environments. The standing assumption is that the cost of local computation is negligible compared to the cost of communication. Frequently, operation implementations rely on complex computations that should not be ignored.
Thus, in some cases, proposed solutions either require restrictions in the system or incur high computation overhead, resulting in solutions that are not practical.

We investigate implementations that reduce both communication and computation demands. Examining the best two algorithms, in terms of communication demands, that implement atomic SWMR memory, CCFAST [8] and OHSAM (Section 4.2), we observe that both solutions have trade-offs. In particular, CCFAST achieves optimal communication by allowing each operation to complete in one round trip, with polynomial computation requirements. However, it relies on strict limitations on the number of participating readers. On the other hand, algorithm OHSAM performs negligible computation, imposes no restrictions on the system, but it provides operations that always require one-and-a-half rounds before completion.

In the light of these shortcomings, we present two SWMR algorithms that implement multi-speed operations and without imposing any restrictions on the system. In particular, we present algorithm CCHYBRID that adopts the fast reads presented in [8] and makes clients to switch to a slow mode whenever the system is congested. Additionally, we give a second algorithm, named OHFAST, that pushes the responsibility of deciding the communication latency of the operations to the servers. This allows the algorithm to utilize fast operations presented in [8] and one-and-a-half-rounds operations of algorithm OHSAM, as necessary.

To assess the practicality of the proposed algorithms, we simulate them alongside comparable solutions using NS3 and compare their performance in terms of operation latency, and ratio of slow over fast operations performed under various scenarios, topologies and operation loads.
1.3.3 Quorum Systems vs Atomic Implementations

We focus on the gap between one-round and two-round algorithms that implement an atomic read/write register abstraction. Thus far we considered the impact of unconstrained constructions – in terms of client participation and replica organization – on the efficiency of implementations. Here, we explore how the organization of the replica hosts is related to or affects the efficiency of the operations in the system. In particular, we investigate the possibility and the cost of efficient implementations of atomic read/write registers where read operations can take at most one-and-a-half-rounds in a system with unconstrained quorum construction and reader participation. Quorums [67] are well-known mathematical tools that have been widely used for coordination and collaboration between processes in a distributed system.

Our work builds on the one-and-a-half-rounds solutions (Section 4.1) and the techniques presented by Georgiou et al. [32, 30]. In particular, authors in [32] introduced Quorum Views, client-side tools that examine the distribution of the latest value among the replica servers in order to enable fast read operations under read and write operation concurrency. Authors derived an atomic SWMR implementation called SLIQ. A later work [30] generalized the client-side decision tools and presented a MWMR algorithm, named CwFR, that also allows fast read operations.

We combine the above techniques and we obtain algorithms for both the SWMR and the MWMR setting. The proposed solutions allow one and one-and-a-half-rounds operations. The SWMR implementation, called ERATO, improves the three-exchange read protocol of algorithm OHSAM and the two or four-exchange read protocol of algorithm SLIQ by allowing reads to terminate in either two or three exchanges. Similarly to ABD, writes take two exchanges.
Using the SWMR algorithm as the basis in combination with the iterative technique on the quorum views of algorithm CWFr we devise an algorithm for the MWMR setting, called ERATO-MW. In the resulted implementation reads take either two or three exchanges to complete. Write operations are similar to ABD-MW and take four exchanges (cf. [55]).

In order to observe how the analytical results of the proposed algorithms are reflected in practical efficiency, using NS3, we simulate our quorum-based SWMR and MWMR algorithms with existing comparable ones.

1.4 Thesis Organization

Up to now we presented the motivation behind this work, we described the general landscape for implementing consistent shared memory services and we stated the research contributions of this dissertation. The rest of the dissertation is organized as follows:

Chapter 2 describes the general distributed setting for implementing consistent shared memory services, defines atomic consistency, and describes the measures of efficiency. In Chapter 3 we give an overview of the proposed consistency semantics and then we present several approaches that implement consistent shared memory in static services for the SWMR and the MWMR setting respectively. Additionally, we survey several approaches for providing consistent shared memory in more dynamic systems.

Chapter 4 introduces the first SWMR and MWMR implementations, called OHSAM and OHMAM, where atomic operations do not necessarily require complete communication round trips to complete, by introducing server-to-server communication. Next, Chapter 5 investigates implementations that reduce both communication and computation demands, and presents two “multi-speed” algorithms, called CCHYBRID and OHFAST, for the SWMR setting. Following,
in Chapter 6, we assume and employ general quorum constructions, and by combining multiple prior techniques we obtain algorithms ERATO and ERATO-MW. The presented algorithms, perform a modest amount of local computation, and allow one and one-and-a-half round-trip operations without imposing any restriction on the participants in the service. In order to observe how the analytical results of all the proposed algorithms are reflected in practical efficiency, in each chapter we present a comparative study of the algorithms with comparable ones. Chapter 7 discusses possible future extensions of this work and concludes this thesis.
Chapter 2

Model of Computation

This chapter presents the formal model, definitions and notations we use. The model of the distributed system is described in Section 2.1. This model is applied to all the algorithms that follow. Definitions of the data types and the consistency semantics are given in Section 2.2. Definitions of complexity measures are presented in Section 2.3. Fastness in operations and implementations is defined in Section 2.4. Notations used throughout this thesis are given in Section 2.5. We encourage the reader to use this chapter as a reference.

2.1 Distributed System

We model the system as a collection of interconnected computers (or processors), that communicate by sending point-to-point messages. Each processor has a unique identifier from some well-ordered set $I$, local storage, and it can perform local computation.

**Executions.** An algorithm $A$ is a collection of processes, where process $A_p$ is assigned to processor $p \in I$. The *state* of processor $p$ is determined over a set of state variables, and the state of $A$ is a vector that contains the state of each process. Algorithm $A$ performs a *step*, when
some process $p$ (i) receives a message, (ii) performs local computation, (iii) sends a message. Each such action causes the state at $p$ to change. An execution $\xi$ is an alternating sequence of states and actions of $A$ starting with the initial state and ending in a state. We denote the set of all the executions of implementation $A$ as $\text{execs}(A)$.

**Failures.** An unspecified external entity, the adversary, determines which components of the system may fail. The adversary determines the type of the faults the components suffer and at which step during the computation those faults occur. We assume that the adversary has a complete knowledge of the computation and it is capable of making instant and dynamic decisions during the course of the computation. Thus, we say that the adversary is omniscient and on-line. The power of the adversary is restricted by the failure model $\mathfrak{F}$. As an example, a constraint would be the maximum number of processes that can fail during an execution $\xi$ of an implementation $A$. We let $\text{execs}(A, \mathfrak{F})$ denote the set of all the executions of implementation $A$ from $\text{execs}(A)$.

A processor may fail by crashing at any point of the computation. Any processor that crashes stops operating: it does not perform any local computation, it does not send any messages, and any messages sent to it are not delivered. Common approaches to implementing resilient in the face of failures algorithm specify failure models that provide qualitative or quantitative restrictions on the power of adversaries, e.g., by limiting the adversary to causing at most $f$ crashes for some algorithm-specific parameter $f$. In this thesis we consider only crash failures.

**Communication.** The system is asynchronous, and the processors have no access to a global clock or synchronization mechanisms. This means that relative processing speeds at the processors can be arbitrary, and that the processors do not know the upper bound on time that
it takes to perform a local computation. The message delays can also be arbitrary, and the processors do not know bounds on message latency (although such bounds may exist). Thus algorithms may not rely on assumptions about global time or delays.

We assume that messages can be reordered in transit, however, the messages cannot be corrupted, duplicated, or generated spontaneously. If a message is received then it must have been previously sent. The messages are not lost, but message loss can be modeled as long delays (we do not address techniques for constructing more dependable communication services).

**Processes and Operations.** The processors with ids in the set $\mathcal{I}$ include a set of writers $\mathcal{W}$, a set of readers $\mathcal{R}$, and a set of replica servers $\mathcal{S}$. These sets do not need to be disjoint, but it is helpful to view separately the roles a participant in the service can play. A writer process $w \in \mathcal{W}$ may perform write operations, and a reader process $r \in \mathcal{R}$ may perform read operations on the shared object. Both readers and writers can be referred as clients. Due to failures, survivability of the shared object is ensured through replication. Each object is replicated at a server process.

A read operation is denoted by $\rho$ and a write operation by $\omega$. We use operation $\pi$ to denote any type of operation, either a read or a write. For the single-writer/multiple-reader static setting (SWMR) $|\mathcal{W}| = 1$ and $|\mathcal{R}| \geq 1$ whereas for the multiple-writer/multiple-reader static environment (MWMR) $|\mathcal{W}| > 1$ and $|\mathcal{R}| \geq 1$.

**Environment.** We categorize a distributed networked system as either static or dynamic as follows. In the static system the set of participating processors is fixed, and each processor may know the identity of all participants; crashes (or voluntary departures) may remove processors from the system. Static algorithms are commonly designed to tolerate up to $f < |\mathcal{S}|/2$ server crashes and arbitrary number of crashes among readers and writers. In the dynamic
system the set of processors may be unbounded, and the set of participating processors may completely change over time as the result of crashes, departures, and new processors joining. Thus, the failure models considered in dynamic settings are much more complicated, given that the system may dramatically evolve over time.

**Quorum Systems.** We now provide some background and basic definition of quorum systems [67]. Quorum systems are basic mathematical tools that are used to reason about distributed implementations of data objects, i.e., read/write storage. We are interested on quorum systems over the set of server identifiers $S$.

Given a set of servers from $S$, a quorum system is a collection of subsets of servers, called quorums, where every two of which intersect. A quorum system is formally defined as follows.

**Definition 2.1.1 (Quorum System)** A quorum system $Q \subset 2^S$ is a set of subsets of $S$, called quorums $Q$, such that:

- $\forall Q \in Q : Q \subseteq S$, and
- $\forall Q, Q' \in Q : Q \cap Q' \neq \emptyset$

In order to enforce consistency, implementations of atomic memory rely on quorum systems over the set of servers $S$. In particular, such solutions assume a failure model where during any execution $\xi$, the adversity may crash *all but one* quorum $Q$ of a quorum system $Q$. Next, we define quorum system failures with respect to server crashes. For a quorum system $Q$ over the set of identifiers $S$, quorum $Q$ becomes *faulty* during an execution $\xi$ if process $i$ that belongs to $Q$, $i \in Q$, crashes.

**Definition 2.1.2 (Quorum Failure)** Let $Q$ be a quorum system defined over the set of replica servers $S$, $Q \subset 2^S$. A quorum $Q$ that belongs to $Q$, $Q \in Q$ becomes *faulty* if a process $p$ that
belongs to $Q$, $p \in Q$, contains a crash event fail$_p$ during an execution $\xi$ of an implementation $A, \xi \in \text{execs}(A, \mathcal{F})$.

If quorum $Q$ is not faulty in a state of an execution, then we say that quorum $Q$ is correct.

Next, we define quorum system failure during an execution.

**Definition 2.1.3 (Quorum System Failure)** Let $Q$ be a quorum system defined over the set of replica servers $\mathcal{S}, Q \subset 2^\mathcal{S}$. Then quorum system $Q$ becomes faulty if every quorum $Q$ that belongs in the quorum system $Q, \forall Q \in Q$, are faulty during an execution $\xi$ of an implementation $A, \xi \in \text{execs}(A, \mathcal{F})$.

As we discussed earlier, in parts of this work we assume a failure model $\mathcal{F}$ where during any execution $\xi$, the adversity may crash all but one quorum $Q$ of a quorum system $Q$. In other words, this failure model implies that no read or write operation can wait messages from more than one full quorum of server replicas. In case we let an operation $\pi$ to wait responses from more than one full quorum of server replicas before termination, then liveness (termination) may be violated since it may never receive those responses. Notice that the correct quorum $Q$ it is not known to any participating process $p \in \mathcal{I}$.

Works by Peleg and Wool in [62] and Naor and Wool in [59], 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, subsequent 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 [65] and by Gifford in [35], Matrix Quorum Systems used by Vitanyi and Awerbuch in [66], Crumbling Walls by Peleg and Wool in [62], Byzantine Quorum Systems by Malkhi and Reiter in [56], and Refined Quorum Systems by Guerraoui and Vukolić in [40].

In parts of this work we are interested in quorum-based implementations. In particular, those are implementations that use quorum systems to specify the subsets of servers that each read and write operation may access.

### 2.2 Consistency: Atomic Object Semantics

In this work we are interested in devising algorithms that implement atomic read/write registers. In this section we give a formal presentation of the consistency property of the distributed shared object implementations in terms of atomicity. The clients, readers and writers, of the atomic distributed shared memory service are modeled as sequential processes that access the shared objects through read and write operations. Note that the different processes accesses to a shared object may happen concurrently.

Let $X$ be the set of all the shared atomic read/write registers. 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 $x$. Let $i$ be the unique id of a server process.
Each memory access operation, read or write, starts with an *invocation* step and concludes with a *response*. In particular, a read operation that accesses object \( x \) at server \( i \) includes the invocation \( \text{read}x,i \) and the corresponding response, \( \text{readAck}(v)_{i,x} \). Similarly, a write operation that tries to write value \( v \) at object \( x \) maintained at server \( i \) has the invocation \( \text{write}(v)_{x,i} \) and the matching response, \( \text{writeAck}_{i,x} \). A read/write register \( x \in X \), is modeled with input actions as the invocation steps \( \{ \text{read}x,i, \text{write}(v)_{x,i} \} \) and matching output actions as the response steps \( \{ \text{readAck}(v')_{i,x}, \text{writeAck}_{i,x} \} \) where \( v,v' \in V_x \) and \( i \in I \). A complete distributed atomic shared memory implementation \( A \) is constructed as the composition of *countable, compatible* read/write registers \( x \), for \( x \in X \).

A read operation \( \rho \) is invoked from reader \( r \in R \) on object \( x \in X \), if during an execution \( \xi \) of the implementation \( A \), \( \xi \in \text{execs}(A) \), the invocation step \( \text{read}_{x,r} \) appears in \( \xi \). Read operation \( \rho \) is completed once the corresponding response step \( \text{readAck}(v)_{x,r} \) appears later in \( \xi \). Similarly for a write operation \( \omega \), we say that it is invoked from \( w \in W \) on object \( x \in X \) writing the value \( v \) in an execution \( \xi \) of the implementation \( A \), \( \xi \in \text{execs}(A) \), if the invocation step \( \text{write}(v)_{x,w} \) appears in \( \xi \). Write operation \( \omega \) terminates once the corresponding response step \( \text{writeAck}_{x,w} \) appears later in \( \xi \).

For any operation \( \pi \) either read or write, we denote its invocation step by \( \text{inv}(\pi) \) and the matching response step with \( \text{res}(\pi) \). Next, we define *operation completeness*.

**Definition 2.2.1 (Operation Completeness)** Operation \( \pi \) is *complete* in an execution \( \xi \) of an implementation \( A \), \( \xi \in \text{execs}(A) \), if execution \( \xi \) contains both \( \text{inv}(\pi) \) and its matching response step \( \text{res}(\pi) \) for \( \pi \). Otherwise, we say that \( \pi \) is *incomplete*. 

21
We assume that any processor invokes one operation at a time. In particular, a process does not invoke a new operation until it receives the response for a previously invoked operation.

We now define well-formed executions.

**Definition 2.2.2 (Well-Formedness)** An execution $\xi$ of an implementation $A$, $\xi \in \text{execs}(A)$, is well-formed if for any process $p$ that invokes an operation $\pi$ with $\text{inv}(\pi)$, then $\xi$ does not contain $\text{inv}(\pi')$ for an operation $\pi'$ at process $p$ before the response step $\text{res}(\pi)$ for $\pi$.

We use the characterizations precedes, succeeds and concurrent in order to describe the relation of two operations based on their invocation and response steps [49]. In particular, in an execution we say that operation $\pi_1$ precedes operation $\pi_2$, or that $\pi_2$ succeeds operation $\pi_1$, if the response step of $\pi_1$ precedes the invocation step of $\pi_2$ in the execution $\xi$. This is denoted by $\pi_1 \rightarrow \pi_2$. If neither of the operations precedes the other then we say that operations are concurrent. A formal definition for this notion is given below.

**Definition 2.2.3 (Precedence Relations)** Let operations $\pi_1$ and $\pi_2$ take place in an execution $\xi$ of an implementation $A$, $\xi \in \text{execs}(A)$. Then, we say that

- $\pi_1$ precedes $\pi_2$, denoted by $\pi_1 \rightarrow \pi_2$, if $\text{res}(\pi_1)$ appears before $\text{inv}(\pi_2)$ in $\xi$,

- $\pi_1$ succeeds $\pi_2$, denoted by $\pi_2 \rightarrow \pi_1$, if $\text{inv}(\pi_1)$ appears after $\text{res}(\pi_2)$ in $\xi$,

- $\pi_1$ is concurrent with $\pi_2$, denoted by $\pi_2 \leftrightarrow \pi_1$, if neither $\pi_1 \rightarrow \pi_2$ or $\pi_2 \rightarrow \pi_1$ appears in $\xi$.

Correctness of an implementation of an atomic shared read/write object is defined in terms of the properties of termination (liveness) and atomicity (safety). Termination ensures that an
an operation invoked from a process $p$ is going to terminate as long as $p$ is correct and the system obeys the failure model. This can be expressed more formally by the following definition.

**Definition 2.2.4 (Termination)** We say that an implementation $A$ in a given model of computation it satisfies termination if for any execution $\xi$ of the implementation $A$, $\xi \in \text{execs}(A)$, either $\xi$ is finite, or if $\xi$ contains an invocation step for an operation at a correct process $p$, then $\xi$ contains the corresponding response step.

Atomic consistency definition involves “shrinking” the duration of each operation in any execution to a chosen serialization point between the operation’s invocation and response, and requiring that the ordering of the operations according to the serialization points preserves their real-time ordering, and the resulting behavior of the object is consistent with its sequential specification. In particular, if a read is invoked after a write completes, then the read is guaranteed to return either the value of that write, or a value written by subsequent write that precedes the read. Additionally, if a read is invoked after another read completes, it returns the same or a “newer” value than the preceding read.

Whereas atomicity is often defined in terms of an equivalence with a serial memory, the definition given below implies this equivalence (as shown in in Lemma 13.16 in [54]), and is more convenient to use because it provides a usable recipe for proving atomic consistency. The definition is given in terms of a partial order on operations in any well-formed execution.

**Definition 2.2.5 (Atomicity [54])** An implementation $A$ of an object is atomic, if for any execution $\xi \in \text{execs}(A)$, if all the read and write operations that are invoked on an object complete, then the read and write operations for the object can be partially ordered by an ordering $\prec$, so that the following conditions are satisfied:
A1. The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations \( \pi_1 \) and \( \pi_2 \) such that \( \pi_1 \) completes before \( \pi_2 \) starts, yet \( \pi_2 \prec \pi_1 \).

A2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

A3. Every read operation ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns the initial value.

Atomicity is *compositional*. In particular, if a system is composed of multiple atomic object implementations then it follows that it preserves atomicity. Thus, it is possible to give a single atomic object implementation, and then provide a complete memory system by composing the implementations for individual objects.

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

### 2.3 Efficiency, Rounds and Message Exchanges

In assessing the efficiency of read and write operations of an implementation, we measure *communication latency*, *local computation time*, and *message complexity* of operations.

Communication latency of an operation is measured in terms of *communication rounds* or *communication exchanges*. The protocol implementing each operation involves a collection of sends of typed messages and the corresponding receives. A communication round is defined following [22].
Definition 2.3.1 (Communication Round [22]) A process $p$ performs a communication round during an operation $\pi$ in an execution $\xi$ of an implementation $A$, $\xi \in \text{execs}(A)$, if all the following hold:

1. process $p$ sends message(s) for operation $\pi$ to a set of processes $Z \subseteq I$,

2. upon the delivery of the message for $\pi$ to process $q$, $q \in Z$, $q$ sends a reply for $\pi$ to $p$ without waiting for any other messages, and

3. when $p$ receives the collection of replies that is deemed sufficient by the implementation, it terminates the round. After this either $p$ starts a new round or $\pi$ completes.

A communication exchange is defined as follows:

Definition 2.3.2 (Communication Exchange) Within an execution $\xi$ of implementation $A$, $\xi \in \text{execs}(A)$, a communication exchange is the set of sends and corresponding matching receives for a specific type of message within the protocol.

We can observe that a round in Definition 2.3.1 is composed of two exchanges: the first is comprised of sends in item (1) and the corresponding receives in item (2), and the second is comprised of the reply sends in item (2) and the corresponding receives in item (3). Thus, in essence each exchange constitutes “one half” of a round. Traditional implementations in the style of ABD are structured in terms of communication rounds, cf. [10, 33], each consisting of two exchanges. The first is a broadcast from a reader or writer process to the servers, and the second is a convergecast in which the servers send corresponding responses to the initiating process.
Computation time accounts for all local computation within an operation; here time complexity of local computation may be significant. When local computation is not more than a constant time per each message send and receive, we consider this to be insignificant relative to the communication latency of an operation. Otherwise, computation time needs to be assessed in addition to communication latency.

Message complexity measures the worst-case total number of messages exchanged during an operation \( \pi \), either read or write. Notice that the number of messages that a process \( p \) expects during a convergecast depends on the implementation.

### 2.4 Fastness

In this subsection we use the definitions of a *Round* (Def. 2.3.1) and an *Exchange* (Def. 2.3.2) to define *fast operations* and *fast implementations* as in [22]:

**Definition 2.4.1 (Fast Operations)** Consider an operation \( \pi \) invoked by process \( p \) in an execution \( \xi \) of some implementation \( A, \xi \in execs(A) \). We say that \( \pi \) is a *fast operation* if it completes when processor \( p \) performs a single communication round, or equivalent, two communication exchanges, between the invocation step and the response steps of \( \pi \). Otherwise, we say that \( \pi \) is slow.

**Definition 2.4.2 (Fast Implementation)** An implementation \( A \) is called *fast implementation* if in every execution \( \xi \) of \( A, \xi \in execs(A) \), it contains only fast operations.
2.5 Notation

For each symbol used in this thesis a short description is given in Table 1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Totally-ordered set of identifiers</td>
<td>2.1</td>
</tr>
<tr>
<td>R</td>
<td>Set of reader identifiers</td>
<td>2.1</td>
</tr>
<tr>
<td>r</td>
<td>Reader process</td>
<td>2.1</td>
</tr>
<tr>
<td>ρ</td>
<td>Read operation</td>
<td>2.1</td>
</tr>
<tr>
<td>W</td>
<td>Set of writer identifiers</td>
<td>2.1</td>
</tr>
<tr>
<td>w</td>
<td>Writer process</td>
<td>2.1</td>
</tr>
<tr>
<td>ω</td>
<td>Write operation</td>
<td>2.1</td>
</tr>
<tr>
<td>S</td>
<td>Set of replica server identifiers</td>
<td>2.1</td>
</tr>
<tr>
<td>s</td>
<td>Server process</td>
<td>2.1</td>
</tr>
<tr>
<td>π</td>
<td>Any operation (read or write)</td>
<td>2.1</td>
</tr>
<tr>
<td>ξ</td>
<td>An execution or an execution fragment</td>
<td>2.1</td>
</tr>
<tr>
<td>A</td>
<td>An atomic shared memory implementation</td>
<td>2.2</td>
</tr>
<tr>
<td>execs(A)</td>
<td>Set of all the executions of implementation A</td>
<td>2.1</td>
</tr>
<tr>
<td>F</td>
<td>Failure model assumed</td>
<td>2.1</td>
</tr>
<tr>
<td>execs(A, F)</td>
<td>Executions that adversity obeys failure model F in A</td>
<td>2.1</td>
</tr>
<tr>
<td>Q</td>
<td>A quorum system</td>
<td>2.1</td>
</tr>
<tr>
<td>Q</td>
<td>A quorum</td>
<td>2.1</td>
</tr>
<tr>
<td>X</td>
<td>Set of atomic read/write registers</td>
<td>2.2</td>
</tr>
<tr>
<td>x</td>
<td>Unique name of a read/write registers</td>
<td>2.2</td>
</tr>
<tr>
<td>V</td>
<td>Set of all the values</td>
<td>2.2</td>
</tr>
<tr>
<td>Vx</td>
<td>Set of values that atomic register x can be assigned</td>
<td>2.2</td>
</tr>
<tr>
<td>v</td>
<td>A single value</td>
<td>2.2</td>
</tr>
<tr>
<td>inv(π)</td>
<td>Invocation step of operation π</td>
<td>2.2</td>
</tr>
<tr>
<td>res(π)</td>
<td>Response step of operation π</td>
<td>2.2</td>
</tr>
<tr>
<td>Ei</td>
<td>The i&lt;sup&gt;th&lt;/sup&gt; communication exchange</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Table 1: List of symbols and descriptions.
Chapter 3

Consistent Distributed Memory Services

We begin by reviewing consistency semantics in Section 3.1. Next, we present several algorithmic approaches that implement consistent shared memory in static services for both SWMR and MWMR settings in Section 3.2. Lastly, we survey several approaches for providing consistent shared memory in dynamic systems in Section 3.3. The solutions we discuss are representative of the various design choices available for implementing distributed memory services, and we emphasize the trade-offs present in the different approaches.

3.1 Consistency

Lamport [49] defined three consistency semantics for a read/write register abstraction in the SWMR environment: safe, regular, and atomic. The safe register semantic ensures that (a) if a read operation is not concurrent with any write operation, it returns the last value written on the register; and, (b) if the read is concurrent with a write operation, then it returns any arbitrary value that is allowed to be written to the register. This consistency semantic is insufficient for
a distributed storage system since we can observe from property (b), that 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 defined, that is, the regular register. Similarly to the safe register, regularity ensures property (a). Additionally, 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 both cases, 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 [63].

Atomic semantics preserve all the properties of the regular register and overcome the above problem by ensuring that a read operation does not return an older value than the one returned by a preceding read operation. Atomicity provides the illusion of a single-copy object.

Herlihy and Wing in [45] 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 [48] and seriazability by Papadimitriou in [60]. A detailed comparison between sequential consistency and linearizability was conducted by Attiya and Welch in [11]. Subsequent works revisited and redefined the definitions provided
in [49, 45] for more specialized distributed systems. Lynch in [54] provided an equivalent definition of atomicity of [49] to describe atomic read/write objects in the MWMR environment. The new definition, totally orders write operations, and partially orders read operations with respect to the write operations. The definition is formally presented in Section 2.2.5.

3.2 Atomic Memory Under Crash Failures in Static Settings

We now survey several approaches that implement atomic shared memory in the asynchronous, message-passing, crash-prone, static setting. In Section 3.2.1 we present several solutions developed for the SWMR setting and in Section 3.2.2 solutions developed for the MWMR setting.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Model</th>
<th>Write Exch.</th>
<th>Read Exch.</th>
<th>Wrt Msg Comp</th>
<th>Rd Msg Comp</th>
<th>Client Participation</th>
<th>Local Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABD [10]</td>
<td>SWMR</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>S</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>FAST [22]</td>
<td>SWMR</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>S</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>SF [33]</td>
<td>SWMR</td>
<td>2 or 3</td>
<td>2</td>
<td>S</td>
<td></td>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>SLIQ [32]</td>
<td>SWMR</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>S</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>ccFAST [8]</td>
<td>SWMR</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>S</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>OHSAM (Section 4.2)</td>
<td>SWMR</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>S</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>OHSAM’ (Section 4.4)</td>
<td>SWMR</td>
<td>2 or 3</td>
<td>2</td>
<td>S</td>
<td></td>
<td>3</td>
<td>S</td>
</tr>
<tr>
<td>CCHYBRID (Section 5.1)</td>
<td>SWMR</td>
<td>2</td>
<td>2 or 4</td>
<td>2</td>
<td>S</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>OHFAST (Section 5.2)</td>
<td>SWMR</td>
<td>2</td>
<td>2 or 3</td>
<td>2</td>
<td>S</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>MR [57]</td>
<td>SWMR</td>
<td>2</td>
<td>2 or 3 or 4</td>
<td></td>
<td>S</td>
<td>^2</td>
<td>4</td>
</tr>
<tr>
<td>ERATO (Section 6.2)</td>
<td>SWMR</td>
<td>2</td>
<td>2 or 3</td>
<td>2</td>
<td>S</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ABD-MW [10, 55]</td>
<td>MWMR</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>S</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>SFW [23]</td>
<td>MWMR</td>
<td>2 or 4</td>
<td>2 or 4</td>
<td>4</td>
<td>S</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>CWF [30]</td>
<td>MWMR</td>
<td>4</td>
<td>2 or 4</td>
<td>4</td>
<td>S</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>OHMAM (Section 4.3)</td>
<td>MWMR</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>S</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>OHMAM’ (Section 4.4)</td>
<td>MWMR</td>
<td>4</td>
<td>2 or 3</td>
<td>4</td>
<td>S</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>ERATO-MW (Section 6.3)</td>
<td>MWMR</td>
<td>4</td>
<td>2 or 3</td>
<td>4</td>
<td>S</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2: Model, Communication Exchanges, Message Complexities, Participation Bounds, and Predicate Computational Class.
3.2.1 The SWMR Setting

Algorithms designed for single-writer static settings assume a fixed set known participants and accommodate some dynamic behaviors, such as asynchrony, transient failures, and permanent crashes within certain limits. A summary of the most relevant results for this setting is given in the first half of Table 2.

The seminal work of Attiya, Bar-Noy, and Dolev [10] provides an algorithm, colloquially referred to as ABD, that implements SWMR atomic objects in message-passing crash-prone asynchronous environments. This work won the Dijkstra Prize in 2011. In ABD, replication helps achieve fault-tolerance and availability, and the implementation replicates objects at servers, and it tolerates \( f \) replica servers crashes, provided a majority of replicas do not fail, i.e., \( |S| > 2f \). Read and write operations are ordered using logical timestamps associated with each written value. These timestamps totally order write operations, and therefore determine the values that read operations return. All operations terminate provided a majority of replicas do not crash.

A pseudocode for ABD is given in Algorithm 1; in referring to the numbered lines of code we use the prefix “L” to stand for “line”. The value of the object and its associated timestamp, as known by each process, are stored in variables \( v \) and \( ts \) respectively. Write operations involve a single communication round-trip consisting of two communication exchanges. The writer broadcasts its request to all replica servers during the first exchange and terminates once it collects acknowledgments from some majority of servers in the second exchange (L19-23). Each read operation takes two rounds involving in four communication exchanges. The reader broadcasts a read request to all replica servers in the first exchange, collects acknowledgments
from some majority of servers in the second exchange, and it discovers the maximum timestamp (L3-7). In order to ensure that any subsequent read will return a value associated with a timestamp at least as high as the discovered maximum, the reader propagates the value associated with the maximum timestamp to at least a majority of servers before completion (L8-11).

The correctness of this implementation, that is, atomicity, relies on the fact that any two majorities have a non-empty intersection. The local computation at readers, writers and servers in ABD incurs insignificant computational overhead.

Algorithm 1 Reader, Writer, and Server Protocols for SWMR algorithm ABD

1. At each reader $r$
2. function $READ(v$: output)
3. Get: broadcast $(get, i)$ to all replica servers
4. await responses $(get-ack, v', ts')$
5. from some majority of servers
6. Let $v$ be the value associated with the maximum timestamp $\maxts$ received
7. Put: broadcast $(put, v, \maxts, i)$ to all servers
8. await responses $(put-ack)$
9. from some majority of servers
10. return $(v)$
11. At each server $s$
12. State $v$ init $\bot$, $ts$ init $0$
13. Upon receive $(get, j)$
14. send $(get-ack, v, ts)$ to $j$

16. At each writer $w$
17. State $ts$ init $0$
18. function $WRITE(v$: input)
19. Put: $ts \leftarrow ts + 1$
20. broadcast $(put, v, ts, i)$ to all servers
21. await responses $(put-ack)$
22. from some majority of servers
23. return ()
24. At each server $s$
25. Upon receive $(put, v', ts', j)$
26. if $ts' > ts$ then
27. $(ts, v) \leftarrow (ts', v')$
28. send $(put-ack)$ to $j$

Following ABD, a folklore belief developed that in atomic memory implementations, “reads must write.” The work by Dutta et al. [22] refuted this belief by presenting an algorithm, called FAST, in which all read and write operations involve only two communication exchanges. Recall that such operations are called fast. To avoid the second round in read operations, FAST uses two mechanisms: (i) a recording mechanism at the servers, and (ii) a predicate that uses the server records at the readers. Here, each server records in a set all processes that witness its local timestamp and resets it whenever it learns a new timestamp. Each
reader explores the sets from the different server replies to determine whether “enough” processes witnessed the maximum observed timestamp. If the predicate holds, the reader returns the value associated with the maximum timestamp. Otherwise it returns the value associated with the previous timestamp. The predicate takes into account which processes witnessed the latest timestamp as it examines the intersection of the received sets.

It was also shown in [22] that atomic memory implementations are only possible when the number of readers is constrained in with respect to the number of replicas servers and in inverse proportion to the number of crashes as stated in the following theorem.

**Theorem 3.2.1 ([22])** Let $f \geq 1$, $|W| = 1$ and $|R| \geq 2$. If $|R| \geq \frac{|S|}{f} - 2$, then there is no fast atomic register implementation.

Fernández Anta, Nicolaou, and Popa [8], show that, although the result in [22] is efficient in terms of communication, it requires reader processes to evaluate a computationally hard predicate. The authors abstracted the predicate used in FAST as a computational problem that they show to be NP-hard via a reduction from the decision version of the Maximum Edge Biclique Problem [61], which is NP-Complete. This suggests the existence of a trade-off between communication efficiency and computational overhead in atomic memory implementations.

Given the inherent limitation on the number of readers in fast single-writer implementations, Georgiou et al. [33] sought a solution that would remove the limit on the number of readers, in exchange for slowing down some operations, i.e., the goal is to enable fast operations, but allow slower operations, taking more than two communication exchanges, when this is unavoidable. They provided a SWMR algorithm, named SF, that adopts an approach to implementing readers similar to the one in [22], but uses a polynomial time predicate to determine
whether it is safe for a read operation to terminate after two exchanges. In order not to place bounds on the number of readers, the authors group readers into abstract entities, called virtual nodes, serving as enclosures for multiple readers. This refinement has a non-trivial challenge of maintaining consistency among readers within the same virtual node. This solution trades communication for the scalability in the number of participating readers. In $S_F$ significant computational overheads incur in order to determine the speed of an operation (to evaluate the mentioned predicate). At most a single complete read operation performs four exchanges for each write operation. Writes and any read operation that precedes or succeeds a four exchange read, is fast. This development motivated creating a new class of implementations, called semifast implementations. Informally, an implementation is semifast if either all reads are fast or all write operations fast. Algorithm $S_F$ becomes fast (same as [22]) when each virtual node contains one reader.

Georgiou et al. [32] showed that fast and semifast quorum-based SWMR implementations are possible if and only if a common intersection exists among all quorums. Because a single point of failure exists in such solutions (i.e., any server in the common intersection), this renders such implementations not fault-tolerant. The same work introduced Quorum Views, client-side tools that examine the distribution of the latest value among the replicas in order to enable fast read operations (two exchanges) under read and write operation concurrency. The authors derived a SWMR algorithm, called SLIQ, that requires at least one single slow read per any write operation, and where all writes are fast. No bound is placed on the number of readers. SLIQ trades communication for the scalability in the number of participating readers. Here only insignificant computation effort is needed to examine the distribution of object values among the replicas that the reader receives during the read operation.
Another algorithm, called ccFAST, with a new predicate, is given by Fernández Anta et al. [8], that allows the operations to be fast with only polynomial computation overhead. The idea of the new predicate is to examine the replies received in the first communication round of a read operation and determine how many (instead of which [22]) processes witnessed the maximum timestamp among those replies. With this modification, the predicate takes polynomial time to decide the value to be returned and it reduces the size of each message sent by the replica nodes. Algorithm ccFAST is more practical than [22], but it has the same constraint on the number of readers.

A recent work by Mostefaoui and Raynal [57] defines what a time-efficient implementation of atomic registers is based on two different synchrony assumptions. The first assumes bounded message delays and is expressed in terms of delays, and the second assumes round-based synchrony. Authors then present a time-efficient implementation of atomic registers while trying to keep its design spirit as close as possible to ABD. We refer to this solution as algorithm MR. In the algorithm a write operation takes two communication exchanges and a read operation takes two, or three, or four exchanges. The heart of the given algorithm is the wait predicate that takes place on the servers side and it is associated with write operations. The wait predicate ensures both atomicity and the fact that the implementation is time-efficient. The trade-off between ABD and this implementation lies in the message complexity of write operations, which for ABD is linear and for MR is quadratic in to the number of replica servers. Algorithm MR is particularly interesting for registers used in read-dominated applications.
3.2.2 The MWMR Setting

We now discuss the multi-writer/multi-reader (MWMR) implementations of atomic memory. Whereas logical timestamps alone are sufficient to order write operations in the single-writer algorithms, the existence of multiple writers requires a somewhat different approach. The simplest approach is instead of a timestamp to use pairs consisting of a timestamp and processor id to order the written values. Such a pair is termed a *tag*. When a writer performs a write operation it associates the value with a tag \( \langle ts, id \rangle \), where \( ts \) is a logical timestamp, and \( id \) is the writer’s unique id that distinguishes the current write operation from all others. Tags are ordered lexicographically in establishing an order on the operations. A summary of the most relevant results for this setting is given in the second half of Table 2.

The work of Lynch and Schwarzmann [55] presented a multi-writer extension of algorithm ABD (and also introduced the notion of reconfigurable memory, where the set of replica servers can be dynamically reconfigured). The static version of their MWMR implementation, that we call ABD-mw, is given in Algorithm 2. In contrast with ABD, where the sole writer generates new timestamps without any communication, the writers in ABD-mw start a write operation by performing an additional round in which the replica servers are queried for their latest tags. Once tags are received from a majority of servers, the writer increments the timestamp of the highest detected timestamp to produce its new tag. The second round is performed as in ABD.

In more detail, the writer performs the “Get” round, broadcasting its request to the servers in the first exchange (L18). Servers reply with their latest timestamps in the second exchange.
The writer determines the highest timestamp among the replies, increments it, produces a new tag that includes its id, and then performs the “Put” round in which it in the third exchange broadcasts the new tag and the new value to all servers (L23-26). On the server side, if the incoming message contains a higher tag, then the server updates its local information and sends an acknowledgment in the fourth communication exchange (L27-31). The write protocol completes once the writer collects acknowledgments from a majority of servers. The first two exchanges ensure that the writer produces a tag that is higher than that of any preceding write. Thus a write operation for ABD-MW takes four exchanges in comparison with the two exchanges in ABD. The read protocol is identical to the four-exchange protocol in ABD, the only difference being that tags are used instead of timestamps. The correctness (atomicity) of this implementation, relies on the fact that any two majorities have a non-empty intersection and that in each round, the read and write protocols await responses from at least a majority of servers.

This algorithm places no constrains on the number of readers and writers, and it performs a modest amount of local computation, resulting in negligible computation overhead. This algorithm can also be used with quorum systems instead of majorities [55, 67], because the only property of majorities that is used is that any two majorities have a non-empty intersection, just like any two quorums. The failure model for the quorum based solution is that any pattern of crashes is tolerated, provided that the servers in at least one quorums do not crash.

Algorithm ABD-MW established that two rounds are sufficient to implement atomic read and write operations. The question of whether fast implementations are possible was answered
Algorithm 2 Reader, Writer, and Server Protocols for MWMR algorithm ABD-MW

1. At each reader \( r \)
2. \[ \text{function READ}(v; \text{output}) \]
3. \[ \text{Get: broadcast} \langle \text{get}, i \rangle \text{ to all replica servers} \]
4. \[ \text{await responses} \langle \text{get-ack}, v', \text{tag}' \rangle \]
5. \[ \text{from some majority of servers} \]
6. \[ \text{Let } v \text{ be the value associated with the} \]
7. \[ \text{maximum tag maxtag received} \]
8. \[ \text{Put: broadcast} \langle \text{put}, v, \text{maxtag}, r \rangle \text{ to all servers} \]
9. \[ \text{await responses} \langle \text{put-ack} \rangle \]
10. \[ \text{from some majority of servers} \]
11. \[ \text{return}(v) \]

12. At each server \( s \)
13. \[ \text{State } v \text{ init } \bot, \text{tag init } \langle 0, \bot \rangle \]
14. \[ \text{Upon receive} \langle \text{get}, j \rangle \]
15. \[ \text{send} \langle \text{get-ack}, v, \text{tag} \rangle \text{ to } j \]

16. At each writer \( w \)
17. \[ \text{function WRITE}(v; \text{input}) \]
18. \[ \text{Get: broadcast} \langle \text{get}, i \rangle \text{ to all replica servers} \]
19. \[ \text{await responses} \langle \text{get-ack}, v', \text{tag}' \rangle \]
20. \[ \text{from some majority of servers} \]
21. \[ \text{Let maxtag } = \langle ts, pid \rangle \text{ be the max tag} \]
22. \[ \text{Let newtag } = \langle ts + 1, w \rangle \]
23. \[ \text{Put: broadcast} \langle \text{put}, v, \text{newtag}, w \rangle \text{ to all servers} \]
24. \[ \text{await responses} \langle \text{put-ack} \rangle \]
25. \[ \text{from some majority of servers} \]
26. \[ \text{return}() \]
27. At each server \( s \)
28. \[ \text{Upon receive} \langle \text{put}, v', \text{tag}', j \rangle \]
29. \[ \text{if } \text{tag}' > \text{tag} \text{ then} \]
30. \[ (\text{tag}, v) \leftarrow (\text{tag}', v') \]
31. \[ \text{send} \langle \text{put-ack} \rangle \text{ to } j \]

in the negative in [22], where it was shown that fast reads are possible only in the single-writer model SWMR. In particular, fast MWMR implementations are impossible when the set of readers \( \mathcal{R} \) and the set of writers \( \mathcal{W} \) contain more than two nodes each.

**Theorem 3.2.2 ([22])** Let \( |\mathcal{W}| \geq 2, |\mathcal{R}| \geq 2, \) and \( f \geq 1. \) Any atomic register implementation has a run in which some complete read or write operation is not fast.

Moreover, Georgiou et al. [33] showed that semifast implementations (recall from Section 3.2.1 that in a semifast implementation either all reads are fast or all the write operations fast) are impossible in the MWMR setting.

**Theorem 3.2.3 ([33])** If \( |\mathcal{W}| \geq 2, |\mathcal{R}| \geq 2, \) and the number of server crashes \( f \geq 1, \) then semifast atomic register implementation is impossible.

These impossibility results motivated the development of algorithms that allow some operations to complete in less than two rounds (less than four communication exchanges). The work from Englert et al. [23] proposed hybrid approaches where some operations complete
in two and others in four exchanges. Their algorithm, called SFW, uses quorum systems and enables some reads and writes to be fast. In order to decide whether an operation can terminate after its first round, the algorithm employs two specialized predicates.

A later work of Georgiou et al. [30] showed that the predicates used in SFW are computationally hard (NP-hard), and fast write operations are enabled only if the quorum system satisfies certain quorum intersection properties, rendering the algorithm impractical. In order to make the evaluation of the predicates computational feasible, the authors presented a polynomial log-approximation algorithm and showed how to use it with algorithm SFW. In the same paper, they presented a MWMR algorithm, called CWFR, that allows fast read operations. The algorithm uses a generalization of client-side decision tools, Quorum Views, developed for the SWMR setting [32], to analyze the distribution of a value within a quorum of replies from servers to determine whether fast termination is safe. Since multiple writes can occur concurrently, an iterative technique is used to discover the latest potentially complete write operation. Here read operations terminate in either two or four communication exchanges. Algorithm CWFR does not impose constrains on participation and it performs a modest amount of local computation, resulting in negligible computation overhead.

### 3.3 Atomic Memory Under Crash Failures in Dynamic Settings

Additional challenges arise when a shared memory system must be long-lived and must ensure data longevity. A storage system may be able to tolerate failures of some servers, but over a long period it is conceivable that all servers may need to be replaced, because no servers are infallible, and also due to unavoidable changes or planned upgrades. Additionally, in mobile settings, e.g., remote search-and-rescue or military operations, it may be necessary to
provide migration of data from one collection of servers to another, so that the data can move as the needs dictate. Whether our concern is data longevity or mobility, the storage system must provide seamless runtime migration of data: one cannot stop the world and reconfigure the system in response to failures and changing environment.

We now survey several approaches for providing consistent shared memory in more dynamic systems, that is, where nodes may not only crash or depart voluntarily, but where new nodes may join the service, and where the entire collection of servers need to be replaced. In general, the set of object replicas can substantially evolve over time, ultimately migrating to a completely different set of replica hosts. Thus, an implementation designed for static settings, e.g., algorithm ABD, cannot be used directly in dynamic settings because it relies on the majority of original replica hosts to always be available. In order to use an ABD-like approach in dynamic settings, one must provide some means for managing the collections of replica hosts, and to ensure that readers and writers contact suitable such collections.

It is noteworthy that dealing with dynamic settings and managing collections of nodes does not directly address the provision of consistency in memory services. Instead, these issues are representative of the broader challenges present in the realm of dynamic distributed computing. It is illustrative that implementations of consistent shared memory services can sometimes be constructed using distributed building blocks, such as those designed for managing collections of participating nodes, for providing suitable communication primitives, and for reaching agreement (consensus) in dynamic distributed settings. A tutorial covering several of these topics is presented by Aguilera et al. [6].

We start by presenting the consensus problem because it provides a natural basis for implementing an atomic memory service by establishing an agreed-upon order of operations, and
because consensus is used in other ways in atomic memory implementations. Next we present group communication services (GCS) solutions that use strong communication primitives, such as totally ordered broadcast, to order operations. Finally we focus on approaches that extend the ideas of algorithm ABD to dynamic settings with explicit management of the evolving collections of replica hosts.

### 3.3.1 Consensus

Reaching agreement in distributed settings is a fundamental problem of computer science. The agreement problem in distributed settings is called consensus [55]. Here a collection of processes need to agree on a value, where each process may propose a value for consideration. Any solution must satisfy the following properties: Agreement: no two processes decide on different values; Validity: the value decided was proposed by some process; Termination: all correct processes reach a decision. Consensus is a powerful tool in designing distributed services [54], however, consensus is a notoriously difficult problem to solve in asynchronous systems, where termination cannot be guaranteed in the presence of even a single process crash [26] (this is the seminal FLP impossibility result of Fischer, Lynch, and Paterson); thus consensus must be used with care.

Consensus algorithms can be used directly to implement an atomic data service by enabling the participants to agree on a global total ordering of all operations [50]. The correctness (atomicity) here is guaranteed regardless of the choice of a specific consensus implementation, but the understanding of the underlying platform characteristics can guide the choice of the implementation for the benefit of system performance (for a tour de force of implementations see [54]). Nevertheless, using consensus for each operation is a heavy-handed approach, especially
given that perturbations may delay or even prevent termination. Thus, when using consensus, one must avoid invoking it in conjunction with individual memory operations, and make operations independent of the termination of consensus. We note that achieving consensus is a more difficult problem than implementing atomic read/write objects. In particular, consensus cannot be solved for two or more processes by using atomic read/write registers [44, 53].

3.3.2 The RAMBO Framework

RAMBO is a dynamic memory service supporting MWMR objects [36]. The name RAMBO stands for Reconfigurable Atomic Memory for Basic Objects. This algorithm uses configurations, each consisting of a set of replica hosts plus a quorum system defined over these hosts, and supports reconfiguration, by which configurations can be replaced. Notably, any quorum configuration may be installed at any time, and quorums from distinct configurations are not required to have non-empty intersections. The algorithm ensures atomicity in all executions. During quiescent periods when there are no reconfigurations, the algorithm operates similarly to algorithm ABD [9, 55]. To enable long-term operation of the service, quorum configurations can be reconfigured. Reconfigurations are performed concurrently with any ongoing read and write operations, and do not directly affect such operations. Additionally, multiple reconfigurations may be in progress concurrently. Reconfiguration involves two decoupled protocols: (1) introduction of a new configuration by the component called Recon, and (2) upgrade to the new configuration and garbage collection of obsolete configuration(s). Recon always emits a unique new configuration. Different reconfiguration proposals are reconciled by executing consensus among the members of an existing configuration. Termination of read and write
operations does not depend on termination of reconfiguration. It is the duty of a decoupled upgrade protocol to garbage collect old configurations and propagate the information about the object to the latest locally-known configuration. The main algorithm performs read and write operations using a two-phase strategy. The first, gathers information from the quorums of active configurations, then the second propagates information to the quorums of active configurations. During each phase new configurations may be discovered. To handle this each phase is terminated by a fixed point condition that involves a quorum from each active configuration.

Lastly, RAMBO is used as a framework for refinements and optimizations, and several subsequent works focused on practical considerations [37, 16, 28, 29, 46]. GeoQuorums [21] is an approach to implementing atomic shared memory on top of a physical platform that is based on mobile nodes moving in arbitrary patterns. The algorithm simplifies reconfiguration of RAMBO by using a finite set of possible configurations, and as the result it avoids the use of consensus. Here it is sufficient for a mobile node to discover the latest configuration, and contact and propagate the latest register information to all configurations.

### 3.3.3 Dynastore

Dynastore [5] is an implementation of a dynamic atomic memory service for multi-writer/multi-reader objects. The participants start with a default local configuration, that is, some common set of replica hosts. The algorithm supports three kinds of operations: read, write, and reconfig. The read and write operations involve two phases, and in the absence of reconfigurations, the protocol is similar to ABD. If a participant wishes to change its current configuration, it uses the reconfig operation and supplies with it a set of incremental changes.
The implementation of reconfig involves traversals of DAG’s representing possible sequences of changed configurations. In each traversal the DAG may be revised to reflect multiple changes to the same configuration. The assumption that a majority of the involved hosts are not removed and do not crash ensures that there is a path through the DAG that is guaranteed to be common among all hosts. The traversal terminates when a sink node is reached. The reconfig protocol involves two phases. The goal of the first phase is similar to the Get phase of ABD: discover the latest value-tag pair for the object. The goal of the second phase is similar to the Put phase of ABD: convey the latest value-tag pair to a suitable majority of replica hosts. The main difference is that these two phases are performed in the context of applying the incremental changes to the configuration, while at the same time discovering the changes submitted by other participants. This “bootstraps” possible new configurations. Given that all of this is done by traversing all possible paths—and thus configurations—in the DAG’s ensures that the common path is also traversed.

The read follows the implementation of reconfig, with the differences being: (a) the set of configuration changes is empty, and (b) the discovered value is returned to the client. The write also follows the implementation of reconfig, with the differences being: (a) the set of changes is empty, (b) a new, higher tag is produced upon the completion of the first phase, and (c) the new value-tag pair is propagated in the second phase.

We note that DynaStore implementation does not incorporate consensus for reconfiguration. On the other hand, reconfigurations are accomplished by additions and removals of individual nodes and this may lead to larger overheads as compared to approaches that evolve the system by replacing a complete configuration with another. Thus the latency of read and write
operations are more dependent on the rate of reconfigurations. Finally, in order to guarantee termination, DynaStore assumes that reconfigurations eventually subside.

3.3.4 Group Communication Services

Among the most important building blocks for distributed systems are group communication services (GCS) [12]. GCSs enable processes at different nodes of a network to operate collectively as a group by means of multicast services that deliver messages to the members of the group, and offer various guarantees about the order and reliability of delivery. The basis of a GCS is a group membership service. Each process, at any time, has a unique view of the group that includes a list of the processes in the group. Views can change over time, and may become different at different processes. Another important concept introduced by the GCS approach is virtual synchrony, where an essential requirement is that processes that proceed together through two consecutive views deliver the same set of messages between these views. This allows the recipients to take coordinated action based on the message, the membership set, and the rules prescribed by the application [12].

GCSs offer one approach for implementing shared memory. For example, one can implement a global totally ordered multicast service on top of a view-synchronous GCS [25]. The ordered multicast is used to impose an order on the memory access operations, yielding atomic memory. The main disadvantage in such solutions is that forming a new view takes time, and client memory operations are delayed (or aborted) during the view-formation period.

Another approach is to integrate a GCS with algorithm ABD as done in the dynamic primary configuration GCS of [19] that implements atomic memory by using techniques of [9] within each configuration, where configurations include a group view and a quorum system.
Here a \textit{configuration} combines a group view with a quorum system. Like other solutions based on GCSs, reads and writes are delayed during reconfiguration.

A general methodology for dynamic service replication is presented in [13]. This reconfiguration model unifies the virtual synchrony approach with state machine replication, as used in consensus solutions, in particular, Paxos [50].

\textbf{Discussion.} Providing efficient atomic implementations remains challenging for dynamic settings. Here the expectation is that solutions are found by integrating static algorithms with a reconfiguration framework so that during periods of relative stability one benefits from the efficiency of static algorithms, and where during the more turbulent times performance degrades gracefully when reconfigurations are needed. An open question here is whether consensus is truly necessary for implementing consistent memory services for long-lived dynamic systems.

The technical challenges and performance overheads in the dynamic setting may be the reasons why the existing distributed storage solutions shy away from atomic consistency guarantees. Commercial solutions, such as Google’s File System (GFS) [34], Amazon’s Dynamo [20], and Facebook’s Cassandra [47], provide less-than-intuitive, unproved guarantees. The concepts discussed in Section 3.3 are echoed in the design decisions of production systems. For instance, consensus is used in GFS [34] to ensure agreement on system configuration as it is done in \textsc{Rambo}; global time is used in Spanner [18] as it is done in \textsc{GeoQuorums}; replica access protocols in Dynamo [20] use quorums as in some approaches surveyed here. These examples provide motivation for pursuing rigorous algorithmic approaches in the study of consistent data services for dynamic networked systems. For a more detailed discussion, we direct the interested reader to related work that surveys atomic shared implementations for dynamic settings [58].
Chapter 4

Efficient Survivable Distributed Storage Implementations

In this chapter we present a new family of atomic read/write shared register implementations where read operations are able to complete in *three* communication exchanges without imposing constraints on the number of participants, i.e., we allow *unbounded* participation in the service. The aim is *One and a Half Rounds Atomic Memory*, hence the name OHRAM.

In Section 4.1 we discuss the communication pattern and the techniques used that allow us to devise atomic implementations where operations do not necessarily require complete communication round trips to terminate. In Sections 4.2 and 4.3, we present algorithms OHSAM and OHMAM for the SWMR and the MWMR setting, respectively. In Section 4.4 we revise the proposed algorithms to obtain implementations where read operations complete in either *two* or *three* communication exchanges. We rigorously reason about the correctness of all the proposed algorithms. Finally, in order to assess the practicality of the proposed algorithms, we simulate them along with existing comparable solutions using the NS3 simulator, and compare their performance in terms of operation latency under various scenarios, topologies and operation loads. Simulation results are discussed in Section 4.5.
4.1 OHRAM: One-and-a-Half Round Atomic Memory

From the literature we surveyed for the static setting (Section 3.2), we can observe that there exists a gap between the number of rounds that an operation of each algorithm takes to complete. An observation made in [22] suggests that atomic memory may be implemented (using a max/min technique) so that each read and write operation completes in three communication exchanges. In particular, while the replica servers update their local value to the one associated with the maximum timestamp received, the reader returns the value associated with the minimum timestamp discovered in the set of the received acknowledgment messages. We are interested in elaborating on the inherent limitations that such a technique may impose on the distributed system. We focus on the gap between one-round and two-rounds algorithmic solutions by presenting atomic memory implementations where read operations can take “one and a half rounds,” i.e., complete in three communication exchanges. In particular, we tackle the following problems by answering the research questions stated below:

**Research Question 4.1** Can we devise an atomic read/write shared objects implementation for the asynchronous, crash-prone, message-passing, static SWMR setting with unbounded participation, such that all read operations take three communication exchanges to complete? (Section 4.2).

**Research Question 4.2** Is it possible using the same three communication exchanges read protocol developed for the SWMR setting to devise a static MWMR implementation under the same assumptions? (Section 4.3).
**Research Question 4.3** Is it feasible to revise the read protocol implementing read operations for both SWMR and MWMR algorithms to yield a protocol that implements read operations that terminate in either two or three communication exchanges? (Section 4.4).

**Research Question 4.4** How the analytical results of the proposed algorithms are reflected in practical efficiency? (Section 4.5).

The answers to the above questions are presented in detail in the sections that follow.

### 4.2 The SWMR Setting

In this section, we present an algorithm that implements atomic shared memory for the static single-writer, multiple-reader (SWMR) shared register setting. The existence of only a single writer in this setting along with the fact that the writer performs at most one write operation at any given time (*well-formedness*), naturally leads to the total ordering of the write operations. In particular, following [10], each value written to the register is associated with a logical *timestamp*, which is a natural number used for totally ordering the write operations. This makes SWMR implementations somewhat more straightforward to reason about. However, it is still challenging to develop efficient implementations while tolerating processor failures and coping with asynchrony.

#### 4.2.1 Description of SWMR Algorithm OHSAM

We now present our SWMR algorithm OHSAM: *One and a Half rounds Single-writer Atomic Memory*. The write operations are fast, that is, they take two communication exchanges to complete (similarly to ABD [10]). We show that atomic operations do not need
to involve complete communication round trips between clients and servers. In particular, we allow server-to-server communication and we devise read operations that take three communication exchanges using the following communication pattern:

- **Exchange E1**: Initiated by a reader process $r$. Reader $r$ multicasts a request message to the participating replica servers.

- **Exchange E2**: A server process upon receiving the request message it then *relays* the request to a subset of replica servers.

- **Exchange E3**: Once a server receives “sufficient” relay messages for a particular read operation from a subset of servers, it sends an acknowledgment message to reader $r$.

The read completes once the invoker collects a majority of acknowledgment replies. A key idea of the algorithm is that the reader returns the value that is associated with the minimum timestamp. In particular, while the replica servers update their local value to the one associated with the maximum timestamp received, the reader returns the value associated with the minimum timestamp discovered in the set of the received acknowledgment messages. That is the value being written by the last complete write operation. The code for the reader and writer protocols is given in Algorithm 3 and for the server protocol in Algorithm 4. We now give the details of the protocols; in referring to the pseudocode of an algorithm, we use prefix “A” and for numbered lines of code we use the prefix “L” to stand for “line”. For example, A7:L29-31 stands for lines 29 to 31 of Algorithm 7.

Counter variables $read\_op$, $operations$ and $relays$ are used to help processes identify “new” read and write operations, and distinguish “fresh” from “stale” messages (since messages can be reordered). The value of the object and its associated timestamp, as known by each process, are stored in variables $v$ and $ts$ respectively. Set $r\_Ack$, at each reader $r$, stores
Algorithm 3 Reader and Writer Protocols for SWMR algorithm OHSAM

1: At each reader $r$
2: Variables:
3: $ts \in \mathbb{N}^+$: init 0, $minTS \in \mathbb{N}^+$: init 0, $v \in V$: init $\perp$
4: $read_{op} \in \mathbb{N}^+$: init 0, $rAck \subseteq S \times M$: init $\emptyset$
5: function READ
6: $read_{op} \leftarrow read_{op} + 1$
7: $rAck \leftarrow \emptyset$
8: broadcast ($\langle readRequest, r, read_{op} \rangle$) to $S$
9: wait until ($\lvert rAck \rvert = \lfloor \lvert S \rvert / 2 \rfloor + 1$)
10: $minTS \leftarrow \min\{m.ts | m \in rAck\}$
11: $v \leftarrow \{m.val | m \in rAck \land m.ts' = minTS\}$
12: return ($v$)
13: Upon receive $m$ from $s$
14: if $m.read_{op} = read_{op}$ then
15: $rAck \leftarrow rAck \cup \{(s, m)\}$
16: At writer $w$
17: Variables:
18: $ts \in \mathbb{N}^+$: init 0, $v \in V$: init $\perp$, $wAck \subseteq S \times M$: init $\emptyset$
19: function WRITE($val : input$)
20: $(ts, v) \leftarrow (ts + 1, val)$
21: $wAck \leftarrow \emptyset$
22: broadcast ($\langle writeRequest, ts, v, w \rangle$) to $S$
23: wait until ($\lvert wAck \rvert = \lfloor \lvert S \rvert / 2 \rfloor + 1$)
24: return
25: Upon receive $m$ from $s$
26: if $m.ts = ts$ then
27: $wAck \leftarrow wAck \cup \{(s, m)\}$

all the received acknowledgment messages. Variable $minTS$ holds the minimum timestamp discovered in the set of the received acknowledgment messages $rAck$. Below we provide a brief description of the protocol of each participant of the service.

Writer Protocol. Writer $w$ increments its local timestamp $ts$ and broadcasts a writeRequest message to all the participating replica servers $s \in S$ during exchange $E1$ (A3:L20-22). The write operation completes once the writer receives writeAck messages from at least a majority of servers, $\lfloor \lvert S \rvert / 2 \rfloor + 1$, during exchange $E2$ (A3:L23-23).

Reader Protocol. When a read process $r$ invokes a read operation it first monotonically increases its local read operation counter $read_{op}$ and empties the set of the received acknowledgment messages, $rAck$ (A3:L6-7). Then, it creates a readRequest message in which it
Algorithm 4 Server Protocols for SWMR algorithm OhtSAM

28. At server $s_i$:
29. Variables:
30. $ts \in \mathbb{N}^+$: init 0, $v \in V$: init ⊥
31. $operations[1..|R|+1]$ : array of int, $relays[1..|R|+1]$: array of int
32. Initialization:
33. $operations[i] \leftarrow 0$ for $i \in \mathcal{R}$, $relays[i] \leftarrow 0$ for $i \in \mathcal{R}$
34. Upon receive((readRequest, $r$, read_op))
35. broadcast((readRelay, $ts, v, r$, read_op, $s_i$)) to $S$
36. Upon receive((writeRequest, $ts', v'$, $w$))
37. if ($ts < ts'$) then
38. ($ts,v) \leftarrow (ts',v')$
39. send(⟨writeAck, $ts,v,s_i$⟩) to $w$
40. Upon receive((readRelay, $ts', v', r$, read_op, $s_i$))
41. if ($ts < ts'$) then
42. ($ts,v) \leftarrow (ts',v')$
43. if ($operations[r] < read_op$) then
44. $operations[r] \leftarrow read_op$
45. $relays[r] \leftarrow 0$
46. if ($operations[r] = read_op$) then
47. $relays[r] \leftarrow relays[r] + 1$
48. if ($relays[r] = |S|/2 + 1$) then
49. send(⟨readAck, $ts,v$, read_op, $s_i$⟩) to $r$

encloses its id and local read counter and it broadcasts this request message to all the participating servers $s \in S$, forming exchange $E_1$ (A3:L8). It then waits to collect readAck messages from $E_3$ from at least $|S|/2 + 1$ servers. While collecting readAck messages from exchange $E_3$, reader $r$ discards any delayed messages from previous operations due to asynchrony. When “fresh” messages are collected from a majority of servers, then the reader returns the value $v$ associated with the minimum timestamp, $minTS$, among the set of the received acknowledgment messages, $rAck$ (A3:L10-12).

Server Protocol. Each server $s \in S$ expects three types of messages:

(1) Upon receiving a ⟨readRequest, $r$, read_op⟩ message the server creates a readRelay message, containing its $ts$, $v$, and its id $s$, and it broadcasts it to all the servers $S$ (A4:L34-35).

(2) Upon receiving a ⟨readRelay, $ts'$, $v'$, $r$, read_op⟩ message, server $s$ compares its local timestamp $ts$ with $ts'$ enclosed in the message. If $ts < ts'$, then $s$ sets its local timestamp
and value to those enclosed in the message (A4:L41-42). In any other case, no updates are taking place. As a next step s checks if the received readRelay message marks a new read operation by r. This is achieved by checking if reader’s r operation counter is newer than the local one, i.e., read_op > operations[r] (A4:L43). If this holds, then s: a) sets its local read operation counter for reader r to be equal to the received counter, i.e., operations[r] = read_op; and b) re-initializes the relay counter for r to zero, i.e., relays[r] = 0 (A4:L43-45).

Server s also updates the number of collected readRelay messages regarding the read request created by reader r (A4:L46-47). When s receives ⟨readRelay, ts, v, read_op, ri⟩ messages from a majority of servers, it creates a ⟨readAck, ts, v, read_op, s⟩ message in which it encloses its local timestamp and value, its id, and the reader’s operation counter and sends it to the requesting reader r (A4:L48-49).

(3) Upon receiving a ⟨writeRequest, ts', v', w⟩ message, server s compares its local timestamp ts with the received one, ts'. If ts < ts', then the server sets its local timestamp and value to be equal to those in the received message (A4:L37-38). In any other case, no updates are taking place. Finally, the server sends an acknowledgement, writeAck, message to the requesting writer (A4:L39).

4.2.2 Correctness of OHSAM

To prove correctness of algorithm OHSAM we reason about its liveness (termination) and atomicity (safety).

Liveness. Termination holds with respect to our failure model: up to f servers may fail by crashing, where f < |S|/2 and each operation waits for messages from some majority of servers. We now give more detail on how each operation satisfies liveness.
Write Operation. Per algorithm OHSAM, writer \( w \) creates a writeRequest message and then it broadcasts it to all servers in exchange \( E_1 \) (A3:L22). Writer \( w \) then waits for writeAck messages from a majority of servers from \( E_2 \) (A3:L23). Since in our failure model up to \( f < \frac{|S|}{2} \) servers may crash, writer \( w \) collects writeAck messages form a majority of live servers during \( E_2 \) and the write operation \( \omega \) terminates.

Read Operation. The reader \( r \) begins by broadcasting a readRequest message all the servers forming exchange \( E_1 \). Since \( f < \frac{|S|}{2} \), then at least a majority of servers receives the readRequest message sent in \( E_1 \). Any server \( s \) that receives this message it then broadcasts a readRelay message to all the servers, forming \( E_2 \), (A4:L34-35), and no server ever discards any incoming readRelay messages. Any server, whether it is aware or not of the readRequest, always keeps a record of the incoming readRelay messages and takes action as if it is aware of the readRequest. The only difference between server \( s_i \) that received a readRequest message and server \( s_k \) that did not, is that \( s_i \) is able to broadcast a readRelay message, and \( s_k \) broadcasts a readRelay message when it receives the corresponding readRequest message (A4:L34-35). Each non-failed server receives readRelay messages from a majority of servers during \( E_2 \) and sends a readAck message to the requesting reader \( r \) at \( E_3 \) (A4:L46-47). Therefore, reader \( r \) collects readAck messages from a majority of servers during \( E_3 \), and the read operation terminates (A3:L10-12).

Based on the above, it is always the case that acknowledgment messages readAck and writeAck are collected from at least a majority of servers in any read and write operation, thus ensuring liveness.

Atomicity. To prove atomicity we order the operations with respect to timestamps written and returned. More precisely, for each execution \( \xi \) of the algorithm there must exist a partial order
≺ on the operations in on the set of completed operations \( \Pi \) that satisfy conditions A1, A2, and A3 as given in Definition 2.2.5 (Section 2.2). Let \( ts_\pi \) be the value of the timestamp at the completion of \( \pi \) when \( \pi \) is a write, and the timestamp computed as the maximum \( ts \) at the completion of a read operation \( \pi \). With this, we denote the partial order on operations as follows. For two operations \( \pi_1 \) and \( \pi_2 \), when \( \pi_1 \) is any operation and \( \pi_2 \) is a write, we let \( \pi_1 \prec \pi_2 \) if \( ts_{\pi_1} < ts_{\pi_2} \). For two operations \( \pi_1 \) and \( \pi_2 \), when \( \pi_1 \) is a write and \( \pi_2 \) is a read we let \( \pi_1 \prec \pi_2 \) if \( ts_{\pi_1} \leq ts_{\pi_2} \). The rest of the order is established by transitivity and reads with the same timestamps are not ordered. We now state and prove a series of lemmas leading to the main correctness result.

It is easy to see that the \( ts \) variable in each server \( s \) is monotonically increasing. This leads to the following lemma.

**Lemma 4.2.1** In any execution \( \xi \) of OHSAM, the variable \( ts \) maintained by any server \( s \) in the system is non-negative and monotonically increasing.

**Proof.** When a server \( s \) receives a timestamp \( ts \) then \( s \) updates its local timestamp \( ts_s \) if and only if \( ts > ts_s \) (A4:L37-38 and A4:L41-42). Thus the local timestamp of the server monotonically increases and the lemma follows. \( \square \)

Next, we show that if a read operation \( \rho_2 \) succeeds read operation \( \rho_1 \), then \( \rho_2 \) always returns a value at least as recent as the one returned by \( \rho_1 \).

**Lemma 4.2.2** In any execution \( \xi \) of OHSAM, if \( \rho_1 \) and \( \rho_2 \) are two read operations such that \( \rho_1 \) precedes \( \rho_2 \), i.e., \( \rho_1 \rightarrow \rho_2 \), and \( \rho_1 \) returns the value for timestamp \( ts_1 \), then \( \rho_2 \) returns the value for timestamp \( ts_2 \geq ts_1 \).
Proof. Let the two operations $\rho_1$ and $\rho_2$ be invoked by processes with identifiers $r_1$ and $r_2$ respectively (not necessarily different). Also, let $RSet_1$ and $RSet_2$ be the sets of servers that sent a readAck message to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$.

Assume by contradiction that read operations $\rho_1$ and $\rho_2$ exist such that $\rho_2$ succeeds $\rho_1$, i.e., $\rho_1 \rightarrow \rho_2$, and the operation $\rho_2$ returns a timestamp $ts_2$ that is smaller than the $ts_1$ returned by $\rho_1$, i.e., $ts_2 < ts_1$. According to our algorithm, $\rho_2$ returns a timestamp $ts_2$ that is smaller than the minimum timestamp received by $\rho_1$, i.e., $ts_1$, if $\rho_2$ obtains $ts_2$ and $v$ in the readAck message of some server $s_x \in RSet_2$, and $ts_2$ is the minimum timestamp received by $\rho_2$.

Let us examine if $s_x$ replies with $ts'$ and $v'$ to $\rho_1$, i.e., $s_x \in RSet_1$. By Lemma 4.2.1, and since $\rho_1 \rightarrow \rho_2$, then it must be the case that $ts' \leq ts_2$. According to our assumption $ts_1 > ts_2$, and since $ts_1$ is the smallest timestamp sent to $\rho_1$ by any server in $RSet_1$, then it follows that $r_1$ does not receive the readAck message from $s_x$, and hence $s_x \notin RSet_1$.

Now let us examine the actions of the server $s_x$. From the algorithm, server $s_x$ collects readRelay messages from a majority of servers in $S$ before sending a readAck message to $\rho_2$ (L48-49). Let $RRSet_{s_x}$ denote the set of servers that sent readRelay to $s_x$. Since, both $RRSet_{s_x}$ and $RSet_1$ contain some majority of the servers then it follows that $RRSet_{s_x} \cap RSet_1 \neq \emptyset$.

Thus there exists a server $s_i \in RRSet_{s_x} \cap RSet_1$, which sent (i) a readAck to $r_1$ for $\rho_1$, and (ii) a readRelay to $s_x$ during $\rho_2$. Note that $s_i$ sends a readRelay for $\rho_2$ only after it receives a read request from $\rho_2$ (L34-35). Since $\rho_1 \rightarrow \rho_2$, then it follows that $s_i$ sent the readAck to $\rho_1$ before sending the readRelay to $s_x$. By Lemma 4.2.1, if $s_i$ attaches a timestamp $ts_{s_i}$ in the readAck to $\rho_1$, then $s_i$ attaches a timestamp $ts'_{s_i}$ in the readRelay message to $s_x$, such that $ts'_{s_i} \geq ts_{s_i}$. Since $ts_1$ is the minimum timestamp received by $\rho_1$, then $ts_{s_i} \geq ts_1$, and
hence $ts'_1 \geq ts_1$ as well. By Lemma 4.2.1, and since $s_x$ receives the readRelay message from $s_i$ before sending a readAck to $\rho_2$, it follows that $s_x$ sends a timestamp $ts_2 \geq ts'_i$. Thus, $ts_2 \geq ts_1$ and this contradicts our initial assumption. □

The next lemma shows that any read operation following a write operation receives readAck messages from servers where each included timestamp is at least as large as the one returned by the complete write operation.

**Lemma 4.2.3** In any execution $\xi$ of OHSAM, if a read operation $\rho$ succeeds a write operation $\omega$ that writes $ts$ and $v$, i.e., $\omega \rightarrow \rho$, and receives readAck messages from a majority of servers $\mathcal{R}$, then each $s \in \mathcal{R}$ sends a readAck message to $\rho$ with a timestamp $ts_s$ s.t. $ts_s \geq ts$.

**Proof.** Let $\mathcal{W}$ be the set of servers that send a writeAck message in $\omega$ and let $\mathcal{R}$ be the set of servers that send readRelay messages to server $s$.

By Lemma 4.2.1, if a server $s$ receives timestamp $ts$ from process $p$, then $s$ includes timestamp $ts'$ s.t. $ts' \geq ts$ in any subsequent message. This, means that every server in $\mathcal{W}$, sends a writeAck message to $\omega$ with a timestamp greater or equal to $ts$. Hence, every server $s_x \in \mathcal{W}$ has timestamp $ts_s$ s.t. $ts_s \geq ts$. Let us now examine timestamp $ts_s$ that server $s \in \mathcal{R}$ sends in read operation $\rho$.

Before server $s$ sends a readAck message in $\rho$, it must receive readRelay messages from the majority of servers, $\mathcal{R}$ (L48-49). Since both $\mathcal{W}$ and $\mathcal{R}$ contain a majority of servers, then $\mathcal{W} \cap \mathcal{R} \neq \emptyset$. By Lemma 4.2.1, any server $s_x \in \mathcal{W} \cap \mathcal{R}$ has a timestamp $ts_{s_x}$ s.t. $ts_{s_x} \geq ts$. Since server $s_x \in \mathcal{R}$ and from the algorithm, server’s $s$ timestamp is always updated to the highest timestamp it receives (A4:L41-42), then when server $s$ receives the message from $s_x$, it updates its timestamp $ts_s$ s.t. $ts_s \geq ts_{s_x}$. Thus, by
Lemma 4.2.1, each $s \in RSet$ sends a readAck (A4:L48-49) in $\rho$ with a timestamp $ts_s$ s.t. $ts_s \geq ts_{x_s} \geq ts$. Therefore, $ts_s \geq ts$ holds and the lemma follows.

Next we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one that was written.

**Lemma 4.2.4** In any execution $\xi$ of OHSAM, if a read $\rho$ succeeds a write operation $\omega$ that writes timestamp $ts$, i.e. $\omega \rightarrow \rho$, and $\rho$ returns a timestamp $ts'$, then $ts' \geq ts$.

**Proof.** Suppose that read $\rho$ receives readAck messages from a majority of servers $RSet$. By lines 10-12 of the algorithm, it follows that $\rho$ decides on the minimum timestamp, $ts' = ts_{\min}$, among all the timestamps in the readAck messages in $RSet$. From Lemma 4.2.3, $ts_{\min} \geq ts$ holds, where $ts$ is the timestamp written by the last complete write operation $\omega$, then $ts' = ts_{\min} \geq ts$ also holds. Thus, $ts' \geq ts$ holds and the lemma follows.

We now show the correctness of algorithm OHSAM.

**Theorem 4.2.5** Algorithm OHSAM implements an atomic SWMR object.

**Proof.** We now use the lemmas stated above and the operations order definition to reason about each of the three atomicity conditions A1, A2 and A3 as given in Definition 2.2.5 [54].

**A1** For any $\pi_1, \pi_2 \in \Pi$ such that $\pi_1 \rightarrow \pi_2$, it cannot be that $\pi_2 \prec \pi_1$.

When the two operations $\pi_1$ and $\pi_2$ are reads and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 4.2.2 it follows that the timestamp returned from $\pi_2$ is always greater or equal to the one returned from $\pi_1$, $ts_{\pi_2} \geq ts_{\pi_1}$. If $ts_{\pi_2} > ts_{\pi_1}$ then by the ordering definition $\pi_1 \prec \pi_2$ is satisfied. When $ts_{\pi_2} = ts_{\pi_1}$ then the ordering is not defined, thus it cannot be the case that $\pi_2 \prec \pi_1$. If $\pi_2$ is a write, the sole writer generates a new timestamp by incrementing the largest timestamp in the
system. By well-formedness, any timestamp generated by the writer for any write operation that precedes \( \pi_2 \) must be smaller than \( ts_{\pi_2} \). Since \( \pi_1 \rightarrow \pi_2 \), then it holds that \( ts_{\pi_1} < ts_{\pi_2} \). Hence, by the ordering definition it cannot be the case that \( \pi_2 \prec \pi_1 \). Lastly, when \( \pi_2 \) is a read and \( \pi_1 \) a write and \( \pi_1 \rightarrow \pi_2 \) holds, then from Lemmas 4.2.3 and 4.2.4 it follows that \( ts_{\pi_2} \geq ts_{\pi_1} \). By the ordering definition, it cannot hold that \( \pi_2 \prec \pi_1 \) in this case either.

**A2** For any write \( \omega \in \Pi \) and any operation \( \pi \in \Pi \), then either \( \omega \prec \pi \) or \( \pi \prec \omega \).

If the timestamp returned from \( \omega \) is greater than the one returned from \( \pi \), i.e. \( ts_{\omega} > ts_{\pi} \), then \( \pi \prec \omega \) follows directly. Similarly, if \( ts_{\omega} < ts_{\pi} \) holds, then \( \omega \prec \pi \) follows. If \( ts_{\omega} = ts_{\pi} \), then it must be that \( \pi \) is a read and \( \pi \) discovered \( ts_{\omega} \) as the minimum timestamp in at least a majority of servers. Thus, \( \omega \prec \pi \) follows.

**A3** Every read operation returns the value of the last write preceding it according to \( \prec \) (or the initial value if there is no such write).

Let \( \omega \) be the last write preceding read \( \rho \). From our definition it follows that \( ts_{\rho} \geq ts_{\omega} \). If \( ts_{\rho} = ts_{\omega} \), then \( \rho \) returned the value written by \( \omega \) on a majority of servers. If \( ts_{\rho} > ts_{\omega} \), then it means that \( \rho \) obtained a larger timestamp. However, the larger timestamp can only be originating from a write that succeeds \( \omega \), thus \( \omega \) is not the preceding write and this cannot be the case. Lastly, if \( ts_{\rho} = 0 \), no preceding writes exist, and \( \rho \) returns the initial value. \( \square \)

### 4.2.3 Performance of OHSAM

We now assess the performance of OHSAM in terms of (i) latency of read and write operations as measured by the number of communication exchanges, (ii) the message complexity of read and write operations and (iii) computational complexity.
In brief, for algorithm OHSAM write operations take 2 communication exchanges and read operations take 3 communication exchanges. The (worst case) message complexity of read operations is $|S|^2 + 2|S|$ and the (worst case) message complexity of write operations is $2|S|$. This follows directly from the structure of the algorithm. We now give additional details.

**Operation Latency.** We study the operation latency, in terms of the number of communication exchanges required by each operation.

*Write operation latency:* According to algorithm OHSAM, writer $w$ sends a writeRequest message to all the servers in exchange $E_1$, and, awaits for writeAck messages from at least a majority of servers in exchange $E_2$. Once the writeAck messages are received, no further communication is required and the write operation terminates. Therefore, any write operation consists of 2 communication exchanges.

*Read operation latency:* A reader sends a readRequest message to all the servers in the first communication exchange $E_1$. Once a server receives a readRequest message, it broadcasts a readRelay message to all the servers in exchange $E_2$. Any active servers now await readRelay messages from at least a majority of servers, and then, the servers send a readAck message to the reader during communication exchange $E_3$. We note that servers do not reply to any incoming readRelay messages. Thus, a read operation consists of 3 communication exchanges.

**Message Complexity.** We measure operation message complexity as the worst case number of exchanged messages in each read and write operation. The worst case number of messages corresponds to failure-free executions where all participants send messages to all destinations according to the protocols.
**Write operation:** A single write operation in algorithm OHSAM takes 2 communication exchanges. In the first exchange $E_1$, the writer sends a writeRequest message to all the servers in $S$. The second exchange $E_2$, occurs when all servers in $S$ send a writeAck message to the writer. Thus, at most $2|S|$ messages are exchanged in a write operation.

**Read operation:** Read operations take 3 communication exchanges. Exchange $E_1$ occurs when a reader sends a readRequest message to all the servers in $S$. Exchange $E_2$ occurs when servers in $S$ send a readRelay message to all the servers in $S$. The last exchange, $E_3$, occurs when servers in $S$ send a readAck message to the requesting reader. Therefore, $|S|^2 + 2|S|$ messages are exchanged during a read operation.

**Computational Complexity.** Algorithm OHSAM performs a modest amount of local computation, resulting in negligible computation overhead.

### 4.3 The MWMR Setting

In the SWMR setting the focus is primarily on the ordering of concurrent reads. Since the solo writer performs at most one write operation at any given time, this ensures the total ordering on the write operations. At this point, we relax the constraint on the number of writer processes that can participate in the system and invoke write operations, leading to the static Multiple-Writer, Multiple-Reader (MWMR) shared register setting.

In this setting, the ordering of write operations becomes more challenging, due to the fact that multiple write operations may be invoked on the shared register concurrently. In particular, a series of natural questions arise in this new relaxed environment:

- *How one can order concurrent writes and how a reader may distinguish the values written by different writers?*
What is the value of the shared register when concurrent writes may access different replicas in a different order?

What is the “safe” value to be returned from a reader process during periods of read/write concurrency while ensuring atomicity?

Atomicity requires that the shared register implementations must establish strategies to totally order concurrent write operations, while reads must be ordered with respect to the order defined for the writes. Following [55], to impose an ordering on the values written by the writers we associate each value with a tag $tg$ defined as the pair $(ts, id)$, where $ts$ is a timestamp and $id$ is the identifier of a writer. Tags are ordered lexicographically, that is, a tag $tg_a > tg_b$ if either (i) $(tg_a.ts > tg_b.ts)$ or (ii) $(tg_a.ts = tg_b.ts) \land (tg_a.id > tg_b.id)$ holds.

4.3.1 Description of MWMR Algorithm OHMAM

We seek a solution for the MWMR setting that involves three communications exchanges per read operation and four exchanges per write operation. We now present our MWMR algorithm OHMAM: One and a Half rounds Multi-writer Atomic Memory. The read protocol is identical to the one presented in Section 4.2.1 for algorithm OHSAM (except that tags are used instead of timestamps). Thus, we provide the full code for OHMAM in Algorithms 5 and 6 but we describe only the protocols that differ, and that is, the writer and server protocols.

**Writer Protocol.** This protocol is similar to the one presented in [55]. When a write operation is invoked, a writer $w$ monotonically increases its local write operation counter write_op, empties the set $mAck$ that holds the received acknowledgment messages (A5:L20-21), and it broadcasts a discover message to all servers $s \in S$ (A5:L22) in exchange E1. It then waits to collect
discoverAck messages from a majority of servers, $|\mathcal{S}|/2 + 1$, from exchange $E_2$. While collecting discoverAck messages from $E_2$, writer $w$ checks the $write_op$ variable that is included in the message $m$ and discards any message where the value of $write_op < m.write_op$ (A5:L31 - 33). This, happens in order to avoid any delayed discoverAck messages sent during previous write operations. Once the discoverAck messages are collected, writer $w$ determines the maximum timestamp $maxTS$ from the tags included in the received messages (A5:L24) and creates a new local tag $tg$, in which it assigns its id and sets the timestamp to one higher than the maximum one, $tg = \langle maxTS + 1, w \rangle$ (A5:L25). The writer then broadcasts a writeRequest message during $E_3$, including the updated tag $tg$, the value $v$ to be written, its write operation counter $write_op$, and its id $w$, to all the participating servers (A5:L28). It then waits to collect $|\mathcal{S}|/2 + 1$ writeAck messages from $E_4$ before completion (A5:L29).

**Server Protocol.** Each replica server $s$ is waiting for four types of messages: a) $readRequest$; b) $discover$; c) $writeRequest$; and d) $readRelay$. Servers react to messages regarding read operations, i.e., $readRequest$ and $readRelay$ messages, exactly as in algorithm OHSAM (Section 4.2.1). Thus, here we describe server actions for $discover$ and $writeRequest$ messages.

1. Upon receiving a $\langle discover, write_op, w \rangle$ message, server $s$ attaches its local tag $tg$ and local value $v$ in a new discoverAck message that it sends back to the requesting writer $w$ (A6:L56-57).

2. Upon receiving $\langle writeRequest, \langle tg', v' \rangle, write_op, w \rangle$ message, server compares its local tag $tg$ with the received tag $tg'$. If the message is not stale and server's local tag is older, $tg < tg'$, it updates its local timestamp and local value to those received (A6:L59-61). Otherwise, no update takes place. Server $s$ acknowledges the requesting writer $w$ by creating and sending it a writeAck message (A6:L62).
Algorithm 5 Reader and Writer Protocols for MWMR algorithm OHAMAM

1. At each reader $r$
2. Components:
   3. $t_g \in (\mathbb{N}^+, I)$: init $0, r$
   4. $\text{minTAG} \in (\mathbb{N}^+, \mathbb{N}^+)$: init $0, 0$
   5. $v \in V$: init $\bot$, $\text{readOp} \in \mathbb{N}^+$: init $0$
   6. $\text{rAck} \subseteq S \times M$: init $\emptyset$

5. function $\text{READ}$
6. $\text{readOp} \leftarrow \text{readOp} + 1$, $\text{rAck} \leftarrow \emptyset$
7. broadcast $(\text{readRequest}, r, \text{readOp})$ to $S$
8. wait until $(|\text{rAck}| = |S|/2 + 1)$
9. $\text{minTAG} \leftarrow \min \{(m.tg' | m \in \text{rAck})\}$
10. $v \leftarrow \{m.val | m \in \text{rAck} \land m.tg' = \text{minTAG}\}$
11. return($v$)

12. Upon receive $m$ from $s$
13. if $m.\text{readOp} = \text{readOp}$ then
14. $\text{rAck} \leftarrow \text{rAck} \cup \{(s, m)\}$

At each writer $w$

Variables:
17. $t_g \in (\mathbb{N}^+, I)$: init $0, w$
18. $\text{writeOp} \in \mathbb{N}^+$: init $0$
19. $\text{maxTS} \in \mathbb{N}^+$: init $0$
20. $\text{mAck} \subseteq S \times M$: init $\emptyset$

21. function $\text{WRITE}(\text{val} : \text{input})$
22. $\text{writeOp} \leftarrow \text{writeOp} + 1$
23. $\text{mAck} \leftarrow \emptyset$
24. broadcast($(\text{discover, writeOp, w})$) to $S$
25. wait until $(|\text{mAck}| = |S|/2 + 1)$
26. $\text{maxTS} \leftarrow \max \{m.tg.ts' | m \in \text{mAck}\}$
27. $(\text{tag, v}) \leftarrow \{(\text{maxTS} + 1, w), \text{val}\}$
28. $\text{writeOp} \leftarrow \text{writeOp} + 1$
29. $\text{mAck} \leftarrow \emptyset$
30. broadcast($(\text{writeRequest, (tg, v), writeOp, w})$) to $S$
31. wait until $(|\text{mAck}| = |S|/2 + 1)$
32. return

Upon receive $m$ from $s$
33. if $m.\text{writeOp} = \text{writeOp}$ then
34. $\text{mAck} \leftarrow \text{mAck} \cup \{(s, m)\}$

4.3.2 Correctness of OHAMAM

To prove correctness of algorithm OHAMAM we reason about its liveness (termination) and atomicity (safety).

Liveness. Similarly to OHSAM, termination holds with respect to our failure model: up to $f$ servers may fail by crashing, where $f < |S|/2$ and each operation waits for messages from some majority of servers. We now give additional details.
Algorithm 6 Server Protocol for MWMR algorithm OHMAM

At each server $s_i$

Variables:

- $t_g \in (\mathbb{N}^+, T)$: init $(0, s_i)$, $v \in V$: init $\perp$,
- $write_{ops}[1...|W|+1]$: array of int
- $operations[1...|R|+1]$: array of int
- $relays[1...|R|+1]$: array of int

Initialization:

- $write_{ops}[i] \leftarrow 0$ for $i \in W$
- $operations[i] \leftarrow 0$ for $i \in R$
- $relays[i] \leftarrow 0$ for $i \in R$

Upon receive $(\langle readRequest, r, read_{op} \rangle)$

- broadcast $(\langle readRelay, (t_g, v), r, read_{op}, s_i \rangle)$ to $S$

Upon receive $(\langle readRelay, (t_g', v'), r, read_{op}, s_i \rangle)$

- if $(t_g < t_g')$ then
  - $(t_g, v) \leftarrow (t_g', v')$
- if $(operations[r] < read_{op})$ then
  - $operations[r] \leftarrow read_{op}$
  - $relays[r] \leftarrow 0$
- if $(operations[r] = read_{op})$ then
  - $relays[r] \leftarrow relays[r] + 1$

- if $(relays[r] = |S|/2 + 1)$ then
  - Send $(\langle readAck, (t_g, v), read_{op}, s_i \rangle)$ to $r$

Upon receive $(\langle discover, write_{op}, w \rangle)$

- Send $(\langle discoverAck, (t_g, v), write_{op}, s_i \rangle)$ to $w$

Upon receive $(\langle writeRequest, t_g', v', write_{op}, w \rangle)$

- if $(t_g < t_g') \land (write_{op}[w] < write_{op})$ then
  - $(t_g, v) \leftarrow (t_g', v')$
  - $write_{ops}[w] \leftarrow write_{op}$
- send $(\langle writeAck, (t_g, v), write_{op}, s_i \rangle)$ to $w$

Write Operation. Writer $w$ finds the maximum tag by broadcasting a discover message to all servers forming exchange $E_1$, and waiting to collect discoverAck replies from a majority of servers during exchange $E_2$ (A5:L22-24 and A6:L56-57). Since we tolerate $f < \frac{|S|}{2}$ crashes, then at least a majority of live servers will collect the discover messages from $E_1$ and reply to writer $w$ in $E_2$. Once the maximum timestamp is determined, then writer $w$ updates its local tag and broadcasts a writeRequest message to all the servers forming $E_3$. Writer $w$ then waits to collect writeAck replies from a majority of servers before completion. Again, at least a majority of servers collects the writeRequest message during $E_3$, and acknowledges to the requesting writer $w$ in $E_4$. 

65
Read Operation. A read operation of algorithm OHMAM differs from OHSAM only by using tags instead of timestamps (to impose an ordering on the values written). The structure of the read protocol is identical to OHSAM, thus liveness is ensured as reasoned in section 4.2.2.

Based on the above, any read or write operation collects a sufficient number of messages to terminate, guaranteeing liveness.

Atomicity. In the MWMR setting we use tags instead of timestamps, and here we show how algorithm OHMAM satisfies atomicity using tags. More precisely, for each execution $\xi$ of the algorithm there must exist a partial order $\prec$ on the operations in on the set of completed operations $\Pi$ that satisfy conditions A1, A2, and A3 as given in Definition 2.2.5 in Section 2.2. Let $tg_\pi$ be the value of the tag at the completion of $\pi$ when $\pi$ is a write, and the tag computed as the maximum $tg$ at the completion of a read operation $\pi$. With this, we denote the partial order using tags on operations as follows. For two operations $\pi_1$ and $\pi_2$, when $\pi_1$ is any operation and $\pi_2$ is a write, we let $\pi_1 \prec \pi_2$ if $tg_{\pi_1} < tg_{\pi_2}$. For two operations $\pi_1$ and $\pi_2$, when $\pi_1$ is a write and $\pi_2$ is a read we let $\pi_1 \prec \pi_2$ if $tg_{\pi_1} \leq tg_{\pi_2}$. The rest of the order is established by transitivity and reads with the same timestamps are not ordered. We now state and prove a series of lemmas leading to the main correctness result.

It is easy to see that the $tg$ variable in each server $s$ is monotonically increasing. This leads to the following lemma.

**Lemma 4.3.1** In any execution $\xi$ of OHMAM, the variable $tg$ maintained by any server $s$ in the system is non-negative and monotonically increasing.
Proof. When server $s$ receives a tag $tg$ then $s$ updates its local tag $tg_s$ iff $tg > tg_s$ (A6:L47-48 and A6:L59-61). Thus the local tag of the server monotonically increases and the lemma follows. \hfill \Box

Next, we show that if a read operation $\rho_2$ succeeds read operation $\rho_1$, then $\rho_2$ always returns a value at least as recent as the one returned by $\rho_1$.

Lemma 4.3.2 In any execution $\xi$ of OHMAM, If $\rho_1$ and $\rho_2$ are two read operations such that $\rho_1$ precedes $\rho_2$, i.e., $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns a tag $tg_1$, then $\rho_2$ returns a tag $tg_2 \geq tg_1$.

Proof. Let the operations $\rho_1$ and $\rho_2$ be invoked by processes $r_1$ and $r_2$ respectively (not necessarily different). Let $RSet_1$ and $RSet_2$ be the sets of servers that reply to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$ respectively.

Suppose, for purposes of contradiction, that read operations $\rho_1$ and $\rho_2$ exist such that $\rho_2$ succeeds $\rho_1$, i.e., $\rho_1 \rightarrow \rho_2$, and the operation $\rho_2$ returns a tag $tg_2$ which is smaller than the $tg_1$ returned by $\rho_1$, i.e., $tg_2 < tg_1$.

According to our algorithm, $\rho_2$ returns a tag $tg_2$ which is smaller than the minimum tag received by $\rho_1$, i.e., $tg_1$, if $\rho_2$ discovers tag $tg_2$ and value $v$ in the readAck message of some server $s_x \in RSet_2$, and $tg_2$ is the minimum tag received by $\rho_2$.

Assume that server $s_x$ replies with tag $tg'$ and value $v'$ to read operation $\rho_1$, i.e., $s_x \in RSet_1$. By monotonicity of the timestamp at the servers (Lemma 4.3.1), and since $\rho_1 \rightarrow \rho_2$, then it must be the case that $tg' \leq tg_2$. According to our assumption $tg_1 > tg_2$, and since $tg_1$ is the smallest tag sent to $\rho_1$ by any server in $RSet_1$, then it follows that $r_1$ does not receive the readAck message from $s_x$, and hence $s_x \notin RSet_1$. 

67
Now examine the actions of server $s_x$. From the algorithm, server $s_x$ collects readRelay messages from a majority of servers in $S$ before sending readAck message to $\rho_2$ (A6:L54-55). Let $RRSet_{s_x}$ denote the set of servers that send readRelay to $s_x$. Since, both $RRSet_{s_x}$ and $RSet_1$ contain some majority of the servers then it follows that $RRSet_{s_x} \cap RSet_1 \neq \emptyset$.

This above means that there exists a server $s_i \in RRSet_{s_x} \cap RSet_1$ that sends (i) a readAck message to $r_1$ for $\rho_1$, and (ii) a readRelay message to $s_x$ during $\rho_2$. Note that $s_i$ sends a readRelay message for $\rho_2$ only after it receives a read request from $\rho_2$ (A6:L44-45). Since $\rho_1 \rightarrow \rho_2$, then it follows that $s_i$ sends the readAck message to $\rho_1$ before sending the readRelay message to $s_x$. Thus, by Lemma 4.3.1, if $s_i$ attaches a tag $tg_{s_i}$ in the readAck to $\rho_1$, then $s_i$ attaches a tag $tg'_{s_i}$ in the readRelay message to $s_x$, such that $tg'_{s_i} \geq tg_{s_i}$. Since $tg_1$ is the minimum tag received by $\rho_1$, then $tg_{s_i} \geq tg_1$, then $tg'_{s_i} \geq tg_1$ as well. By Lemma 4.3.1, and since $s_x$ receives the readRelay message from $s_i$ before sending a readAck to $\rho_2$, it follows that $s_x$ sends a tag $tg_2 \geq tg'_{s_i}$. Therefore, $tg_2 \geq tg_1$ and this contradicts our initial assumption and completes our proof. \hfill \square

Next, we reason that if a write operation $\omega_2$ succeeds write operation $\omega_1$, then $\omega_2$ writes a value associated with a tag strictly higher than $\omega_1$.

**Lemma 4.3.3** In any execution $\xi$ of OHHAM, if a write operation $\omega_1$ writes a value with tag $tg_1$ then for any succeeding write operation $\omega_2$ that writes a value with tag $tg_2$ we have $tg_2 > tg_1$.

**Proof.** Let $WSet_1$ be the set of servers that send a writeAck message within write operation $\omega_1$. Let $Disc_2$ be the set of servers that send a discoverAck message within write operation $\omega_2$. 68
Based on the assumption, write operation $\omega_1$ is complete. By Lemma 4.3.1, we know that if a server $s$ receives a tag $tg$ from a process $p$, then $s$ includes tag $tg'$ s.t. $tg' \geq tg$ in any subsequently message. Thus the servers in $WSet_1$ send a writeAck message within $\omega_1$ with tag at least tag $tg_1$. Hence, every server $s_x \in WSet$ obtains tag $tg_{sx} \geq tg_1$.

When write operation $\omega_2$ is invoked, it obtains the maximum tag, $max_tag$, from the tags stored in at least a majority of servers. This is achieved by sending discover messages to all servers and collecting discoverAck replies from a majority of servers forming set $Disc_2$ (A5:L22-24 and A6:L56-57).

Sets $WSet_1$ and $Disc_2$ contain a majority of servers, and so $WSet_1 \cap Disc_2 \neq \emptyset$. Thus, by Lemma 4.3.1, any server $s_k \in WSet \cap Disc_2$ has a tag $tg_{sk}$ s.t. $tg_{sk} \geq tg_{sx} \geq tg_1$. Furthermore, the invoker of $\omega_2$ discovers a $max_tag$ s.t. $max_tag \geq tg_{sk} \geq tg_{sx} \geq tg_1$. The invoker updates its local tag by increasing the maximum tag it discovered, i.e. $tg_2 = (max_tag + 1, v)$ (A5:L25), and associating $tg_2$ with the value to be written. We know that, $tg_2 > max_tag \geq tg_1$, hence $local_tag > tg_1$.

Now the invoker of $\omega_2$ includes its tag $local_tag$ with writeRequest message to all servers, and terminates upon receiving writeAck messages from a majority of servers. By Lemma 4.3.1, $\omega_2$ receives writeAck messages with a tag $tg_2$ s.t. $tg_2 \geq local_tag \geq tg_1$ hence $tg_2 > tg_1$, and the lemma follows. \hfill \Box

At this point we have to show that any read operation which succeeds a write operation, will receive readAck messages from the servers where each included timestamp will be greater or equal to the one that the complete write operation returned.
**Lemma 4.3.4** In any execution $\xi$ of OHMAM, if a read operation $\rho$ succeeds a write operation $\omega$ that writes value $v$ with tag $tg$, i.e., $\omega \rightarrow \rho$, and $\rho$ receives readAck messages from a majority of servers $RSet$, then each $s \in RSet$ sends a readAck message to $\rho$ with a tag $tg_s$ s.t. $tg_s \geq tg$.

**Proof.** Let $WSet$ be the set of servers that send a writeAck message to the write operation $\omega$ and let $RRSet$ be the set of servers that sent readRelay messages to server $s$.

It is given that write operation $\omega$ is complete. By Lemma 4.3.1, we know that if server $s$ receives a tag $tg$ from process $p$, then $s$ includes a tag $tg'$ s.t. $tg' \geq tg$ in any subsequent message. Thus a majority of servers, forming $WSet$, send a writeAck message in $\omega$ with tag greater or equal to tag $tg$. Hence, every server $s_x \in WSet$ has a tag $tg_{s_x} \geq tg$. Let us now examine tag $tg_{s}$ that server $s$ sends to read operation $\rho$.

Before server $s$ sends a readAck message to $\rho$, it must receive readRelay messages for the majority of servers, $RRSet$ (A6:L47-48). Since both $WSet$ and $RRSet$ contain a majority of servers, then it follows that $WSet \cap RRSet \neq \emptyset$. Thus, by Lemma 4.3.1, any server $s_x \in WSet \cap RRSet$ has a tag $tg_{s_x}$ s.t. $tg_{s_x} \geq tg$.

Since server $s_x \in RRSet$ and by the algorithm, server’s $s$ tag is always updated to the highest tag it observes (A6:L47-48), then when server $s$ receives the message from $s_x$, it updates its tag $tg_s$ s.t. $tg_s \geq tg_{s_x}$. Furthermore, server $s$ creates a readAck message where it includes its local tag $tg_s$ and its local value $v_s$, and sends this readAck message within the read operation $\rho$ (A6:L54-55). Each $s \in RSet$ sends a readAck to $\rho$ with a tag $tg_s$ s.t. $tg_s \geq tg_{s_x} \geq tg$. Therefore, $tg_s \geq tg$ and the lemma follows. \[\square\]

Next we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.
Lemma 4.3.5 In any execution $\xi$ of OHMAM, if read operation $\rho$ succeeds write operation $\omega$, i.e., $\omega \rightarrow \rho$, that writes value $v$ associated with tag $tg$, and $\rho$ returns tag $tg'$, then $tg' \geq tg$.

Proof. Suppose that read operation $\rho$ receives readAck messages from a majority of servers $RSet$ and decides on a tag $tg'$ associated with value $v$ and terminates.

In this case, by Algorithm (A5:L9-11) it follows that read $\rho$ decides on a tag $tg'$ that belongs to a readAck message among the messages from servers in $RSet$; and it is the minimum tag among all the tags that are included in messages of servers $RSet$, hence $tg' = \min_{tag}$.

Furthermore, since $tg' = \min_{tag}$ holds and from Lemma 4.3.4, $\min_{tag} \geq tg$ holds, where $tg$ is the tag returned from the last complete write operation $\omega$, then $tg' = \min_{tag} \geq tg$ also holds. Therefore, $tg' \geq tg$ holds and the lemma follows. \qed

Next we show that if a write operation succeeds a read operation, then it writes a value associated with a tag greater than the one returned by the read operation.

Lemma 4.3.6 In any execution $\xi$ of OHMAM, if a write $\omega$ succeeds a read operation $\rho$ that reads tag $tg$, i.e. $\rho \rightarrow \omega$, and $\omega$ writes with a tag $tg'$, then $tg' > tg$.

Proof. Let $RR$ be the set of servers that sent readRelay messages to $\rho$. Let $dAck$ be the set of servers that sent discoverAck messages to $\omega$. Let $wAck$ be the set of servers that sent writeAck messages to $\omega$ and let $RA$ be the set of servers that sent readAck messages to $\rho$. It is not necessary that $a \neq b \neq c$ holds.

Based on the read protocol, the read operation $\rho$ terminates when it receives readAck messages from a majority of servers. It follows that $\rho$ decides on the minimum tag, $tg = \min_{TG}$, among all the tags in the readAck messages of the set $RA$ and terminates. Writer $\omega$, initially it broadcasts a discover message to all servers, and it then awaits for “fresh” discoverAck
messages from a majority of servers, that is, set $dAck$. Each of $RA$ and $dAck$ sets are from majorities of servers, and so $RA \cap dAck \neq \emptyset$. By Lemma 4.3.1, any server $s_k \in RA \cap dAck$ has a tag $tg_{s_k}$ s.t. $tg_{s_k} \geq tg$. Since $\omega$ generates a new local tag-value $(tg', v)$ pair in which it assigns the timestamp to be one higher than the one discovered in the maximum tag from set $dAck$, it follows that $tg' > tg$. Furthermore, $\omega$ broadcasts the value to be written associated with $tg'$ in a writeRequest message to all servers and it awaits for writeAck messages from a majority of servers before completion, set $wAck$. Observe that each of $dAck$ and $wAck$ sets are from majority of servers, and so $dAck \cap wAck \neq \emptyset$. By Lemma 4.3.1, any server $s_k \in dAck \cap wAck$ has a tag $tg_{s_k}$ s.t. $tg_{s_k} \geq tg'$ and the result follows. □

We now show the correctness of algorithm OHMAM.

**Theorem 4.3.7** Algorithm OHMAM implements an atomic MWMR object.

**Proof.** We use the above lemmas and the operations order definition to reason about each of the three atomicity conditions A1, A2 and A3.

**A1** For any $\pi_1, \pi_2 \in \Pi$ such that $\pi_1 \rightarrow \pi_2$, it cannot be that $\pi_2 \prec \pi_1$.

If both $\pi_1$ and $\pi_2$ are writes and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 4.3.3 it follows that $tg_{\pi_2} > tg_{\pi_1}$. By the ordering definition $\pi_1 \prec \pi_2$ is satisfied. When $\pi_1$ is a write, $\pi_2$ a read and $\pi_1 \rightarrow \pi_2$ holds, then from Lemmas 4.3.4 and 4.3.5 it follows that $tg_{\pi_2} \geq tg_{\pi_1}$. By definition $\pi_1 \prec \pi_2$ is satisfied. If $\pi_1$ is a read, $\pi_2$ a write and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 4.3.6 it follows that $\pi_2$ always returns a tag $tg_{\pi_2}$ s.t. $tg_{\pi_2} > tg_{\pi_1}$. By the ordering definition $\pi_1 \prec \pi_2$ is satisfied. If both $\pi_1$ and $\pi_2$ are reads and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 4.3.2 it follows that the tag returned from $\pi_2$ is always greater or equal to the one returned from $\pi_1$. $tg_{\pi_2} \geq tg_{\pi_1}$.
If \( tg_{\pi_2} > tg_{\pi_1} \), then by the ordering definition \( \pi_1 \prec \pi_2 \) holds. When \( tg_{\pi_2} = tg_{\pi_1} \) then the ordering is not defined but it cannot be that \( \pi_2 \prec \pi_1 \).

**A2** For any write \( \omega \in \Pi \) and any operation \( \pi \in \Pi \), then either \( \omega \prec \pi \) or \( \pi \prec \omega \).

If \( tg_{\omega} > tg_{\pi} \), then \( \pi \prec \omega \) follows directly. Similarly, if \( tg_{\omega} < tg_{\pi} \) holds, then it follows that \( \omega \prec \pi \). When \( ts_{\omega} = ts_{\pi} \) holds, then the uniqueness of each tag that a writer creates ensures that \( \pi \) is a read. In particular, remember that each tag is a \( \langle ts, id \rangle \) pair, where \( ts \) is a natural number and \( id \) a writer identifier. Tags are ordered lexicographically, first with respect to the timestamp and then with respect to the writer id. Since the writer ids are unique, then even if two writes use the same timestamp \( ts \) in the tag pairs they generate, the two tags cannot be equal as they will differ on the writer id. Furthermore, if the two tags are generated by the same writer, then by well-formedness it must be the case that the latter write will contain a timestamp larger than any timestamp used by that writer before. Since \( \pi \) is a read operation that receives readAck messages from a majority of servers, then the intersection properties of majorities ensure that \( \omega \prec \pi \).

**A3** Every read operation returns the value of the last write preceding it according to \( \prec \) (or the initial value if there is no such write).

Let \( \omega \) be the last write preceding read \( \rho \). From our definition it follows that \( tg_{\rho} \geq tg_{\omega} \). If \( tg_{\rho} = tg_{\omega} \), then \( \rho \) returned a value written by \( \omega \) in some servers. Read \( \rho \) waited for readAck messages from a majority of servers and the intersection properties of majorities ensure that \( \omega \) was the last complete write operation. If \( tg_{\rho} > tg_{\omega} \) holds, it must be the case that there is a write \( \omega' \) that wrote \( tg_{\rho} \) and by definition it must hold that \( \omega \prec \omega' \prec \rho \). Thus, \( \omega \) is not the
preceeding write and this cannot be the case. Lastly, if \( tg_p = 0 \), no preceding writes exist, and \( \rho \) returns the initial value.

4.3.3 Performance of OHMAM

We assess the performance of OHMAM in terms of (i) latency of read and write operations as measured by the number of exchanges, (ii) the message complexity of reads and writes and (iii) computational complexity. Briefly, for algorithm OHMAM write operations take 4 communication exchanges and read operations take 3 exchanges. The (worst case) message complexity of read operations is \( |S|^2 + 2|S| \) and the (worst case) message complexity of write operations is \( 4|S| \). This follows directly from the structure of the algorithm. We now give additional details.

**Operation Latency.** Similarly as in Section 4.2.3, we study the operation latency, in terms of the number of communication exchanges required by each operation.

*Write operation latency:* Per algorithm OHMAM, writer \( w \) broadcasts a discover message to all the servers during exchange \( E_1 \), and awaits for discoverAck messages from a majority of servers during \( E_2 \). Once the discoverAck messages are received, then \( w \) broadcasts a writeRequest message to all servers in exchange \( E_3 \). Lastly, it waits for writeAck messages from a majority of servers in \( E_4 \). Thus any write consists of 4 exchanges.

*Read operation latency:* The structure of the read protocol of OHMAM is identical to OHSAM, thus a read operation consists of 3 exchanges as reasoned in Section 4.2.3.

**Message Complexity.** Similarly as in Section 4.2.3, we measure operation message complexity as the worst case number of exchanged messages in each read and write operation.
Write operation: The first and the third exchanges, E₁ and E₃, occur when a writer sends discover and writeRequest messages respectively to all servers \( s \in S \). The second and fourth exchanges, E₂ and E₄, occur when servers send discoverAck and writeAck messages back to the writer. Thus, in a write operation there are \( 4|S| \) messages exchanged.

Read operation: The structure of the read protocol of OHMAM is identical to OHSAM thus, as reasoned in Section 4.2.3, during a read operation, \( |S|^2 + 2|S| \) messages are exchanged.

Computational Complexity. Algorithm OHMAM performs a modest amount of local computation, resulting in negligible computation overhead.

4.4 Reducing the Latency of Read Operations

In this section we revise the protocol implementing read operations of algorithms OHSAM and OHMAM to reduce the latency of read operations, in terms of communication exchanges. In particular, we study the cases under which operations can terminate fast without violating atomicity and we devise protocols for both the SWMR and the MWMR setting. By doing so, in both settings, the revised protocols implement read operations that take either two or three communication exchanges before completion.

4.4.1 Obtaining Algorithms OHSAM′ and OHMAM′

The objective of the revised protocols is to capture the conditions under which operations can terminate fast without violating the property of atomicity. In particular, the key idea here is to let the reader \( r \) determine “quickly” that a majority of replica servers hold the same timestamp \( ts \) (or tag \( tg \) for the MWMR setting) and its associated value \( v \), and let reader \( r \) be fast and return \( v \). This indicates a potentially complete write operation associated with
timestamp \( ts \). Notice that, due to asynchrony, it is possible that the write operation associated with \( ts \) did not receive acknowledgment messages from a majority of servers, thus it may still be on-going, i.e., not completed. However, by the non-empty intersection property of majorities, this guarantees that any subsequent read operation, fast or not, before termination it will obtain from a majority of replica servers a value \( v' \) associated with a timestamp \( ts' \) (or tag \( tg' \)) at least as recent as the one associated with \( v \). Thus, atomicity is not violated.

This is achieved by having the servers send relay messages to each other as well as to the readers. While a reader collects the relays and the read acknowledgment messages, if it observes in the set of the received relay messages that a majority of servers holds the same timestamp (or tag), then it safely returns the associated value and the read operation terminates in two communication exchanges. If that was not the case, then the reader proceeds similarly to algorithm OHSAM and terminates in three communication exchanges. We refer to the revised algorithms as OHSAM' and OHMAM'.

The code for OHSAM' that presents the revised read protocol is given in Algorithm 7. In addition, for the servers protocol we present only the changes from algorithm OHSAM.

**Revised Server Protocol.** The idea is that the server sends a readRelay message to all servers and to the invoker of the read operation. This is captured in lines A7:L144-145.

**Revised Reader Protocol.** Here, we let the reader to await for either (a) \( |S|/2 + 1 \) readAck messages or (b) \( |S|/2 + 1 \) readRelay messages that include the same timestamp \( ts \). In either case we check the enclosed counter variables to ensure “freshness” as in OHSAM. For case (b) the reader returns the value \( v \) associated with the common timestamp \( ts \) found in readRelay messages from a majority of servers. Otherwise, when case (a) holds and the read protocol proceeds as in OHSAM.
Algorithm 7 Read and Server Protocol Changes for SWMR algorithm OHSAM'

At each reader $r$

Variables:
- $ts \in \mathbb{N}^+$, $minTS \in \mathbb{N}^+$, $v \in V$
- $read_op \in \mathbb{N}^+$, $rRelay, rAck \subseteq S \times M$

Initialization:
- $ts \leftarrow 0$, $minTS \leftarrow 0$, $v \leftarrow \bot$, $read_op \leftarrow 0$

function READ()
- $read_op \leftarrow read_op + 1$, $rAck, rRelay \leftarrow (\emptyset, \emptyset)$
- broadcast $(readRequest, r, read_op)$ to $S$

wait until $(|rAck| = |S|/2 + 1)$ OR
- $(\exists Z \subseteq rRelay : (|Z| \geq |S|/2 + 1) \land (\forall (m', s'), (m'', s'') \in Z : m'.ts = m''.ts))$

- if ($rAck = |S|/2 + 1$) then
  - $minTS \leftarrow \min\{m.ts | m \in rAck\}$
  - $v \leftarrow \{m.val | m \in rAck \land m.ts = minTS\}$
  - return($v$)
- else
  - $v \leftarrow \{m.val | m \in rRelay\}$
  - return($v$)

Upon receive $m$ from $s$
- if ($m.read_op = read_op$) then
    - if ($m.type = readAck$) then
      - $rAck \leftarrow rAck \cup \{(s, m)\}$
    - else
      - $rRelay \leftarrow rRelay \cup \{(s, m)\}$

At each server $s_i$

Upon receive $(readRequest, r, read_op)$
- broadcast $(readRelay, ts, v, r, read_op, s_i)$ to $S \cup \{r\}$

Algorithm OHMAM' is obtained similarly to OHSAM' by (i) using tags instead of timestamps in the revised read protocol of OHSAM' and (ii) using the write protocol of OHMAM without any modifications.

4.4.2 Correctness of OHSAM'

We first prove liveness (termination) and then atomicity (safety).

Liveness. Termination of Algorithm OHSAM' is guaranteed with respect to our failure model: up to $f$ servers may fail by crashing, where $f < |S|/2$, and any type of operation awaits for messages from a majority of servers before completion. We now provide additional details.
Read Operation. A read operation of algorithm OHSAM' terminates when the client either (i) collects readRelay messages from at least a majority of serves and all of them include the same timestamp; or (ii) collects readAck messages from a majority of servers. When case (i) occurs then the operation terminates immediately (and faster). Otherwise, case (ii) holds, the read operation proceeds identically to algorithm OHSAM and its termination is ensured as reasoned in Section 4.2.2.

Write Operation. A write operation of algorithm OHSAM' is identical to that of algorithm OHSAM, thus liveness is guaranteed as reasoned in Section 4.2.2.

Atomicity. Due to the similarity of the writer and server protocols of algorithm OHSAM' to those in OHSAM, we state the lemmas and we omit some of the proofs. Lemma 4.4.1 shows that the timestamp variable $ts$ maintained by each server $s$ in the system is monotonically non-decreasing.

**Lemma 4.4.1** In any execution $\xi$ of OHSAM', the variable $ts$ maintained by any server $s$ in the system is non-negative and monotonically increasing.

**Proof.** Lemma 4.2.1 for algorithm OHSAM also applies to algorithm OHSAM' since the modification in line 145 does not affect the update of the timestamp $ts$ at the server protocol.

Next, we show that if a read operation $\rho_2$ succeeds read operation $\rho_1$, then $\rho_2$ always returns a value at least as recent as the one returned by $\rho_1$.

**Lemma 4.4.2** In any execution $\xi$ of OHSAM', if $\rho_1$ and $\rho_2$ are two read operations such that $\rho_1$ precedes $\rho_2$, i.e., $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns the value for timestamp $ts_1$, then $\rho_2$ returns the value for timestamp $ts_2 \geq ts_1$. 

78
**Proof.** Since read operations in algorithm OHSAM terminate in 3 communication exchanges then from Lemma 4.2.2 we know that any two non-concurrent 3-exchange read operations satisfy this. Thus we have to show that the lemma holds for the cases where (i) a 2-exchange read operation $\rho_1$ precedes a 2-exchange read operation $\rho_2$; (ii) a 2-exchange read operation $\rho_1$ precedes a 3-exchange read operation $\rho_2$; and (iii) a 3-exchange read operation $\rho_1$ precedes a 2-exchange read operation $\rho_2$. Let the two operations $\rho_1$ and $\rho_2$ be invoked by processes with identifiers $r_1$ and $r_2$ respectively (not necessarily different).

**Case (i).** Let $RRSet_1$ and $RRSet_2$ be the sets of servers that send a readRelay message to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$. Assume by contradiction that 2-exchange read operations $\rho_1$ and $\rho_2$ exist such that $\rho_2$ succeeds $\rho_1$, i.e., $\rho_1 \rightarrow \rho_2$, and the operation $\rho_2$ returns a timestamp $t_s_2$ that is smaller than the $t_s_1$ returned by $\rho_1$, i.e., $t_s_2 < t_s_1$. According to our algorithm, $\rho_2$ returns a timestamp $t_s_2$ that is smaller than the timestamp that $\rho_1$ returned i.e., $t_s_1$, if $\rho_2$ received $|S|/2 + 1$ readRelay messages that all included the same timestamp $t_s_2$ and $t_s_2$ is smaller than $t_s_1$, which is included in $|S|/2 + 1$ readRelay messages received by $\rho_1$. Since, both $RRSet_1$ and $RRSet_2$ contain some majority of servers then it follows that $RRSet_1 \cap RRSet_2 \neq \emptyset$. And since by Lemma 4.4.1 the timestamp variable $t_s$ maintained by servers is monotonically increasing, then it is impossible that $\rho_2$ received $|S|/2 + 1$ readRelay messages that all included the same timestamp $t_s_2$ and $t_s_2 < t_s_1$. In particular, since at least a majority of servers have a timestamp at least as $t_s_1$ then $\rho_2$ can receive only $|S|/2$ readRelay messages with a timestamp $t_s_2$ s.t. $t_s_2 < t_s_1$. Therefore, this contradicts our assumption.

**Case (ii).** Since $\rho_1$ is a 2-exchange operation, then $r_1$ receives at least $|S|/2 + 1$ readRelay messages that includes the same timestamp $t_s_1$. Thus after the completion of $\rho_1$ at least a majority of servers hold a timestamp at least as $t_s_1$. In addition, we know that the servers relay...
to each other and wait for readRelay messages from a majority of servers before they send a readAck message to the reader \( r_2 \). By Lemma 4.4.1 the timestamp variable \( ts \) maintained by servers is monotonically increasing then each server \( s_i \) that sends a readAck message to \( r_2 \) must include a timestamp \( ts_{s_i} \) s.t. \( ts_{s_i} \geq ts_1 \). Therefore, the minimum timestamp \( ts_2 \) that \( r_2 \) can observe in each readAck message received from \( s_i \) must be \( ts_2 \geq ts_{s_i} \geq ts_1 \). Since a 3-exchange read operation decides on the minimum timestamp observed in the readAck responses, then reader \( r_2 \) will decide on a timestamp \( ts_2 \) s.t. \( ts_2 \geq ts_1 \).

Case (iii). Since \( \rho_1 \) is a 3-exchange operation, then \( r_1 \) receives at least \( |S|/2 + 1 \) readAck messages that include the minimum timestamp \( ts_1 \). Servers relay to each other before they send a readAck message to \( \rho_1 \) and timestamps in servers are monotone (Lemma 4.4.1), thus after the completion of \( \rho_1 \) at least a majority of servers, \( s_i \in RSet \), hold a timestamp no smaller than \( ts_1 \). Let \( RRSet \) be the set of servers that send a readRelay message to \( r_2 \) during \( \rho_2 \). In order for \( \rho_2 \) to terminate, based on the read protocol the size of \( RRSet \) must be at least \( |S|/2 + 1 \). Let \( ts_{s_i} \) be a timestamp received from a server \( s_i \in RRSet \). Since \( RSet \cap RRSet \neq \emptyset \) then the 2-exchange operation \( \rho_2 \) that succeeds \( \rho_1 \) can receive at most \( |S|/2 \) (minority) readRelay messages with a timestamp \( ts_{s_i} \) s.t. \( ts_{s_i} < ts_1 \). Thus, when \( \rho_2 \) terminates it must return a timestamp \( ts_2 \) s.t. \( ts_2 \geq ts_1 \) and the lemma follows. \( \square \)

Now we show that if a read operation succeeds a write operation, then the read returns a value at least as recent as the one written.

**Lemma 4.4.3** In any execution \( \xi \) of the algorithm, if a read \( \rho \) succeeds a write operation \( \omega \) that writes timestamp \( ts \), i.e. \( \omega \rightarrow \rho \), and \( \rho \) returns a timestamp \( ts' \), then \( ts' \geq ts \).
**Proof.** From Lemma 4.2.4 for algorithm OHSAM we know that lemma holds if a 3-exchange read operation succeeds a write operation. We now show that the same holds in case where the read operation terminates in 2 exchanges.

Assume by contradiction that a 2-exchange read operation $\rho$ and a write operation $\omega$ exist such that $\rho$ succeeds $\omega$, i.e. $\omega \rightarrow \rho$, and $\rho$ returns a timestamp $ts'$ that is smaller than $ts$ that $\omega$ wrote, $ts' < ts$. From our algorithm, in order for this to happen, $\rho$ receives $|S|/2 + 1$ readRelay messages that all include the same timestamp $ts'$ and $ts' < ts$. Since $\omega$ is complete it means that at least a majority of servers hold a timestamp $ts_s$ s.t. $ts_s \geq ts$. Since any two majorities have a non empty intersection, this contradicts the assumption that $\rho$ received $|S|/2 + 1$ readRelay messages that all included the same timestamp $ts'$ where $ts' < ts$ and the lemma follows.

We now show the correctness of algorithm OHSAM'.

**Theorem 4.4.4** Algorithm OHSAM' implements an atomic SWMR object.

**Proof.** We use the lemmas stated above and the operations order definition to reason about each of the three atomicity conditions A1, A2 and A3.

**A1** For any $\pi_1, \pi_2 \in \Pi$ such that $\pi_1 \rightarrow \pi_2$, it cannot be that $\pi_2 \prec \pi_1$.

When the two operations $\pi_1$ and $\pi_2$ are reads and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 4.4.2 it follows that the timestamp returned from $\pi_2$ is always greater or equal to the one returned from $\pi_1$, $ts_{\pi_2} \geq ts_{\pi_1}$. If $ts_{\pi_2} > ts_{\pi_1}$ then by the ordering definition $\pi_1 \prec \pi_2$ is satisfied. When $ts_{\pi_2} = ts_{\pi_1}$ then the ordering is not defined, thus it cannot be the case that $\pi_2 \prec \pi_1$. If $\pi_2$ is a write, the sole writer generates a new timestamp by incrementing the largest timestamp in the system. By well-formedness, any timestamp generated by the writer for any write operation
that precedes $\pi_2$ must be smaller than $ts_{\pi_2}$. Since $\pi_1 \rightarrow \pi_2$, then it holds that $ts_{\pi_1} < ts_{\pi_2}$.

Hence, by the ordering definition it cannot be the case that $\pi_2 \prec \pi_1$. Lastly, when $\pi_2$ is a read and $\pi_1$ a write and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 4.4.3 it follows that $ts_{\pi_2} \geq ts_{\pi_1}$. By the ordering definition, it cannot hold that $\pi_2 \prec \pi_1$ in this case either.

**A2** For any write $\omega \in \Pi$ and any operation $\pi \in \Pi$, then either $\omega \prec \pi$ or $\pi \prec \omega$.

If the timestamp returned from $\omega$ is greater than the one returned from $\pi$, i.e. $ts_\omega > ts_\pi$, then $\pi \prec \omega$ follows directly. Similarly, if $ts_\omega < ts_\pi$ holds, then $\omega \prec \pi$ follows. If $ts_\omega = ts_\pi$, then it must be that $\pi$ is a read and $\pi$ either discovered $ts_\omega$ as the minimum timestamp in at least a majority of servers or returned fast $ts_\omega$ because it was noticed in at least a majority of servers.

Thus, $\omega \prec \pi$ follows.

**A3** Every read operation returns the value of the last write preceding it according to $\prec$ (or the initial value if there is no such write).

Let $\omega$ be the last write preceding read $\rho$. From our definition it follows that $ts_\rho \geq ts_\omega$. If $ts_\rho = ts_\omega$, then $\rho$ returned the value written by $\omega$ on a majority of servers. If $ts_\rho > ts_\omega$, then it means that $\rho$ obtained a larger timestamp. However, the larger timestamp can only be originating from a write that succeeds $\omega$, thus $\omega$ is not the preceding write and this cannot be the case. Lastly, if $ts_\rho = 0$, no preceding writes exist, and $\rho$ returns the initial value.

4.4.3 Correctness of OHMAM' 

We first prove liveness (termination) and then atomicity (safety).
**Liveness.** Termination of Algorithm OHMAM′ is guaranteed with respect to our failure model: up to \( f \) servers may fail, where \( f < \frac{|S|}{2} \), and operations wait for messages only from a majority of servers. We now give additional details.

*Write Operation.* Since the write protocol of algorithm OHMAM′ is identical to the one that algorithm OHMAM uses, *liveness* is guaranteed as discussed in Section 4.3.2.

*Read Operation.* A read operation of OHMAM′ differs from OHSAM′ by using tags instead of timestamps in order to impose an ordering on the values written. The structure of the read protocol is identical to OHSAM′, thus *liveness* is ensured as reasoned in Section 4.4.2.

**Atomicity.** Due to the similarity of the writer and server protocols of algorithm OHMAM′ to those in OHMAM, we state the lemmas and we omit some of the proofs. Lemma 4.4.5 shows that the tag \( tg \) maintained by each server \( s \) in the system is monotonically non-decreasing.

**Lemma 4.4.5** In any execution \( \xi \) of OHMAM′, the \( tg \) maintained by any server \( s \) in the system is non-negative and monotonically increasing.

**Proof.** Lemma 4.3.1 for algorithm OHMAM also applies to algorithm OHMAM′ because the modification in line 145 does not affect the update of the tag \( tg \) at the server protocol. \( \square \)

Next, we show that if a read operation \( \rho_2 \) succeeds read operation \( \rho_1 \), then \( \rho_2 \) always returns a value at least as recent as the one returned by \( \rho_1 \).

**Lemma 4.4.6** In any execution \( \xi \) of OHMAM′, If \( \rho_1 \) and \( \rho_2 \) are two read operations such that \( \rho_1 \) precedes \( \rho_2 \), i.e., \( \rho_1 \rightarrow \rho_2 \), and \( \rho_1 \) returns a tag \( tg_1 \), then \( \rho_2 \) returns a tag \( tg_2 \geq tg_1 \).

**Proof.** Since read operations in algorithm OHMAM terminate in 3 communication exchanges then from Lemma 4.3.2 we know that the any two non-concurrent 3-exchange satisfy this. Thus
we now have to show that the lemma holds for the cases where (i) a 2-exchange read operation
\( \rho_1 \) precedes a 2-exchange read operation \( \rho_2 \); (ii) a 2-exchange read operation \( \rho_1 \) precedes a
3-exchange read operation \( \rho_2 \); and (iii) a 3-exchange read operation \( \rho_1 \) precedes a 2-exchange
read operation \( \rho_2 \). Let the two operations \( \rho_1 \) and \( \rho_2 \) be invoked by processes with identifiers \( r_1 \)
and \( r_2 \) respectively (not necessarily different).

Case (i). Assume by contradiction that 2-exchange read operations \( \rho_1 \) and \( \rho_2 \) exist such that
\( \rho_2 \) succeeds \( \rho_1 \), i.e., \( \rho_1 \to \rho_2 \), and operation \( \rho_2 \) returns a tag \( tg_2 \) that is smaller than tag \( tg_1 \)
returned by \( \rho_1 \), i.e., \( tg_2 < tg_1 \). Since both operations \( \rho_1 \) and \( \rho_2 \) complete in 2 exchanges, they
both have to collect \( |S|/2 + 1 \) readRelay messages with the same tag \( tg_1 \) and \( tg_2 \) respectively.
We know that after the completion or \( \rho_1 \) at least \( |S|/2 + 1 \) servers have a tag at least as
\( tg_1 \). By monotonicity of the tag at the servers (Lemma 4.4.5) and the fact that \( \rho_1 \) is completed it follows
that it is impossible for \( \rho_2 \) to collect \( |S|/2 + 1 \) readRelay messages with the same tag \( tg_2 \) s.t.
\( tg_2 < tg_1 \). In particular, \( \rho_2 \) can receive only \( |S|/2 \) readRelay messages with a timestamp \( tg_2 \)
s.t. \( tg_2 < tg_1 \). Therefore, this contradicts our assumption.

Case (ii). We know that since \( \rho_1 \) is a 2-exchange operation then \( r_1 \) receives at least \( |S|/2 + 1 \)
readRelay messages that include the same tag \( tg_1 \). Thus after the completion of \( \rho_1 \) at least
a majority of servers hold a timestamp at least as \( tg_1 \). Servers relay to each other and wait
for readRelay messages from a majority of servers before they send a readAck message to the
reader \( r_2 \). By Lemma 4.4.5 since the tag variable \( tg_s \) maintained by servers is monotonically
increasing then each server \( s_i \) that sends a readAck message to \( r_2 \) must include a tag \( tg_s \), s.t.
\( tg_{s_i} \geq tg_1 \). Therefore, the minimum tag \( tg_2 \) that \( r_2 \) can observe in each readAck message
received from \( s_i \) must be \( tg_2 \geq tg_{s_i} \geq tg_1 \). Since a 3-exchange read operation decides on
the minimum tag observed in the readAck responses, reader \( r_2 \) decides on a timestamp \( t g_2 \) s.t. \( t g_2 \geq t g_1 \).

**Case (iii).** Since \( \rho_1 \) is a 3-exchange operation, \( r_1 \) receives at least \( |S|/2 + 1 \) readAck messages that include the minimum tag \( t g_1 \). Servers relay to each other before they send a readAck message to \( \rho_1 \), then, by the monotonicity of tags at servers (Lemma 4.4.5), after the completion of \( \rho_1 \) at least a majority of servers \( s_i \in RSet \) hold a tag at least as \( t g_1 \). Let \( RRSet \) be the set of servers that send a readRelay message to \( r_2 \) during \( \rho_2 \). In order for \( \rho_2 \) to terminate the size of \( RRSet \) must be at least \( |S|/2 + 1 \) based on the read protocol. Let \( t g_{s_i} \) be a tag received from a server \( s_i \in RRSet \). Since \( RSet_1 \cap RRSet \neq \emptyset \) then the 2-exchange operation \( \rho_2 \) that succeeds \( \rho_1 \) can receive at most \( |S|/2 \) (minority) readRelay messages with a tag \( t g_{s_i} \) s.t. \( t g_{s_i} < t g_1 \). Thus, when \( \rho_2 \) terminates it must return a tag \( t g_2 \) s.t. \( t g_2 \geq t g_1 \) and the lemma follows. \( \square \)

Now we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

**Lemma 4.4.7** In any execution \( \xi \) of \( OHMAM' \), if read operation \( \rho \) succeeds write operation \( \omega \) (i.e., \( \omega \rightarrow \rho \)) that writes value \( v \) associated with tag \( t g \) and \( \rho \) returns tag \( t g' \), then \( t g' \geq t g \).

**Proof.** From Lemma 4.3.5 for algorithm \( OHMAM \) we know that the lemma holds if a 3-exchange read operation succeeds a write operation. We now show that the same holds for 2-exchange read operations.

Assume by contradiction that a 2-exchange read operation \( \rho \) and a write operation \( \omega \) exist such that \( \rho \) succeeds \( \omega \), i.e. \( \omega \rightarrow \rho \), and \( \rho \) returns a tag \( t g' \) that is smaller than the tag \( t g \) that \( \omega \) wrote, \( t g' < t g \). From the algorithm, in order for this to happen, \( \rho \) receives \( |S|/2 + 1 \) readRelay

85
messages that all include the same tag $tg'$ and $tg' < tg$. Since $\omega$ is complete it means that at least a majority of servers hold tag $tg_s$ s.t. $tg_s \geq tg$. Since any two majorities intersect, this contradicts the assumption that $\rho$ receives $|S|/2 + 1$ readRelay messages that all include the same timestamp $tg'$ where $tg' < tg$ and the lemma follows. \hfill \Box

Next, we reason that if a write operation $\omega_2$ succeeds write operation $\omega_1$, then $\omega_2$ writes a value associated with a tag strictly higher than $\omega_1$.

**Lemma 4.4.8** In any execution $\xi$ of OHMAM', if a write operation $\omega_1$ writes a value with tag $tg_1$ then for any succeeding write operation $\omega_2$ that writes a value with tag $tg_2$ we have $tg_2 > tg_1$.

**Proof.** The modifications of OHMAM' do not have an impact on the write operations thus this lemma follows directly from lemma 4.3.3 of OHMAM. \hfill \Box

Next we show that if a write operation succeeds a read operation, then it writes a value associated with a tag greater than the one returned by the read operation.

**Lemma 4.4.9** In any execution $\xi$ of OHMAM', if a write $\omega$ succeeds a read operation $\rho$ that reads tag $tg$, i.e. $\rho \rightarrow \omega$, and $\rho$ returns a tag $tg'$, then $tg' > tg$.

**Proof.** The case where the read operation takes three communication exchanges to terminate is identical as in lemma 4.3.6 of algorithm OHMAM. Thus, we are interested to examine the case where the read terminates in two communication exchanges.

Let $RR$ be the set of servers that sent readRelay messages to $\rho$. Let $dAck$ be the set of servers that sent discoverAck messages to $\omega$. Let $wAck$ be the set of servers that sent writeAck
messages to \( \omega \) and let \( RA \) be the set of servers that sent readAck messages to \( \rho \). It is not necessary that \( a \neq b \neq c \) holds.

In the case we examine, the read operation \( \rho \) terminates when it receives readRelay messages from a majority of servers and \( \rho \) decides on a tag that all servers attached in the set \( RA \) and lastly it terminates. Writer \( \omega \), initially it broadcasts a discover message to all servers, and it then awaits for “fresh” discoverAck messages from a majority of servers, that is, set \( dAck \). Each of \( RA \) and \( dAck \) sets are from majorities of servers, and so \( RA \cap dAck \neq \emptyset \). By Lemma 4.4.5, any server \( s_k \in RA \cap dAck \) has a tag \( t g_{s_k} \) s.t. \( t g_{s_k} \geq t g \). Since \( \omega \) generates a new local tag-value \( (tg',v) \) pair in which it assigns the timestamp to be one higher than the one discovered in the maximum tag from set \( dAck \), it follows that \( t g' > t g \). Furthermore, \( \omega \) broadcasts the value to be written associated with \( t g' \) in a writeRequest message to all servers and it awaits for writeAck messages from a majority of servers before completion, set \( wAck \). Observe that each of \( dAck \) and \( wAck \) sets are from majority of servers, and so \( dAck \cap wAck \neq \emptyset \). By Lemma 4.3.1, any server \( s_k \in dAck \cap wAck \) has a tag \( t g_{s_k} \) s.t. \( t g_{s_k} \geq t g' > t g \) and the result follows.

We now show the correctness of algorithm \( OHMAM' \).

**Theorem 4.4.10** Algorithm \( OHMAM' \) implements an atomic MWMR object.

**Proof.** We use the above lemmas and the operations order definition to reason about each of the three atomicity conditions A1, A2 and A3.

**A1** For any \( \pi_1, \pi_2 \in \Pi \) such that \( \pi_1 \rightarrow \pi_2 \), it cannot be that \( \pi_2 \prec \pi_1 \).

If both \( \pi_1 \) and \( \pi_2 \) are writes and \( \pi_1 \rightarrow \pi_2 \) holds, then from Lemma 4.4.9 it follows that \( t g_{\pi_2} > t g_{\pi_1} \). By the ordering definition \( \pi_1 < \pi_2 \) is satisfied. When \( \pi_1 \) is a write, \( \pi_2 \) a read and
\(\pi_1 \rightarrow \pi_2\) holds, then from Lemma 4.4.7 it follows that \(tg_{\pi_2} \geq tg_{\pi_1}\). By definition \(\pi_1 \prec \pi_2\) is satisfied. If \(\pi_1\) is a read, \(\pi_2\) a write and \(\pi_1 \rightarrow \pi_2\) holds, then from Lemma 4.4.9 it follows that \(\pi_2\) always returns a tag \(tg_{\pi_2}\) s.t. \(tg_{\pi_2} > tg_{\pi_1}\). By the ordering definition \(\pi_1 \prec \pi_2\) is satisfied.

If both \(\pi_1\) and \(\pi_2\) are reads and \(\pi_1 \rightarrow \pi_2\) holds, then from Lemma 4.4.6 it follows that the tag returned from \(\pi_2\) is always greater or equal to the one returned from \(\pi_1\). \(tg_{\pi_2} \geq tg_{\pi_1}\).

If \(tg_{\pi_2} > tg_{\pi_1}\), then by the ordering definition \(\pi_1 \prec \pi_2\) holds. When \(tg_{\pi_2} = tg_{\pi_1}\) then the ordering is not defined but it cannot be that \(\pi_2 \prec \pi_1\).

A2 For any write \(\omega \in \Pi\) and any operation \(\pi \in \Pi\), then either \(\omega \prec \pi\) or \(\pi \prec \omega\).

If \(tg_{\omega} > tg_{\pi}\), then \(\pi \prec \omega\) follows directly. Similarly, if \(tg_{\omega} < tg_{\pi}\) holds, then it follows that \(\omega \prec \pi\). When \(ts_{\omega} = ts_{\pi}\) holds, then the uniqueness of each tag that a writer creates ensures that \(\pi\) is a read. In particular, remember that each tag is a \(\langle ts, id\rangle\) pair, where \(ts\) is a natural number and \(id\) a writer identifier. Tags are ordered lexicographically, first with respect to the timestamp and then with respect to the writer id. Since the writer ids are unique, then even if two writes use the same timestamp \(ts\) in the tag pairs they generate, the two tags cannot be equal as they will differ on the writer id. Furthermore, if the two tags are generated by the same writer, then by well-formedness it must be the case that the latter write will contain a timestamp larger than any timestamp used by that writer before. Since \(\pi\) is a read operation that receives either (i) readAck messages from a majority of servers, or (ii) readRelay messages from a majority of servers with the same \(tg\), then the intersection properties of majorities ensure that \(\omega \prec \pi\).

A3 Every read operation returns the value of the last write preceding it according to \(\prec\) (or the initial value if there is no such write).
Let $\omega$ be the last write preceding read $\rho$. From our definition it follows that $tg_\rho \geq tg_\omega$. If $tg_\rho = tg_\omega$, then $\rho$ returned a value written by $\omega$ in some servers. Read $\rho$ waited either (i) for readAck messages from a majority of servers, or (ii) readRelay messages from a majority of servers with the same $tg$. Thus the intersection properties of majorities ensure that $\omega$ was the last complete write operation. If $tg_\rho > tg_\omega$ holds, it must be the case that there is a write $\omega'$ that wrote $tg_\rho$ and by definition it must hold that $\omega \prec \omega' \prec \rho$. Thus, $\omega$ is not the preceding write and this cannot be the case. Lastly, if $tg_\rho = 0$, no preceding writes exist, and $\rho$ returns the initial value.

\[\square\]

### 4.4.4 Performance of the Revised Algorithms

In algorithm OHSAM$'$ write operations take 2 exchanges and read operations take 2 or 3 exchanges. The (worst case) message complexity of read operations is $|S|^2 + 3|S|$ and the (worst case) message complexity of write operations is $2|S|$.

In algorithm OHMAM$'$ write operations take 4 exchanges and read operations take 2 or 3 exchanges. The (worst case) message complexity of read operations is $|S|^2 + 3|S|$ and the (worst case) message complexity of write operations is $4|S|$. Both algorithms perform a modest amount of local computation, resulting in negligible computation overhead. These results follows directly from the structure of the algorithms.

### 4.5 Experimental Evaluation

In order to explore Research Question 4.4 we did a comparative study of our algorithms by simulating them using the NS3 discrete event simulator [4]. NS3 is a highly customizable and extensible simulator that allows us to gain full control over the event scheduler and the
deployment environment. Thus, it allows us to investigate the performance of our algorithms under exact environmental conditions.

In particular, we implemented the following three comparable SWMR algorithms: ABD [10], OHSAM, and OHSAM'. We also implemented the corresponding three algorithms for the MWMR setting: ABD-MW (following the multi-writer extension [55]), OHMAM, and OHMAM'. For a better comparison, we implemented benchmark LB that mimics the minimum message requirements: LB performs two communication exchanges for read and write operations thus providing a lower bound on performance in simulated scenarios. Note that LB does not satisfy Atomicity and its use is strictly for comparison purposes.
**Experimentation Setup.** The experimental configuration consists of a single (SWMR) or multiple (MWMR) writers, a set of readers, and a set of servers. For our evaluation, we use simulations representing two different topologies, *Series* and *Star*, that use the same array of routers, but differ in the deployment of server and client nodes. Clients are connected to routers over 5Mbps links with 4ms delay and the routers over 10Mbps links with 6ms delay. In *Series* topology, Fig.1(a), a server is connected to each router over 10Mbps bandwidth with 2ms delay, modeling a network where servers are separated and appear to be in different networks. In *Star* topology, Fig.1(b), servers are connected to a single router over 50Mbps links with 2ms delay, modeling a network where servers are in close proximity and well-connected, e.g., a datacenter. Clients are located uniformly with respect to the routers. To subject the system to high communication traffic, no failures are assumed (ironically, crashes reduce network traffic). Communication among the nodes is established via point-to-point bidirectional links implemented with a DropTail queue. We ran NS3 on a Mac OS X with 2.5Ghz Intel Core i7 processor. The results are compiled as averages over five samples per each scenario.

**Performance.** We assess algorithms in terms of operation latency that depends on communication delays and local computation time. For operation latency we combine two clocks: the simulation clock to measure communication delays, and a real time clock for computation delays. The sum of the two yields latency.

**Scenarios.** The scenarios are designed to test the following:

– the scalability of the algorithms as the number of readers, writers, and servers increases;

– the contention effect on efficiency, and

– the effects of chosen topologies on the efficiency.
For scalability we test with the number of readers $|R|$ from the set $\{10, 20, 40, 80, 100\}$ and the number of servers $|S|$ from the set $\{10, 15, 20, 25, 30\}$. For the MWMR setting we use at most 80 readers and we range the number of writers $|W|$ over the set $\{10, 20, 40\}$. We issue reads (and writes) every $rInt$ (and $wInt$ respectively) from the set of $\{2, 3, 4\}$ seconds. We issue reads every $rInt = \{2, 3, 4, 6\}$ seconds and write operations every $wInt = 4$ seconds.

To test contention we define two invocation schemes: fixed and stochastic. In the fixed scheme all operations are scheduled periodically at a constant interval. In the stochastic scheme reads are scheduled randomly from the intervals $[1...rInt]$ and write operations from the intervals $[1..wInt]$. In order to test the effects of topology we run our simulations using the Series and the Star topologies.

**Results.** We generally observe that the proposed algorithms outperform algorithms ABD and ABD-MW in most scenarios by a factor of 2. A closer examination of the results yields the following observations.

**Scalability:** As seen in Figures 2(b) and 2(c), increasing the number of readers and servers increases latency in the SWMR algorithms. Additionally, for the MWMR setting we increase the number of the participating writers. The same observation holds for the MWMR algorithms. When the number of the participating readers and writers is reduced then not surprisingly the latency improves, but the relative performance of the algorithms remains the same. For both settings the impact is higher on algorithms ABD and ABD-MW.

**Contention:** We examine the efficiency of our algorithms under different concurrency schemes. We set the operation frequency to be constant and we observe that in the stochastic scheme read operations complete faster than in the fixed scheme; see Figures 2(c) and 2(d) for the SWMR setting, and Figures 3(g) and 3(h) for the MWMR setting. This is expected as the
fixed scheme causes congestion. For the stochastic scheme the invocation time intervals are distributed uniformly, this reduces congestion and improves latency.

Topology: Figures 2(a) and 2(b) for the SWMR setting, and Figures 3(e) and 3(f) for the MWMR setting show that topology substantially impacts performance. For both the SWMR and MWMR settings our algorithms outperform algorithms ABD and ABD-MW by a factor of at least 2 in Star topology where servers are well-connected. Our SWMR algorithms perform much better than ABD also in the Series topology. For the MWMR setting and Series topology, we
Figure 3: Simulation Results for the MWMR Setting.

note that ABD-MW generally outperforms algorithm OHMAM, however the revised algorithm OHMAM′ noticeably outperforms ABD-MW.

Lastly we compare the performance of algorithms OHSAM and OHMAM with revised versions OHSAM′ and OHMAM′. We note that OHSAM′ and OHMAM′ outperform all other algorithms in Series topologies. However, and perhaps not surprisingly, OHSAM and OHMAM outperform OHSAM′ and OHMAM′ in Star topology. This is explained as follows. In Star
topology readRelay and readAck messages are exchanged quickly at the servers and thus delivered quickly to the clients. On the other hand, the bookkeeping mechanism used in the revised algorithms incur additional computational latency, resulting in worse latency.

An important observation is that while algorithms $\text{OHSAM}'$ and $\text{OHMAM}'$ improve the latencies of some operations (allowing some reads to complete in two exchanges), their performance relative to algorithms $\text{OHSAM}$ and $\text{OHMAM}$ depends on the deployment setting. Simulations show that $\text{OHSAM}$ and $\text{OHMAM}$ are more suitable for datacenter-like deployment, while in the “looser” settings algorithms $\text{OHSAM}'$ and $\text{OHMAM}'$ perform better.
Communication cost is the most commonly used metric in assessing the efficiency of operations in distributed algorithms for message-passing environments. In doing so, the standing assumption is that the cost of local computation is negligible compared to the cost of communication. However, in many cases operation implementations rely on complex computations that should not be ignored. Thus, in some cases, proposed solutions either require participation restriction in the system or incur in high computation demands, resulting in solutions that are not suitable to be used in practice.

We are interested in devising implementations that reduce both communication and computation demands. Examining the best two, in terms of communication demands, algorithms that implement atomic SWMR atomic memory, CCFAST [8] and OHSAM (Section 4.2), we observe that both solutions come with their trade-offs. In particular, algorithm CCFAST achieves optimal communication by allowing each operation to complete in one round trip, and has light computation requirements. However, it relies on strict limitations on the number of participating readers. On the other hand, algorithm OHSAM performs negligible computation, imposes
no participation restrictions on the system, but it provides operations that always require one-and-a-half communication rounds trips before completion.

In the light of these shortcomings, we present two algorithms that implement multi-speed operations without imposing any restrictions on the number of participants in the system. In particular, in Section 5.1, we present algorithm CCHYBRID that adopts the fast writes presented in [8], and makes clients to switch to a slow mode whenever the system is congested. Moreover, in Section 5.2 we present algorithm OHFAST, which pushes the responsibility of deciding for the speed switch to the servers. This allows algorithm OHFAST to utilize the fast operations presented in [8] and the one-and-a-half-rounds operations of OHSAM, whenever is necessary. We rigorously reason about the correctness of all the proposed algorithms.

Lastly, in order to assess the practical efficiency of the proposed algorithms, we simulate them with existing comparable solutions using the NS3 simulator. We compare their performance in terms of operation latency and ratio of slow over fast operations performed under various scenarios, topologies and operation loads. Simulation results are analyzed in Section 5.3.

5.1 Switching between One and Two Rounds

As discussed in Section 3.2.1, Dutta et al. [22] presented an algorithm called FAST, in which all read and write operations involve only two communication exchanges. To avoid the second round in read operations, FAST uses two mechanisms: (i) a recording mechanism at the servers, and (ii) a predicate that uses the server records at the readers. A recent work by Fernández Anta, Nicolaou, and Popa [8], has shown that although the result in [22] is efficient
in terms of communication exchanges, it requires reader processes to evaluate a computationally hard predicate. The authors abstracted the predicate used in FAST as a computational problem that they show to be NP-hard via a reduction from the decision version of the Maximum Edge Biclique Problem [61], which is NP-Complete, as stated in the following theorem. In the same work, authors modified the predicate of FAST [22] and proposed a new algorithm, called $\text{CCFAST}$, that allows the operations to be fast with only polynomial computation overhead. The idea of the new predicate is to examine the replies received in the first communication round of a read operation and determine how many (instead of which [22]) processes witnessed the maximum timestamp among those replies. With this modification, the predicate takes polynomial time to decide the value to be returned and it reduces the size of each message sent by the replica nodes. Algorithm $\text{CCFAST}$ is more practical than FAST, but it has the same constraint on the number of readers. It guarantees correctness only when the number of readers is constrained with respect to the number of replicas servers and in inverse proportion to the number of crashes, i.e., $|\mathcal{R}| < \frac{|\mathcal{S}|}{T} - 2$.

Here, we examine whether we can combine the techniques presented by Attiya et al. in [10] and Fernández Anta et al. [8] to obtain a SWMR algorithm that allows one or two round-trip operations while removing the constraint on the number of participating readers, i.e., permit unbounded participation in the system. In particular, this section aims to tackle this problem by examining the following research questions:

**Research Question 5.1** Can we devise an atomic read/write shared objects implementation for the asynchronous, crash-prone, message-passing, static SWMR setting with unbounded
participation; such that read operations take either two or four communication exchanges to complete?

**Research Question 5.2** How the analytical results of the proposed algorithm are reflected in practical efficiency?

We elaborate on these research questions in the sections that follow.

### 5.1.1 Description of SWMR Algorithm CCHYBRID

Algorithm CCHYBRID aims to allow unbounded number of readers to participate in the service while allowing operations to complete in either two or four communication exchanges. In particular, CCHYBRID combines ideas from algorithms CCFast [8] and ABD [10]: (i) it exploits timestamp-value pairs to order the write operations, (ii) it uses the predicate proposed by CCFast to determine the value returned by a fast read and (iii) it propagates the maximum timestamp-value pair to a majority of servers during a slow read as in algorithm ABD.

The biggest challenge in CCHYBRID is to determine when a second phase is necessary, and ensure that such a strategy does not violate atomicity. The key idea of CCHYBRID is to have the reader examine if the number of processes that observed the latest value is under the bound $|S| - 2$. If this holds, then CCHYBRID evaluates the predicate proposed in CCFast over the replies, to determine the value to return. This happens on the readers side while receiving messages from exchange E2, resulting to read operations that take two exchanges to complete. Otherwise, if the number of processes that observed the latest value is over the bound, the reader proceeds to a propagation phase to write the latest discovered value on at least a majority of replica servers before completion. Whenever this propagation phase is needed, it adds an
overhead of 2 exchanges on a read operation, resulting to reads that take four exchanges to complete. Additionally, in order to prevent readers from propagating an already propagated value, servers maintain a flag that indicates whether a timestamp has been already propagated. Notice that algorithm CC HYBRID performs equally to CCFAST when the number of readers that return the same value (not necessarily the same readers for each value) satisfies the bound required by CCFAST. In any other case, a single complete, slow read operation (similar to [33]) is necessary per write operation.

The code for the reader and writer protocols is given in Algorithm 8 and for the server protocol in Algorithm 9. We now give the details of the protocols; in referring to the pseudocode of an algorithm, we use prefix “A” and for numbered lines of code we use the prefix “L” to stand for “line”.

Counter variables rcounter, wcounter and Counter are used to help processes identify “new” read and write operations, and distinguish “fresh” from “stale” messages (since messages can be reordered). The value of the object and its associated timestamp, as known by each process, are stored in variables v and ts respectively. Variable vp is used to store the value associated with timestamp maxTS − 1. Set srvAck, at each reader r, stores all the received acknowledgment messages. Variable maxTS holds the maximum timestamp discovered in the set of the received acknowledgment messages srvAck. The set maxAck holds all the received messages that contain maxTS. The set maxViews at each reader r, holds the maximum number of server processes that witnessed the maximum timestamp maxTS, recorded by a server process and sent in an acknowledgment message to the reader r. Each server records all the processes that witness its local timestamp, in a set called seen. The use of the prop flag allows any read that succeeds a slow read, and returns the same value, to be
fast, as: (i) The slow read propagates the maxTS to \(|S| - f\) servers, (ii) a succeeding read receives replies from \(|S| - f\) servers, and (iii) the read discovers prop = True for maxTS in more than \(|S| - 2f > f + 1\) servers. Below we provide a brief description of the predicate and the protocol of each participant of the service.

**Predicate.** The purpose of the predicate is to allow a read operation to predict the value that was potentially returned by a preceding read operation. To understand the idea behind the predicate consider the following execution, \(\xi_1\). Let the writer perform a write operation \(\omega\) and receive replies from a set \(S_1\) of \(|S| - f\) servers. Let a reader follow and perform a read operation \(\rho_1\) that receives replies from a set of servers \(S_2\) again of size \(|S| - f\) that misses \(f\) servers that replied to the write operation \(\omega\). Due to asynchrony, an operation may miss a set of servers if the messages of the operation are delayed to reach any servers in that set. So the two sets intersect in \(|S_1 \cap S_2| = |S| - 2f\) servers. Consider now \(\xi_2\) where the write operation \(\omega\) is not complete and only the servers in \(S_1 \cap S_2\) receive the write requests. If \(\rho_1\) receive replies from the same set \(S_2\) in \(\xi_2\) then it won’t be able to distinguish the two executions. In \(\xi_1\) however the read has to return the value written, as the write in that execution proceeds the read operation. Thus, in \(\xi_2\) the read has to return the value written as well. If we extend \(\xi_2\) by another read operation \(\rho_2\) from a third process, then it may receive replies from a set \(S_3\) missing \(f\) servers in \(|S_1 \cap S_2|\). Thus it may see the value written in \(|S_1 \cap S_2 \cap S_3| = |S| - 3f\) servers. But since there is another read that saw the value from these servers (read \(\rho_1\)) then \(\rho_2\) has to return the written value to preserve atomicity. Observe now that \(\rho_1\) saw the written value from \(|S| - 2f\) servers and each server replied to both \(\{w, \rho_1\}\), and \(\rho_2\) saw the written value from \(|S| - 3f\) and each server replied to all three \(\{\omega, \rho_1, \rho_2\}\). By continuing with the same logic, we derive
Algorithm 8 Reader and Writer Protocols for SWMR algorithm CCHYBRID

1: at each reader $r$

2: Variables:
3: $ts \in \mathbb{N}^+, \maxTS \in \mathbb{N}^+, v \in V, vp \in V, rcounter \in \mathbb{N}^+$
4: $propSet \subseteq S, \text{srvAck} \subseteq S \times M, \maxAck \subseteq S \times M, \maxViews \in \mathbb{N}^+$

5: Initialization:
6: $ts \leftarrow 0; \maxTS \leftarrow 0; v \leftarrow \bot; vp \leftarrow \bot, rcounter \leftarrow 0$
7: $propSet \leftarrow \emptyset, \text{srvAck} \leftarrow \emptyset, \maxAck \leftarrow \emptyset, \maxViews \leftarrow 0$

8: function READ()
9:  $rcounter \leftarrow rcounter + 1$
10:  $\text{send}(\text{readRequest}, ts, v, vp, r, rcounter) \text{ to } S$
11:  $\text{wait until } (|\text{srvAck}| = |S| - f)$
12:  $\maxTS \leftarrow \max(\{m.ts' | (s,m) \in \text{srvAck}\})$
13:  $\maxAck \leftarrow \{(s,m) | (s,m) \in \text{srvAck} \land m.ts' = \maxTS\}$
14:  $(ts, v, vp) \leftarrow m.(ts', v', vp') \text{ for } (s,m) \in \maxAck$
15:  $\maxViews \leftarrow \max(\{m.seen | (s,m) \in \maxAck\})$
16:  $\text{propSet} \leftarrow \{s | (s,m) \in \maxAck \land m.prop = \text{True}\}$
17:  if $(\maxViews > \frac{|S|}{T} - 2) \lor (\text{propSet} \neq \emptyset)$ then
18:     if $(|\text{propSet}| < f)$ then
19:         $rcounter \leftarrow rcounter + 1$
20:         $\text{srvAck} \leftarrow \emptyset$
21:         $\text{send}(\text{writeRequest}, ts, v, vp, r, rcounter) \text{ to } S$
22:         $\text{wait until } (|\text{srvAck}| = |S| - f)$
23:     else
24:         $\text{return}(v)$
25: else
26:     if $\exists \alpha \in [1, \frac{|S|}{T} - 2] \text{ s.t. } MS = \{s | (s,m) \in \maxAck \land m.seen \geq \alpha\} \land |MS| \geq |S| - \alpha f$ then
27:         $\text{return}(v)$
28:     else
29:         $\text{return}(vp)$

30: Upon receive $m$ from $s$
31: if $(m.counter = rcounter)$ then
32: $\text{srvAck} \leftarrow \text{srvAck} \cup \{(s,m)\}$

33: At writer $w$
34: Variables:
35: $ts \in \mathbb{N}^+, v, vp \in V, wcounter \in \mathbb{N}^+$
36: Initialization:
37: $ts \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, wcounter \leftarrow 0$
38: function WRITE($val: \text{input}$)
39:  $vp \leftarrow v$
40:  $v \leftarrow val$
41:  $ts \leftarrow ts + 1$
42:  $\text{wcounter} \leftarrow \text{wcounter} + 1$
43:  $\text{wAck} \leftarrow \emptyset$
44:  $\text{broadcast}(\text{writeRequest}, ts, v, vp, w, wcounter) \text{ to } S$
45:  $\text{wait until } (|\text{wAck}| = |S| - f)$
46:  $\text{return}$

47: Upon receive $m$ from $s$
48: if $(m.counter = wcounter)$ then
49: $\text{wAck} \leftarrow \text{wAck} \cup \{(s,m)\}$
Algorithm 9 Server Protocol for SWMR algorithm cCHYBRID

1: at each server $s$
2: Components:
3: $ts \in \mathbb{N}^+$, $seen \subseteq \mathcal{R} \cup \{w\}$, $v \in V, vp \in V, prop \in \{True, False\}$
4: $Counter[1..|\mathcal{R}| + 2]$: array of int
5: Initialization:
6: $ts \leftarrow 0, seen \leftarrow \emptyset, v \leftarrow \bot, vp \leftarrow \bot, prop \leftarrow False$
7: $Counter[i] \leftarrow 0$ for $i \in \mathcal{R} \cup \{w\}$
8: Upon receive($\langle writeRequest, ts', v', vp', w, wcounter \rangle$)
9: if ($Counter[w] < wcounter$) then
10: $Counter[w] \leftarrow wcounter$
11: if ($ts < ts'$) then
12: $\langle ts, v, vp \rangle \leftarrow \langle ts', v', vp' \rangle$
13: $seen \leftarrow \{w\}$
14: $prop \leftarrow False$
15: else
16: $seen \leftarrow seen \cup \{w\}$
17: send($\langle writeAck, Counter[w], s \rangle$) to $w$
18: Upon receive($\langle readRequest, ts', v', vp', r, rcounter \rangle$)
19: if ($Counter[r] < rcounter$) then
20: $Counter[w] \leftarrow wcounter$
21: if ($ts' > ts$) then
22: $\langle ts, v, vp \rangle \leftarrow \langle ts', v', vp' \rangle$
23: $seen \leftarrow \{q\}$
24: $prop \leftarrow False$
25: else
26: $seen \leftarrow seen \cup \{r\}$
27: if ($ts' = ts$) then
28: $prop \leftarrow True$
29: send($\langle readAck, ts, v, vp, |seen|, prop, Counter[r] \rangle$) to $r$

the predicate that if a read sees a value written in $|S| - \alpha f$ servers and each of those servers sent this value to $\alpha$ other processes then we return the written value.

Notice that in order for a subsequent operation to obtain a written value from at least a single server, it must be the case that the current operation observes the value in $|S| - \alpha f > f$. Solving this equation results in $\alpha < \frac{|S|}{f} - 1$. But $\alpha$ is the number of processes in the system. As the maximum number of processes is $|\mathcal{R}| + 1$, hence the bound on the number of possible reader participants $|\mathcal{R}| < \frac{|S|}{f} - 2$. 

103
**Writer Protocol.** Writer $w$ increments its local timestamp and broadcasts a writeRequest message to all the participating servers $s \in S$ in exchange $E_1$ (A8:L40-43). Once the writer receives writeAck messages from $|S| - f$ servers during $E_2$, the operation completes (A8:L44).

**Reader Protocol.** When a read process $r$ invokes a read operation it sends readRequest messages to all the servers in exchange $E_1$ and waits to collect messages from $|S| - f$ servers from $E_2$ (A8:L10-11). When readAck messages are received from a majority of servers, the reader discovers the maximum timestamp, $maxTS$, among the replies (A8:L12), the set of messages $maxAck$ that contained $maxTS$ (A8:L13), and the maximum views reported in those messages (A8:L15). If the maximum views are less than $\frac{|S|}{f} - 2$ and no reader propagated the maximum timestamp, $propSet = \emptyset$, (A8:L17), then the reader evaluates the predicate as in algorithm $CCFAST$ to decide which value to return. Otherwise, the reader will return the value $v$ associated with the $maxTS$. However, before doing so, the reader checks if at least $f + 1$ of the messages that contain $maxTS$ also contain $prop = True$. Meaning that $maxTS$ is already propagated to a majority of servers. If this is the case, the reader returns $fast$ the value $v$ associated with the $maxTS$ without performing any further actions. If not, then the reader performs a second phase propagating the maximum timestamp-value pair to $|S| - f$ servers before completion (A8:L18-L22).

**Server Protocol.** The server protocol is the most involved. In addition to the replica state (timestamp and value), a server $s$ maintains a set $seen$ to record the processes that requested this replica, and a flag $prop$ that, as we explained earlier, its use is to optimize read operations. Each server $s \in S$ expects two types of messages:

1. Upon receiving a $\langle readRequest, ts', v', vp', r, rcounter \rangle$ message from reader $r$ server $s$ updates its local replica state and $seen$ set appropriately. Additionally, server compares its
local timestamp to the one enclosed in the message and if the attached timestamp is greater than its local timestamp, it also sets \textit{prop} flag to \textit{False} (A9:L22-23). In case the timestamp of the message is not greater than the local timestamp of \textit{s}, then the server records the sender in its \textit{seen} set (A9:L26). Server \textit{s} sets \textit{prop} = \textit{True} when it receives a message from a reader that contained a \textit{timestamp-value} pair equal to the one that is locally stored at \textit{s} (A9:L28). Notice that a reader propagates a \textit{timestamp-value} pair in every phase. So, \textit{s} may set \textit{prop} during the first or second phase of a read. Lastly, reader acknowledges the requesting reader with a readAck message (A9:L17).

(2) Upon receiving a \langle \textit{writeRequest}, \textit{ts}', \textit{v}', \textit{vp}', \textit{w}, \textit{wcounter} \rangle message the server updates its local replica state and \textit{seen} set appropriately. In case the timestamp in the request is greater than its local timestamp it also sets \textit{prop} flag to \textit{False} (A9:L11-16). It then acknowledges the requesting writer with a writeAck message (A9:L17).

5.1.2 Correctness of \textsc{CCHybrid}

To prove correctness of algorithm \textsc{CCHybrid} we reason about its \textit{liveness} (termination) and \textit{atomicity} (safety).

\textbf{Liveness.} Termination holds with respect to our failure model: \(|S| - f\) servers do not fail and and each operation waits for no more than \(|S| - f\) messages for completion. We now give additional details.

\textbf{Write Operation.} Per algorithm \textsc{CCHybrid}, writer \textit{w} creates a \textit{writeRequest} message and then it broadcasts it to all servers in exchange \textit{E1} (A8:L43). Writer \textit{w} then waits for \textit{writeAck} messages from \(|S| - f\) servers from \textit{E2} (A8:L44). According to our failure model \(|S| - f\)
servers do not fail and can receive writeRequest and send writeAck messages to the requesting writer, thus a write operation \( \omega \) terminates.

**Read Operation.** Each operation \( \rho \) sends readRequest messages to all the servers in exchange \( E_1 \) (A8:L10) and waits for \( |S| - f \) readAck messages from exchange \( E_2 \) (A8:L11). According to our failure model \( |S| - f \) servers do not fail and can receive the readRequest messages and reply back with a readAck message to the requesting reader. In cases where the reader must perform a second round to propagate the maximum timestamp before termination, then \( \rho \) sends writeRequest messages to all the servers in exchange \( E_3 \) (A8:L21) and waits for \( |S| - f \) writeAck messages from exchange \( E_4 \) (A8:L22). Since at least \( |S| - f \) servers receive the messages from \( r \) during exchanges \( E_1 \) and \( E_3 \) and at least \( |S| - f \) send an acknowledgment message to \( r \) during exchanges \( E_2 \) and \( E_4 \), and \( r \) awaits for no more than \( |S| - f \) messages, then termination of \( \rho \) is always guaranteed.

**Atomicity.** We use the association between the timestamps and the partial order as given in Section 4.2.2. We now state and prove a series of lemmas.

Monotonicity allows the ordering of the values according to their associated timestamps. So Lemma 5.1.2 shows that the \( ts \) variable maintained by each server process in the system is monotonically increasing. Let us first make the following observation:

**Lemma 5.1.1** In any execution \( \xi \) of CCHYBRID, if a server \( s \) replies with a timestamp \( ts \) at time \( T \), then \( s \) replies with a timestamp \( ts' \geq ts \) at any time \( T' > T \).
Proof. A server attaches in each reply its local timestamp. Its local timestamp in turn is updated only whenever the server receives a higher timestamp. So the server local timestamp is monotonically non-decreasing and the lemma follows.

The following is also true for a server process.

Lemma 5.1.2 In any execution $\xi$ of CCHYBRID, if a server $s$ receives a timestamp $ts$ at time $T$ from a process $p$, then $s$ replies with a timestamp $ts' \geq ts$ at any time $T' > T$.

Proof. If the local timestamp of the server $s$, $ts_s$, is smaller than $ts$, then $ts_s = ts$. Otherwise $ts_s$ does not change and remains $ts_s \geq ts$. In any case $s$ replies with a timestamp $ts_s \geq ts$ to $\pi$. Since the timestamp of $s$ is monotonically incrementing, then $s$ attaches a timestamp $ts' \geq ts_s$, and hence $ts' \geq ts$, to any subsequent reply.

Now we show that the timestamp is monotonically non-decreasing for the writer and the reader processes.

Lemma 5.1.3 In any execution $\xi$ of CCHYBRID, the variable $ts$ stored in any process is non-negative and monotonically non-decreasing.

Proof. The lemma holds for the writer as it changes its local timestamp by incrementing it every time it performs a write operation. The timestamp at each reader becomes equal to the largest timestamp the reader discovers from the server replies. So it suffices to show that in any two subsequent reads from the same reader, say $\rho_1, \rho_2$ s.t. $\rho_1 \rightarrow \rho_2$, then $\rho_2$ returns a $ts'$ that is bigger or equal to the timestamp $ts$ returned by $\rho_1$. This can be easily shown by the fact that $\rho_2$ attaches the maximum timestamp discovered by the reader before the execution of $\rho_2$. Say this is $ts$ discovered during $\rho_1$. By Lemma 5.1.2 any server that will receive the message from
ρ₂ will reply with a timestamp τₛ ≥ τₛ. So ρ₂ will discover a maximum timestamp τₛ′ ≥ τₛ. If τₛ′ = τₛ then the predicate will hold for α = 1 for ρ₂ and thus it stores τₛ′ = τₛ. If τₛ′ > τₛ then ρ₂ stores either τₛ′ or τₛ′ − 1. In either case it stores a timestamp greater or equal to τₛ and the lemma follows.

Next, we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

**Lemma 5.1.4** In any execution ξ of CCHYBRID, if a read ρ from r₁ succeeds a write operation ω that writes timestamp τₛ from the writer w, i.e. ω → ρ, and ρ returns a timestamp τₛ′, then τₛ′ ≥ τₛ.

**Proof.** Per algorithm CCHYBRID a read operation ρ that succeeds a write operation ω can be either (a) fast or (b) slow.

**Case 1:** We now examine the case where the read operation is fast. According to the algorithm, the write operation ω communicates with a set of |Sₛₗ| = |S| − f servers before completing. Let |S₁| = |S| − f be the number of servers that replied to the read operation ρ. The intersection of the two sets is |Sₛₗ ∩ S₁| ≥ |S| − 2f and since f < |S|/2 there exists at least a single server s that replied to both operations. Each server s ∈ Sₛₗ ∩ S₁ replies to ω before replying to ρ. Thus, by Lemma 5.1.2 and since s receives the message from ω before replying to any of the two operations, then it replies to ρ with a timestamp τₛ ≥ τₛ. Thus there are two cases to investigate on the timestamp: 1(a) τₛ > τₛ, and 1(b) τₛ = τₛ.

Case 1(a): In the case where τₛ > τₛ, ρ will observe a maximum timestamp maxTS ≥ τₛ. Since ρ returns either τₛ′ = maxTS or τₛ′ = maxTS − 1, then τₛ′ ≥ τₛ − 1. Thus, τₛ′ ≥ τₛ as desired.
Case 1(b): In this case all the servers in $S_w \cap S_1$ reply with a timestamp $ts_s = ts$. The read $\rho$ may observe a maximum timestamp $maxTS \geq ts_s$. If $maxTS > ts_s$, then, with similar reasoning as in Case 1, we can show that $\rho$ returns $ts' \geq ts$. So it remains to investigate the case where $maxTS = ts_s = ts$. In this case, at least $|S_w \cap S_1| = |S| - 2f$ servers replied with $maxTS$ to $\rho$. Also for each $s \in S_w \cap S_1$, $s$ included both the writer identifier $w$ and $r_1$ before replying to $\omega$ and $\rho$ respectively. So $s$ replied with a size at least $s.views \geq 2$ to $\rho_2$. Thus, given that $|R| \geq 2$, the predicate holds for $\alpha = 2$ and the set $S_w \cap S_1$ for $\rho$, and hence it returns a timestamp $ts' = ts$.

Case 2: On the other hand, if $\rho$ is slow or observed $prop = True$ in more than $f + 1$ servers, then it returns $maxTS$. Since $|S| - f$ servers received $\omega$, and since $\rho$ contacts $|S| - f$ servers during its first phase, then there is at least a single server, say $s$ that received the message for $\omega$ before replying to $\rho$. According to Lemma 5.1.2, $s$ replies to $\rho$ with a timestamp $ts_s \geq ts_\omega$, the timestamp it received from $\omega$. Thus, $\rho$ observes a $maxTS \geq ts_s \geq ts_\omega$, and hence returns $ts' = maxTS \geq ts_\omega$ and the lemma follows.

Using the next three lemmas, we show that if a read operation $\rho_2$ succeeds read operation $\rho_1$, then $\rho_2$ always returns a value at least as recent as the one returned by $\rho_1$.

Lemma 5.1.5 In any execution $\xi$ of $\text{CCHYBRID}$, if $\rho_1$ and $\rho_2$ are two read operations such that $\rho_1 \rightarrow \rho_2$, $\rho_1$ is fast satisfying the predicate for $maxTS = ts_1$, then $\rho_2$ receives a $maxTS = ts_2$ s.t. $ts_2 \geq ts_1$.

Proof. Since the predicate holds for $\rho_1$, hence there exists an $\alpha \in [1, \frac{|S|}{f} - 2]$, and $MS_1 \subseteq S$ s.t. $|MS_1| = |S| - \alpha f$, and $\forall s \in MS_1$, $s.ts = ts_1$ and $s.views \geq \alpha$. Performing the substitutions
follows that:

$$|MS_1| \geq |S| - \left(\frac{|S|}{f} - 2\right)f \Rightarrow |MS_1| > f$$

Since $\rho_2$ receives replies from $|S_2| = |S| - f$ servers, then there exists a server $s \in MS_1 \cap S_2$ that replies to both $\rho_1$ and $\rho_2$. Since $\rho_1 \rightarrow \rho_2$ then $s$ replies to $\rho_1$ before replying to $\rho_2$. Since $s$ replies with $ts_1$ to $\rho_1$, then according to Lemma 5.1.2, it replies with a timestamp $ts_s \geq ts_1$ to $\rho_2$. Thus, $\rho_2$ observes a timestamp $maxTS \geq ts_1$ and hence $ts_2 \geq ts_1$.

Lemma 5.1.6 In any execution $\xi$ of CCHYBRID, if $\rho_1$ and $\rho_2$ are two fast read operations such that $\rho_1 \rightarrow \rho_2$, and both observe $maxViews \leq \frac{|S|}{f} - 2$ and check the predicate, and $\rho_1$ returns $ts_1$, then $\rho_2$ returns $ts_2 \geq ts_1$.

Proof. Let the two operations $\rho_1$ and $\rho_2$ be executed from the same process, say $r_1$. As explained in Lemma 5.1.3, $\rho_2$ will discover a maximum timestamp $maxTS \geq ts_{\rho_1}$. If $maxTS > ts_{\rho_1}$, then $\rho_2$ returns either $ts_{\rho_2} = maxTS$ or $ts_{\rho_2} = maxTS - 1$, and thus in both cases $ts_{\rho_2} \geq ts_{\rho_1}$. It remains to examine the case where $maxTS = ts_{\rho_1}$. Since $\rho_1 \rightarrow \rho_2$, then any message sent during $\rho_2$ contains timestamp $ts_{\rho_1}$. By Lemma 5.1.2, every server $s$ that receives the message from $\rho_2$ replies with a timestamp $ts_s \geq ts_{\rho_1}$. Since $maxTS = ts_{\rho_1}$, then it follows that all $|S| - f$ servers that replied to $\rho_2$, sent the timestamp $ts_{\rho_1}$. Before each server replies adds $r_1$ in their seen set. So they include a $views \geq 1$ in their messages. Thus, the predicate holds for $\rho_2$ for $\alpha = 1$ and returns $ts_{\rho_2} = maxTS = ts_{\rho_1}$.

For the rest of the proof we assume that the read operations are invoked from two different processes $r_1$ and $r_2$ respectively. Let $maxTS_1$ be the maximum timestamp discovered by $ts_{\rho_1}$. We have two cases to consider: (1) $\rho_1$ returns $ts_{\rho_1} = maxTS_1 - 1$, or (2) $\rho_1$ returns $ts_{\rho_1} = maxTS_1$. 

110
**Case 1:** In this case $\rho_1$ returns $ts_{\rho_1} = maxTS_1 - 1$. It follows that there is a server $s$ that replied to $\rho_1$ with a timestamp $maxTS_1$. This means that the writer invoked the write operation that tries to write a value with timestamp $maxTS_1$. Since the single writer invokes a single operation at a time (by well-formedness), it must be the case that the writer completed writing timestamp $maxTS_1 - 1$ before the completion of $\rho_1$. Let that write operation be $\omega$.

Since, $\rho_1 \rightarrow \rho_2$, then it must be the case that $\omega \rightarrow \rho_2$ as well. So by Lemma 5.1.4, $\rho_2$ returns a timestamp $ts_{\rho_2}$ greater or equal to the timestamp written by $\omega$, and thus $ts_{\rho_2} \geq maxTS_1 - 1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$.

**Case 2:** This is the case where $\rho_1$ returns $ts_{\rho_1} = maxTS_1$. So it follows that the predicate is satisfied for $\rho_1$, and hence $\exists \alpha \in [1, \ldots, |R|]$ and a set of servers $M_1$ such that every server $s \in M_1$ replied with the maximum timestamp $maxTS_1$ and a seen set size $s.views \geq \alpha$, and $|M_1| \geq |S| - \alpha f$. We know that $\rho_2$ receives replies from a set of servers $|S_2| = |S| - f$ before completing. Let $M_2$ be the set of servers that replied to $\rho_2$ with a maximum timestamp $maxTS_2$. Since $|R| < \frac{|S|}{f} - 2$, then

$$|M_1| > |S| - \left(\frac{|S|}{f} - 2\right)f \Rightarrow |M_1| > f$$

Hence, $S_2 \cap M_1 \neq \emptyset$ and by Lemma 5.1.2 every server $s \in S_2 \cap M_1$ replies to $\rho_2$ with a timestamp $ts_s \geq maxTS_1$. Therefore $maxTS_2 \geq maxTS_1$. If $maxTS_2 > maxTS_1$, then $\rho_2$ returns a timestamp $ts_{\rho_2} \geq maxTS_2 - 1 \Rightarrow ts_{\rho_2} \geq maxTS_1$ and hence $ts_{\rho_2} \geq ts_{\rho_1}$.

It remains to investigate the case where $maxTS_2 = maxTS_1$. Notice that any server in $s \in S_2 \cap M_1$ is also in $M_2$. Since $\rho_2$ may skip $f$ servers that reply to $\rho_1$, then $|M_1 \cap M_2| \geq |S| - (a + 1)f$. Recall that for each server $s \in M_1 \cap M_2$, $s$ replied with a size $s.views \geq a$ to
\(\rho_1\). Also \(s\) adds \(r_2\) in its seen set before replying to \(\rho_2\). So there are two subcases to examine:

(a) either \(r_2\) was already in the seen set of \(s\), or (b) \(r_2\) was not a member of \(s\).seen.

**Case 2(a):** If \(r_2\) was already a part of the seen set of \(s\), then the size of the set remains the same. It also means that \(r_2\) obtained \(maxTS_1\) from \(s\) in a previous read operation, say \(\rho'_2\) from \(r_2\). Since each process satisfies well-formedness, it must be the case that \(r_2\) completed \(\rho'_2\) before invoking \(\rho_2\). All the messages sent by \(\rho_2\) contained \(maxTS_1\). So by Lemma 5.1.2 any server \(s \in S_2\) replies to \(r_2\) with a timestamp \(ts_s = maxTS_2 = maxTS_1\). In this case \(|S| - f\) servers replied with \(maxTS_2\) and their seen set contains at least \(r_2\), having \(s.views \geq 1\). Thus, the predicate is valid with \(\alpha = 1\) for \(\rho_2\) which returns \(ts_{\rho_2} = maxTS_2 = maxTS_1 = ts_{\rho_1}\).

**Case 2(b):** This case may arise if \(r_2\) is not part of the seen set of every server \(s \in M_1 \cap M_2\). If \(r_2\) is part of the seen set of some server \(s' \in M_1 \cap M_2\), then this is resolved by case 2(a). So each server \(s \in M_1 \cap M_2\) inserts \(r_2\) in their seen sets before replying to \(\rho_2\). So if the size of the set \(s.views = \alpha\) when \(s\) replied to \(\rho_1\), \(s\) includes a size \(s.views \geq a + 1\) when replying to \(\rho_2\). Notice here that if \(\alpha = |R| + 1\) for \(\rho_1\), then it means that \(r_2\) was already part of the seen set of \(s\) when \(s\) replied to \(\rho_1\). This case is similar to 2(a). So we assume that \(\alpha < |R| + 1\), in which case \(\alpha + 1 \leq |R| + 1\). Since every server \(s \in M_1 \cap M_2\) replies with \(s.views \geq \alpha + 1\) to \(\rho_2\) and since \(|M_1 \cap M_2| \geq |S| - (\alpha + 1)f\), then the predicate holds for \(\alpha + 1 \leq |R| + 1\) and the set \(MS = M_1 \cap M_2\) for \(\rho_2\), and thus \(\rho_2\) returns \(ts_{\rho_2} = maxTS_2 = maxTS_1 = ts_{\rho_1}\) in this case as well. And this completes our proof. \(\square\)

**Lemma 5.1.7** In any execution \(\xi\) of CCHYBRID, if \(\rho_1\) and \(\rho_2\) are two read operations such that \(\rho_1 \rightarrow \rho_2\), and \(\rho_1\) returns \(ts_1\), then \(\rho_2\) returns \(ts_2 \geq ts_1\).
Proof. A read operation has two modes: fast and slow. Thus, we need to examine all the possible combinations of the speeds of $\rho_1$ and $\rho_2$. There are four cases to investigate: (1) $\rho_1$ is fast, and $\rho_2$ is fast, (2) $\rho_1$ is fast, and $\rho_2$ is slow, (3) $\rho_1$ is slow, and $\rho_2$ is slow, and (4) $\rho_1$ is slow, and $\rho_2$ is fast. Let $maxTS_i$ be the maximum timestamp observed by a read $\rho_i$, for $i \in \{1, 2\}$, during its first phase.

Case 1: In case both operations are fast then, according to CCHYBRID, either they observe $maxViews \leq \frac{|S|}{f} - 2$ and $propSet = \emptyset$, or they observe an $|propSet| \geq f + 1$. If both observe $maxViews \leq \frac{|S|}{f} - 2$ and check the predicate, then with the same reasoning as in Lemma 5.1.6, it follows that $ts_2 \geq ts_1$.

If $\rho_1$ observes $|propSet| \geq f + 1$ then since $\rho_2$ receives replies from $|S_2| = |S| - f$ servers, then there exists a server $s \in propSet \cap S_2$ such that $s$ replies to both $\rho_1$ and $\rho_2$. Since $\rho_1 \rightarrow \rho_2$, then $s$ replies to $\rho_1$ before replying to $\rho_2$. Since $s$ replies with $maxTS_1$ to $\rho_1$, then by Lemma 5.1.2, $s$ replies with a timestamp $ts_s \geq maxTS_1$ to $\rho_2$. So $maxTS_2 \geq ts_s$ and hence $maxTS_2 \geq maxTS_1$. If $maxTS_2 = maxTS_1$ then $s$ will reply with $ts_s = maxTS_1$ and $prop = True$. In this case $\rho_2$ will return $ts_2 = maxTS_1 = ts_1$. If $maxTS_2 > maxTS_1$ then $\rho_2$ returns either $maxTS_2$ or $maxTS_2 - 1$ and thus $ts_2 \geq ts_1$.

It remains to examine the case where $\rho_1$ observes $maxViews \leq \frac{|S|}{f} - 2$ and $propSet = \emptyset$, and $\rho_2$ observes $|propSet| \geq f + 1$. If the predicate holds for $\rho_1$ then by Lemma 5.1.5, $\rho_2$ observes $maxTS_2 \geq maxTS_1$. Since $\rho_2$ observes $|propSet| \geq f + 1$ then it returns $ts_2 = maxTS_2$, and thus $ts_2 \geq ts_1$. If the predicate does not hold for $\rho_1$ then we know that the write operation propagating $maxTS_1 - 1$ completed before or during $\rho_1$. Since $\rho_1 \rightarrow \rho_2$ then this write completed before $\rho_2$ as well. Thus, by Lemma 5.1.4, $\rho_2$ observes $maxTS_2 \geq$
\[ \text{maxTS}_1 - 1. \] Since \( \rho_2 \) observes \(|\text{propSet}| \geq f + 1\), then it returns \( ts_2 = \text{maxTS}_2 \Rightarrow ts_2 \geq \text{maxTS}_1 - 1 \Rightarrow ts_2 \geq ts_1 \).

**Case 2:** Since \( \rho_1 \) in this case is fast then \( \rho_1 \) returns either: (i) \( \text{maxTS}_1 - 1 \), or (ii) \( \text{maxTS}_1 \).

In (i), since \( \rho_1 \) observed \( \text{maxTS}_1 \) and since we have a single writer, it follows that the write operation that wrote timestamp \( \text{maxTS}_1 - 1 \), say \( \omega_1 \), proceeds or is concurrent to \( \rho_1 \), and completes before the response step of \( \rho_1 \). Since \( \rho_1 \rightarrow \rho_2 \), then \( \omega_1 \rightarrow \rho_2 \). Since \( \rho_2 \) is slow, then it returns the maximum timestamp it observes, i.e. \( ts_2 = \text{maxTS}_2 \). Moreover, since \( \omega_1 \rightarrow \rho_2 \), and since both operations wait for \(|S| - f \) replies, then according to our failure model, there exist at least a single server \( s \) that replies to both operations, first to \( \omega_1 \) and then to \( \rho_2 \). According to Lemma 5.1.2, \( s \) sends a timestamp \( ts_s \geq \text{maxTS}_1 - 1 \) to \( \rho_2 \). Thus, \( \text{maxTS}_2 \geq \text{maxTS}_1 - 1 \), and therefore \( ts_2 \geq ts_1 \).

In (ii) it follows that either the predicate holds for \( \rho_1 \), or \( \rho_1 \) observes \(|\text{propSet}| \geq f + 1\). Since \( \rho_2 \) is slow and returns \( ts_2 = \text{maxTS}_2 \), then by Lemma 5.1.5 and with similar reasoning as in Case (a) for when \( \rho_1 \) observes \(|\text{propSet}| \geq f + 1\), we can show that \( \text{maxTS}_2 \geq \text{maxTS}_1 \) and hence \( ts_2 \geq ts_1 \).

**Case 3:** The case where both reads are slow is simple and resembles the behavior of the reads in ABD [10]. Here each read \( \rho_i \), for \( i \in [1, 2] \), returns \( \text{maxTS}_i \) and before completing it propagates \( \text{maxTS}_i \) to \(|S| - f \) servers. Thus, \( \rho_1 \) returns \( ts_1 = \text{maxTS}_1 \), and before completing propagates \( \text{maxTS}_1 \) to \(|P_1| = |S| - f \) servers. Since \( \rho_1 \rightarrow \rho_2 \), and since \( \rho_2 \) receives \(|S_2| = |S| - f \) replies, then it is going to receive a timestamp \( ts_s \geq \text{maxTS}_1 \) from at least a single server \( s \in P_1 \cap S_2 \). Thus, \( \rho_2 \) returns \( ts_2 = \text{maxTS}_2 \geq \text{maxTS}_1 \), and \( ts_2 \geq ts_1 \).
Case 4: So it remains to investigate the case where \( \rho_1 \) is slow and \( \rho_2 \) is fast. Observe that this case is possible when a server \( s \) is “saturated” by concurrent reads (more than \( \frac{|S|}{f} - 2 \)) and \( s \) replies to \( \rho_1 \) but does not reply to \( \rho_2 \). Now we have two cases to investigate: either \( \rho_2 \) observes \( \text{maxTS}_2 \geq \text{maxTS}_1 \), or \( \text{maxTS}_2 = \text{maxTS}_1 - 1 \). If \( \rho_2 \) observes a \( \text{maxTS}_2 \geq \text{maxTS}_1 \), it may either return \( ts_2 = \text{maxTS}_2 \) or \( ts_2 = \text{maxTS}_2 - 1 \). In either case \( ts_2 \geq \text{maxTS}_1 - 1 \Rightarrow ts_2 \geq ts_1 \).

Let us examine now the case where \( \text{maxTS}_2 = \text{maxTS}_1 - 1 \). Since \( \rho_1 \) is slow and returns \( \text{maxTS}_1 - 1 \), then before completing it propagates \( \text{maxTS}_1 - 1 \) to \( |S| - f \) servers. Let \( P_1 \) be the set of servers that received the messages and replied to the second phase of \( \rho_1 \). Moreover, \( |S_2| = |S| - f \) are the servers that received messages and replied to \( \rho_2 \). So by Lemma 5.1.2, every server \( s \in P_1 \cap S_2 \) replies to both \( \rho_1 \) and then to \( \rho_2 \), with a timestamp \( ts_s \geq \text{maxTS}_1 - 1 \). In addition \( s \) sets \( \text{prop} = \text{True} \) before replying to \( \rho_1 \). Since \( \text{maxTS}_2 = \text{maxTS}_1 - 1 \), then \( s \) replies with \( ts_s = \text{maxTS}_1 - 1 \) to \( \rho_2 \), and thus the \( \text{propSet} \) contains at least \( s \) in \( \rho_2 \). According to the algorithm \( \rho_2 \) returns \( ts_2 = \text{maxTS}_2 \) in this case and hence \( ts_2 \geq ts_1 \).

We now show the correctness of algorithm CCHYBRID.

**Theorem 5.1.8** Algorithm CCHYBRID implements an atomic SWMR object.

**Proof.** We now use the lemmas stated above and the operations order definition to reason about each of the three atomicity conditions A1, A2 and A3 as given in Definition 2.2.5.

**A1** For any \( \pi_1, \pi_2 \in \Pi \) such that \( \pi_1 \rightarrow \pi_2 \), it cannot be that \( \pi_2 \prec \pi_1 \).

When the two operations are reads and \( \pi_1 \rightarrow \pi_2 \) holds, then from Lemma 5.1.7 it follows that the timestamp of \( \pi_2 \) is no less than the one of \( \pi_1 \), i.e. \( ts_2 \geq ts_1 \). If \( ts_2 > ts_1 \), then by the ordering definition \( \pi_1 \prec \pi_2 \) is satisfied. When \( ts_2 = ts_1 \) then the ordering is not
defined, thus it cannot be the case that $\pi_2 \prec \pi_1$. If $\pi_2$ is a write, the sole writer generates a new timestamp by incrementing the largest timestamp in the system. By well-formedness, any timestamp generated in any write operation that precedes $\pi_2$ must be smaller than $ts_2$. Since $\pi_1 \rightarrow \pi_2$, then it holds that $ts_1 < ts_2$. Hence, by the ordering definition it cannot be the case that $\pi_2 \prec \pi_1$. Lastly, when $\pi_2$ is a read and $\pi_1$ a write, then by Lemma 5.1.4 it follows that $ts_2 \geq ts_1$. By the ordering definition, it cannot hold that $\pi_2 \prec \pi_1$ in this case either.

**A2** For any write $\omega \in \Pi$ and any operation $\pi \in \Pi$, then either $\omega \prec \pi$ or $\pi \prec \omega$.

If the timestamp returned from $\omega$ is greater than the one returned from $\pi$, i.e. $ts_\omega > ts_\pi$, then $\pi \prec \omega$ follows directly. Similarly, if $ts_\omega < ts_\pi$ holds, then $\omega \prec \pi$ follows. If $ts_\omega = ts_\pi$, then it must be that $\pi$ is a read and either (i) discovered $ts_\omega$ from a propagation set, propSet, written by $\omega$, or (ii) discovered $ts_\omega$ from a set of servers and the predicate is satisfied, or (iii) $\pi$ discovered $ts_\omega + 1$ but the predicate is not satisfied. Thus, $\omega \prec \pi$ follows.

**A3** Every read operation returns the value of the last write preceding it according to $\prec$ (or the initial value if there is no such write).

Let $\omega$ be the last write preceding read $\rho$. From our definition it follows that $ts_\rho \geq ts_\omega$. If $ts_\rho = ts_\omega$, then $\rho$ either: (i) discovered $ts_\omega$ from a propagation set, propSet, written by $\omega$, or (ii) discovered $ts_\omega$ from a set of servers and the predicate is satisfied, or (iii) $\pi$ discovered $ts_\omega + 1$ but the predicate is not satisfied. If case (i) holds then, it is clear that $\omega$ is the last preceding write since $\rho$ discovered $ts_\omega$ as the maximum timestamp maxTS and either (a) it was propagated to a set of servers and $\rho$ returns $ts_\omega$ without any further actions or (b) $\rho$ propagates $ts_\omega$ to a set of servers before completion. When case (ii) holds, then it is clear that $\omega$ is the last preceding write. If (iii) holds then by Lemma 5.1.4, and since $ts_\rho = ts_\omega$, it must be the case that $\rho$ is concurrent with $\omega'$ and hence $\omega$ is again the last preceding write. If
$ts_\rho > ts_\omega$, then it means that $\rho$ obtained a larger timestamp. However, the larger timestamp can only be originating from a write that succeeds $\omega$, thus $\omega$ is not the preceding write and this cannot be the case. Lastly, if $ts_\rho = 0$ as the maximum timestamp, then the predicate holds for $\alpha = 1$ and thus $ts_\rho \geq 0$, returning in the worst case the initial value.

5.1.3 Performance of CCHYBRID

Performance. We now assess the performance of CCHYBRID in terms of (i) latency of read and write operations as measured by the number of communication exchanges (ii) the message complexity of read and write operations and (iii) computational complexity.

In brief, for algorithm CCHYBRID write operations take 2 communication exchanges and read operations take either 2 or 4 communication exchanges. The (worst case) message complexity of read operations is $4|\mathcal{S}|$ and the (worst case) message complexity of write operations is $2|\mathcal{S}|$. This follows directly from the structure of the algorithm. We now give additional details.

Operation Latency. We study the operation latency, in terms of the number of communication exchanges required by each operation.

Write operation latency: Per algorithm CCHYBRID, writer $w$ sends a writeRequest message to all the servers in exchange $E_1$, and, awaits for writeAck messages from at least a majority of servers in exchange $E_2$. Once the writeAck messages are received, no further communication is required and the write operation terminates. Therefore, any write operation consists of 2 communication exchanges.

Read operation latency: A reader sends a readRequest message to all the servers in the first communication exchange $E_1$ and, awaits for readAck messages from at least a majority of
servers in exchange \( E_2 \). Once the readAck messages are received, under some conditions, the read operation will either terminate or perform an additional communication round. In the case where a propagation round is required to ensure atomicity, the reader sends a writeRequest message to all the servers in exchange \( E_3 \), and, awaits for writeAck messages from at least a majority of servers in exchange \( E_4 \). Once the writeAck messages are received, no further communication is required and the read operation terminates.

**Message Complexity.** We measure operation message complexity as the worst case number of exchanged messages in each read and write operation. The worst case number of messages corresponds to failure-free executions where all participants send messages to all destinations according to the protocols.

*Write operation:* A single write operation in algorithm CCHYBRID takes 2 communication exchanges. In the first exchange \( E_1 \), the writer sends a writeRequest message to all the servers in \( S \). The second exchange \( E_2 \), occurs when all servers in \( S \) send a writeAck message to the writer. Thus, at most \( 2|S| \) messages are exchanged in a write operation.

*Read operation:* A single read operation in algorithm CCHYBRID in the worst case it takes 4 communication exchanges. In the first exchange \( E_1 \), the writer sends a readRequest message to all the servers in \( S \). The second exchange \( E_2 \), occurs when all servers in \( S \) send a readAck message to the reader. Then, in exchange \( E_3 \), the reader sends a writeRequest message to all the servers in \( S \) to propagate the maximum timestamp-value pair. Lastly, the fourth exchange \( E_4 \), occurs when all servers in \( S \) respond with a readAck message to the requesting reader. Thus, at most \( 2|S| \) messages are exchanged in a write operation.
Predicate Computational Complexity. Computation is minimal at the writer and server protocols. The most computationally intensive procedure is the computation of the predicate during a read operation. To analyze the computation complexity of CCHYBRID we design and analyze an algorithm to compute the predicate during a read operation.

**Algorithm 10 Linear Algorithm for Predicate Computation.**

1. `function ISVALIDPREDICATE(srvAck, maxTS)`
2. `buckets ← Array[1 ... |R| + 1], initially [0, ..., 0]`
3. for all `s ∈ srvAck` do
   4. if `s.ts = maxTS` then
      5. `buckets[s.views] ← buckets[s.views] + 1`
   6. for `α ← |R| + 1` down to `2` do
      7. if `buckets[α] ≥ (|S| − αf)` then
         8. return `True`
      9. else
         10. `buckets[α − 1] ← buckets[α − 1] + buckets[α]`
     11. if `buckets[1] = (|S| − f)` then
         12. return `True`
     13. return `False`

Algorithm 10 presents the formal specification of the predicate. Briefly, we assume that the input of the algorithm is a set `srvAck` and a value `maxTS` which indicate the servers that reply to a read operation and the maximum timestamp discovered among the replies, respectively. The algorithm uses a set of `|R| + 1` “buckets” each of which is initialized to 0. Running through the set of replies, `srvAck`, a bucket `k` is incremented whenever a server replied with the maximum timestamp and reports that this timestamp is seen by `k` processes (A10:L3-5). At the end of the parsing of the `srvAck` set, each bucket `k` holds how many servers reported the maximum timestamp and they sent this timestamp to `k` processes. Once we accumulate this information we check if the number of servers collected in a bucket `k` are more than `|S| − kf`. If they are, the procedure terminates returning `True`; else the number of servers in bucket `k` is added to the number of servers of bucket `k − 1` and we repeat the check of the condition (A10:L6-10). At this point the number kept at bucket `k − 1` indicates the total number of
servers that reported that their timestamp was seen by more or equal to $k - 1$ processes. This procedure continues until the above condition is satisfied or we reach the smallest bucket. If none of the buckets satisfies the condition the procedure returns False.

The complexity of Algorithm 10 specifies the computational complexity of the CCHYBRID. Algorithm 10 traverses once the set $srvAck$ and once the array of $|R| + 1$ buckets. Since, the set $srvAck$ may contain at most $|S|$ servers, and $|R|$ is bounded by $|S|$ when the predicate is evaluated, Algorithm 10 takes $O(|S|)$ time. This results to the following theorem.

**Theorem 5.1.9** The computational complexity of the predicate (algorithm 10) is $O(|S|)$.

### 5.2 Switching between One and One-and-a-Half Rounds

Our focus is still on the gap between one-round and two-round algorithms. Earlier in Section 4.2 we presented an algorithm for the SWMR model, called OHSAM, where each operation takes one-and-a-half-rounds to complete. Additionally, as discussed in Section 3.2.1, Fernández Anta et al. [8] presented an algorithm, called CCFAST, with a new predicate, that allows the operations to be fast with only polynomial computation overhead. In particular, the authors in [8] answered the following question: “Can we preserve atomicity if we know how many and not which processes read the latest value of a server?” Answering this question in positive yield two important benefits: (i) reduced the size of messages exchanged between the participants, and (ii) reduced the computation time of the predicate, it takes polynomial time to decide the value to be returned. Algorithm CCFAST is more practical than [22], but it has the same constraint on the number of participating readers.
Here, we examine whether we can combine the techniques used in algorithms OHSAM and CCFAST to obtain a SWMR algorithm that allows one and one-and-a-half round trip operations while removing the constraint on the number of participating readers, i.e., permit unbounded participation in the system. This section examines the following research questions:

**Research Question 5.3** Can we devise an atomic read/write shared objects implementation for the asynchronous, crash-prone, message-passing, static SWMR setting with unbounded participation; such that all read operations take at most three exchanges to complete?

**Research Question 5.4** How the analytical results of the proposed algorithm are reflected in practical efficiency?

Answers to the above research questions are presented in detail in the sections that follow.

### 5.2.1 Description of SWMR Algorithm OHFAST

Algorithm OHFAST aims to allow unbounded number of readers to participate in the service while allowing operations to complete in either two or three communication exchanges. In contrast to the classic approach of the four exchanges per read operation, OHFAST tries to further reduce the communication required by reads. Thus, OHFAST combines ideas suggested by CCFAST [8] and the one-and-a-half-rounds operations of OHSAM (Section 4.2).

Like in OHSAM, servers assume the responsibility of propagating the value of the timestamp instead of the reader. Similarly, in OHFAST we move the decision whether a slow read operation is necessary or not to the servers. In particular, the servers record the processes that requested their timestamp. If the recording set becomes “large” then a server relays a read to the other servers before replying to the reader. Thus, some of the servers may reply directly
to the requesting reader whereas some others, for the same read may perform a relay phase. However, there is a major departure from OHSAM: the servers that receive relay messages do not broadcast relays to all the servers but just to the servers that send them a relay. Therefore, only a single server may relay for a read operation keeping the message complexity of the algorithm low in cases of low contention.

When a server that relays a timestamp gets appropriate relays from the other servers, it marks the timestamp as secured, and sends a reply to the reader. When now the reader receives the replies from \(|S| - f\) servers, it collects the messages with the highest timestamp. If there is a server that declares this timestamp as secured then the read immediately returns the value associated with this timestamp; otherwise the reader evaluates the predicate of CCFast on the replies to determine the value to return. Notice that, it is possible for a read operation to terminate before receiving a reply from a relaying server. The code for the reader and writer protocols is given in Algorithm 11 and for the server protocol in Algorithm 12. We now give the details of the protocols.

Counter variables \(r_{counter}\), \(w_{counter}\) and \(Counter\) are used to help processes identify “new” read and write operations, and distinguish “fresh” from “stale” messages (since messages can be reordered). The value of the object and its associated timestamp, as known by each process, are stored in variables \(v\) and \(ts\) respectively. Variable \(vp\) is used to store the value associated with \(maxTS - 1\). Set \(srvAck\), at each reader \(r\), stores all the received acknowledgment messages. Variable \(maxTS\) holds the maximum timestamp discovered in the set of the received acknowledgment messages \(srvAck\). The set \(maxAck\) holds all the received messages that contain \(maxTS\). The set \(maxViews\) at each reader \(r\), holds the maximum number of server processes that witnessed the maximum timestamp \(maxTS\), recorded by a
server process and sent in an acknowledgment message to the reader $r$. Each server records all the processes that witness its local timestamp, in a set called $seen$. Additionally, each server maintains a $Relays$ array where it stores the latest timestamp it relayed for each reader. Below we provide a brief description of the protocol of each participant of the service.

**Writer Protocol.** Writer $w$ increments its local timestamp and broadcasts a $writeRequest$ message to all the participating servers $s \in S$ in exchange $E_1$ (A11:L34-37). Once the writer receives $writeAck$ messages from $|S| - f$ servers from $E_2$, the operation completes (A11:L38).

**Reader Protocol.** When a read process $r$ invokes a read operation it sends $readRequest$ messages to all the servers during $E_1$ and waits to collect messages from $|S| - f$ servers (A11:L11-12). Once those replies are received the reader discovers the maximum timestamp $maxTS$ among the replies (A11:L13), and collects all the messages that contain $maxTS$ in the set $maxAck$ (A11:L14). If some message in $maxAck$ indicates that $maxTS$ is secured, i.e., the value $v$ associated with $maxTS$ was sufficiently propagated, then the reader returns $v$ associated with $maxTS$ (A11:L17-18). Otherwise, the reader evaluates the predicate, that $ccFAST$ [8] uses, on the messages that belong in $maxAck$ to decide on which value to return. If the predicate is holds, then the reader returns the value $v$ associated with $maxTS$, otherwise the value $vp$ associated with $maxTS - 1$ (A11:L19-22).

**Server Protocol.** The server protocol is the most involved. The server’s state is composed of the state of the replica, the recording set $seen$, a flag $securedts$ which indicates whether a timestamp has been relayed to a majority of servers, and a $Relays$ array storing the latest timestamp the server relayed for each reader. Each server $s \in S$ expects three types of messages:

---

1Notice that, this is another departure from OHSAM as each reader in OHSAM returns the smallest discovered timestamp.
Algorithm 11 Reader and Writer Protocols for SWMR algorithm OHFAST

1: At each reader \( r \)

2: Variables:

3: \( ts \in \mathbb{N}^+, \text{max}TS \in \mathbb{N}^+, v, vp \in V, rcounter \in \mathbb{N}^+ \)

4: \( \text{srvAck} \subseteq S \times M, \text{maxAck} \subseteq S \times M, \text{maxViews} \in \mathbb{N}^+ \)

5: Initialization:

6: \( ts \leftarrow 0, \text{max}TS \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, rcounter \leftarrow 0, \text{maxViews} \leftarrow 0 \)

7: function READ()

8: \( rcounter \leftarrow rcounter + 1 \)

9: \( \text{srvAck} \leftarrow \emptyset \)
10: \( \text{maxAck} \leftarrow \emptyset \)

11: if \( \exists (s, m) \in \text{maxAck} \text{ s.t. } m.\text{secured} = \text{True} \) then

12: return \( v \)

13: else

14: return \( \bot \)

15: Upon receive \( m \) from \( s \)

16: if \( (m.\text{rcounter} = rcounter) \) then

17: \( \text{srvAck} \leftarrow \text{srvAck} \cup \{(s, m)\} \)

18: At writer \( w \)

19: Variables:

20: \( ts \in \mathbb{N}^+, v, vp \in V, wcounter \in \mathbb{N}^+ \)

21: Initialization:

22: \( ts \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, wcounter \leftarrow 0 \)

23: function WRITE(val : input)

24: \( \text{vp} \leftarrow v \)

25: \( v \leftarrow \text{val} \)

26: \( ts \leftarrow ts + 1 \)

27: \( wcounter \leftarrow wcounter + 1 \)

28: \( \text{wAck} \leftarrow \emptyset \)

29: broadcast(\( \text{writeRequest}, ts, v, vp, w, wcounter \)) to \( S \)

30: wait until \( (|\text{wAck}| = |S| - f) \)

31: return

32: Upon receive \( m \) from \( s \)

33: if \( (m.\text{wcounter} = wcounter) \) then

34: \( \text{wAck} \leftarrow \text{wAck} \cup \{(s, m)\} \)

(1) Upon receiving a \( \langle \text{readRequest}, ts', v', vp', r, rcounter \rangle \) message from reader \( r \) server \( s_j \) updates its local replica state and \( \text{seen} \) set appropriately. Additionally, server compares its local timestamp to the one enclosed in the message and if the attached timestamp is greater
Algorithm 12 Server Protocol for SWMR algorithm OhFAST

1: at each server $s_j$
2: Variables:
3: $ts \in \mathbb{N}^+, v \in V, vp \in V, scounter \in \mathbb{N}^+, securedts \in \{\text{True, False}\}, seen \subseteq R \cup \{w\}, srvRelay \subseteq S$
4: $\text{Relays}[1..|R| + 1]$: array of int, $\text{Counter}[1..|R| + 2]$: array of int

5: Initialization:
6: $ts \leftarrow 0, v \leftarrow \perp, vp \leftarrow \perp, scounter \leftarrow 0, securedts \leftarrow \text{False}, seen \leftarrow \emptyset, srvRelay \leftarrow 0$
7: $\text{Counter}[i] \leftarrow 0$ for $i \in R \cup \{w\}, \text{Relays}[i] \leftarrow 0$ for $i \in R$

8: Upon receive($\langle \text{writeRequest, } ts', v', vp', w, wcounter \rangle$)
9: if (Counter[$w$] < wcounter) then
10: $\text{Counter}[$w$] \leftarrow wcounter
11: if ($ts < ts'$) then
12: $\langle ts, v, vp \rangle \leftarrow \langle ts', v', vp' \rangle$
13: seen $\leftarrow \{w\}, \text{securedts} \leftarrow \text{False}$
14: else
15: seen $\leftarrow$ seen $\cup \{w\}$
16: send(($\text{writeAck}, wcounter, s_j$)) to $w$

17: Upon receive($\langle \text{readRequest, } ts', v', vp', r, rcounter \rangle$)
18: if (Counter[$r$] < rcounter) then
19: $\text{Counter}[$r$] \leftarrow rcounter
20: if ($ts < ts'$) then
21: $\langle ts, v, vp \rangle \leftarrow \langle ts', v', vp' \rangle$
22: seen $\leftarrow \{r\}, \text{securedts} \leftarrow \text{False}$
23: else
24: seen $\leftarrow$ seen $\cup \{r\}$
25: if ($\text{seen} > \frac{|S|}{2} - 2$) $\land$ (securedts = False) $\land$ (Relays[$r$] < $ts$) then
26: scounter $\leftarrow$ scounter + 1
27: Relays[$r$] $\leftarrow$ $ts$
28: srvRelay $\leftarrow$ 0
29: send(($\text{readRelay, } ts, v, vp, r, s_j, rcounter, scounter$)) to $S$
30: else
31: send(($\text{readAck, } ts, v, vp, \text{seen}, \text{rcounter, securedts}$)) to $r$

32: Upon receive($\langle \text{readRelay, } ts', v', vp', r, s, cl, c2 \rangle$)
33: if (Counter[$s$] < $c2$) then
34: Counter[$s$] $\leftarrow c2$
35: if ($ts > ts'$) then
36: $\langle ts, v, vp \rangle \leftarrow \langle ts', v', vp' \rangle$
37: seen $\leftarrow \{r\}$
38: else if ($ts = ts'$) then
39: seen $\leftarrow$ seen $\cup \{r\}$
40: if (Relays[$r$] = $ts'$) then
41: srvRelay $\leftarrow$ srvRelay $\cup \{s\}$
42: if ($|\text{srvRelay}| = |S| - f$) then
43: if ($ts = ts'$) then
44: securedts $\leftarrow$ True
45: send(($\text{readAck, } ts', v', vp', 0, cl, securedts$)) to $r$
46: else
47: scounter $\leftarrow$ scounter + 1
48: send(($\text{readRelay, } ts', v', vp'$), $r, s_j, cl, scounter$)) to $s$
than its local timestamp, it also sets `securedts` flag to `False` (A12:L21-22). If not, then then server \( s_j \) adds the sender to the seen set (A12:L24). Next, \( s_j \) must decide whether to relay the received timestamp or not. In particular, \( s_j \) relays a timestamp to all the servers (A12:L29) if:

(i) it sent this timestamp to more than \( \frac{|S|}{f} - 2 \) reader processes, (ii) the timestamp has not already being relayed (i.e., `securedts = False`) and (iii) the server has not yet relayed this timestamp for the same reader (A12:L25). Otherwise, if any of these conditions does not hold then \( s \) just replies to the sender with its local timestamp (A12:L31). In a `readRelay` message \( s_j \) includes its local replica state, the id of the reader that initiated the relay, and its own id.

(2) Upon receiving a \( \langle \text{readRelay}, ts', v', vp', r, s, c1, c2 \rangle \) message from server \( s \) a server \( s_j \) first checks if the attached timestamp is strictly greater than its local one. If that holds, then \( s_j \) updates its local timestamp and value to the ones collected and resets the `seen` set to include only the requesting reader \( r \). Otherwise, \( s_j \) adds the requesting reader in the `seen` set without resetting it (A12:L35-39). Then \( s_j \) checks if it also sent a relay with the same timestamp for the same reader (A12:L40). If this holds, server \( s_j \) adds sender \( s \) in the servers that received its relay (A12:L41). When the server receives \( |S| - f \) relays, then it sends a `readAck` message to the reader that initiated the relay along with the timestamp that it initially relayed (not its local timestamp). Lastly, if its local timestamp is the same as the relayed timestamp, then \( s_j \) also sets `securedts = True` (A12:L42-45). In the case where the server did not sent a relay with the same timestamp for the same reader, then the server sends the `readRelay` message back to sender \( s \) and completes (A12:L48).

(3) Upon receiving a \( \langle \text{writeRequest}, ts', v', vp', w, wccounter \rangle \) message the server updates its local replica state and `seen` set appropriately. In case the timestamp in the request is greater
than its local timestamp it also sets `securedts` flag to `False` (A12:L11-15). It then acknowledges the requesting writer with a `writeAck` message (A12:L16).

### 5.2.2 Correctness of O\text{HFAST}

We first show *liveness* (termination) and then *atomicity* (safety).

**Liveness.** Termination holds with respect to our failure model: $|S| - f$ servers do not fail and each operation waits for no more than $|S| - f$ messages for completion. We now give additional details.

**Write Operation.** Per algorithm O\text{HFAST}, writer $w$ creates a `writeRequest` message and then it broadcasts it to all servers in exchange $E1$ (A11:L37). Writer $w$ then waits for `writeAck` messages from $|S| - f$ servers from $E2$ (A11:L38). According to our failure model $|S| - f$ servers do not fail and can receive `writeRequest` and send `writeAck` messages to the requesting writer, thus a write operation $\omega$ terminates.

**Read Operation.** Each operation $\rho$ sends `readRequest` messages to all the servers (A11:L11) and waits for $|S| - f$ replies before terminating (A11:L12). Thus termination of such process is prevented if less than $|S| - f$ servers reply to $r$ for operation $\rho$.

When a server receives a `readRequest` for a read operation it may perform one of two actions: (i) replies to the requesting reader with a `readAck` message that includes its local timestamp-value pair, or (ii) sends `readRelay` messages to other servers and replies to the reader with a `readAck` message when it collects $|S| - f$ relays that contain its local-timestamp. Thus, a read operation terminates if a correct server is guaranteed to send a `readAck` message to the reader in both cases. Notice that when a server $s'$ receives a `readRelay` message from $s$ with a timestamp $ts$ it either, (a) sends a `readRelay` to $s$ (A12:L45), or (b) appends its local `srvRelay`
set with the sender if Relays[r] = ts (A12:L41). In (a) it is clear that s′ replies to s with a readRelay that contains ts. However it is not clear if s′ sends a readRelay message to s in (b). Notice that (b) is only possible if Relays[r] = ts, where ts the timestamp enclosed in the readRelay message. Server s′ sets Relays[r] = ts only when it sends readRelay messages for r for timestamp ts to all the servers (A12:L29). So in that line s′ sends readRelay message to s as well. Therefore, in any case (a) or (b), a readRelay message is sent by s′ to s with timestamp ts. So s eventually receives |S| − f readRelay messages that contain ts and thus the check in A12:L42 is satisfied and replies with a readAck message to the read operation. Thus, the reader collects a readAck message from a server in both cases (i) and (ii). Hence, the reader receives at least |S| − f readAck messages and the read operation ρ terminates.

Atomicity. We use the association between the timestamps and the partial order as given in Section 4.2.2. We now state and prove the following lemmas.

Monotonicity allows the ordering of the values according to their associated timestamps. So Lemma 5.2.2 shows that the ts variable maintained by each server process in the system is monotonically increasing. Let us first make the following observation:

Lemma 5.2.1 In any execution ξ of OHFAST, if a server s replies with a timestamp ts at time T, then s replies with a timestamp ts′ ≥ ts at any time T′ > T.

Proof. A server attaches in each reply its local timestamp. Its local timestamp in turn is updated only whenever the server receives a higher timestamp So the server local timestamp is monotonically non-decreasing and the lemma follows. □

The following is also true for a server process.
Lemma 5.2.2 In any execution $\xi$ of OHFAST, if a server $s$ receives a timestamp $ts$ at time $T$ from a process $p$, then $s$ replies with a timestamp $ts' \geq ts$ at any time $T' > T$.

**Proof.** If the local timestamp of the server $s$, $ts_s$, is smaller than $ts$, then server updates to $ts_s = ts$. Otherwise, not updates take place and it remains $ts_s \geq ts$. In any case $s$ replies with a timestamp $ts_s \geq ts$ to $\pi$. By Lemma 5.2.1 the server $s$ attaches a timestamp $ts' \geq ts_s$, and hence $ts' \geq ts$ to any subsequent reply. $\square$

Now, we want to show that when a server receives a readRelay message that contains a timestamp $ts$ then it sends a timestamp $ts_s \geq ts$ from that point onward.

Lemma 5.2.3 In any execution $\xi$ of OHFAST, if a server $s$ receives a readRelay message with a timestamp $ts$ at time $T$ from a server $s'$, then $s$ attaches a timestamp $ts' \geq ts$ to any message it sends at any time $T' > T$.

**Proof.** If the local timestamp of the server $s$, $ts_s$, is smaller than $ts$ when it receives a readRelay message, then $s$ updates $ts_s = ts$ (A12:L36). Otherwise $ts_s$ does not change and remains $ts_s \geq ts$. In any case $s$ replies with a timestamp $ts_s \geq ts$ to $s'$. By monotonicity of the timestamps, $s$ attaches a timestamp $ts' \geq ts_s$, and hence $ts' \geq ts$ to any subsequent message. $\square$

Next, we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

Lemma 5.2.4 In any execution $\xi$ of OHFAST, if a read $\rho$ from $r$ succeeds a write operation $\omega$ that writes timestamp $ts_\omega$ from the writer $w$, i.e. $\omega \rightarrow \rho$, and $\rho$ returns a timestamp $ts_\rho$, then $ts_\rho \geq ts_\omega$. 


**Proof.** There are two cases to investigate: (i) $\rho$ returns after examining the predicate, or (ii) $\rho$ returns because it received a secured timestamp.

**Case (i):** Per algorithm, the write operation $\omega$ communicates with a set of $|S_w| = |S| - f$ servers before completing. Let $|S_1| = |S| - f$ be the number of servers that replied to the read operation $\rho$. The intersection of the two sets is $|S_w \cap S_1| \geq |S| - 2f$ and since $f < |S|/2$ there exists at least a single server $s$ that replied to both operations. Each server $s \in S_w \cap S_1$ replies to $\omega$ before replying to $\rho$. Thus, by Lemma 5.2.2 and since $s$ receives the message from $\omega$ before replying to any of the two operations, then it replies to $\rho$ with a timestamp $ts_s \geq ts$.

Thus there are two cases to investigate on the timestamp: (a) $ts_s > ts$, and (b) $ts_s = ts$.

We now examine sub case (a). In the case where $ts_s > ts$, $\rho$ will observe a maximum timestamp $maxTS \geq ts_s$. Since $\rho$ returns either $ts' = maxTS$ or $ts' = maxTS - 1$, then $ts' \geq ts_s - 1$. Thus, $ts' \geq ts$ as desired.

We now examine sub case (b). In this case all the servers in $S_w \cap S_1$ reply with a timestamp $ts_s = ts$. The read $\rho$ may observe a maximum timestamp $maxTS \geq ts_s$. If $maxTS > ts_s$, then, with similar reasoning as in Case 1, we can show that $\rho$ returns $ts' \geq ts$. So it remains to investigate the case where $maxTS = ts_s = ts$. In this case, at least $|S_w \cap S_1| = |S| - 2f$ servers replied with $maxTS$ to $\rho$. Also for each $s \in S_w \cap S_1$, $s$ included both the writer identifier $w$ and $r_1$ before replying to $\omega$ and $\rho_2$ respectively. So $s$ replied with a size at least $s.views \geq 2$ to $\rho_2$. Thus, given that $|R| \geq 2$, the predicate holds for $\alpha = 2$ and the set $S_w \cap S_1$ for $\rho$, and hence it returns a timestamp $ts' = ts$.

**Case (ii):** In this case $\rho$ received a message that contained a timestamp $ts_s = maxTS$ and a secured flag equal to True. According to the algorithm $\rho$ returns $ts_\rho = maxTS$. Since $|S| - f$
servers received $\omega$, and since $\rho$ contacts $|S| - f$ servers during its first phase, with $f < \frac{|S|}{2}$, then there is at least a single server, say $s$, that received the message for $\omega$ before replying to $\rho$. According to Lemmas 5.2.2 and 5.2.3, $s$ replies to $\rho$ with a timestamp $t_s \geq t_s\omega$, the timestamp it received from $\omega$. Thus, $\rho$ observes a $\max TS \geq t_s \geq t_s\omega$, and hence returns $ts_\rho = \max TS \geq t_s\omega$. \hfill \Box

Next, we prove a lemma showing that if a timestamp $ts$ is secured from a server $s$, then at least $|S| - f$ servers have a timestamp $ts' > ts$.

**Lemma 5.2.5** In any execution $\xi$ of OHFAST, if a server $s$ sets $secured ts = True$ for a timestamp $ts$ at time $T$ then $\exists S' \subseteq S$ at $T$, s.t. $|S'| \geq |S| - f$ and $\forall s' \in S'$, the local timestamp of $s'$ is $ts' \geq ts$.

**Proof.** This lemma follows from the way that a relay round is implemented by a server. In particular, when a server $s$ relays a timestamp $ts$, it sends a readRelay message to all the servers. Each server $srvr'$ that receives such a relay replies with a timestamp $ts' = ts$. Before replying, $s'$ either sets its timestamp to $ts$ or has a larger timestamp. So when $s$ sets $secured ts = True$ has received a set $|S'| \geq |S| - f$ of replies, and every server $s' \in S'$ has a timestamp $ts' \geq ts$, by Lemma 5.2.3. Thus the lemma follows. \hfill \Box

Next, we show that if a read operation $\rho_2$ succeeds read operation $\rho_1$, then $\rho_2$ always returns a value at least as recent as the one returned by $\rho_1$.

**Lemma 5.2.6** In any execution $\xi$ of OHFAST, if $\rho_1$ and $\rho_2$ are two read operations such that $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns $ts_{\rho_1}$, then $\rho_2$ returns $ts_{\rho_2} \geq ts_{\rho_1}$. 
Proof. A read operation may decide on the value to return in two ways in OHFAST: (i) it receives a secured timestamp, or (ii) it evaluates the predicate. Let us first examine what happens when the two reads are invoked by the same reader (i.e. \( r_1 = r_2 \)). During \( \rho_2 \), \( r_1 \) includes a timestamp \( ts_{r_1} \geq ts_{\rho_1} \) in every message it sends to servers. According to Lemma 5.2.2 every server \( s \) replies with a timestamp \( ts_s \geq ts_{\rho_1} \). Thus, \( maxTS_2 \geq ts_{\rho_1} \). If \( maxTS_2 > ts_{\rho_1} \) then since \( ts_{\rho_2} = maxTS_2 \) or \( ts_{\rho_2} = maxTS_2 - 1 \) it follows that \( ts_{\rho_2} \geq ts_{\rho_1} \) in either case. If \( maxTS_2 = ts_{\rho_1} \) then every server adds \( r_1 \) in their \( seen \) set before replying to \( \rho_2 \). So the predicate is valid for \(|MS| \geq |S| - f\) and \( \alpha = 1 \). Hence, \( \rho_2 \) returns \( ts_{\rho_2} = maxTS_2 = ts_{\rho_1} \) in any case (i) or (ii).

So we need now to examine all the possible combinations for the two reads \( \rho_1 \) and \( \rho_2 \) when \( r_1 \neq r_2 \). We examine the following four cases: (1) \( \rho_1 \) evaluates the predicate, and \( \rho_2 \) receives a secured \( maxTS_2 \), (2) \( \rho_1 \) receives a secured \( maxTS_1 \), and \( \rho_2 \) evaluates the predicate, (3) \( \rho_1 \) receives a secured \( maxTS_1 \), and \( \rho_2 \) receives a secured \( maxTS_2 \), and (4) both \( \rho_1 \) and \( \rho_2 \) evaluate the predicate.

Case 1: In this case, \( \rho_1 \) evaluates the predicate, and \( \rho_2 \) returns \( ts_{\rho_2} = maxTS_2 \) as it received a reply with \( maxTS_2 \) and \( secured = True \). There are two subcases to examine: (a) \( \rho_1 \) returns \( maxTS_1 \), and (b) \( \rho_1 \) returns \( maxTS_1 - 1 \).

Case 1a: If \( \rho_1 \) returns \( maxTS_1 \) it follows that the predicate is valid for \( \rho_1 \). Hence:

\[
\exists \alpha \in [1, \left\lceil \frac{|S|}{f} \right\rceil - 2] \text{ and } MS \subseteq S \text{ s.t. } (1) \]

\[
MS = \{ s : s.ts = maxTS_1 \land s.views \geq \alpha \} \land |MS| \geq |S| - \alpha f \quad (2)
\]
Moreover, since $\rho_1$ examines the predicate, then none of the servers that replied with $\text{maxTS}_1$ sends secured = True. Therefore, $\forall s \in MS$, it must be true that $s.\text{views} \leq \frac{S}{f} - 2$ before replying to $\rho_1$ (A12:L24), otherwise $s$ would proceed to relay and secure $\text{maxTS}_1$. Since every $s.\text{views} \leq \frac{S}{f} - 2$, then it must be the case that $\alpha \leq \frac{S}{f} - 2$ as well. Thus substituting:

$$|MS| \geq |S| - \alpha f \Rightarrow |MS| \geq |S| - (\frac{S}{f} - 2)f \Rightarrow |MS| > f$$

Since $\rho_2$ receives replies from $|S_2| = |S| - f$ servers then $S_2 \cap MS \neq \emptyset$. Also notice that since $\rho_1 \rightarrow \rho_2$, then a server $s \in S_2 \cap MS$ replies to $\rho_1$ with $\text{maxTS}_1$ before replying to $\rho_2$. By Lemma 5.2.2, $s$ replies to $\rho_2$ with a timestamp $ts_s \geq \text{maxTS}_1$. Thus, $\text{maxTS}_2 \geq ts_s \Rightarrow \text{maxTS}_2 \geq \text{maxTS}_1$ and $\rho_2$ returns $ts_{\rho_2} \geq \text{maxTS}_1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$.

**Case 1b:** Assume now the case where $\rho_1$ returns $\text{maxTS}_1 - 1$. Since $\rho_1$ received $\text{maxTS}_1$, and since the sole writer invokes one operation at a time, then it follows that the write operation that wrote $\text{maxTS}_1 - 1$, say $\omega$, completed during or before $\rho_1$. Since though $\rho_1 \rightarrow \rho_2$, then it follows that $\omega \rightarrow \rho_2$. Since $\omega$ communicates with $|S| - f$ servers before completing, and since $\rho_2$ waits for $|S| - f$ replies, then there is a server $s$ that replies to $\omega$ before replying to $\rho_2$. By Lemma 5.2.2, $s$ replies with a timestamp $ts_s \geq \text{maxTS}_1 - 1$ to $\rho_2$. Thus $\rho_2$ observes a $\text{maxTS}_2 \geq \text{maxTS}_1 - 1$, and hence $ts_{\rho_2} \geq \text{maxTS}_1 - 1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$ in this case as well.

**Case 2:** Here, $\rho_1$ returns $ts_{\rho_1} = \text{maxTS}_1$ as it received a message that contained $\text{maxTS}_1$ and secured = True. Read $\rho_2$ evaluates the predicate to decide on the value to return. We have two subcases to examine again: (a) $\rho_2$ returns $\text{maxTS}_2$, or (b) $\rho_2$ returns $\text{maxTS}_2 - 1$. Since $\rho_1$ returned a secured timestamp, then it received $\text{maxTS}_1$ and secured = True from some server $s$. By Lemma 5.2.5, a set $|S'| \geq |S| - f$ of servers have a timestamp $ts' \geq \text{maxTS}_1$ before $s$ replies to $\rho_1$. Since $\rho_2$ receives replies from $|S_2| = |S| - f$ servers, then $S' \cap S_2 \neq \emptyset$. 133
Then by Lemmas 5.2.2 and 5.2.3, any server in $s' \in S' \cap S_2$ replies to $\rho_2$ with a timestamp $ts_{s'} \geq maxTS_1$. Thus, $\rho_2$ observes a $maxTS_2 \geq maxTS_1$. If $maxTS_2 > maxTS_1$ and since $\rho_2$ returns either $maxTS_2$ or $maxTS_2 - 1$, then in either case $ts_{\rho_2} \geq ts_{\rho_1}$.

So it remains to examine what happens when $maxTS_2 = maxTS_1$. If $\rho_2$ returns $ts_{\rho_2} = maxTS_2$ then $ts_{\rho_2} \geq ts_{\rho_1}$. Let us examine now if $\rho_2$ may return $maxTS_2 - 1$. As we said before every server $s'$ in $S' \cap S_2$ replies with $ts_{s'} \geq maxTS_1$ to $\rho_2$. Since $|S'| \geq |S| - f$ and $|S_2| \geq |S| - f$ then $|S' \cap S_2| \geq |S| - 2f$. Also by the algorithm, every server in $S'$ adds $r_1$ in its seen set before replying to the relay message from $s$ (A12:L42). Furthermore, every server in $S_2$ adds $r_2$ in its seen set before replying to $\rho_2$. So every server $s' \in S' \cap S_2$ replies with a $s'.views \geq 2$. Thus, the predicate holds for at least $|MS| = |S' \cap S_2| \geq |S| - 2f$ and $\alpha = 2$. Hence $\rho_2$ will return $maxTS_2$ contradicting our assumption that returns $maxTS_2 - 1$. So returning $maxTS_2 - 1$ is not possible.

**Case 3:** In this case both $\rho_1$ and $\rho_2$ return a secured timestamp. Let $s_1$ be the server that send $maxTS_1$ and secured = True to $\rho_1$, and $s_2$ (not necessarily different than $s_1$) be the server that sent $maxTS_2$ and secured = True to $\rho_2$. By Lemma 5.2.5, there exists a set $S'$ s.t. every server $s \in S'$ has a timestamp $ts_s \geq maxTS_1$ before $s_1$ replies to $\rho_1$. As explained in Case 2, $S' \cap S_2 \neq \emptyset$. Hence there exists a server that replied both to the relay message of $s_1$ and to $\rho_2$. By Lemma 5.2.3, each server $s' \in S' \cap S_2$ replies to $\rho_2$ with a timestamp $ts_{s'} \geq maxTS_1$. Hence, $maxTS_2 \geq maxTS_1$. Since $\rho_2$ returns a secured timestamp, then it returns $maxTS_2$. Therefore, $ts_{\rho_2} = maxTS_2 \Rightarrow ts_{\rho_2} \geq maxTS_1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$.
Case 4: In this case both $\rho_1$ and $\rho_2$ evaluate the predicate. Let $\text{maxTS}_1$ be the maximum timestamp discovered by $ts_1$. We have two subcases to consider: (a) $\rho_1$ returns $ts_1 = \text{maxTS}_1 - 1$, or (b) $\rho_1$ returns $ts_1 = \text{maxTS}_1$.

Case 4(a): In this case $\rho_1$ returns $ts_1 = \text{maxTS}_1 - 1$. It follows that there is a server $s$ that replied to $\rho_1$ with a timestamp $\text{maxTS}_1$. This means that the writer invoked the write operation that tries to write a value with timestamp $\text{maxTS}_1$. Since the single writer invokes a single operation at a time (by well-formedness), it must be the case that the writer completed writing timestamp $\text{maxTS}_1 - 1$ before the completion of $\rho_1$. Let that write operation be $\omega$.

Since, $\rho_1 \rightarrow \rho_2$, then it must be the case that $\omega \rightarrow \rho_2$ as well. So by Lemma 5.2.4, $\rho_2$ returns a timestamp $ts_2$ greater or equal to the timestamp written by $\omega$, and thus $ts_2 \geq \text{maxTS}_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Case 4(b): This is the case where $\rho_1$ returns $ts_1 = \text{maxTS}_1$. So it follows that the predicate is satisfied for $\rho_1$, and hence $\exists \alpha \in [1, \ldots, |\mathcal{R}|]$ and a set of servers $M_1$ such that every server $s \in M_1$ replied with the maximum timestamp $\text{maxTS}_1$ and a seen set size $s.\text{views} \geq \alpha$, and $|M_1| \geq |S| - \alpha f$. We know that $\rho_2$ receives replies from a set of servers $|S_2| = |S| - f$ before completing. Let $M_2$ be the set of servers that replied to $\rho_2$ with a timestamp $\text{maxTS}_2$. Since $|\mathcal{R}| < \frac{|S|}{f} - 2$, then

$$|M_1| > |S| - (\frac{|S|}{f} - 2)f \Rightarrow |M_1| > f$$

Hence, $S_2 \cap M_1 \neq \emptyset$ and by Lemma 5.2.2 every server $s \in S_2 \cap M_1$ replies to $\rho_2$ with a timestamp $ts_s \geq \text{maxTS}_1$. Therefore $\text{maxTS}_2 \geq \text{maxTS}_1$. If $\text{maxTS}_2 > \text{maxTS}_1$, then $\rho_2$ returns a timestamp $ts_2 \geq \text{maxTS}_2 - 1 \Rightarrow ts_2 \geq \text{maxTS}_1$ and hence $ts_2 \geq ts_1$. 

135
It remains to investigate the case where \( \text{max}T S_2 = \text{max}T S_1 \). Notice that any server in \( s \in S_2 \cap M_1 \) is also in \( M_2 \). Since \( \rho_2 \) may skip \( f \) servers that reply to \( \rho_1 \), then \( |M_1 \cap M_2| \geq |S| - (a + 1)f \). Recall that for each server \( s \in M_1 \cap M_2 \), \( s \) replied with a size \( s.\text{views} \geq a \) to \( \rho_1 \). Also \( s \) adds \( r_2 \) in its seen set before replying to \( \rho_2 \). So there are two subcases to examine: (i) either \( r_2 \) was already in the seen set of \( s \), or (ii) \( r_2 \) was not a member of \( s.\text{seen} \).

**Case 4(b)(i):** If \( r_2 \) was already a part of the seen set of \( s \), then the size of the set remains the same. It also means that \( r_2 \) obtained \( \text{max}T S_1 \) from \( s \) in a previous read operation, say \( \rho'_2 \) from \( r_2 \). Since each process satisfies well-formedness, it must be the case that \( r_2 \) completed \( \rho'_2 \) before invoking \( \rho_2 \). All the messages sent by \( \rho_2 \) contained \( \text{max}T S_1 \). So by Lemma 5.2.2 any server \( s \in S_2 \) replies to \( r_2 \) with a timestamp \( t s_s = \text{max}T S_2 = \text{max}T S_1 \). In this case \( |S| - f \) servers replied with \( \text{max}T S_2 \) and their seen set contains at least \( r_2 \), having \( s.\text{views} \geq 1 \). Thus, the predicate is valid with \( \alpha = 1 \) for \( \rho_2 \) which returns \( t s_2 = \text{max}T S_2 = \text{max}T S_1 = t s_1 \).

**Case 4(b)(ii):** This case may arise if \( r_2 \) is not part of the seen set of every server \( s \in M_1 \cap M_2 \). If \( r_2 \) is part of the seen set of some server \( s' \in M_1 \cap M_2 \), then this is resolved by case 2(a). So each server \( s \in M_1 \cap M_2 \) inserts \( r_2 \) in their seen sets before replying to \( \rho_2 \). So if the size of the set \( s.\text{views} = \alpha \) when \( s \) replied to \( \rho_1 \), \( s \) includes a size \( s.\text{views} \geq a + 1 \) when replying to \( \rho_2 \). Notice here that if \( \alpha = |\mathcal{R}| + 1 \) for \( \rho_1 \), then it means that \( r_2 \) was already part of the seen set of \( s \) when \( s \) replied to \( \rho_1 \). This case is similar to 2(a). So we assume that \( \alpha < |\mathcal{R}| + 1 \), in which case \( \alpha + 1 \leq |\mathcal{R}| + 1 \). Since every server \( s \in M_1 \cap M_2 \) replies with \( s.\text{views} \geq \alpha + 1 \) to \( \rho_2 \) and since \( |M_1 \cap M_2| \geq |S| - (a + 1)f \), then the predicate holds for \( \alpha + 1 \leq |\mathcal{R}| + 1 \) and the set \( MS = M_1 \cap M_2 \) for \( \rho_2 \), and thus \( \rho_2 \) returns \( t s_2 = \text{max}T S_2 = \text{max}T S_1 = t s_1 \) in this case as well. And this completes the proof.

We now show the correctness of algorithm OHFAST.
**Theorem 5.2.7** Algorithm OHFAST implements an atomic SWMR object.

**Proof.** We now use the lemmas stated above and the operations order definition to reason about each of the three *atomicity* conditions A1, A2 and A3 as given in Definition 2.2.5.

**A1** For any \( \pi_1, \pi_2 \in \Pi \) such that \( \pi_1 \to \pi_2 \), it cannot be that \( \pi_2 \prec \pi_1 \).

When the two operations are reads and \( \pi_1 \to \pi_2 \) holds, then from Lemma 5.2.6 it follows that the timestamp of \( \pi_2 \) is no less than the one of \( \pi_1 \), i.e. \( ts_2 \geq ts_1 \). If \( ts_2 > ts_1 \), then by the ordering definition \( \pi_1 \prec \pi_2 \) is satisfied. When \( ts_2 = ts_1 \) then the ordering is not defined, thus it cannot be the case that \( \pi_2 \prec \pi_1 \). If \( \pi_2 \) is a write, the sole writer generates a new timestamp by incrementing the largest timestamp in the system. By well-formedness, any timestamp generated in any write operation that precedes \( \pi_2 \) must be smaller than \( ts_2 \). Since \( \pi_1 \to \pi_2 \), then it holds that \( ts_1 < ts_2 \). Hence, by the ordering definition it cannot be the case that \( \pi_2 \prec \pi_1 \). Lastly, when \( \pi_2 \) is a read and \( \pi_1 \) a write, then by Lemma 5.2.4 it follows that \( ts_2 \geq ts_1 \). By the ordering definition, it cannot hold that \( \pi_2 \prec \pi_1 \) in this case either.

**A2** For any write \( \omega \in \Pi \) and any operation \( \pi \in \Pi \), then either \( \omega \prec \pi \) or \( \pi \prec \omega \).

If the timestamp returned from \( \omega \) is greater than the one returned from \( \pi \), i.e. \( ts_\omega > ts_\pi \), then \( \pi \prec \omega \) follows directly. Similarly, if \( ts_\omega < ts_\pi \) holds, then \( \omega \prec \pi \) follows. If \( ts_\omega = ts_\pi \), then it must be that \( \pi \) is a read and either (i) \( \rho \) discovered \( ts_\omega \) from a set of messages that contained \( ts_\omega \) as the maximum timestamp, i.e., \( ts_\omega = maxTS \), and it was propagated to a set of servers \( (maxTS = ts_\omega = secured) \), or (ii) discovered \( ts_\omega \) from a set of servers and the predicate is satisfied, or (iii) \( \pi \) discovered \( ts_\omega + 1 \) but the predicate is not satisfied. Thus, \( \omega \prec \pi \) follows.

**A3** Every read operation returns the value of the last write preceding it according to \( \prec \) (or the initial value if there is no such write).
Let $\omega$ be the last write preceding read $\rho$. From our definition it follows that $ts_\rho \geq ts_\omega$. If $ts_\rho = ts_\omega$, then $\rho$ either: (i) $\rho$ discovered $ts_\omega$ from a set of messages that contained $ts_\omega$ as the maximum timestamp, i.e., $ts_\omega = \max TS$, and it was propagated to a set of servers ($\max TS = ts_\omega = \text{secured}$), or (ii) discovered $ts_\omega$ as the maximum timestamp from some servers and their replies satisfied the predicate, or (iii) discovered the value written by some write $\omega'$ with timestamp $ts_\omega + 1$ but the replies received did not satisfy the predicate. If case (i) holds, $\omega$ is the last preceding write since $\rho$ discovered $ts_\omega$ as the maximum timestamp, $ts_\omega = \max TS$ and it was propagated to a set of servers and $\rho$ returns $ts_\omega$ without any further actions. When case (ii) holds, then it is clear that $\omega$ is the last preceding write. If (iii) holds then by Lemma 5.2.4, and since $ts_\rho = ts_\omega$, it must be the case that $\rho$ is concurrent with $\omega'$ and hence $\omega$ is again the last preceding write. If $ts_\rho > ts_\omega$, then it means that $\rho$ obtained a larger timestamp. However, the larger timestamp can only be originating from a write that succeeds $\omega$, thus $\omega$ is not the preceding write and this cannot be the case. Lastly, if $ts_\rho = 0$ as the maximum timestamp, then the predicate holds for $\alpha = 1$ and thus $ts_\rho \geq 0$, returning in the worst case the initial value.

5.2.3 Performance of OHFAST

**Performance.** We now assess the performance of OHFAST in terms of (i) latency of read and write operations as measured by the number of communication exchanges, (ii) the message complexity of read and write operations and (iii) computational complexity.

In brief, for algorithm OHFAST write operations take 2 communication exchanges and read operations take 3 communication exchanges. The (worst case) message complexity of
read operations is $|S|^2 + 2|S|$ and the (worst case) message complexity of write operations is $2|S|$. This follows directly from the structure of the algorithm. We now give additional details.

Operation Latency. We study the operation latency, in terms of the number of communication exchanges required by each operation.

Write operation latency: Per algorithm OHFAST, writer $w$ sends a writeRequest message to all the servers in exchange $E_1$, and, awaits for writeAck messages from at least a majority of servers in exchange $E_2$. Once the writeAck messages are received, no further communication is required and the write operation terminates. Therefore, any write operation consists of 2 communication exchanges.

Read operation latency: A reader sends a readRequest message to all the servers in the first communication exchange $E_1$. Once a server receives a readRequest message, it either broadcasts a readRelay message to all the servers or reply back with a readAck message to the requesting reader forming exchange $E_2$. An active server that receives a readRelay message it will either bounce a readRelay message back to the server sender or reply back to the requesting reader with a readAck message. Both cases form communication exchange $E_3$. Thus, a read operation consists of 3 communication exchanges.

Message Complexity. We measure operation message complexity as the worst case number of exchanged messages in each read and write operation. The worst case number of messages corresponds to failure-free executions where all participants send messages to all destinations according to the protocols.

Write operation: A single write operation in algorithm OHFAST takes 2 communication exchanges. In the first exchange $E_1$, the writer sends a writeRequest message to all the servers
in $\mathcal{S}$. The second exchange $E_2$, occurs when all servers in $\mathcal{S}$ send a writeAck message to the writer. Thus, at most $2|\mathcal{S}|$ messages are exchanged in a write operation.

Read operation: Read operations take 3 communication exchanges. Exchange $E_1$ occurs when a reader sends a readRequest message to all the servers in $\mathcal{S}$. Exchange $E_2$ occurs when servers in $\mathcal{S}$ send a readRelay message to all the servers in $\mathcal{S}$. The last exchange, 3, occurs when servers in $\mathcal{S}$ send a readAck message to the requesting reader. Therefore, $|\mathcal{S}|^2 + 2|\mathcal{S}|$ messages are exchanged during a read operation.

Computational Complexity. Algorithm OHFAST uses the polynomial-time predicate to decide the communication latency of a read operation as reasoned for algorithm CCHYBRID in section 5.1.3.

5.3 Experimental Evaluation

In order to answer Research Questions 5.2 and 5.4 we did a comparative study of our algorithms by simulating them using the NS3 discrete event simulator [4]. The approach is similar to the one presented in Section 4.5 with minor modifications. In particular, we present empirical results that we obtained by implementing algorithms ABD [10], OHSAM (Section 4.2), SF [33], CCHYBRID, and OHFAST. We now give additional details.

Experimentation Platform. The general testbed of our experiments consists of a single writer, a set of readers, and a set servers. We assume that $f = 1$ servers may fail. This assumption was chosen to subject the system to high communication traffic, since every operation would wait for all but one servers to reply (ironically, crashes reduce network traffic). Communication between the nodes is established via point to point bidirectional links implemented with a DropTail queue. For the purpose of the experimental evaluation, we developed simulations
representing two different topologies, Series and Star, which mainly differ on the deployment of server nodes.

Figure 4 presents the two topologies. In both topologies the clients are divided evenly and are connected on a series of router nodes. Clients are connected to the routers with 5Mbps links and 2ms delay, and routers are connected with 10Mbps links and 4ms delay. In the Series topology, Fig.4(a), a server is connected to each router with 10Mbps bandwidth and 2ms delay. This topology demonstrates a network where servers are separated and appear to be in different networks. In the Star topology, Fig.4(b), all the servers are connected to a single router with 50Mbps links and 2ms delay, modeling a network where servers are in close proximity and well-connected, e.g., a datacenter. Clients are located uniformly with respect to the routers. We ran NS3 on a Macintosh machine running OS X El Capitan, with 2.5Ghz Intel Core i7 processor and 16GB of RAM. The average of 5 samples per scenario provided the stated operation latencies.

**Performance.** The performance of the algorithms is measured in terms of the ratio of the number of fast over slow R/W operations - *communication burden*; and the total time it takes for an operation to complete - *operation latency*. Operation latency is affected by both communication and computation latencies. For operation latency we combine two clocks: the simulation clock to measure communication delays, and a real time clock for computation delays. The sum of the two yields latency.

**Scenarios.** Measurements of the performance involves multiple execution scenarios. The scenarios were designed to test (i) the scalability of the algorithms as the number of readers and servers increases; (ii) the contention effect on efficiency, by running different concurrency scenarios; and (iii) the relation of the efficiency with the topology of the network that we use. To
test scalability we range the number of readers $|\mathcal{R}| \in [10, 20, 40, 80, 100]$ and the number of servers $|\mathcal{S}| \in [10, 15, 20, 25, 30]$. To test contention we specify the frequency of read operation and we run our algorithm for different read intervals ($rInt \in [2.3, 4.6, 6.9]$ seconds). We issue write operations every 4 seconds. To test contention we define two invocation schemes: fixed and stochastic. In the fixed scheme all operations are scheduled periodically at a constant interval. In the stochastic scheme reads are scheduled randomly from the time intervals $[1...rInt]$. Finally, to test the effects of topology we run our algorithms using both the Series and Star topologies.

**Results.** We now present our experimentation results. Our discussion is accompanied by sample plots. In the next paragraphs we analyse how the read operation latency is affected by the
scenarios we discussed earlier. As a general observation, the proposed algorithms outperform all the other algorithms in most scenarios. In particular, it is clear that CHYBRID and OHFAST outperform algorithms ABD and OHSAM. In addition, the two algorithms appear to achieve similar operation latencies as SF. A closer examination reveals that in many scenarios SF does not perform any slow reads, whereas in the same executions both CHYBRID and OHFAST require some slow reads. The fact that the two algorithms perform the same as SF, despite the slow reads, demonstrates that the computation overhead of the two presented algorithms is much less than the computation needed by SF. Thus, in executions where SF will perform more slow operations, clearly this will result in even worse operation latencies. More in detail, we conclude to the following observations:

**Scalability:** The increasing number of readers and the servers have a negative impact on all the algorithms. Sample plots that present the average latency of read operations for this scenario appear in Fig. 5(b) and (c). In particular, the read latency for algorithms ABD and OHSAM increases dramatically even when few servers participate in the system, Fig. 5(b) and the latency becomes even higher when we double the participating servers as shown in Fig. 5(c). On the other hand, we notice algorithms SF, CHYBRID and OHFAST, with almost identical behavior, to perform better and be more efficient than ABD and OHSAM. The increase of the latency in read operations still exists as the number of participants grows but in lower levels.

**Contention:** Contention is generated by: (i) operation frequencies, and (ii) concurrency schemes. We observe that operation frequency affects the latency of the operations in the fix scheme where operations are invoked at a constant interval. This can be seen in Fig. 5(a) and (b). Algorithms ABD and OHSAM are not affected (as all of their reads are slow), but the multi-speed algorithms SF, CHYBRID and OHFAST, are affected negatively. This behavior is due to the
fact that these algorithms perform a slow read operation per write operation. When the read interval is close to the write interval, e.g., \( rInt = 4.6 \), most of the reads are concurrent to the write and thus more reads are slow Fig. 6(l). This is not observed when \( rInt = 2.3 \) (or \( rInt = 6.9 \)), see Fig. 6(k). The impact of the slow reads on the operation latency can be also seen in Figs. 6(h) and (i) where algorithms perform much better when the read interval is not close to the write interval, i.e., \( rInt = 6.9 \). Notice that the same behavior is not being observed when the stochastic scheme is used, as randomness prevents the operations to be invoked at exactly the same time, see Figs. 5(d) and (e). Hence, a slow read operation may complete before any read operations that return the same value are invoked. Therefore, according to the multi-speed algorithms, once a slow read is completed, any read operation that succeeds such a read will be fast. This results in a low percentage of slow reads, see Fig. 6(m).

Finally, when the operation frequency is constant, it appears that in the stochastic scheme each operation completes almost two times faster than in the fix scheme, as shown in Figs. 5(b) and 5(e). Algorithms, ABD and OhSAM, can be used as points of reference as they have the same computation and communication requirements in both fix and stochastic scenarios. The difference can be explained due to the congestion that the fix scheme introduces in the network. On the contrary, a stochastic scheme distributes the invocation time intervals of the read operations uniformly, reducing the network congestion, and hence operation latency.

*Topology:* Now, we are interested to examine what is the impact of the topology on our algorithms. Pair of plots 5(e)(f) and 6(g)(h) show that topology has an impact on the performance and the efficiency of all the algorithms. Most importantly, we can observe that OhSAM and OhFAST are the two algorithms that are affected the most. In particular, while in Fig. 5(e) OhSAM performs better than ABD and OhFAST performs similar to cCHYBRID and SF, we
notice that in Fig. 5(f) OHSAM performs worse than ABD and OHFAST worse than CCHY-BRID and SF. Same observation can be noticed in Figs. 6(g) and (h). This behavior is expected as both OHSAM and OHFAST need to exchange messages between the servers during a relay phase. The results show clearly that the algorithms using server-to-server communication perform better in a Star topology, where servers are well-connected using high bandwidth links. However, notice that OHFAST performs much better OHSAM since operation relays are not performed for every read operation.
Figure 5: SWMR Simulation Results Part I.
Figure 6: SWMR Simulation Results Part II.
Chapter 6

Quorum-based Atomic Implementations

We now address the gap between one-round and two-rounds algorithms in quorum-based implementations. In Section 4.2 we presented an algorithm for the SWMR model, called OHSAM where each read operation takes one-and-a-half-rounds before completion. This is achieved by introducing server-to-server communication in the system during the second communication exchange. Additionally, in Section 4.3 we showed how the three communication exchanges read protocol of algorithm OHSAM can be adapted in the MWMR setting, resulting to algorithm OHMAM. Both algorithmic solutions, OHSAM and OHMAM, do not impose any constraints on the reader participation in the system and they perform a modest amount of local computation, resulting in negligible computation overhead.

Furthermore, as discussed in Section 3.2.1, Georgiou et al. [32] introduced Quorum Views, client-side tools that examine the distribution of the latest value among the replicas in order to enable fast read operations under read and write operation concurrency. Authors devised an atomic SWMR implementation, called SLIQ, that utilizes quorum views and allows some read operations to be fast. Algorithm SLIQ needs insignificant computation effort in order to
examine the distribution of object values among the replica servers that a reader process receives during a read operation. The key idea is to try to determine the state of a write operation i.e., whether it is complete or incomplete, and terminate fast whenever possible. In addition, a later work of Georgiou et al. [30] generalized the client-side decision tools and presented an algorithm, called CWFr, that enables fast read operations for the MWMR setting.

Here, we examine whether we can combine the techniques presented in Section 4.1 and by Georgiou et al. in [32, 30] to obtain algorithms both for the SWMR and the MWMR setting that allow one and one-and-a-half round-trip operations without imposing any restriction on the participants in the service. In particular, this chapter aims to tackle these problems by examining the following research questions:

**Research Question 6.1** Can we devise an atomic read/write shared objects implementation for the asynchronous, crash-prone, message-passing, static SWMR setting with unbounded participation, such that all read operations take at most three exchanges to complete?

**Research Question 6.2** Under the same assumptions, can we devise an atomic read/write shared objects implementation for the static MWMR setting such that all read operations take either two or three exchanges to complete?

**Research Question 6.3** How the analytical results of the proposed algorithms are reflected in practical efficiency?

We give details regarding Quorum Views and the server-to-server communication scheme in Section 6.1. In Section 6.2 we present a SWMR algorithm called ERATO and in Section 6.3 we present algorithm ERATO-MW, developed for the MWMR setting. Lastly, in Section 6.4 we analyze results and observations we obtained through the different empirical study cases.
6.1 Incorporating Prior Techniques

Since we use quorum views as a design element in the algorithms that follow, as a first step we present the idea behind Quorum Views as presented by Georgiou et al. in [32].

Quorum views. A quorum view refers to the distribution of the highest timestamp among the servers, $maxTS$, that a read operation witnesses during a communication exchange, and can be used as a client-side tool to determine the state of a write operation i.e., whether it is complete or incomplete. Following [32], quorum views are defined as follows:

Definition 6.1.1 (Quorum Views [32]) Suppose that during an exchange, a read $\rho$ strictly receives timestamp and value pairs $(s.ts, v)$ from each server $s \in Q_i$. Under the assumption that servers always maintain the largest timestamp they receive, a reader process $\rho$ can distinguish three different quorum views that may reveal the state of the write operation that writes value $v$ associated with $maxTS$:

- $QV(1)$: $\forall s \in Q_i : s.ts = maxTS$

- $QV(2)$: $\forall Q_j \in Q, i \neq j, \exists A \subseteq Q_i \cap Q_j, s.t. A \neq \emptyset$ and $\forall s \in A : s.ts < maxTS$

- $QV(3)$: $\exists s' \in Q_i : s'.ts < maxTS$ and $\exists Q_j \in Q, i \neq j$ s.t. $\forall s \in Q_i \cap Q_j : s.ts = maxTS$

Figure 7: (a) $QV(1)$, (b) $QV(2)$, (c) $QV(3)$ incomplete write, (d) $QV(3)$ complete write
As an example, consider the quorum system \( \mathcal{Q} \) consisting of three quorums, \( \{Q_i, Q_j, Q_z\} \). Figure 7 illustrates the three possible quorum views that can be observed assuming that the read operation \( \rho \) receives replies from each server \( s \in Q_i \). Dark circles represent servers that contain the \( \max TS \), and light colored ones represent any older timestamp. If \( QV(1) \) is observed, Fig. 7(a), it means that the read operation \( \rho \) obtains only one timestamp-value pair from all servers in quorum \( Q_i \), that is value \( v \) associated with \( \max TS \). This implies that the write associated with \( \max TS \) has potentially been completed. Thus, we say that \( QV(1) \) implies a potentially complete write operation.

If \( QV(2) \) is observed, Fig. 7(b), then read operation \( \rho \) witnessed a subset of servers that maintain a timestamp older than \( \max TS \) in each intersection of \( Q_i \). This implies that the write operation (that propagates \( \max TS \)) has not yet received replies from any full quorum. Thus, it indicates that the write associated with \( \max TS \) is still in progress. From this observation, \( QV(2) \) reveals an incomplete write operation.

Lastly, if \( QV(3) \) is observed, the distribution of timestamps does not provide sufficient information for the state of the write operation associated with \( \max TS \). This is because there are two possibilities as shown in Fig. 7(c) and 7(d). In Fig. 7(c) the write is incomplete while in Fig. 7(d) the write completes in quorum \( Q_z \), however, in both executions, every server in the intersection of \( Q_z \cap Q_i \) replies with \( \max TS \) to read operation \( \rho \).

In the MWMR setting, we use tags instead of timestamps to order operations. Therefore, to comply with the ordering scheme of algorithms developed for the MWMR setting, the authors revised Definition 6.1.1 to examine tags instead of timestamps. The revised definition is the following:
Definition 6.1.2 (Quorum Views in the MWMR setting [30]) Suppose that during an exchange, a read $\rho$ strictly receives tag and value pairs $\langle s.ts, s.id, v \rangle$ from each server $s \in Q_i$. Notice that $s.id$ is the $id$ of the writer that writes value $v$. A reader process $\rho$ can distinguish three different quorum views that may reveal the state of the write operation that writes value $v$ associated with $maxTAG$:

- $QV(1)$: $\forall s \in Q_i : \langle s.ts, s.id \rangle = maxTAG$
- $QV(2)$: $\forall Q_j \in Q, i \neq j, \exists A \subseteq Q_i \cap Q_j, \text{s.t. } A \neq \emptyset \text{ and } \forall s \in A : \langle s.ts, s.id \rangle < maxTAG$
- $QV(3)$: $\exists s' \in Q_i : \langle s'.ts, s'.id \rangle < maxTS \text{ and } \exists Q_j \in Q, i \neq j \text{ s.t. } \forall s \in Q_i \cap Q_j : \langle s.ts, s.id \rangle = maxTAG$.

With the same reasoning, $QV(1)$ implies the potential completion of the write operation that writes value $v$ associated with $maxTS$. View $QV(2)$ imposes its non-completion and $QV(3)$ does not reveal any information regarding the status of the write operation.

Communication Scheme. The second major design element of the algorithms presented here is the one-and-a-half rounds communication scheme that processes follow. Recall from Section 4.1 that it involves three communication exchanges as follows:

- **EXCHANGE E1**: Initiated by a reader process $r$. Reader $r$ multicasts a request message to a subset of replica servers.
- **EXCHANGE E2**: A server process upon receiving the request message it then relays the request to a subset of replica servers.
- **EXCHANGE E3**: Once a server receives “sufficient” relay messages for a particular read operation from a subset of servers, it sends a message to the requesting reader $r$. 

152
6.2 The SWMR Setting

In this section, we design an algorithm that implements atomic shared memory in the SWMR setting while incorporating the two abovementioned techniques.

6.2.1 Description of SWMR Algorithm ERATO

We present ERATO\(^1\), Efficient Reads for ATomic Objects, a SWMR algorithm for atomic objects in the asynchronous message-passing model with processor crashes. We improve the three-exchange read protocol of OHSAM (Section 4.2) to allow reads to terminate in either two or three exchanges using quorum views.

In a nutshell, when the reader receives messages during E2, it analyses the timestamps to determine whether to terminate fast or wait for the conclusion of E3 before completion. Due to asynchrony it is possible for the message from E3 to arrive at the reader before messages from E2. In this case the reader still terminates in three exchanges. A key idea of the algorithm is when the reader is “slow”, i.e., it terminates by the conclusion of E3, then the reader returns the value associated with the minimum timestamp. That is, the value associated with the previous write which is guaranteed to be complete (cf. Section 4.1). Writes are identical to algorithm ABD [10], and take two exchanges to complete. The code for the reader and writer protocols is given in Algorithm 13 and for the server protocol in Algorithm 14.

Counter variables read\textsubscript{op}, operations and relays are used to help processes identify “new” read and write operations, and distinguish “fresh” from “stale” messages (since messages can be reordered). The value of the object and its associated timestamp, as known by

\(^1\)E\textsubscript{ρατω} is a Greek Muse according to Greek mythology.
each process, are stored in variables $v$ and $ts$ respectively. Below we provide a brief description of the protocol of each participant of the service.

**Writer Protocol.** Writer $w$ increments its local timestamp $ts$ and broadcasts a writeRequest message to all servers in exchange $E_1$. It completes once writeAck messages are received from some servers that together they compose a full quorum $Q$ during $E_2$ (A13:L41-45).

**Reader Protocol.** Each reader $r$ maintain several temporary variables. Key variable include $minTS$ and $maxTS$ hold the minimum and the maximum timestamps discovered during the read operation. Sets $RR$ and $RA$ hold the received readRelay and readAck messages respectively. The ids of servers that sent these messages are stored in sets $RRsrv$ and $RAsrv$ respectively. The set $maxTSrv$ keeps the ids of the servers that sent a readRelay message with the timestamp equal to the maximum timestamp $maxTS$.

Reader $r$ starts its operation by broadcasting a readRequest message to the servers in exchange $E_1$. It then collects readRelay messages from exchange $E_2$ and readAck messages from exchange $E_3$. The reader uses counter $read\_op$ to distinguish fresh message from stale message from prior operations. The incoming readRelay and readAck messages are stored into sets $RR$ and $RA$ and their senders into sets $RRsrv$ and $RAsrv$ respectively. The messages are collected until messages of the same type are received from some quorum $Q$ of servers (A13:L7-10). If readRelay messages are received from a full quorum $Q$, then the reader examines the timestamps to determine what quorum view is observed (recall Section 6.1). If $QV(1)$ is observed, then the distribution of the timestamps at the servers indicates the existence of one and only timestamp in $Q$, and that is, the maximum timestamp. This means that the write operation associated with the timestamp is complete, and it is safe to return the value associated with it without exchange $E_3$ (A13:L18-19). If $QV(2)$ is observed, then the write associated with the
Algorithm 13 Reader and Writer Protocols for SWMR algorithm ERATO

1: At each reader \( r \)
2: Variables:
3: \( \min TS, \max TS \in \mathbb{N}; read_{op} \in \mathbb{N} \) init \( 0 \)
4: \( RR, RA, \max ACK \subseteq S \times M \)
5: \( v \in V; RRsrv, RAsrv, maxTSrv \subseteq S \)

6: function READ()
7: \( read_{op} \leftarrow read_{op} + 1 \)
8: \( (RR, RA, RRsrv, RAsrv) \leftarrow (\emptyset, \emptyset, \emptyset) \)
9: \( \text{bcast} ((\text{readRequest}, r, read_{op})) \to S \)
10: \( \text{wait until } (\exists Q \in Q : Q \subseteq RRsrv \lor Q \subseteq RAsrv) \)
11: if \( (\exists Q \in Q : Q \subseteq RRsrv) \)
12: \( \min TS \leftarrow \min \{(m.ts) : (s, m) \in RA \land s \in Q \} \)
13: \( \text{return}(m.v \text{ s.t. } (s, m) \in RA \land m.ts = \min TS) \)
14: else if \( (\exists Q \in Q : Q \subseteq RAsrv) \)
15: \( \max ACK \leftarrow \max \{(m.ts) : (s, m) \in RR \land s \in Q \} \)
16: \( \text{return}(m.v \text{ s.t. } (s, m) \in \max ACK) \)
17: if \( (Q \subseteq maxTSrv) \)
18: \( \text{return}(m.v \text{ s.t. } (s, m) \in maxACK) \)
19: if \( (\exists Q \in Q, Q^\prime \neq Q \text{ s.t. } Q \cap Q^\prime \subseteq maxTSrv) \)
20: \( \text{wait until } (\exists Q \in Q : Q \subseteq RRsrv) \)
21: \( \min TS \leftarrow \min \{(m.ts) : (s, m) \in RA \land s \in Q \} \)
22: \( \text{return}(m.v \text{ s.t. } (s, m) \in RA \land s \in Q \land m.ts = \min TS) \)
23: else
24: \( \max ACK \leftarrow \{ (s, m) \in RR : s \in Q \land m.ts = \max TS - 1 \} \)
25: \( \text{return}(m.v \text{ s.t. } (s, m) \in \max ACK) \)

27: Upon receive \( m \) from \( s \)
28: if \( (m.read_{op} = read_{op}) \)
29: if \( (m.type = \text{readRelay}) \)
30: \( RR \leftarrow RR \cup \{ (s, m) \} \)
31: \( RRsrv \leftarrow RRsrv \cup \{ s \} \)
32: else if \( (m.type = \text{readAck}) \)
33: \( RA \leftarrow RA \cup \{ (s, m) \} \)
34: \( RAsrv \leftarrow RAsrv \cup \{ s \} \)

35: At writer \( w \)
36: Variables:
37: \( ts \in \mathbb{N}^+, v \in V, wAck \subseteq S \)
38: Initialization:
39: \( ts \leftarrow 0, v \leftarrow \bot, wAck \leftarrow \emptyset \)
40: function WRITE(val : input)
41: \( (ts, v) \leftarrow (ts + 1, val) \)
42: \( wAck \leftarrow \emptyset \)
43: \( \text{broadcast} ((\text{writeRequest}, ts, v, w)) \to S \)
44: \( \text{wait until } (\exists Q \in Q : Q \subseteq wAck) \)
45: \( \text{return}() \)

46: Upon receive \( m \) from \( s \)
47: if \( (m.ts = ts) \)
48: \( wAck \leftarrow wAck \cup \{ s \} \)
Algorithm 14 Server Protocol for SWMR algorithm Erato

At server $s$

Variables:

- $t_s \in \mathbb{N}$ init 0 ; $v \in V$ init $\perp$
- $D \subseteq S$ init $\{s' : Q \in Q \land (s,s' \in Q)\}$
- $\text{operations} : R \rightarrow \mathbb{N}$ init $0^{\mid R\mid}$
- $\text{relays} : R \rightarrow 2^S$ init $0^{\mid R\mid}$

Upon receive $((\text{readRequest}, r, \text{read}_\text{op}))$

broadcast($(\text{readRelay}, t_s, v, r, \text{read}_\text{op}, s)$) to $D \cup r$

Upon receive $((\text{writeRequest}, t_{s'}, v', w))$

if $(t_s < t_{s'})$ then

$(t_s, v) \leftarrow (t_{s'}, v')$

send $(\langle \text{writeAck}, t_s, s \rangle)$ to writer $w$

Upon receive $(\langle \text{readRelay}, t_{s'}, v', r, \text{read}_\text{op}, s \rangle)$

if $(t_s < t_{s'})$ then

$(t_s, v, vp) \leftarrow (t_{s'}, v')$

if $(\text{operations}[r] < \text{read}_\text{op})$ then

$\text{operations}[r] \leftarrow \text{read}_\text{op}$

$\text{relays}[r] \leftarrow \emptyset$.

if $(\text{operations}[r] = \text{read}_\text{op})$ then

$\text{relays}[r] \leftarrow \text{relays}[r] \cup \{s\}$

if $(\exists Q \in Q : Q \subseteq \text{relays}[r])$ then

send $(\langle \text{readAck}, t_s, v, \text{read}_\text{op}, s \rangle)$ to reader $r$

maximum timestamp $\text{maxTS}$ is not complete. But because there is a sole writer, it is safe to to
return the value associated with timestamp $\text{maxTS} - 1$, i.e., the value of the preceding complete
write, again without exchange E3 (A13:L24-26). If $QV(3)$ is observed, then the write associated
with the maximum timestamp $\text{maxTS}$ is in progress or complete. Since the reader is unable to
decide which case applies, it waits for readAck messages from E3 from some quorum $Q$ be-
fore completion (A13:L20-23). In this case, the reader returns the value associated with the
minimum timestamp observed. This ensures that the value to be returned was propagated to
“enough” servers, and that is, the value that was written during the last complete operation.
It is possible, due to asynchrony, that messages from E3 arrive from a quorum before enough
messages from E2 are gathered. Here the reader decides as above for E3 (A13:L11-13).
**Server Protocol.** Server \( s \) stores the value of the replica \( v \) and its associated timestamp \( ts \). The \textit{relays} array is used to store sets of processes that relayed to \( s \) regarding a read operation. Destinations set \( D \) is initialized to set containing all servers from every quorum that contains \( s \) and it is used for sending relay messages during exchange \( E2 \). Each server \( s \in S \) expects three types of messages:

1. Upon receiving a \( \langle \text{readRequest}, r, read\_op \rangle \) message from exchange \( E1 \) of a read operation, the server creates a \textit{readRelay} message, containing its local information and it broadcasts it in exchange \( E2 \) to destinations in \( D \) and the requesting reader \( r \) (A14:L55-56).

2. Upon receiving a \( \langle \text{readRelay}, ts', v', r, read\_op, s \rangle \) message from exchange \( E2 \), server \( s \) compares its local local information with the information enclosed in the message. If the information is “fresh” i.e., \( ts < ts' \), then \( s \) sets its local value and timestamp to those enclosed in the message (A14:L62-63). In any other case, no updates are taking place. Next, \( s \) checks if the received \textit{readRelay} marks a new read operation by \( r \), i.e., \( read\_op > \text{operations}[r] \). If this holds, then \( s \): (a) sets its local counter for \( r \) to the enclosed one, \( \text{operations}[r] = read\_op \); and (b) re-initializes the relay set for \( r \) to an empty set, \( \text{relays}[r] = \emptyset \) (A14:L64-66). It then adds the sender of the \textit{readRelay} message to the set of servers that informed it regarding the read invoked by \( r \) (A14:L67-68). Once \textit{readRelay} messages are received from a full quorum \( Q \), \( s \) creates a \textit{readAck} message and sends it to \( r \) in exchange \( E3 \) of the read operation (A14:L69-70).

3. Upon receiving a \( \langle \text{writeRequest}, ts', v', w \rangle \) message, server \( s \) compares its local information with the received one and if \( ts < ts' \), then \( s \) sets its local timestamp and value to those received. In any other case, no updates are taking place. Finally, the server acknowledges the requesting writer with a \textit{writeAck} message \( s \) compares its \( ts \) to the received one (A14:L57-60).
6.2.2 Correctness of ERATO

We first show liveness (termination) and then atomicity (safety).

Liveness. Termination of Algorithm ERATO is guaranteed with respect to our failure model: at least one quorum $Q$ is non-faulty and each operation waits for messages from a single quorum. Let us consider this in more detail.

Write Operation. Showing liveness is straightforward. Per algorithm ERATO, writer $w$ creates a writeRequest message and then it broadcasts it to all servers. Writer $w$ then waits for writeAck messages from a full quorum of servers. Since in our failure model at least one quorum is non-faulty, then writer $w$ collects writeAck messages from a full quorum of live servers and write operation $\omega$ terminates.

Read Operation. The reader $r$ begins by broadcasting a readRequest message all servers and waiting for responses. A read operation of the algorithm ERATO terminates when the reader $r$ either (i) collects readAck messages from full quorum of servers or (ii) collects readRelay messages from a full quorum and notices $Q(1)$ or $Q(2)$. Let’s analyze case (i). Since a full quorum $Q$ is non-faulty then at least a full quorum of servers receives the readRequest message. Any server $s$ that receives this message broadcasts readRelay message to every server that belongs to the same quorum with, and the invoker $r_i$. That is its destinations set $D \cup \{r_i\}$ (A14:L56-56). In addition, no server ever discards any incoming readRelay messages. Any server, whether it is aware or not of the readRequest, always keeps a record of the incoming readRelay messages and takes action as if it is aware of the readRequest. The only difference between server $s_i$ that received a readRequest message and server $s_k$ that does not, is that $s_i$ is able to broadcast readRelay messages, and $s_k$ broadcasts readRelay messages when $s_k$
receives the readRequest message. Each non-failed server receives readRelay messages from a full quorum of servers and sends a readAck message to reader $r$ (A13:L69-70). Therefore, reader $r$ can always collect readAck messages from a full quorum of servers, decide on a value to return, and terminate (A13:L11-13). In case where case (ii) never holds then the algorithm will always terminate from case (i). Thus, since any read or write operation will collect a sufficient number of messages and terminate then liveness is satisfied.

Based on the above, it is always the case that acknowledgment messages readAck and writeAck are collected from a full quorum of servers in any read and write operation, thus ensuring liveness.

**Atomicity.** We use the association between the timestamps and the partial order as given in Section 4.2.2. We now state and prove a series of lemmas leading to the correctness result.

It is easy to see that the $ts$ variable in each server $s$ is monotonically increasing. This leads to the following lemma.

**Lemma 6.2.1** In any execution $\xi$ of ERATO, the variable $ts$ maintained by any server $s$ in the system is non-negative and monotonically increasing.

**Proof.** Upon receiving a timestamp $ts$, a server $s$ updates its local timestamp $ts_s$ iff $ts > ts_s$, (A14:L58-59, 62-63), and the lemma follows. $\square$

Next, we show that any read operation that follows a write operation, it receives readAck messages the servers where each included timestamp is at least as the one returned by the complete write operation.
Lemma 6.2.2 In any execution $\xi$ of ERATO, if a read operation $\rho$ succeeds a write operation $\omega$ that writes $ts$ and $v$, i.e., $\omega \rightarrow \rho$, and receives readAck messages from a quorum $Q$ of servers, set $RA$, then each $s \in RA$ sends a readAck message to $\rho$ with a timestamp $ts_s$ s.t. $ts_s \geq ts$.

Proof. Let $wAck$ be the set of servers from a quorum $Q_a$ that send a writeAck message to $\omega$, let $RelaySet$ be the set of servers from a quorum $Q_b$ that sent readRelay messages to server $s$, and let $RA$ be the set of servers from a quorum $Q_c$ that send a readAck message to $\rho$. Notice that it is not necessary that $a \neq b \neq c$ holds.

Write operation $\omega$ is completed. By Lemma 6.2.1, if a server $s$ receives a timestamp $ts$ from a process $p$ at some time $T$, then $s$ attaches a timestamp $ts'$ s.t. $ts' \geq ts$ in any message sent at any time $T' \geq T$. Thus, every server in $wAck$, sent a writeAck message to $\omega$ with a timestamp greater or equal to $ts$. Hence, every server $s_x \in wAck$ has a timestamp $ts_{s_x} \geq ts$.

Let us now examine a timestamp $ts_s$ that server $s$ sends to read operation $\rho$.

Before server $s$ sends a readAck message to $\rho$, it must receive readRelay messages from a full quorum $Q_b$ of servers, $RelaySet$ (A14:L69-70). Since both $wAck$ and $RelaySet$ contain messages from a full quorum of servers, and by definition, any two quorums have a non-empty intersection, then $wAck \cap RelaySet \neq \emptyset$. By Lemma 6.2.1, any server $s_x \in aAck \cap RelaySet$ has a timestamp $ts_{s_x}$ s.t. $ts_{s_x} \geq ts$. Since server $s_x \in RelaySet$ and from the algorithm, server’s $s$ timestamp is always updated to the highest timestamp it noticed (A14:L62-62), then when server $s$ receives the message from $s_x$, it will update its timestamp $ts_s$ s.t. $ts_s \geq ts_{s_x}$.

Server $s$ creates a readAck message where it encloses its local timestamp and its local value,
(ts_s, v_s) (A14:L70). Each s ∈ RA sends a readAck to ρ with a timestamp ts_s s.t. ts_s ≥ ts_{s_x} ≥ ts. Thus, ts_s ≥ ts, and the lemma follows.

Now, we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

**Lemma 6.2.3** In any execution ξ of ERATO, if a read ρ succeeds a write operation ω that writes timestamp ts, i.e. ω → ρ, and ρ returns a timestamp ts', then ts' ≥ ts.

**Proof.** A read operation ρ terminates when it either receives (a) readRelay messages from a full quorum Q or (b) readAck messages from a full quorum Q (A13:L7-10).

We first examine case (b). Let’s suppose that ρ receives readAck messages from a full quorum Q of servers, RA. By lines 11 - 13, it follows that ρ decides on the minimum timestamp, ts' = minTS, among all the timestamps in the readAck messages of the set RA. From Lemma 6.2.2, minTS ≥ ts holds, where ts is the timestamp written by the last complete write operation ω. Then ts' = minTS ≥ ts also holds. Thus, ts' ≥ ts.

Now we examine case (a). In particular, case (a) terminates when the reader process notices either (i) QV(1) or (ii) QV(2) or (iii) QV(3). Let wAck be the set of servers from a quorum Q_a that send a writeAck message to ω. Since the write operation ω, that wrote value v associated with timestamp ts is complete, and by monotonicity of timestamps in servers (Lemma 6.2.1), then at least a quorum Q_a of servers has a timestamp ts_a s.t. ts_a ≥ ts. In other words, every server in wAck, sent a writeAck message to ω with a timestamp ts_a greater or equal to ts.

Let’s suppose that ρ receives readRelay messages from a full quorum Q_b of servers, RR. Since both wAck and RR contain messages from a full quorum of servers, quorums Q_a and Q_b, and by definition any two quorums have a non-empty intersection, then wAck ∩ RR ≠ ∅.
Since every server in $wAck$ has a timestamp $ts_a \geq ts$ then any server $s_x \in wAck \cap RR$ has a timestamp $ts_{s_x}$ s.t. $ts_{s_x} \geq ts_a \geq ts$.

If $\rho$ noticed $QV(1)$ in $RR$, then the distribution of the timestamps indicates the existence of one and only timestamp in $RR$, $ts'$. Hence, $ts' \geq ts_{s_x} \geq ts_a \geq ts$. Based on the algorithm (A13:L18-19), the read operation $\rho$ returns value $v$ associated with $ts'$ and $ts' \geq ts$ holds.

Based on the definition of $QV(2)$, if it is noticed in $RR$, then there must exist at least two servers in $wAck \cap RR$ with different timestamps and one of them holds the maximum timestamp. Let $s_k$ be the one that holds the maximum timestamp $ts_{s_k}$ (or $maxTS$) and $s_m$ the server that holds the timestamp $ts_{s_m}$ s.t. $maxTS = ts_{s_k} > ts_{s_m}$. Since (a) any server $s_x \in wAck \cap RR$ has a timestamp $ts_{s_x}$ s.t. $ts_{s_x} \geq ts$, and (b) $s_k \in wAck \cap RR$ holds the maximum timestamp $ts_{s_k}$ (or $maxTS$), and (c) $s_m \in wAck \cap RR$ and (d) $maxTS = ts_{s_k} > ts_{s_m}$ then it follows that $maxTS = ts_{s_k} > ts_{s_m} \geq ts$. Thus, $ts_{s_k}$ (or $maxTS$) must be strictly greater from $ts$, $maxTS = ts_{s_k} > ts$. Based on the algorithm, when $\rho$ notices $QV(2)$ in $RR$ then it returns the value $v$ associated with the previous maximum timestamp, that is the value associated with $maxTS-1$ (A13:L24-26). Since $maxTS = ts_{s_k} > ts$, then for the previous maximum timestamp, denoted by $ts'$, which is only one unit less than $maxTS$, then the following holds, $maxTS > maxTS - 1 = ts' \geq ts$. Therefore, in this case $\rho$ returns a value $v$ associated with $ts'$ and $ts' \geq ts$ holds.

Lastly, when $QV(3)$ is noticed then $\rho$ waits for readAck messages from a full quorum $Q$ before termination, (A13:L20-23), proceeds identically as in case (b) and the lemma follows. □

In the following three lemmas we show that if a read operation $\rho_2$ succeeds a read $\rho_1$, then $\rho_2$ returns a value associated with a timestamp $ts_2$ s.t. $ts_2 \geq ts_1$. 
Lemma 6.2.4  In any execution $\xi$ of ERATO, if $\rho_1$ and $\rho_2$ are two semi-fast read operations, take 3 exchanges to complete, such that $\rho_1$ precedes $\rho_2$, i.e., $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns the value for timestamp $ts_1$, then $\rho_2$ returns the value for timestamp $ts_2 \geq ts_1$.

Proof. Let the two operations $\rho_1$ and $\rho_2$ be invoked by processes with identifiers $r_1$ and $r_2$ respectively (not necessarily different). Also, let $RA_1$ and $RA_2$ be the sets of servers from quorums $Q_a$ and $Q_b$ (not necessarily different) that sent a readAck message to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$.

Assume by contradiction that read operations $\rho_1$ and $\rho_2$ exist such that $\rho_2$ succeeds $\rho_1$, i.e., $\rho_1 \rightarrow \rho_2$, and the operation $\rho_2$ returns a timestamp $ts_2$ that is smaller than the $ts_1$ returned by $\rho_1$, i.e., $ts_2 < ts_1$. Based on the algorithm, $\rho_2$ returns a timestamp $ts_2$ that is smaller than the minimum timestamp received by $\rho_1$, i.e., $ts_1$, if $\rho_2$ obtains $ts_2$ and $v$ in the readAck message of some server $s_x \in RA_2$, and $ts_2$ is the minimum timestamp received by $\rho_2$.

Let us examine if $s_x$ sends a readAck message to $\rho_1$ with timestamp $ts_x$, i.e., $s_x \in RA_1$. By Lemma 6.2.1, and since $\rho_1 \rightarrow \rho_2$, then it must be the case that $ts_x \leq ts_2$. According to our assumption $ts_1 > ts_2$, and since $ts_1$ is the smallest timestamp sent to $\rho_1$ by any server in $RA_1$, then it follows that $r_1$ does not receive the readAck message from $s_x$, and hence $s_x \notin RA_1$.

Now let us examine the actions of the server $s_x$. From the algorithm, server $s_x$ collects readRelay messages from a full quorum $Q_c$ of servers before sending a readAck message to $\rho_2$ (A14:L69-69). Let $RRSet_{s_x}$ denote the set of servers from the full quorum $Q_c$ that sent readRelay to $s_x$. Since, both $RRSet_{s_x}$ and $RA_1$ contain messages from full quorums, $Q_c$ and $Q_a$, and since any two quorums have a non-empty intersection, then it follows that $RRSet_{s_x} \cap RA_1 \neq \emptyset$. 163
Thus there exists a server $s_i \in RRS_{s_x} \cap RA_1$, that sent (i) a readAck to for $\rho_1$, and (ii) a readRelay to $s_x$ during $\rho_2$. Note that $s_i$ sends a readRelay for $\rho_2$ only after it receives a read request from $\rho_2$. Since $\rho_1 \rightarrow \rho_2$, then it follows that $s_i$ sent the readAck to $\rho_1$ before sending the readRelay to $s_x$. By Lemma 6.2.1, if $s_i$ attaches a timestamp $ts_{s_i}$ in the readAck to $\rho_1$, then $s_i$ attaches a timestamp $ts'_{s_i}$ in the readRelay message to $s_x$, such that $ts'_{s_i} \geq ts_{s_i}$. Since $ts_1$ is the minimum timestamp received by $\rho_1$, then $ts_{s_i} \geq ts_1$, and hence $ts'_{s_i} \geq ts_1$ as well. By Lemma 6.2.1, and since $s_x$ receives the readRelay message from $s_i$ before sending a readAck to $\rho_2$, it follows that $s_x$ sends a timestamp $ts_2$ s.t. $ts_2 \geq ts'_{s_i} \geq ts_1$. Thus, $ts_2 \geq ts_1$ and this contradicts our initial assumption.

Lemma 6.2.5 In any execution $\xi$ of ERATO, if $\rho_1$ and $\rho_2$ are two fast read operations, take 2 exchanges to complete, such that $\rho_1$ precedes $\rho_2$, i.e., $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns the value for timestamp $ts_1$, then $\rho_2$ returns the value for timestamp $ts_2 \geq ts_1$.

Proof. Let the two operations $\rho_1$ and $\rho_2$ be invoked by processes with identifiers $r_1$ and $r_2$ respectively (not necessarily different). Also, let $RR_1$ and $RR_2$ be the sets of servers from quorums $Q_a$ and $Q_b$ (not necessarily different) that sent a readRelay message to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$.

The algorithm terminates in two communication exchanges when a read operation $\rho$ receives readRelay messages from a full quorum $Q$ and based on the distribution of the timestamp it either notices (a) $QV(1)$ or (b) $QV(2)$. We now examine the four cases.

Case (i), $\rho_1 \rightarrow \rho_2$ and both $\rho_1$ and $\rho_2$ notice $QV(1)$. It is known that all the servers in $RR_1$ replied to $\rho_1$ with timestamp $ts_1$. Since by definition, any two quorums have a non-empty intersection it follows that $RR_1 \cap RR_2 \neq \emptyset$. From that and by Lemma 6.2.1, then every server
Case (ii), $\rho_1 \rightarrow \rho_2$ and $\rho_1$ notices $QV(1)$ and $\rho_2$ notices $QV(2)$. It is known that all the servers in $RR_1$ replied to $\rho_1$ with timestamp $ts_1$. Since by definition, any two quorums have a non-empty intersection it follows that $RR_1 \cap RR_2 \neq \emptyset$. From that and by Lemma 6.2.1, then every server $s_x \in RR_1 \cap RR_2$ has a timestamp $ts'$ such that $ts' \geq ts_1$. Since $\rho_2$ notices $QV(2)$ in $RR_2$, then there must exist at least two servers in $RR_1 \cap RR_2$ with different timestamps and one of them holds the maximum timestamp. Let $s_k$ be the one that holds the maximum timestamp $ts_{s_k}$ (or $maxTS$) and $s_m$ the server that holds the timestamp $ts_{s_m}$ s.t. $maxTS = ts_{s_k} > ts_{s_m}$. Since (a) any server $s_x \in RR_1 \cap RR_2$ has a timestamp $ts'$ s.t. $ts' \geq ts_1$, and (b) $s_k \in RR_1 \cap RR_2$ holds the maximum timestamp $ts_{s_k}$ (or $maxTS$), and (c) $s_m \in RR_1 \cap RR_2$ and (d) $maxTS = ts_{s_k} > ts_{s_m}$ then it follows that $maxTS = ts_{s_k} > ts_{s_m} \geq ts_1$. Thus, $ts_{s_k}$ (or $maxTS$) must be strictly greater from $ts_1$, $maxTS = ts_{s_k} > ts_1$. Based on the algorithm, when $\rho$ notices $QV(2)$ in $RR_2$ then it returns the value $v$ associated with the previous maximum timestamp, that is the value associated with $maxTS-1$ (A13:L24-26). Since $maxTS = ts_{s_k} > ts_1$, then for the previous maximum timestamp, denoted by $ts_2$, which is only one unit less than $maxTS$, then the following holds, $maxTS > maxTS - 1 = ts_2 \geq ts_1$, thus $ts_2 \geq ts_1$.

Case (ii), $\rho_1 \rightarrow \rho_2$ and $\rho_1$ notices $QV(2)$ and $\rho_2$ notices $QV(1)$. Since $\rho_1$ notices $QV(2)$ in $RR_1$ then there exist a subset of servers $S_{max}$, $S_{max} \subset RR_1$, that hold the maximum timestamp $maxTS$ and a subset of servers $S_{pre}$, $S_{pre} \subset RR_1$, that hold timestamp $maxTS-1$. Based on the algorithm, $\rho_1$ returns $ts_1$ s.t. $ts_1 = maxTS - 1$ from the set of servers in $S_{pre}$. Since $RR_1 \cap RR_2 \neq \emptyset$, and $QV(1)$ indicates the existence of one and only timestamp, then $\rho_2$
can notice \(QV(1)\) in two cases; (a) all the servers in \(RR_1 \cap RR_2 \subseteq Spre\) or (b) all the servers in \(RR_1 \cap RR_2 \subseteq Smax\). By Lemma 6.2.1, and if (a) holds then \(\rho_2\) returns \(ts_2\) s.t. \(ts_2 \geq ts_1\); else, if (b) holds then \(\rho_2\) returns \(ts_2\) s.t. \(ts_2 > ts_1\).

Case (i), \(\rho_1 \rightarrow \rho_2\) and both \(\rho_1\) and \(\rho_2\) notice \(QV(2)\). The distribution of the timestamps that \(\rho_1\) notices, indicates that the write operation associated with the \textit{maximum} timestamp, \(maxTS\), is on-going, i.e., not completed. By the property of \textit{well-formedness} and the existence of a sole writer in the system then we know that \(ts_1\) corresponds to the latest complete write operation, \(ts_1 = maxTS - 1\). By Lemma 6.2.3, \(\rho_2\) will not be able to return a timestamp \(ts_2\) s.t. \(ts_2 < maxTS - 1\). Thus \(ts_2 \geq ts_1\) holds and the lemma follows. \(\square\)

\begin{lemma}
In any execution \(\xi\) of \textsc{Erato}, if \(\rho_1\) and \(\rho_2\) are two read operations such that \(\rho_1\) precedes \(\rho_2\), i.e., \(\rho_1 \rightarrow \rho_2\), and \(\rho_1\) returns timestamp \(ts_1\), then \(\rho_2\) returns a timestamp \(ts_2\) s.t. \(ts_2 \geq ts_1\).
\end{lemma}

**Proof.** We are interested to examine the cases where one of the read operation is \textit{fast} and the other is \textit{semifast}. In particular, cases (i) \(\rho_1 \rightarrow \rho_2\) and \(\rho_1\) is \textit{semifast} and \(\rho_2\) is \textit{fast} and (ii) \(\rho_1 \rightarrow \rho_2\) and \(\rho_1\) is \textit{fast} and \(\rho_2\) is \textit{semifast}.

Let the two operations \(\rho_1\) and \(\rho_2\) be invoked by processes with identifiers \(r_1\) and \(r_2\) respectively (not necessarily different). Also, let \(RR_1, RA_1\) and \(RR_2, RA_2\) be the sets of servers from full quorums (not necessarily different) that sent a \texttt{readRelay} and \texttt{readAck} message to \(\rho_1\) and \(\rho_2\) respectively.

We start with case (i). Since read operation \(\rho_1\) is \textit{semifast}, then based on the algorithm, the timestamp \(ts_1\) that is returned it is also the \textit{minimum} timestamp noticed in \(RA_1\). Before a server \(s\) sent \texttt{readAck} messages to \(\rho_1\) (that form \(RA_1\)), it must receive \texttt{readRelay} messages from a
full quorum of servers. Thus, by Lemma 6.2.1 monotonicity of the timestamps at the servers we know that the minimum timestamp that a full quorum has by the end of $\rho_1$ is $t_{s_1}$. Read operation $\rho_2$ receives readRelay messages from a full quorum of servers, $RR_2$. By definition of quorums, since both $RA_1$ and $RR_2$ are from a full quorum of servers then it follows that $RA_1 \cap RR_2 \neq \emptyset$. Thus every server $s_x \in RA_1 \cap RR_2$ holds a timestamp $t_{s'}$ s.t. $t_{s'} \geq t_{s_1}$.

If $\rho_2$ notices $QV(1)$ in $RR_2$ then the distribution of the timestamps in $RR_2$ indicates the existence of one and only timestamp, $maxTS$. From the above, it follows that for the timestamp $t_{s_2}$ that $\rho_2$ returns $maxTS = t_{s_2} \geq t_{s'} \geq t_{s_1}$ holds.

On the other hand, if $\rho_2$ notices $QV(2)$ in $RR_2$, then based on the distributions of the timestamps in $QV(2)$ there must exist at least two servers in $RA_1 \cap RR_2$ with different timestamps and the one must be the maximum. Since every server $s_x \in RA_1 \cap RR_2$ holds a timestamp $t_{s'}$ s.t. $t_{s'} \geq t_{s_1}$ then the maximum timestamp $maxTS$ cannot be equal to $t_{s_1}$. If that was the case, $\rho_2$ would have noticed $QV(1)$. In particular, now $maxTS > t_{s_1}$ holds. Based on the algorithm, when $\rho$ notices $QV(2)$ in $RR_2$ then it returns the value $v$ associated with the previous maximum timestamp, that is the value associated with $maxTS-1$ (A13:L24-26). Since $maxTS > t_{s_1}$, then for the previous maximum timestamp, denoted by $t_{s_2}$, which is only one unit less than $maxTS$, then the following holds, $maxTS > maxTS - 1 = t_{s_2} \geq t_{s_1}$, thus $t_{s_2} \geq t_{s_1}$.

We now examine case (ii). Since $\rho_1$ is fast, it follows that it has either noticed $QV(1)$ or $QV(2)$ in $RR_1$. If $QV(1)$ was noticed, and $\rho_1$ returned a value associated with maximum timestamp $t_{s_1}$, then by the completion of $\rho_1$ a full quorum has a timestamp $t_{s'}$ s.t. $t_{s'} \geq t_{s_1}$. Now, since read operation $\rho_2$ is semifast, then based on the algorithm, the timestamp $t_{s_2}$ that is returned it is the minimum timestamp noticed in $RA_2$. Before a server $s$ sends readAck messages to $\rho_2$ (that
form \( RA_2 \), it must receive readRelay messages from a full quorum of servers, RelaySet. By Lemma 6.2.1 monotonicity of the timestamps at the servers and \( RR_1 \cap RelaySet \neq \emptyset \), then every server in \( RA_2 \) has a timestamp \( ts_2 \) s.t. \( ts_2 \geq ts' \geq ts_1 \).

If \( QV(2) \) was noticed in \( RR_1 \), based on the algorithm, \( \rho_1 \) returned a value associated with previous maximum timestamp, that is \( ts_1 \). By the completion of \( \rho_1 \) a full quorum has a timestamp \( ts' \) s.t. \( ts' \geq ts_1 \). Read operation \( \rho_2 \) is semifast, and the returned timestamp \( ts_2 \) is the minimum timestamp noticed in \( RA_2 \). A server \( s \) sends readAck messages to \( \rho_2 \) (that form \( RA_2 \)), when receives readRelay messages from a full quorum of servers, RelaySet. By Lemma 6.2.1 and since \( RR_1 \cap RelaySet \neq \emptyset \), then every server in \( RA_2 \) has a timestamp \( ts_2 \) s.t. \( ts_2 \geq ts' \geq ts_1 \). The rest of the cases are proved in Lemmas 6.2.4 and 6.2.5.

We now show the correctness of ERATO.

**Theorem 6.2.7** Algorithm ERATO implements an atomic SWMR object.

**Proof.** We use the lemmas stated above and the operations order definition to reason about each of the three atomicity conditions A1, A2 and A3 as given in Definition 2.2.5.

**A1** For any \( \pi_1, \pi_2 \in \Pi \) such that \( \pi_1 \rightarrow \pi_2 \), it cannot be that \( \pi_2 \prec \pi_1 \).

When the two operations \( \pi_1 \) and \( \pi_2 \) are reads and \( \pi_1 \rightarrow \pi_2 \) holds, then from Lemma 6.2.6 it follows that the timestamp of \( \pi_2 \) is no less than the one of \( \pi_1 \), \( ts_{\pi_2} \geq ts_{\pi_1} \). If \( ts_{\pi_2} > ts_{\pi_1} \) then by the ordering definition \( \pi_1 \prec \pi_2 \) is satisfied. When \( ts_{\pi_2} = ts_{\pi_1} \) then the ordering is not defined, thus it cannot be the case that \( \pi_2 \prec \pi_1 \). If \( \pi_2 \) is a write, the sole writer generates a new timestamp by incrementing the largest timestamp in the system. By well-formedness, any timestamp generated in any write operation that precedes \( \pi_2 \) must be smaller than \( ts_{\pi_2} \). Since \( \pi_1 \rightarrow \pi_2 \), then it holds that \( ts_{\pi_1} < ts_{\pi_2} \). Hence, by the ordering definition it cannot
be the case that $\pi_2 < \pi_1$. Lastly, when $\pi_2$ is a read and $\pi_1$ a write and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 6.2.3 it follows that $ts_{\pi_2} \geq ts_{\pi_1}$. By the ordering definition, it cannot hold that $\pi_2 < \pi_1$ in this case either.

**A2** For any write $\omega \in \Pi$ and any operation $\pi \in \Pi$, then either $\omega < \pi$ or $\pi < \omega$.

If the timestamp returned from $\omega$ is greater than the one returned from $\pi$, i.e. $ts_{\omega} > ts_{\pi}$, then $\pi < \omega$ follows directly. Similarly, if $ts_{\omega} < ts_{\pi}$ holds, then $\omega < \pi$ follows. If $ts_{\omega} = ts_{\pi}$, then it must be that $\pi$ is a read and $\pi$ discovered $ts_{\omega}$ in a quorum view $QV(1)$ or $QV(3)$. Thus, $\omega < \pi$ follows.

**A3** Every read operation returns the value of the last write preceding it according to $\prec$ (or the initial value if there is no such write).

Let $\omega$ be the last write preceding read $\rho$. From our definition it follows that $ts_{\rho} \geq ts_{\omega}$. If $ts_{\rho} = ts_{\omega}$, then $\rho$ returns the value conveyed by $\omega$ to some servers in a quorum $Q$, satisfying either $QV(1)$ or $QV(3)$. If $ts_{\rho} > ts_{\omega}$, then $\rho$ obtains a larger timestamp, but such a timestamp can only be created by a write that succeeds $\omega$, thus $\omega$ does not precede the read and this cannot be the case. Lastly, if $ts_{\rho} = 0$, no preceding writes exist, and $\rho$ returns the initial value. □

#### 6.2.3 Performance of ERATO

We now assess the performance of ERATO in terms of (i) latency of read and write operations as measured by the number of communication exchanges, (ii) the message complexity of read and write operations, and (iii) computational complexity.

In brief, for algorithm ERATO write operations take 2 exchanges and read operations take either 2 or 3 exchanges. The (worst case) message complexity of write operations is $2|\mathcal{S}|$ and
the (worst case) message complexity of read operations is $|S|^2 + 3|S|$. This follows directly from the structure of the algorithm. We now give additional details.

**Operation Latency.** We study the operation latency, in terms of the number of communication exchanges required by each operation.

*Write operation latency:* According to algorithm ERATO, writer $w$ sends writeRequest messages to all servers during exchange $E_1$ and waits for writeAck messages from a full quorum of servers during $E_2$. Once the writeAck messages are received, no further communication is required and the write operation terminates. Therefore, any write operation consists of 2 communication exchanges.

*Read operation latency:* A reader sends a readRequest message to all the servers in the first communication exchange $E_1$. Once the servers receive the readRequest message they broadcast a readRelay message to all servers and the reader in exchange $E_2$. The reader can terminate at the end of the $E_2$ if it receives readRelay messages and based on the distribution of the timestamp it notices $QV(1)$ or $QV(2)$. If this is not the case, the operation goes into the third exchange $E_3$. Thus read operations terminate after either 2 or 3 communication exchanges.

**Message Complexity.** We measure operation message complexity as the worst case number of exchanged messages in each read and write operation. The worst case number of messages corresponds to failure-free executions where all participants send messages to all destinations according to the protocols.

*Write operation:* A single write operation in algorithm ERATO takes 2 communication exchanges. In the first exchange $E_1$, the writer sends a writeRequest message to all the servers in $S$. The second exchange $E_2$, occurs when all servers in $S$ send a writeAck message to the writer. Thus, at most $2|S|$ messages are exchanged in a write operation.
**Read operation:** Read operations in the worst case take 3 communication exchanges. Exchange \( E_1 \) occurs when a reader sends a readRequest message to all servers in \( S \). The second exchange \( E_2 \) occurs when servers in \( S \) send readRelay messages to all servers in \( S \) and to the requesting reader. The final exchange \( E_3 \) occurs when servers in \( S \) send a readAck message to the reader. Summing together \( |S| + (|S|^2 + |S|) + |S| \), shows that in the worst case, \( |S|^2 + 3|S| \) messages are exchanged during a read operation.

**Computational Complexity.** Algorithm ERATO performs a modest amount of local computation, resulting in negligible computation overhead.

### 6.3 The MWMR Setting

We now extend algorithm ERATO to obtain an atomic shared register implementations for the MWMR setting, called ERATO-mw.

#### 6.3.1 Description of MWMR Algorithm ERATO-MW

We now aim for a MWMR algorithm that involves *two or three* communications exchanges per read operation and *four* exchanges per write operation. The read protocol of algorithm ERATO exploits the single writer and the well-formedness property: based on the distribution of the timestamp in a quorum \( Q \), if the reader knows that the write operation is not complete, then any previous write is complete. In the MWMR setting such an assumption does not hold due to the possibility of concurrent writes. Consequently, algorithm ERATO-MW, in order to allow operations to terminate in either *two or three* communication exchanges, adapts the read protocol from algorithm OHMAM in combination with the iterative technique using quorum
views of CWFr. The latter approach, not only predicts the completion status of a write operation, but also detects the last potentially complete write operation. Writes are identical to algorithm ABD-MW [55], and take four exchanges to complete. The code for the reader and writer protocols is given in Algorithm 15 and for the server protocol in Algorithm 16. Below we provide a brief description of the protocol of each participant in the service.

**Writer Protocol.** Similarly to the four-exchange implementation [55], a writer broadcasts a writeDiscover message to all servers in exchange E1, and awaits “fresh” discoverAck messages from some quorum Q from E2 (A15:L37-40). Among these responses, the writer extracts from the tags the maximum timestamp, denoted by maxTS, increments it, and associates it with its own id creating a new tag. Then it associates the value to be written with this tag and broadcasts it in a writeRequest message to all servers in E3. The write completes when writeAck messages are received from some quorum Q from E4 (A15:L41-46).

**Reader Protocol.** Readers use state variables similarly to algorithm ERATO (Section 6.2). Reader r broadcasts a readRequest message to all servers during exchange E1 and it then awaits either (a) readRelay messages from some quorum from E2, or (b) readAck messages from some quorum from E3 before completion (A15:L6-9).

The key departure here is in how the reader handles case (a) and in particular the subcase where QV(2) is detected, which indicates that the write associated with the maximum tag is not complete. Here the reader considers past history and discovers the tag associated with the last complete write. This is accomplished in an iterative manner, by removing the servers that respond with the maximum tag in the responding quorum Q and repeating the analysis (A15:L14-25). During the iterative process, if r detects QV(1) it returns the value associated with the maximum tag discovered during the current iteration. If no iteration yields QV(1), then
Algorithm 15 Reader and Writer Protocols for MWMR algorithm ERATO-MW

1. **At each reader** $r$
2. **Variables and Initialization:**
3. $v \in V$ init ⊥; $read_{op} \in N$ init 0; $minTAG, maxTAG \in T$ init (0, 0);
4. $RR, RA, maxACK \subseteq S \times M$ init 0; $RRsrv, RAsrv, maxTGsrv \subseteq S$ init 0;
5. function **READ**()
6. \(read_{op} \leftarrow read_{op} + 1\)
7. $(RR, RA, maxACK, RRsrv, RAsrv, maxTGsrv) \leftarrow (0, 0, 0, 0, 0, 0)$
8. **broadcast** ((readRequest, $r, read_{op}$)) to $S$
9. **wait until** (3Q $\in Q: Q \subseteq RRsrv \lor Q \subseteq RAsrv)
10. if (3Q $\in Q: Q \subseteq RAsrv) then$
11. \(minTAG \leftarrow \min\{(m.ts, m.id) : (s, m) \in RA \land s \in Q\}\)
12. **return** (m.v s.t. (s, m) $\in RA \land s \in Q \land (m.ts, m.id) = minTAG)\)
13. else if (3Q $\in Q: Q \subseteq RAsrv) then
14. while (Q $\neq \emptyset$) do
15. \(maxTAG \leftarrow \max\{(m.ts, m.id) : (s, m) \in RA \land s \in Q\}\)
16. \(maxACK \leftarrow \{(s, m) \in RR : s \in Q \land (m.ts, m.id) = maxTAG\}\)
17. \(maxTGsrv \leftarrow \{s \in Q : (s, m) \in maxACK\}\)
18. if Q $\subseteq maxTGsrv$ then \(\Rightarrow**Indicates QView1**\)
19. **return** (m.v s.t. (s, m) $\in maxACK$)
20. if (3Q $\in Q, Q' \neq Q : Q' \cap Q \subseteq maxTGsr)$ then \(\Rightarrow**Indicates QView3**\)
21. **wait until** (3Q $\in Q : Q'' \subseteq RAsrv$)
22. \(minTAG \leftarrow \min\{(m.ts, m.id) : (s, m) \in RA \land s \in Q''\}\)
23. **return** (m.v s.t. (s, m) $\in RA \land s \in Q'' \land (m.ts, m.id) = minTAG)\)
24. else \(\Rightarrow**Indicates QView2**\)
25. \(Q \leftarrow Q - maxTGsrv\)

26. **Upon receive** m from s
27. if (m.read_op = read_op) then
28. if (m.type = readRelay) then
29. \(RR \leftarrow RR \cup \{(s, m)\}, RRsrv \leftarrow RRsrv \cup \{s\}\)
30. else if (m.type = readAck) then
31. \(RA \leftarrow RA \cup \{(s, m)\}, RAsrv \leftarrow RAsrv \cup \{s\}\)

32. **At each writer** w
33. **Variables and Initialization:**
34. ts $\in N$ init 0; v $\in V$ init ⊥; $write_{op} \in N$ init 0; maxTS $\in N$ init 0;
35. Acks $\subseteq S \times M$ init 0; AcksSrv $\subseteq S$ init 0;
36. function **WRITE**(val : input)
37. \(write_{op} \leftarrow write_{op} + 1\)
38. \((Acks, AcksSrv) \leftarrow (\emptyset, 0)\)
39. **broadcast** ((writeDiscover, write_op, w)) to $S$
40. **wait until** (3Q $\in Q : Q \subseteq AcksSrv$)
41. \(maxTS \leftarrow \max\{(m.ts) : (s, m) \in Acks \land s \in Q\}\)
42. \((ts, id, v) \leftarrow (maxTS + 1, i, val)\)
43. \(write_{op} \leftarrow write_{op} + 1\)
44. \((Acks, AcksSrv) \leftarrow (\emptyset, 0)\)
45. **broadcast** ((writeRequest, ts, v, w, write_op)) to $S$
46. **wait until** (3Q $\in Q : Q \subseteq AcksSrv$)
47. **return**()

48. **Upon receive** m from s
49. if m.write_op = write_op then
50. \(Acks \leftarrow Acks \cup \{(s, m)\}, AcksSrv \leftarrow AcksSrv \cup \{s\}\)
Algorithm 16 Server Protocol for MWMR algorithm E

```
51: At server s
52: Variables and Initialization:
53: ts ∈ ℕ init 0; id ∈ W init ⊥; v ∈ V init ⊥;
54: operations : R → ℕ init 0 |R|
55: write_ops : W → ℕ init 0 |W|
56: relays : R → 2^ℕ init ∅ |R|
57: D ⊆ S init {s : (∃Q ∈ Q), (s, s_i ∈ Q)}
58: Upon receive (⟨writeDiscover, write_op, w⟩)
59: send (⟨discoverAck, ts, id, s_i⟩) to w
60: Upon receive (⟨writeRequest, ts', v', id', write_op, w⟩)
61: if write_ops[w] < write_op then
62: write_ops[w] ← write_op
63: if (ts < ts') ∨ (ts = ts' ∧ id < id') then
64: (ts, id, v) ← (ts', id', v')
65: send (⟨writeAck, write_op, s⟩) to w
66: Upon receive (⟨readRequest, r, read_op⟩)
67: bcast (readRelay, ts, id, v, r, read_op, s) to D ∪ r
68: Upon receive (⟨readRelay, ts', id', v', r, read_op, s⟩)
69: if (ts < ts') ∨ (ts = ts' ∧ id < id') then
70: (ts, id, v) ← (ts', id', v')
71: if (operations[r] < read_op) then
72: operations[r] ← read_op; relays[r] ← ∅.
73: if (operations[r] = read_op) then
74: relays[r] ← relays[r] ∪ {s}
75: if (∃Q ∈ Q : Q ⊆ relays[r]) then
76: send (⟨readAck, ts, id, v, read_op, s⟩) to r
```

eventually r observes QV(3). In the last case, QV(3) is detected when a single server remains in some intersection of Q. If so, the reader waits for readAck messages to arrive from some quorum, and returns the value associated with the minimum tag. This ensures that the returned value was propagated “sufficiently” before completion. Notice that, due to asynchrony, it is possible for case (b) to happen before case (a). In this scenario, the reader proceeds identically as in the case where QV(3) is detected.

Server Protocol. Server s stores the value of the replica v and its associated tag tg. The relays array is used to store sets of processes that relayed to s regarding a read operation. Destinations set D is initialized to set containing all servers from every quorum that contains s.
and it is used for sending relay messages during exchange E2. Each server $s \in S$ expects four types of messages:

1. Upon receiving a $\langle \text{readRequest}, r, \text{read_op} \rangle$ message from exchange E1 of a read operation, the server creates a readRelay message, containing its local information and it broadcasts it in exchange E2 to destinations in $D$ and the requesting reader $r$ (A16:L66-67).

2. Upon receiving a $\langle \text{readRelay}, ts', id', v', r, \text{read_op}, s \rangle$ message from exchange E2, server $s$ compares its local local information with the information enclosed in the message. If the information is “fresh” i.e., $tg < tg'$ where $tg' = \langle ts', id' \rangle$, then $s$ sets its local value and tag to those enclosed in the message (A16:L69-70). In any other case, no updates are taking place. Next, $s$ checks if the received readRelay marks a new read operation by $r$, i.e., $\text{read_op} > \text{operations}[r]$. If this holds, then $s$: (a) sets its local counter for $r$ to the enclosed one, $\text{operations}[r] = \text{read_op}$; and (b) re-initializes the relay set for $r$ to an empty set, $\text{relays}[r] = \emptyset$ (A16:L71-72). It then adds the sender of the readRelay message to the set of servers that informed it regarding the read operation invoked by $r$ (A16:L73-73). Once readRelay messages are received from a full quorum $Q$, $s$ creates a readAck message and sends it to $r$ in E3 (A16:L75-76).

3. Upon receiving a $\langle \text{writeDiscover}, \text{write_op}, w \rangle$ message from E1, server $s$ replies to the sender with a discoverAck message containing its local tag and value pair in E2 (A16:L58-59).

4. Upon receiving a $\langle \text{writeRequest}, ts', id', v', \text{write_op}, w \rangle$ message during E3, server $s$ compares lexicographically its local tag with the received one. If $tg < tg'$, where $tg' = \langle ts', id' \rangle$, then $s$ updates its local information to the information attached in the message and acknowledges the requesting writer with a writeAck message forming E4 (A16:L60-65).
6.3.2 Correctness of ERATO-MW

We first show liveness (termination) and then atomicity (safety).

**Termination.** Liveness is satisfied with respect to our failure model: at least one quorum \( Q \) is non-faulty and each operation waits for messages from a single quorum. Let us consider this in more detail.

**Write Operation.** Writer \( w \) finds the maximum tag by broadcasting a discover message to all servers and waits to collect discoverAck replies from a full quorum of servers (A15:L37-40). Since a full quorum of servers is non-faulty, then at least a full quorum of live servers will collect the discover messages and reply to writer \( w \). Once the maximum timestamp is determined, then writer \( w \) updates its local tag and broadcasts a writeRequest message to all servers. Writer \( w \) then waits to collect writeAck replies from a full quorum of servers before it terminates. Again, at a full quorum of servers will collect the writeRequest messages and will reply to writer \( w \) (A15:L41-46).

**Read Operation.** A read operation of the algorithm ERATO-MW terminates when the reader either (i) collects readAck messages from full quorum of servers or (ii) collects readRelay messages from a full quorum and throughout the iterative procedure it notices \( Q^V(1) \) or \( Q^V(3) \) (A15:L14-25). Case (i) is identical as in Algorithm ERATO and liveness is ensured as reasoned in Section 6.2.2. For case (ii), in the worst case, during the iterative analysis the reader will observe \( Q^V(3) \) once one server remains in one of the intersections of the replying quorum. This is identical when \( Q^V(3) \) is observed in case (i) and the result follows.

Based on the above, any read or write operation collect a sufficient number of messages to terminate, guaranteeing liveness.
Atomicity. We show how algorithm ERATO-MW satisfies atomicity using tags. More precisely, for each execution $\xi$ of ERATO-MW there must exist a partial order $\prec$ on the operations in on the set of completed operations $\Pi$ that satisfy conditions A1, A2, and A3 as given in Definition 2.2.5 in Section 2.2. Let $tg_\pi$ be the value of the tag at the completion of $\pi$ when $\pi$ is a write, and the tag computed as the maximum $tg$ at the completion of a read operation $\pi$. With this, we denote the partial order using tags on operations as follows. For two operations $\pi_1$ and $\pi_2$, when $\pi_1$ is any operation and $\pi_2$ is a write, we let $\pi_1 \prec \pi_2$ if $tg_{\pi_1} < tg_{\pi_2}$. For two operations $\pi_1$ and $\pi_2$, when $\pi_1$ is a write and $\pi_2$ is a read we let $\pi_1 \prec \pi_2$ if $tg_{\pi_1} \leq tg_{\pi_2}$. The rest of the order is established by transitivity and reads with the same timestamps are not ordered. We now state and prove a series of lemmas.

It is easy to see that the $tg$ variable in each server $s$ is monotonically increasing. This leads to the following lemma.

Lemma 6.3.1 In any execution $\xi$ of ERATO-MW, the variable $tg$ maintained by any server $s$ in the system is non-negative and monotonically increasing.

Proof. When server $s$ receives a tag $tg$ then $s$ updates its local tag $tg_s$ if and only if $tg > tg_s$ (A16:L63-64, 69-70). \hfill \Box

Next we show that if a write operation succeeds a read operation, then it writes a value associated with a tag greater than the one returned by the read operation.

Lemma 6.3.2 In any execution $\xi$ of ERATO-MW, if a write $\omega$ writes tag $tg'$ and succeeds a read operation $\rho$ that returns a tag $tg$, i.e., $\rho \rightarrow \omega$, then $tg' > tg$.

Proof. Let $RR$ be the set of servers that belong to quorum $Q_a$ and sent readRelay messages to $\rho$. Let $dAck$ be the set of servers from a quorum $Q_b$ that sent discoverAck messages to $\omega$. Let
$w$Ack be the set of servers from a quorum $Q_c$ that sent writeAck messages to $\omega$ and let $R.A$ be the set of servers from a quorum $Q_d$ that sent readAck messages to $\rho$. It is not necessary that $a \neq b \neq c \neq d$ holds.

Based on the read protocol, the read operation $\rho$ terminates when it either receives (a) readRelay messages from a full quorum $Q$ or (b) readAck messages from a full quorum $Q$ (A15:L6-9).

Case (a), based on the algorithm, during the iterative analysis $\rho$ terminates once it notices $Q\langle 1 \rangle$ or $Q\langle 3 \rangle$ in the messages received from $RR$. If $Q\langle 1 \rangle$ is noticed then the distribution of the timestamps indicates the existence of one and only tag in $Q_a$ and that is, the maximum tag in $Q_a$ at the current iteration. Read $\rho$ returns the value associated with the current maximum tag, $tg$ and terminates. The following writer $\omega$, initially it broadcasts a writeDiscover message to all servers, and it then awaits for “fresh” discoverAck messages from a full quorum $Q_b$, that is, set $d.Ack$ (A15:L37-40). Observe that each of $RR$ and $d.Ack$ sets are from a full quorum of servers, $Q_a$ and $Q_b$ respectively, and so $RR \cap d.Ack \neq \emptyset$. By Lemma 6.3.1, any server $s_k \in RR \cap d.Ack$ has a tag $tg_{s_k}$ s.t. $tg_{s_k} \geq tg$. Since $\omega$ generates a new local tag-value $(tg', v)$ pair in which it assigns the timestamp to be one higher than the one discovered in the maximum tag from set $d.Ack$, it follows that $tg' > tg$. Write operation $\omega$ broadcasts the value to be written associated with $tg'$ in a writeRequest message to all servers and it awaits for writeAck messages from a full quorum $Q_c$ before completion, set $w$Ack (A15:L41-46). Observe that each of $d.Ack$ and $w$Ack sets are from a full quorum of servers, $Q_b$ and $Q_c$ respectively, and so $d.Ack \cap w$Ack $\neq \emptyset$. By Lemma 6.3.1, any server $s_k \in d.Ack \cap w$Ack has a tag $tg_{s_k}$ s.t. $tg_{s_k} \geq tg' > tg$ and the result for this case follows.
Now we examine if \( Q V(3) \) is noticed. When that holds, based on the algorithm, the reader awaits readAck messages from a full quorum \( Q \) of servers, set \( RA \). By lines 10 - 12 of Algorithm 15, it follows that \( \rho \) decides on the minimum tag, \( tg = minTG \), among all the tags in the readAck messages of the set \( RA \) and terminates. Again, \( \omega \), initially it broadcasts a writeDiscover message to all servers, and it then awaits for “fresh” discoverAck messages from a full quorum \( Q_b \), that is, set \( dAck \). Each of \( RA \) and \( dAck \) sets are from a full quorum of servers, \( Q_d \) and \( Q_b \) respectively, and so \( RA \cap dAck \neq \emptyset \). By Lemma 6.3.1, any server \( s_k \in RA \cap dAck \) has a tag \( tg_{s_k} \) s.t. \( tg_{s_k} \geq tg \). Since \( \omega \) generates a new local tag-value \( (tg', v) \) pair in which it assigns the timestamp to be one higher than the one discovered in the maximum tag from set \( dAck \), it follows that \( tg' > tg \). Furthermore, \( \omega \) broadcasts the value to be written associated with \( tg' \) in a writeRequest message to all servers and it awaits for writeAck messages from a full quorum \( Q_c \) before completion, set \( wAck \) (A15:L41-46). Observe that each of \( dAck \) and \( wAck \) sets are from a full quorum of servers, \( Q_b \) and \( Q_c \) respectively, and so \( dAck \cap wAck \neq \emptyset \). By Lemma 6.3.1, any server \( s_k \in dAck \cap wAck \) has a tag \( tg_{s_k} \) s.t. \( tg_{s_k} \geq tg' > tg \) and the result for this case follows.

Lastly, case (b) where read \( \rho \) terminates because it received readAck messages from a full quorum of servers \( Q \), it is the same as in case (a) when reader observers \( Q V(3) \) and the lemma follows. \( \square \)

Next, we reason that if a write operation \( \omega_2 \) succeeds write operation \( \omega_1 \), then \( \omega_2 \) writes a value associated with a tag strictly higher than \( \omega_1 \).

**Lemma 6.3.3** In any execution \( \xi \) of ERATO-MW, if a write \( \omega_1 \) writes tag \( tg_1 \) and precedes a write \( \omega_2 \) that writes tag \( tg_2 \), i.e., \( \omega_1 \rightarrow \omega_2 \), then \( tg_2 > tg_1 \).
**Proof.** Let $wAck_1$ be the set of servers from a full quorum $Q_a$ that send a writeAck message within write operation $\omega_1$. Let $dAck_2$ be the set of servers from a full quorum $Q_b$ (not necessarily different from $Q_a$) that send a discoverAck message within write operation $\omega_2$.

Lemma assumes that $\omega_1$ is complete. By Lemma 6.3.1, we know that if a server $s$ receives a tag $tg$ from a process $p$, then $s$ includes tag $tg'$ s.t. $tg' \geq tg$ in any subsequent message. Thus, servers in $wAck_1$ send a writeAck message within $\omega_1$ with tag at least tag $tg_1$.

Once $\omega_2$ is invoked, it collects discoverAck messages from a full quorum of servers in the set, $dAck_2$ (A15:L37-40). Since $Q_a \subseteq wAck_1$ and $Q_b \subseteq dAck_2$ then $wAck_1 \cap dAck \neq \emptyset$. By Lemma 6.3.1, any server $s_k \in wAck_1 \cap dAck_2$ has a tag $tg_{s_k}$ s.t. $tg_{s_k} \geq tg_1$. Thus, the invoker of $\omega_2$ discovers the maximum tag, $maxTG$, from the tags found in $dAck_2$ s.t. $maxTG \geq tg_{s_k} \geq tg_1$ (A15:L41). It then increases the timestamp from in the maximum tag discovered by one, sets its local tag to that and associates it with its id $i$ and the value $val$ to be written, $local\_tag = (maxTS + 1, i, val)$ (A15:L42). We know that, $local\_tag > maxTG \geq tg_1$, hence $local\_tag > tg_1$.

Lastly, $\omega_2$ attaches its local tag $local\_tag$ in a writeRequest message which it broadcasts to all the servers, and terminates upon receiving writeAck messages from a full quorum of servers. By Lemma 6.3.1, $\omega_2$ receives writeAck messages with a tag $tg_2$ s.t. $tg_2 \geq local\_tag > tg_1$ hence $tg_2 > tg_1$. This completes the Proof of the lemma.

We now show that any read operation that follows a write operation, and it receives readAck messages the servers where each included tag is at least as the one returned by the complete write operation.
Lemma 6.3.4 In any execution $\xi$ of ERATO-MW, if a read operation $\rho$ succeeds a write operation $\omega$ that writes $tg$ and $v$, i.e., $\omega \rightarrow \rho$, and receives readAck messages from a quorum $Q$ of servers, set $RA$, then each $s \in RA$ sends a readAck message to $\rho$ with a tag $tg_s$ s.t. $tg_s \geq tg$.

Proof. Let $wAck$ be the set of servers from a quorum $Q_a$ that send a writeAck message to $\omega$, let $RelaySet$ be the set of servers from a quorum $Q_b$ that sent readRelay messages to server $s$, and let $RA$ be the set of servers from a quorum $Q_c$ that send a readAck message to $\rho$. Notice that it is not necessary that $a \neq b \neq c$ holds.

Write operation $\omega$ is completed. By Lemma 6.3.1, if a server $s$ receives a tag $tg$ from a process $p$ at some time $t$, then $s$ attaches a tag $tg'$ s.t. $tg' \geq ts$ in any message sent at any time $t' \geq t$. Thus, every server in $wAck$, sent a writeAck message to $\omega$ with a tag greater or equal to $tg$. Hence, every server $s \in wAck$ has a tag $tg_s \geq tg$. Let us now examine a tag $tg_s$ that server $s$ sends to read operation $\rho$.

Before server $s$ sends a readAck message to $\rho$, it must receive readRelay messages from a full quorum $Q_b$ of servers, $RelaySet$ (A16:L75-76). Since both $wAck$ and $RelaySet$ contain messages from a full quorum of servers, and by definition, any two quorums have a non-empty intersection, then $wAck \cap RelaySet \neq \emptyset$. By Lemma 6.3.1, any server $s_x \in AAck \cap RelaySet$ has a tag $tg_{s_x}$ s.t. $tg_{s_x} \geq tg$. Since server $s_x \in RelaySet$ and from the algorithm, server’s $s$ tag is always updated to the highest tag it noticed (A16:L69-70), then when server $s$ receives the message from $s_x$, it will update its tag $tg_s$ s.t. $tg_s \geq tg_{s_x}$. Server $s$ creates a readAck message where it encloses its local tag and its local value, $(tg_s, v_s)$ (A16:L76). Each $s \in RA$. 

181
sends a readAck to \( \rho \) with a tag \( tg_s \) s.t. \( tg_s \geq tg_x \geq tg \). Thus, \( tg_s \geq tg \), and the lemma follows. \( \square \)

Next we show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

**Lemma 6.3.5** In any execution \( \xi \) of ERATO-MW, if a read \( \rho \) succeeds a write operation \( \omega \) that writes tag \( tg \), i.e. \( \omega \rightarrow \rho \), and returns a tag \( tg' \), then \( tg' \geq tg \).

**Proof.** A read operation \( \rho \) terminates when it either receives (a) readRelay messages from a full quorum \( Q \) or (b) readAck messages from a full quorum \( Q \) (A15:L6-9).

We first examine case (b). Let’s suppose that \( \rho \) receives readAck messages from a full quorum \( Q \) of servers, \( RA \). By lines 10 - 12, it follows that \( \rho \) decides on the minimum tag, \( tg' = minTG \), among all the tag in the readAck messages of the set \( RA \). From Lemma 6.3.4, \( minTG \geq tg \) holds, where \( tg \) is the tag written by the last complete write operation \( \omega \). Then \( tg' = minTG \geq tg \) also holds. Thus, \( tg' \geq tg \).

Now we examine case (a). Case (a) is an iterative procedure that terminates when the reader notices either (i) \( QV(1) \) or (iii) \( QV(3) \). When \( QV(2) \) is observed then it is the case where the write associated with the maximum tag is not yet complete, thus we proceed to the next iteration to discover the latest potentially complete write. This, by removing all the servers with the maximum tag from \( Q \) and repeating the analysis. If no iteration was interrupted because of \( QV(1) \) then eventually \( QV(3) \) will be noticed, when a single server remains in some intersection of \( Q \) (A15:L14-25).

Let \( wAck \) be the set of servers from a quorum \( Q_a \) that send a writeAck message to \( \omega \). Since the write operation \( \omega \), that wrote value \( v \) associated with tag \( tg \) is complete, and by
monotonicity of tags in servers (Lemma 6.3.1), then at least a quorum \( Q_a \) of servers has a tag \( t g_a \) s.t. \( t g_a \geq t g \).

Let’s suppose that \( \rho \) receives readRelay messages from a full quorum \( Q_b \) of servers, \( RR \). Since both \( wA\ck \) and \( RR \) contain messages from a full quorum of servers, quorums \( Q_a \) and \( Q_b \), and by definition any two quorums have a non-empty intersection, then \( wA\ck \cap RR \neq \emptyset \). Since every server in \( wA\ck \) has a tag \( t g_a \geq t g \) then any server \( s_x \in wA\ck \cap RR \) has a tag \( t g_{s_x} \) s.t. \( t g_{s_x} \geq t g_a \geq t g \).

Assume by contradiction that at the \( i^{th} \) iteration \( \rho \) noticed \( QV(1) \) in \( RR \) and returned a tag \( t g' \) s.t. \( t g' < t g \). Since every server \( s_x \in wA\ck \cap RR \) has a tag \( t g_{s_x} \) s.t. \( t g_{s_x} \geq t g \) and since \( QV(1) \) returned a tag \( t g' \) s.t. \( t g' < t g \), then it must be the case that none of the servers in \( wA\ck \cap RR \) were participating in \( QV(1) \). Therefore, it must be the case that all servers in \( wA\ck \cap RR \) were removed during the analysis at a previous iteration \( k \), s.t. \( k < i \). However, we know that the iterative procedure, in the worst case, it will notice \( QV(3) \) once a single server remains in an intersection of the quorum we examine. This contradicts the fact that all servers in \( wA\ck \cap RR \) were removed from \( Q_a \) during the analysis. Thus, if \( QV(1) \) is noticed, then the distribution of the tags yielded the existence of one and only tag, the current maximum tag. At least one server \( s_x \) from \( wA\ck \cap RR \) will participate in \( QV(1) \), hence \( \rho \) will return a tag \( t g' \) s.t. \( t g' = t g_{s_x} \geq t g \).

Lastly, when \( QV(3) \) is noticed during the iterative procedure then \( \rho \) waits for readAck messages from a full quorum \( Q \) before termination, (A15:L20-23), proceeds identically as in case (b) and the lemma follows.

In the following three lemmas we show that if a read operation \( \rho_2 \) succeeds a read \( \rho_1 \), then \( \rho_2 \) returns a value associated with a timestamp \( t s_2 \) s.t. \( t s_2 \geq t s_1 \).
Lemma 6.3.6 In any execution $\xi$ of ERATO-MW, if $\rho_1$ and $\rho_2$ are two semi-fast read operations, take 3 exchanges to complete, such that $\rho_1$ precedes $\rho_2$, i.e., $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns the value for tag $tg_1$, then $\rho_2$ returns the value for tag $tg_2 \geq tg_1$.

Proof. Let the two operations $\rho_1$ and $\rho_2$ be invoked by processes with identifiers $r_1$ and $r_2$ respectively (not necessarily different). Also, let $RA_1$ and $RA_2$ be the sets of servers from quorums $Q_a$ and $Q_b$ (not necessarily different) that sent a readAck message to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$.

Assume by contradiction that read operations $\rho_1$ and $\rho_2$ exist such that $\rho_2$ succeeds $\rho_1$, i.e., $\rho_1 \rightarrow \rho_2$, and the operation $\rho_2$ returns a tag $tg_2$ that is smaller than the $tg_1$ returned by $\rho_1$, i.e., $tg_2 < tg_1$. Based on the algorithm, $\rho_2$ returns a tag $tg_2$ that is smaller than the minimum tag received by $\rho_1$, i.e., $tg_1$, if $\rho_2$ obtains $tg_2$ and $v$ in the readAck message of some server $s_x \in RA_2$, and $tg_2$ is the minimum tag received by $\rho_2$.

Let us examine if $s_x$ sends a readAck message to $\rho_1$ with tag $tg_x$, i.e., $s_x \in RA_1$. By Lemma 6.3.1, and since $\rho_1 \rightarrow \rho_2$, then it must be the case that $tg_x \leq tg_2$. According to our assumption $tg_1 > tg_2$, and since $tg_1$ is the smallest tag sent to $\rho_1$ by any server in $RA_1$, then it follows that $r_1$ does not receive the readAck message from $s_x$, and hence $s_x \notin RA_1$.

Now let us examine the actions of the server $s_x$. From the algorithm, server $s_x$ collects readRelay messages from a full quorum $Q_c$ of servers before sending a readAck message to $\rho_2$ (A16:L67-67). Let $RRSet_{s_x}$ be the set of servers that belong to quorum $Q_c$ and sent readRelay message to $s_x$. Since, both $RRSet_{s_x}$ and $RA_1$ contain messages from full quorums, $Q_c$ and $Q_a$, and since any two quorums have a non-empty intersection, then it follows that $RRSet_{s_x} \cap RA_1 \neq \emptyset$.  

184
Thus there exists a server $s_i \in RRSet_{s_x} \cap RA_1$, that sent (i) a readAck to $\rho_1$, and (ii) a readRelay to $s_x$ during $\rho_2$. Note that $s_i$ sends a readRelay for $\rho_2$ only after it receives a read request from $\rho_2$. Since $\rho_1 \rightarrow \rho_2$, then it follows that $s_i$ sent the readAck to $\rho_1$ before sending the readRelay to $s_x$. By Lemma 6.3.1, if $s_i$ attaches a tag $tg_{s_i}$ in the readAck to $\rho_1$, then $s_i$ attaches a tag $tg'_{s_i}$ in the readRelay message to $s_x$, such that $tg'_{s_i} \geq tg_{s_i}$. Since $tg_1$ is the minimum tag received by $\rho_1$, then $tg_{s_i} \geq tg_1$, and hence $tg'_{s_i} \geq tg_1$ as well. By Lemma 6.3.1, and since $s_x$ receives the readRelay message from $s_i$ before sending a readAck to $\rho_2$, it follows that $s_x$ sends a tag $tg_2$ to $\rho_2$ s.t. $tg_2 \geq tg'_{s_i} \geq tg_1$. Thus, $tg_2 \geq tg_1$ and this contradicts our initial assumption.

Lemma 6.3.7 In any execution $\xi$ of ERATO-MW, if $\rho_1$ and $\rho_2$ are two fast read operations, take 2 exchanges to complete, such that $\rho_1$ precedes $\rho_2$, i.e., $\rho_1 \rightarrow \rho_2$, and $\rho_1$ returns the value for tag $tg_1$, then $\rho_2$ returns the value for tag $tg_2 \geq tg_1$.

Proof. Let the two operations $\rho_1$ and $\rho_2$ be invoked by processes with identifiers $r_1$ and $r_2$ respectively (not necessarily different). Let $RR_1$ and $RR_2$ be the quorums (not necessarily different) that sent a readRelay message to $r_1$ and $r_2$ during $\rho_1$ and $\rho_2$ respectively.

The algorithm terminates in two communication exchanges when a read operation $\rho$ receives readRelay messages from a full quorum $Q$ and based on the distribution of the tags during the $i^{th}$ iteration of the analysis, $i \geq 1$, it notices $QV(1)$.

Observe that if there exists a server $s_k \in RR_1$, that replies with a tag $tg_{s_k}$ s.t. $tg_{s_k} < tg_1$ then $\rho_1$ wouldn’t be able to notice $QV(1)$ and return $tg_1$. Thus, since $QV(1)$ is noticed during the $i^{th}$ iteration then it is known that all the servers in $RR_1$ replied to $\rho_1$ with a tag $tg_{s}$ s.t.
\(tg_s \geq tg_1\). This is clear since every server \(s_x\) that was removed during the iterative analysis at iteration \(j\) s.t. \(j > i\), server \(s_x\) holds a tag \(tg_{s_x} \geq tg_s\).

Since by definition, any two quorums have a non-empty intersection it follows that \(RR_1 \cap RR_2 \neq \emptyset\). From that and by Lemma 6.3.1, then every server \(s_x \in RR_1 \cap RR_2\) has a tag \(tg'\) such that \(tg' \geq tg_1\). When \(\rho_2\) notices \(QV(1)\) in \(RR_2\) at the \(m^{th}\) iteration of the analysis, \(m \geq 1\), we know that \(QV(1)\) consists tags that come from the set \(RR_1 \cap RR_2\). Notice that if \(RR_1 \cap RR_2 = \emptyset\) holds at iteration \(m\), then it means that the algorithm would have stopped at an earlier iteration when either \(RR_1 \cap RR_2 \neq \emptyset\) or \(|RR_1 \cap RR_2| = 1\) holds.

Since the distribution of the tags during \(m^{th}\) iteration indicates the existence of one and only tag and since servers from \(RR_1 \cap RR_2\) participate then \(\rho\) returns a value associated with \(tg_2\) s.t. \(tg_2 \geq tg' \geq tg_1\) and the lemma follows.

\[\square\]

**Lemma 6.3.8** In any execution \(\xi\) of ERATO-MW, if \(\rho_1\) and \(\rho_2\) are two read operations s.t. \(\rho_1\) precedes \(\rho_2\), i.e., \(\rho_1 \rightarrow \rho_2\), and \(\rho_1\) returns tag \(tg_1\), then \(\rho_2\) returns a tag \(tg_2\) s.t. \(tg_2 \geq tg_1\).

**Proof.** We are interested to examine the cases where one of the read operation is *fast* and the other is *semifast*. In particular, cases (i) \(\rho_1 \rightarrow \rho_2\) and \(\rho_1\) is *semifast* and \(\rho_2\) is *fast* and (ii) \(\rho_1 \rightarrow \rho_2\) and \(\rho_1\) is *fast* and \(\rho_2\) is *semifast*.

Let the two operations \(\rho_1\) and \(\rho_2\) be invoked by processes with identifiers \(r_1\) and \(r_2\) respectively (not necessarily different). Also, let \(RR_1, RA_1\) and \(RR_2, RA_2\) be the sets of servers from full quorums (not necessarily different) that sent a *readRelay* and *readAck* message to \(\rho_1\) and \(\rho_2\) respectively.

We start with case (i). Since read operation \(\rho_1\) is *semifast*, then based on the algorithm, the tag \(tg_1\) that is returned it is also the *minimum* tag noticed in \(RA_1\). Before a server \(s\) sends
readAck messages to $\rho_1$ (that form $RA_1$), it must receive readRelay messages from a full quorum of servers. Thus, by Lemma 6.3.1 monotonicity of the tags at the servers we know that the minimum tag that a full quorum has by the end of $\rho_1$ is $tg_1$. Read operation $\rho_2$ receives readRelay messages from a full quorum of servers, $RR_2$. By definition of quorums, since both $RA_1$ and $RR_2$ are from a full quorum of servers then it follows that $RA_1 \cap RR_2 \neq \emptyset$. Thus every server $s_x \in RA_1 \cap RR_2$ holds a tag $tg'$ s.t. $tg' \geq tg_1$.

For $\rho_2$ to notice $QV(1)$ in $RR_2$ at the $m^{th}$ iteration of the analysis, $m \geq 1$, it means that in $QV(1)$ participate servers that belong to $RR_1 \cap RR_2$. Notice that if $RR_1 \cap RR_2 = \emptyset$ holds at iteration $m$, then it means that the algorithm would have stopped at an earlier iteration when either $RR_1 \cap RR_2 \neq \emptyset$ or $|RR_1 \cap RR_2| = 1$ holds. Since the distribution of the tags during $m^{th}$ iteration indicates the existence of one and only tag and since servers from $RR_1 \cap RR_2$ participate then $\rho$ returns a value associated with $tg_2$ s.t. $tg_2 \geq tg' \geq tg_1$ and the case follows. The rest of the cases are proved in Lemmas 6.3.6 and 6.3.7.

We now examine case (ii). Since $\rho_1$ is fast, it follows that it has noticed $QV(1)$ in $RR_1$. If $QV(1)$ was noticed at the $m^{th}$ iteration of the analysis, $m \geq 1$, and $\rho_1$ returned a value associated with maximum tag during $m^{th}$ iteration, $tg_1$, then by the completion of $\rho_1$ a full quorum has a tag $tg'$ s.t. $tg' \geq tg_1$. Now, since read operation $\rho_2$ is semifast, then based on the algorithm, the tag $tg_2$ that is returned it is the minimum tag noticed in $RA_2$. Before a server $s$ sends readAck messages to $\rho_2$ (that form $RA_2$), it must receive readRelay messages from a full quorum of servers, $RelaySet$. By Lemma 6.3.1 monotonicity of the tags at the servers and $RR_1 \cap RelaySet \neq \emptyset$, then every server in $RA_2$ has a tag $tg_2$ s.t. $tg_2 \geq tg' \geq tg_1$ and the case follows. The rest of the cases are proved in Lemmas 6.3.6 and 6.3.7.

We now show the correctness of algorithm $ERATO$-MW. 

187
**Theorem 6.3.9** Algorithm ERATO-MW implements an atomic MWMR object.

**Proof.** We use the lemmas stated above and the operations order definition to reason about each of the three *atomicity* conditions A1, A2 and A3 as given in Definition 2.2.5.

**A1** For any $\pi_1, \pi_2 \in \Pi$ such that $\pi_1 \rightarrow \pi_2$, it cannot be that $\pi_2 \prec \pi_1$.

If both $\pi_1$ and $\pi_2$ are writes and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 6.3.3 it follows that $tg_{\pi_2} > tg_{\pi_1}$. From the definition of order $\prec$ we have $\pi_1 \prec \pi_2$. When $\pi_1$ is a write, $\pi_2$ a read and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 6.3.5 it follows that $tg_{\pi_2} \geq tg_{\pi_1}$. By definition $\pi_1 \prec \pi_2$ holds. If $\pi_1$ is a read, $\pi_2$ is a write and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 6.3.2 it follows that $\pi_2$ returns a tag $tg_{\pi_2}$ s.t. $tg_{\pi_2} > tg_{\pi_1}$. By the order definition $\pi_1 \prec \pi_2$ is satisfied. If both $\pi_1$ and $\pi_2$ are reads and $\pi_1 \rightarrow \pi_2$ holds, then from Lemma 6.3.8 it follows that $tg_{\pi_2} \geq tg_{\pi_1}$. If $tg_{\pi_2} > tg_{\pi_1}$, then by the ordering definition $\pi_1 \prec \pi_2$ holds. When $tg_{\pi_2} = tg_{\pi_1}$ then the ordering is not defined, thus it cannot be that $\pi_2 \prec \pi_1$.

**A2** For any write $\omega \in \Pi$ and any operation $\pi \in \Pi$, then either $\omega \prec \pi$ or $\pi \prec \omega$.

If $tg_{\omega} > tg_{\pi}$, then $\pi \prec \omega$ follows directly. Similarly, if $tg_{\omega} < tg_{\pi}$ holds, then it follows that $\omega \prec \pi$. When $ts_{\omega} = ts_{\pi}$ holds, then because all writer tags are unique (each server increments timestamps monotonically, and the server ids disambiguate among servers) $\pi$ can only be a read. Since $\pi$ is a read and the distribution of the tag written by $\omega$ satisfies either $QV(1)$ or $QV(3)$, it follows that $\omega \prec \pi$.

**A3** Every read operation returns the value of the last write preceding it according to $\prec$ (or the initial value if there is no such write).

Let $\omega$ be the last write preceding read $\rho$. From our definition it follows that $tg_{\rho} \geq tg_{\omega}$. If $tg_{\rho} = tg_{\omega}$, then $\rho$ returned a value written by $\omega$ in some servers in a quorum $Q$. Read $\rho$ either
was fast and during the iterative analysis it noticed a distribution of the tags in $Q$ that satisfied $QV(1)$ or $\rho$ was slow and waited for readAck messages from a full quorum $Q$. In the latter, the intersection properties of quorums ensure that $\omega$ was the last complete write. If $tg_{\rho} > tg_{\omega}$ holds, it must be the case that there is a write $\omega'$ that wrote $tg_{\rho}$ and by definition it must hold that $\omega \prec \omega' \prec \rho$. Thus, $\omega$ is not the preceding write and this cannot be the case. Lastly, if $tg_{\rho} = 0$, no preceding writes exist, and $\rho$ returns the initial value.

6.3.3 Performance of ERATO-MW

By inspection of the code, write operations of algorithm ERATO-MW take 4 exchanges and read operations take either 2 or 3 exchanges. The (worst case) message complexity of write operations is $4|S|$ and of read operations is $|S|^2 + 2|S|$, as follows from the structure of the algorithm. We now provide additional details.

**Operation Latency.** We study the operation latency, in terms of the number of communication exchanges required by each operation.

*Write operation latency:* According to algorithm ERATO-MW, writer $w$ sends discover messages to all servers in exchange $E1$ and waits for discoverAck messages from a full quorum of servers in $E2$. Once the discoverAck messages are received from $E2$, then writer $w$ broadcasts a writeRequest message to all servers in $E3$. It then waits for writeAck messages from a full quorum of servers from $E4$. No further communication is required and the write operation terminates. Thus a write operation consists of 4 communication exchanges.

*Read operation latency:* A reader sends a readRequest message to all the servers in the first communication exchange $E1$. Once the servers receive the readRequest message they broadcast a readRelay message to all servers and the reader in exchange $E2$. The reader can terminate
at the end of the $E_2$ if it receives readRelay messages and based on the distribution of the tags through the iterative procedure it notices $QV(1)$ or $QV(3)$. If not, the operation goes into the third exchange $E_3$. Thus read operations terminate after either 2 or 3 communication exchanges.

**Message Complexity.** We proceed with the analysis of message complexity.

*Write operation:* Write operations in algorithm ERATO-MW take 4 communication exchanges. The first and the third exchanges, $E_1$ and $E_3$, occur when a writer sends discover and writeRequest messages respectively to all servers in $S$. The second and fourth exchanges, $E_2$ and $E_4$, occur when servers in $S$ send discoverAck and writeAck messages respectively back to the writer. Thus $4|S|$ messages are exchanged in any write operation.

*Read operation:* The structure of the read protocol of ERATO-MW is similar as in ERATO, thus, as reasoned in Section 6.2.3, $|S|^2 + 3|S|$ messages are exchanged during a read operation.

**Computational Complexity.** Algorithm ERATO-MW performs a modest amount of local computation, resulting in negligible computation overhead.

### 6.4 Experimental Evaluation

In order to explore Research Question 6.3 we did a comparative study of our algorithms with existing comparable ones. In particular, the following SWMR algorithms: ERATO (Section 6.2), ABD [10], OHSAM (Section 4.2), and SLIQ [32], and the corresponding MWMR algorithms: ERATO-MW (Section 6.3), ABD-MW [55], OHMAM (Section 4.3), and CWF [30] were simulated. For comparison purposes we implemented a benchmark, called LB, that mimics the minimum message requirements. In particular, benchmark LB takes two communication exchanges for a read and a write operation, and it neither performs any computation nor ensures consistency.
Experimentation Setup. The experimentation setup is similar to the one developed for the empirical evaluation of algorithms OHSAM and OHMAM (see Section 4.5) and of algorithms CCHYBRID and OHFAST (see Section 5.3). In particular, minor modifications took place on the Star topology (see Fig. 8) in order to better represent an environment where the servers are placed in a close proximity, are well-connected, and communicate via high capacity links, e.g., a datacenter. In both topologies readers and writer(s) are located uniformly with respect to the routers in the system.
**Scenarios.** Measurements of the performance involves multiple execution scenarios. In brief, the scenarios were designed to test (i) the scalability of the algorithms as the number of readers and servers increases; (ii) the contention effect on efficiency, by running different concurrency scenarios; and (iii) the relation of the efficiency with the topology of the network that we use. Algorithms are evaluated with matrix quorums (unions of rows and columns). Scenarios are similar to the ones developed for the experimental evaluation of algorithms OHSAM and OHMAM. Thus, for additional details we direct the interested reader to Section 4.5.

![Figure 9: Simulation results for the SWMR setting.](image-url)
Figure 10: Simulation results for the MWMR setting.

Results. Generally the new algorithms outperform the competition in most scenarios. A closer examination yields the following observations:

Scalability: Increased number of participating readers, writers, and servers increases latency in both settings. Observe Fig. 9(a),(b) for the SWMR algorithms and Fig. 10(e),(f) for the MWMR algorithms. Not surprisingly, latency is better for smaller numbers of readers, writers, and servers. However, the relative performance of the algorithms remains the same.
**Contention:** The efficiency of the algorithms is examined under different concurrency schemes.

We notice that in the *stochastic* scheme reads complete faster than in the *fixed* scheme – Fig. 9(b) and 9(c) for the SWMR and Fig. 10(f) and 10(g) for the MWMR setting. This outcome is expected as the *fixed* scheme causes congestion. For the *stochastic* scheme the invocation time intervals are distributed uniformly (randomness prevents the operations from being invoked simultaneously), and this reduces congestion in the network and improves latency.

**Topology:** Topology substantially impacts performance and the behavior of the algorithms. This can be seen in Figures 9(b) and 9(d) for the SWMR setting, and Figures 10(f) and 10(h) for the MWMR setting. The results show clearly that the proposed algorithms outperform the competition in the *Star* topology, where servers are well-connected using high bandwidth links.
Chapter 7

Conclusion

The overall objective of this dissertation was to devise new algorithmic solutions that lead to latency-efficient survivable distributed storage implementations with provable performance and correctness guarantees. We examined asynchronous, crash-prone, message-passing systems with static participation and we considered both the SWMR and the MWMR settings. Latency of an operation is determined by the communication delays and the computation time. Computation time accounts for all local computation within an operation. Communication delays are assessed in terms of the number of rounds (or exchanges) required by each operation and is a factor that typically dominates the performance of message-passing systems.

In particular, we focused on the gap between one-round and two-rounds algorithms and we presented a new family of atomic read/write shared register implementations where operations do not necessarily require complete communication rounds to terminate, i.e., operations are able to complete in “one-and-a-half-rounds”. To achieve that, on top of the traditional client-to-server communication pattern we introduced server-to-server communication in the system. We elaborated on the inherent limitations that such a technique may impose on the distributed
system and we performed empirical studies of the proposed algorithms. The objective was to compare the efficiency of different algorithms and understand how the analytical results are reflected in practical efficiency. We now identify future directions in this research area.

7.1 Future Directions

This dissertation investigated latency-efficient algorithms for consistent and fault-tolerant distributed storage implementations. In particular, the various read/write atomic shared register implementations presented deal with systems that tolerate crash failures, assume fixed participation in terms of the known universe of processors and reliable communication. However, real systems may experience variable participation, i.e., dynamic participation, message alterations and arbitrary process failures i.e., Byzantine failures. These environmental parameters introduce new challenges in devising efficient, in terms of communication and computation latency, atomic shared read/write register implementations. In the sequel, we present research directions that build on the results presented in this thesis potentially leading to efficient atomic memory implementations in more hostile environments.

7.1.1 Extensions of the Current Work

Previous works considered only client-server communication round-trips. Our work shows that atomic operations do not necessarily require complete communication round trips, i.e., operations can terminate in one-and-a-half-rounds, and that is achieved by introducing server-to-server communication in the system. The key idea is to let two of the communication exchanges happen between clients and servers, and one among the servers. The development of
such implementations, that may use server-to-server communication, opened new possibilities for practical and applicable distributed atomic storage systems.

**Planetary Scale Implementations**

In order to understand how the analytical efficiency bounds of the algorithms are reflected in the performance of the algorithms in practical settings, in Sections 4.5, 5.3 and 6.4, we presented the comparisons of the simulated performance of the proposed algorithms and the relevant existing algorithms. The simulation results suggest that in practical settings, such as datacenters where the servers are placed in a close proximity, are well-connected, and communicate via high capacity links, the communication overhead is not prohibitive.

To further improve our understanding, in addition to simulations, an immediate future step would be to perform full scale cloud-based experimental evaluations. For the purpose of such implementation we may utilize overlay network infrastructures, like Planetlab [1] and cloud infrastructures such as the ones offered from AWS [2] and Google Cloud [3]. Results obtained from such experiments will yield valuable observations in realistic settings and may help us evaluate the efficiency of the system in real environmental conditions.

Lastly, another aspect of this direction would be to try optimizing the network deployment. The objective is to minimize the communication costs to contact a sufficient subset of replica servers. An example of this direction is the work of Sonderegger [64] where the author strove to optimize the network deployment by examining various parameters of the system such as the distance between the members in a quorum.
Byzantine Failures

Here, we are interested in situations where a processor might exhibit malicious behavior. In particular when processors are subject to Byzantine Failures [51]. Abraham et al. [5] showed a tight lower bound on the communication efficiency of write operations. More in detail, authors showed that in the SWMR model, a write operation needs two rounds when the number of replica servers $|S|$ that participate in the system is less than $|S| \leq 2f + 2b$; otherwise a single round is sufficient. Here $f$ is 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 et al. [39] studied the operation latency of read operations in the SWMR environment under Byzantine failures. In particular, authors showed that when less than $2f + 2b$ replica hosts participate in the system, then read operations that take two rounds to complete are necessary even for the implementation of a safe register. Additionally, authors 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. Both results show tight lower bounds on the operation latency of atomic register implementations.

Guerraoui, Levi and Vucolic [38], showed that “lucky” read/write operations may be fast when $2f + b + 1$ register replicas are used. In that context, 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 Vukolic [40]. In this work, the authors introduce a new family of quorum systems, called Refined Quorum Systems, that allow some fast operations under Byzantine failures. To achieve fastness, the solutions rely on a synchronization assumption.
that requires each operation to wait for a predefined *timeout interval*. For a more detailed discussion, we direct the interested reader to [17].

Building upon our work, it would be interesting to investigate if the one-and-a-half round read protocol of OHSAM (Section 4.2) can by adopted by SWMR algorithms that tolerate Byzantine failures when the participation of replica servers in the system is such that \(|S| \geq 2f + 2b\). In particular, we want to examine possible advantages that may be hidden in the server-to-server communication, and explore the feasibility of obtaining an implementation that tolerates Byzantine failures.

Moreover, it will be interesting to explore this model also in *quorum-based* implementations. Recall that *Quorums Views* (c.f. Section 6.1) examine the distribution of a value within a quorum of replica servers and that quorum’s intersections. If the system tolerates Byzantine failures, then if a malicious server in a single intersection reports a faulty value, two different read operations may witness different values in the same set of replica hosts. Thus, the utilization of quorum views may result in violation of atomicity and direct application of quorum views in an environment that suffers from Byzantine failures is impossible. A possible way to avoid this problem is to combine quorum views with special families of quorum constructions, i.e., like Refined Quorum Systems presented in [40] or Byzantine Quorum Systems presented in [56].

Additionally, it would be interesting to investigate whether we can use Quorum Views and the three exchange read protocol presented in Sections 4.2 and 6.2 to take advantage of the server-to-server communication, and explore the feasibility of obtaining an atomic register implementation that tolerates Byzantine failures. We will seek the properties and possible replica deployment strategies that may result in efficient implementations.
Hybrid Atomic Implementations

Here, we are interested in exploring the feasibility of devising “hybrid” atomic read/write shared object implementations. By this we mean that the client will be offered more than one specific way to perform a read operation. In particular, we want to examine the possibility of combining current existing techniques and read protocols together in such a way that it will be up to the clients to choose the type of the protocol to use in a read operation.

As an example, assume that and we were able to merge the three exchanges read protocol of OHSAM (Section 4.2) with the classical four exchanges read protocol of ABD [10]. Since clients may be aware of a possible congestion in the network, they can avoid “forcing” server-to-server communication in the network by invoking a classical four exchanges read operation of ABD. On the other hand, when the traffic is light, readers can be greedy and invoke a read operation using OHSAM. The big challenge here is to merge more than two algorithmic solutions while maintaining atomic.

Reductions in the quality of service are typically expected when network is congested or its capacity is limited. Congestion usually results in queueing delay or packet loss. This motivates further research and to the best of our knowledge current solutions do not address this problem.

The combination of the three exchanges read protocol of OHSAM (Section 4.2) along with the use of client-side tools, Quorum Views [32] opened new possibilities for practical and applicable quorum-based distributed atomic storage systems. We now present possible future extensions of our work.
7.1.2 Theoretical Bounds

We are interested in showing possible tight bounds for atomic object implementations under different failure models and various assumptions.

Bounds Under Crash Failures

In Section 4.3, we presented a MWMR atomic implementation, called OhMAM, where read operations take three communication exchanges and writes four exchanges to complete. As discussed in Section 3.2.1, Georgiou et al. [33] showed that semifast implementations are impossible in the MWMR setting. Recall that in a semifast implementation either all reads are fast or all the write operations fast. This is expressed in the following theorem.

**Theorem 7.1.1 ([33])** If $|W| \geq 2$, $|R| \geq 2$, and the number of server crashes $f \geq 1$, then semifast atomic register implementation is impossible.

A natural question arises regarding the possibility of devising MWMR implementations where all operations can terminate in three communication exchanges. In particular, we are interested in the following research questions.

**Research Question 7.1** Is it feasible to devise an atomic read/write shared objects implementation for the asynchronous, crash-prone, message-passing, MWMR setting with unbounded participation, such that all operations terminate in *one-and-a-half* round, i.e., take three communication exchanges to complete?

**Research Question 7.2** If the answer to 7.1 is negative, can we argue that it is not possible to obtain an atomic read/write register implementation, where all operations perform three communication exchanges, when $|W| = |R| = 2$, $|S| \geq 3$ and $f = 1$?
If the answer to research question 7.1 is positive, then such an implementation will match the tight bounds of [33] and will be optimal in terms of communication exchanges. On the other hand, if the answer to research question 7.2 is positive, then algorithm OHMAM will match the new bounds and will be optimal in terms of communication exchanges.

**Zero-Delay Operations**

For the static setting, it is also interesting to investigate the possibility of devising consistent implementations with zero-delay operations. That is, where some operations are able to complete without additional communication, perhaps only relying on the knowledge obtained through prior communications. In particular, a study on combining heuristics regarding the status of the latest complete write operation, and/or the current reader, writer, and server participation in the system, along with the server-to-server communication pattern, could possibly help us to understand the inherit limitations of systems utilizing the above techniques. Furthermore, any positive outcome from such a study would assist us on providing specific conditions under which zero-delay operations may be feasible in the asynchronous, message-passing, crash-prone setting. In case this is not feasible, i.e., no atomic implementations can utilize zero-delay operations in this setting, it will still be interesting to understand the possibility of obtaining such implementations either for notions of consistency weaker than atomicity, e.g., eventual consistency, or by weakening the power of adversity, e.g., tolerate a smaller number of failures, or by weakening the strength of the system, e.g., assume a fully or partially synchronous setting. An example of this direction is the work of Chandra et al. [15] that assumes a partially synchronous system and uses synchronized local clocks. Such solutions are particularly interesting for applications that are either read or write dominated.
7.1.3 Dynamic Systems

Until now we considered atomic implementations for the static setting where the set of participating nodes is fixed, and each node may know the identity of all participants. Crashes (or voluntary departures) may remove nodes from the system. In the dynamic system the set of nodes may be unbounded, and the set of participating nodes may completely change over time as the result of crashes, departures, and new nodes joining. The failure models considered in dynamic settings are more complicated, given the systems that substantially evolve over time.

We can study the implications of dynamic systems on atomic register implementations and specify lower bounds on the operation latency of such implementations. In particular, we aim to specify the exact communication demands (i.e., amount of communication exchanges required by each read/write operation) in dynamic systems. Building on these results, if feasible, we will then try to develop optimal, in terms of operation latency, dynamic memory services.

As a first step, we can focus in identifying the factors that differentiate static from dynamic environments. This, will be helpful to determine what characteristics have an impact on the communication demands of algorithms designed for dynamic systems. We want to provide an answer to the following:

**Research Question 7.3** What are the characteristics of a dynamic environment that may affect the communication demands of a read/write atomic register implementation?

Perhaps, the main characteristic that differentiates dynamic from static systems, is that participants are allowed to join and fail/leave the service at any point during the execution. This capability, improves the scalability and longevity of systems. Participant additions and removals lead eventually to the need to reconfigure the set of replica hosts to include or exclude
the new or departed participants. Algorithms like [55, 24, 36] separate the join and reconfig-
inguration protocols. Here one needs to examine the possibility of having fast read or write
operations when these operations are concurrent with a join/removal. Recently, [5] combined
the two protocols and suggested the tracing of the new system view (service participation)
whenever an addition or removal occurred in the system. The categorization boils down to the
replica organization that each algorithm utilizes:

- **Voting:** participants need to know the replica hosts, and

- **Quorums:** Participants need to know the replica hosts and replica organization.

Voting techniques eliminate the necessity of having a dedicated entity to decide and prop-
agate 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 in-
troduce 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**

First, we can examine the incorporation of voting strategies to obtain an atomic register
implementation. In particular, we are interested in the following:
**Research Question 7.4** What is the minimum number of communication exchanges required for a read or write operation when it is invoked concurrently with a join/remove operation?

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. Each operation (read or write) 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 guarantees 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, either SWMR or MWMR. In the SWMR setting, by well formedness, only a single write operation (and thus, a single value) may be in progress by the sole writer. 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” on how many new configurations it encounter along with the value to be written. Potentially, this could help read operations to discover the latest configuration in fewer communication exchanges.

In the MWMR setting multiple writers may perform concurrent write operations, thus, discovery of the latest replica host configuration becomes even more challenging. Communication latency of such implementations can benefit from (i) relaxing the failure model (e.g., \( f < \frac{|S|}{c} \) for a constant \( c > 2 \)) and/or (ii) restricting the number of participants.
Utilizing Quorum Systems

Here, the main concern is the fastness of the read/write operations. In particular, every read and write operation that is concurrent with a reconfiguration needs to ensure that both the old and the 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. The main challenges we need to address are:

Research Question 7.5 How fast a quorum system can be reconfigured?

Research Question 7.6 How fast read/write operations can be during reconfigurations?

Research Question 7.7 How fast read and write operations can be during quiescent periods when there are no reconfigurations?

When assuming a single reconfigurer, then it imposes a total ordering on the series of configurations. Therefore, the next configuration can be obtained locally. 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. Such an approach will allow the reconfiguration to be faster, but it may require extra communication exchanges from each read and write operation that are concurrent with it. It is essential to expedite the reconfiguration process, since this may allow more reads and writes to be faster. Additionally, having access to the order of reconfigurations from the single reconfigurer, may help read and write operations to predict the latest configuration.
Conversely, introducing multiple reconfigurers in the system improves fault-tolerance but it also introduces the need of achieving an agreement between the reconfigurers on the next configuration to be deployed. This will affect negatively the fastness of a reconfiguration process, since extra communication exchanges will be needed for the agreement protocol. 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 their effect on the latency of reads and writes. In addition, it is 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.

Equally interesting, it would be to investigate if it is feasible to use the proposed three exchanges protocols along with current existing techniques (i.e., Quorum Views) in order to devise efficient read/write operations for the dynamic setting. As an example, recall the dynamic memory service that supports MWMR objects, called RAMBO [36] (Section 3.3). The phases in read and write operations in that implementation follow ABD [10, 55], and in fact, during quiescent periods when there are no reconfigurations, the algorithm operates similarly to ABD. A natural question here is if we can obtain a dynamic MWMR implementation where read operations take one-and-a-half rounds to complete during quiescent periods:

**Research Question 7.8** Is it feasible for the three exchanges read protocol of algorithm OHMAM (Section 4.3) to be adopted by an approach based on RAMBO and devise a dynamic memory service that supports MWMR objects?

**Research Question 7.9** Would the resulting dynamic implementation have practical benefits?

This can be answered through comprehensive experimental evaluations.
7.2 Closing Remarks

Providing efficient emulations of atomic read/write shared objects in asynchronous, crash-prone, message-passing systems is a fundamental problem in distributed computing. Our focus is on atomic consistency because it is an intuitive notion that hides the complexities of underlying implementations, presenting a convenient abstraction to the software builders. This is particularly valuable because of the common perception that the shared-memory paradigm is easier to deal with than the message-passing paradigm in designing distributed algorithms.

With the advent of cloud services, distributed storage services are bound to attract more attention. Thus, consistent storage systems continues to be an area of active research and advanced development, and there are good reasons to believe that as high performance memory systems with superior fault-tolerance become available, they will play a significant role in the construction of sophisticated distributed applications. The demand for implementations providing atomic read/write memory will ultimately be driven by the needs of distributed applications that require provable consistency and performance guarantees.
Bibliography


