Computationally Light “Multi-Speed” Atomic Memory

Antonio Fernández Anta[1], Theophanis Hadjistasi[2], and Nicolas Nicolaou[3]

1 **IMDEA Networks Institute, Madrid, Spain**
`antonio.fernandez@imdea.org`

2 **University of Connecticut, Storrs, CT, USA**
`theophanis.hadjistasi@uconn.edu`

3 **IMDEA Networks Institute, Madrid, Spain**
`nicolas.nicolaou@imdea.org`

Abstract

Communication demands are usually the leading factor that defines the efficiency of operations on a read/write shared memory emulation in the message-passing environment. In the quest for minimizing the communication demands, the algorithms proposed either require restrictions in the system or incur high computation demands. As a result, such solutions may be not suitable to be used in practice.

In this paper we focus on the practicality of implementations of *atomic read/write shared memory* emulation in the message-passing environment. In particular we investigate implementations that reduce both *communication* and *computation* demands. We first examine the shortcomings of the best two (in terms of communication demands) known algorithms that implement atomic single-writer multiple-reader (SWMR) atomic memory, [3, 6]. The algorithm ccFast proposed in [3], achieves optimal communication by allowing each operation to complete in one round trip, with light computation requirements. Unfortunately, it relies on strict limitations on the number of readers. On the other hand, algorithm OhSam [6], imposes no restrictions on the system, but provides operations that require one and a half communication rounds. In the light of these shortcomings, we present two algorithms that implement *multi-speed* operations with *light computation,* and *without imposing* any restriction on the system. In particular, algorithm ccHybrid adopts the fast (one-round) writes presented in [3], and makes *clients* to switch to a slow (two-round) mode whenever the system is congested. On the other hand, algorithm OhFast, pushes the responsibility of deciding for the speed switch to the *servers*. This allows the algorithm to utilize the fast operations presented in [3], and the slow one-and-a-half-rounds operations of [6], whenever is necessary. We prove that both new algorithms preserve atomicity. To evaluate the new algorithms we implement five different atomic memory algorithms in the NS3 simulator, and we compare their performance in terms of *operation latency*, and *ratio of slow over fast operations* performed. We test the algorithms over different: (i) topologies, and (ii) operation loads. Our results support that the newly presented algorithms increase the practicality of atomic read/write atomic shared memory implementations in the message-passing, asynchronous environment.

1998 ACM Subject Classification C.3.4 Distributed Systems, C.4 Performance of Systems

Keywords and phrases atomicity, read/write objects, shared memory, operation latency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.29

© Antonio Fernández Anta, Theophanis Hadjistasi, and Nicolas Nicolaou;
licensed under Creative Commons License CC-BY

20th International Conference on Principles of Distributed Systems (OPODIS 2016).
Editors: Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone; Article No. 29; pp. 29:1–29:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Emulating atomic [8] (linearizable [7]) read/write objects in message-passing environments is one of the fundamental problems in distributed computing. The problem becomes more difficult when participants in the service may fail and the environment is asynchronous, i.e., it cannot provide any time guarantees on the delivery of the messages and the computation speeds. To cope with failures, traditional distributed object implementations like [1, 10], use *redundancy* by replicating the object to multiple (possibly geographically dispersed) locations (replica servers). Replication however raises the challenge of consistency, as multiple object copies can be accessed concurrently by multiple processes. Atomicity is the most intuitive consistency semantic, as it provides the illusion of a single-copy object that serializes all accesses: each read operation returns the value of the latest write operation.

Attiya, Bar-Noy, and Dolev [1] were the first to present an algorithm, known as ABD, to implement single-writer multi-reader (SWMR) atomic objects in message-passing, crash-prone, asynchronous environments. The authors associate logical *timestamps* to the values written, to impose an order on the write operations. The propagation of the latest timestamp (and its corresponding value) is based on the assumption that at least a majority of replica servers do not fail. In this setting, ABD has write operations that terminate with a single communication round-trip, and read operations that involve two round-trips. Based on basic value comparisons, ABD incurs almost no computational overhead to the service participants. Atomicity is guaranteed by the intersecting properties of two majorities and the second phase of a read operation. Following ABD, a folklore belief persisted that in asynchronous multi-reader (MR) atomic memory algorithms, "reads must write."

The work by Dutta et al. [2] refuted this belief, by presenting atomic register algorithms in which every operation involves only a *single* round-trip. Such an algorithm is called *fast*. They showed that fast reads are possible only in the single-writer (SW) model, and given that the number of readers R is constrained with respect to the number of replicas S and the maximum number of failures f; in particular, $R < \frac{S}{f} - 2$. A recent work by Fernández Anta, Nicolaou, and Popa [3], has shown that, although the result in [2] is efficient in terms of communication, it requires processes to evaluate a computationally hard (NP-hard) predicate. A new algorithm ccFast, with a new predicate, was proposed in that paper to allow operations terminate with a linear computation overhead. Despite improving the practicality of [2], the algorithm in [3] inherited the same system constraint as [2].

The idea of exploring "multi-speed" read operations is not new. An algorithm is said to be "multi-speed" when different read operations may perform different number of communication rounds before completing. Works like [4, 5] proposed implementations in the SWMR model with *two-speed* operations, in an attempt to relax the constraints proposed in [2], and to allow unbounded number of readers. In particular, the work in [5] presents algorithm Sf, which applies a predicate similar to the one introduced in [2], but on *virtual nodes* (i.e., sets of readers) instead of individual reader processes. In [4], the authors introduced *quorum views*, which are client-side tools that examine the distribution of the latest value among the replicas, in order to enable fast read operations. Both [4, 5] trade communication for scalability. Under conditions of low concurrency, both algorithms allow most reads to complete in a single communication round-trip; otherwise a *two round-trip* operation (similar to ABD) is required. To determine the speed of an operation, both algorithms inflicted significant computational demands: (i) [5] exploited the same predicate as in [2], which is NP-hard [3], and (ii) [4] needed to examine the distribution of the object value within all the possible replica subsets. Thus, a trend appeared in the algorithms that aimed for fast operations:

algorithms with lower communication rounds demanded higher computation overhead at the processes.

Following the above findings, we say that an operation is *fast* if it completes in a single communication round trip, and *slow* if it completes in two round trips. A recent work by Hadjistasi, Nicolaou and Schwarzmann [6] redefines *slowness*, as they present an algorithm for the SWMR model, called OhSam, where each operation takes *one and a half* round-trips to complete. As the number of readers is bounded when all operations are fast [2], the authors claim the optimality of their approach in terms of communication when no constraint is imposed. Furthermore their algorithm relies on basic comparisons, inflicting negligible computation overhead.

Contributions. In this paper, we focus in improving the practicality of SWMR atomic read/write register algorithms, by achieving low communication and computation costs on the atomic operations. We trade communication for scalability, by adopting the predicate presented in [3] and allowing some operations to be slow. Also, we combine ideas presented in both [3] and [6], to introduce implementations that allow only *single* and *one-and-a-half* round operations. Enumerated, our contributions are the following:

- We introduce a new "multi-speed" algorithm, ccHybrid, that allows operations to terminate in *one* or *two* communication round-trips, and does not impose any bounds on the number of readers. ccHybrid uses the predicate introduced in [3] to determine the speed of a read operation, and it requires at most one *complete* slow operation per written value. This is similar to the semifast algorithm Sf [5]. However, in contrast to Sf, in which processes have to decide NP-hard predicates, it incurs light (linear) computation.
- Next we examine whether we can combine the techniques presented in [3] and [6] to obtain a "multi-speed" algorithm that allows *one* and *one-and-a-half* round-trip operations. We present algorithm OhFast, that achieves the targeted performance by moving the decision on whether a slow read operation is necessary to the servers. When servers determine that a slow read is necessary, they perform a *relay* phase to inform other servers before replying to the reader. It is interesting that in OhFast not all the servers need to perform a relay for a single read operation. Some of the servers may reply directly to the read whereas some others may perform a relay phase for the same read. Thus a read operation may terminate before receiving a reply from a relaying server.
- We complement our algorithms with experimental results for five algorithms: ABD, OhSam, ccHybrid, OhFast, and Sf. ABD sets the threshold for the rest of the algorithms, while OhSam sets the threshold on the operations that use one and a half rounds. Algorithm Sf is used to demonstrate whether computation has an impact to the latency of operations. We test our algorithms under different scenarios by changing the number of participants, the frequency of operations, and using two network topologies: (i) a topology where servers are distributed evenly over the network, and (ii) a topology that resembles a datacenter where servers are concentrated in close proximity and communicate through high bandwidth links. Our results show that the proposed algorithms outperform the algorithms with "one speed" operations (i.e., ABD and OhSam) in all scenarios, reducing the latency per operation to less than half in most cases. Compared with the semifast "multi-speed" algorithm Sf, our algorithms achieve a similar read latency, even though the scenarios explored were extremely favorable for Sf, since we observed that practically all its operations were fast and the NP-hard predicate evaluations were not heavy (mainly due to the good communication conditions). Finally, as expected, we observed that the topology has a great impact on the algorithms that use *one and a half* round operations.

2 Model

We assume a system consisting of three distinct sets of processes: a writer process with identifier w, a set $\mathcal{R}$ of readers, and a set $\mathcal{S}$ of replica servers. Let $\mathcal{I} = \{w\} \cup \mathcal{R} \cup \mathcal{S}$. In a read/write object implementation, we assume that the object may take a value from a set V. The writer is the sole process that is allowed to modify the value of the object, the readers are allowed to obtain the value of the object, and each server maintains a copy of the object to ensure the availability of the object in case of failures. We assume an *asynchronous* environment, where processes communicate by exchanging messages. The writer, any subset of readers, and up to $f < \frac{|\mathcal{S}|}{2}$ servers may *crash* without any notice.

An algorithm A is a collection of processes, where process A_p is assigned to processor $p \in \mathcal{I}$. Each processor p has a *state* which is determined over a set of state variables. The state of A is a vector that contains the state of each process. Algorithm A performs a *step*, when some process p atomically:

(i) receives a message,
(ii) performs local computation,
(iii) sends a message.

Each such step causes the state at p to change from a pre-state σ_p to a post-state σ'_p. Hence, the state of A changes from σ to σ' where σ contains state σ_p for p and σ' contains state σ'_p, while the state of every $p' \neq p$ is the same in both σ and σ'. An *execution fragment* is an alternating sequence of states and actions of A ending in a state. An *execution* is an execution fragment that starts with the initial state. An execution fragment ξ' extends an execution fragment ξ if the last state of ξ is the first state of ξ'. A process p *crashes* in an execution if it stops taking steps; otherwise p is *correct*. Each process may perform a read or write operation, and each operation has *invocation* and *response* steps. An operation π is *complete* in an execution ξ, if ξ contains both the invocation and the *matching* response step for π; otherwise π is *incomplete*. An execution ξ is *well formed* if any process p that invokes an operation π in ξ does not invoke any other operation π' before the matching response step of π appears in ξ. An operation π *precedes* an operation π' in an execution ξ, denoted by $\pi \rightarrow \pi'$, if the response step of π appears before the invocation step of π' in ξ. Two operations are *concurrent* if none precedes the other.

Correctness of an implementation of an atomic read/write object is defined in terms of the *atomicity* and *termination* properties. The termination property requires that any operation invoked by a correct process eventually completes. For atomicity we use the definition of [9, Lemma 13.16].

Efficiency Metrics. We measure the complexity of an operation π in terms of:

(i) *message complexity*, i.e. the worst-case number of messages exchanged during π, and
(ii) *operation latency*, i.e. the *computation time* and the *communication delays* incurred by π. Computation time accounts the computation steps the algorithm performs in each operation.

Communication delays are measured in *communication exchanges*, as defined in [6].

In particular, a protocol requires each operation to involve a sequence of sends (or broadcasts) of typed messages and the corresponding receives. A *communication exchange* during an operation π in an execution ξ, is defined as the collection of send and receive actions for a specific typed message (as required by the protocol) between the invocation and response of π in ξ. Using this definition, implementations, such as ABD, are structured in terms of *rounds*, where each round consists of two message exchanges: a *broadcast*, initiated

by the process executing an operation, and a *convergecast* of responses to the initiator. A *fast* operation as in [5, 2] consists of two communication exchanges (or one round), and a *slow* operation as used in [1, 4, 5] consists of four communication exchanges (or two rounds). A read operation as in [6] consists of three communication exchanges (or 1.5 rounds). The number of messages that a process expects during a convergecast depends on the implementation.

3 State-of-the-Art Performance of Atomic Memory Implementations

The algorithm by Dutta et al. in 2004 [2], we refer to it as FAST, was the first to present atomic register implementations where all operations take a *single* communication round before completing. To allow fast reads, FAST deploys a recording mechanism at each server and evaluates a predicate at each reader. It was shown that fast reads are possible only if $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$ readers participate in the service. To avoid the bound on the number of readers, Georgiou, Nicolaou and Shvartsman [5], grouped the readers under logical sets, they called *virtual groups*, and allowed some of the read operations to perform two rounds (or 4 communication exchanges). The predicate of [2] was applied on the virtual groups instead of individual readers, where each group could have an arbitrary size. As expected, the use of the predicate imposed a bound on the number of virtual nodes, for atomicity to be preserved; that is $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 1$.

Fernández Anta, Nicolaou and Popa [3], showed that the predicate used by both [2] and [5], is computationally hard. This was due to the fact that the original predicate was searching among all the subsets of servers to identify if there is some subset of servers that replied to a "large enough" subset of readers. To avoid this computational overhead, they investigate whether it is possible to use *how many* instead of *which* readers obtained the latest value, and still be able to preserve atomicity. Thus, the paper introduced a new algorithm, called CCFAST, that was using the following predicate at the readers:

$$\exists \alpha \in [1, |\mathcal{R}|+1] \text{ s.t. } MS = \{s : (s,m) \in maxAck \ \wedge \ m.views \geq \alpha\} \text{ and } |MS| \geq |\mathcal{S}| - \alpha f .$$

Essentially, each server records the readers that observed its local timestamp in a set *seen*, and whenever requested, it reports the *cardinality* of that set to the requesting process. A reader collects the replies from the servers in each read operation, detects the replies that contain the maximum timestamp (set *maxAck*), and checks the cardinalities reported in those replies (*m.views*). If there are "enough" replies with "sufficiently" large cardinalities, the predicate holds and the reader returns the value associated with the maximum timestamp; otherwise the value associated with the previous timestamp is returned. The evaluation of the predicate can be done in linear time with respect to the number of servers in the system. Their algorithm inherited the necessary bound presented in [2] on the number of readers participants, $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$.

Finally, Hadjistasi, Nicolaou and Schwarzmann [6] closed the gap of the communication of read/write operations by presenting algorithm OHSAM, where writes take just one round (or 2 communication exchanges) and reads always take one and a half round (or 3 communication exchanges) to complete. The main idea of the algorithm is to allow servers to exchange information about the operations, before replying to the invoking process. OHSAM uses negligible computation at the processors, as each operation performs only basic comparisons. However, the server communication in *every* operation makes the algorithm suitable for environments where server communication is being carried out by high capacity links.

Table 1, summarizes the efficiency of each of the algorithms in different efficiency metrics. It also presents any bounds that an algorithm may impose on the participation of the service

Table 1 Communication, Computation, Message Complexities and Participation Bounds. (WE/RE: write/read-communication excanges, WC/RC: write/read-computation, WM/RM: write/read-number of messages). $\mathcal{V}$ is the set of virtual nodes.

Algorithm	WE	RE	WC	RC	WM	RM	Bounds
ABD [1]	2	4	$O(1)$	$O(\|\mathcal{S}\|)$	$2\|\mathcal{S}\|$	$4\|\mathcal{S}\|$	Unbounded
Fast [2]	2	2	$O(1)$	$O(\|\mathcal{S}\|^2 \cdot 2^{\|\mathcal{S}\|})$	$2\|\mathcal{S}\|$	$2\|\mathcal{S}\|$	$\|\mathcal{R}\| < \frac{\|\mathcal{S}\|}{f} - 2$
ccFast [3]	2	2	$O(1)$	$O(\|\mathcal{S}\|)$	$2\|\mathcal{S}\|$	$2\|\mathcal{S}\|$	$\|\mathcal{R}\| < \frac{\|\mathcal{S}\|}{f} - 2$
Sf [5]	2	2 or 4	$O(1)$	$O(\|\mathcal{S}\|^2 \cdot 2^{\|\mathcal{S}\|})$	$2\|\mathcal{S}\|$	$O(4\|\mathcal{S}\|)$	$\|\mathcal{V}\| < \frac{\|\mathcal{S}\|}{f} - 1$
OhSam [6]	2	3	$O(1)$	$O(\|\mathcal{S}\|)$	$2\|\mathcal{S}\|$	$2\|\mathcal{S}\| + \|\mathcal{S}\|^2$	Unbounded
ccHybrid (here)	2	2 or 4	$O(1)$	$O(\|\mathcal{S}\|)$	$2\|\mathcal{S}\|$	$O(4\|\mathcal{S}\|)$	Unbounded
OhFast (here)	2	2 or 3	$O(1)$	$O(\|\mathcal{S}\|)$	$2\|\mathcal{S}\|$	$O(\|\mathcal{S}\|^2)$	Unbounded

Algorithm 1 Write, Read and Server protocols of algorithm ccHybrid.

```
 1: at the writer w
 2: Components:
 3: ts ∈ ℕ+;  v, vp ∈ V; wcounter ∈ ℕ+
 4: Initialization:
 5: ts ← 0; v ← ⊥; vp ← ⊥; wcounter ← 0
 6: function WRITE(val)
 7:   vp ← v; v ← val
 8:   ts ← ts + 1
 9:   wcounter ← wcounter + 1
10:   send(⟨ts, v, vp⟩, w, wcounter) to all servers
11:   wait until |S| − f servers reply
12:   return(OK)
13: end function

14: at each reader r_i
15: Components:
16: ts ∈ ℕ+;  maxTS ∈ ℕ+;  v, vp ∈ V; rcounter ∈ ℕ+
17: srvAck ⊆ S × M
18: Initialization:
19: ts ← 0;  maxTS ← 0; v ← ⊥; vp ← ⊥, rcounter ← 0
20: function READ( )
21:   rcounter ← rcounter + 1
22:   send(⟨ts, v, vp⟩, r_i, rcounter) to all servers
23:   wait until |S| − f servers reply
    ▷ Collect (sid, ⟨⟨ts′, v′, vp′⟩, views, prop⟩) msgs in srvAck
24:   maxTS ← max({m.ts′|(s, m) ∈ srvAck})
25:   maxAck ← {(s, m)|(s, m) ∈ srvAck ∧ m.ts′ = maxTS}
26:   ⟨ts, v, vp⟩ ← m.⟨ts′, v′, vp′⟩ for (∗, m) ∈ maxAck
27:   maxViews ← max({m.views|(s, m) ∈ maxAck})
28:   propSet ← {s|(s, m) ∈ maxAck ∧ m.prop = True}
29:   if maxViews > |S|/f − 2 ∨ propSet ≠ ∅ then
30:     if |propSet| < f + 1 then
                                                  ▷ Phase 2
31:       send(⟨ts, v, vp⟩, r_i, rcounter) to all servers
32:       wait until |S| − f servers reply
33:     end if
34:     return(v)
35:   else
36:     if ∃α ∈ [1, |S|/f − 2] s.t.
37:       MS = {s : (s, m) ∈ maxAck ∧ m.views ≥ α} and
38:       |MS| ≥ |S| − αf then
39:       return(v)
40:     else
41:       return(vp)
42:     end if
43:   end if
44: end function

45: at each server s_i
46: Components:
47: ts ∈ ℕ+; seen ⊆ R ∪ {w};  v, vp ∈ V; prop ∈ {True, False}
48: Counter[|R| + 1] ∈ ℕ+
49: Initialization:
50: ts ← 0;  seen ← ∅;  v, vp ← ⊥; prop ← False
51: Counter[i] ← 0 for i ∈ R ∪ {w}
52: function RCV(⟨ts′, v′, vp′⟩, q, counter)
                              ▷ Called upon reception of a message
53:   if Counter[q] < counter then
54:     if ts′ > ts then
55:       ⟨ts, v, vp⟩ ← ⟨ts′, v′, vp′⟩
56:       seen ← {q}
57:       prop ← False
58:     else
59:       seen ← seen ∪ {q}
60:     end if
61:     if ts′ = ts ∧ q ∈ R then
62:       prop ← True
63:     end if
64:     send(⟨ts, v, vp⟩, |seen|, prop) to q
65:   end if
66: end function
```

in order to be able to provide atomic guarantees. The last two algorithms are the ones we present in this paper. Notice that the goal is to minimize communication without inflicting high computation overheads, or participation bounds in the system.

4 Algorithm ccHybrid: Switching from One to Two Rounds

As discussed in Section 3, algorithm ccFast guarantees correctness only when the number of readers is bounded with respect to the ratio of the number of servers and the number of failures in the system, i.e. $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 1$. In this section we propose a modification to ccFast that removes the bound on the number of readers. To unleash the number of readers, the new algorithm ccHybrid, allows some read operations to complete in two rounds. In particular, ccHybrid combines ideas from ccFast and ABD:

(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) propagates the maximum *timestamp-value* pair to a majority of servers during a *slow* read.

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 idea of ccHybrid is to have the reader examine if the number of processes that observed the latest value is over the bound $\frac{|\mathcal{S}|}{f} - 1$. If not, then ccHybrid evaluates the predicate proposed in ccFast over the replies, to determine the value to return. Otherwise, it proceeds to a propagation phase to send the latest value to a majority of servers. To prevent readers from propagating an already propagated value, servers maintain a flag that indicates whether a timestamp has been propagated.

Algorithm 1 provides the formal pseudocode of ccHybrid. The write protocol remains the same as in both ccFast and ABD: the writer increments its local timestamp (L8) and propagates the *timestamp-value* pair to a majority of servers (L10-11). The server protocol is more 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 explain later, optimizes read operations. A server s waits for read and write requests. When a request is received, s updates its local *timestamp-value* pair (L51-57) if the *timestamp* attached in the received message is greater than its local timestamp. In addition, it initializes its *seen* set to contain the sender process, and sets the *prop* flag to $False$. In case the timestamp of the message is not greater than the local timestamp of s, then the server records the sender in its *seen* set (L59). The server s sets $prop = True$ when it receives a message from a reader that contained a *timestamp-value* pair equal to the one that is locally stored in s. Notice that a reader propagates a *timestamp-value* pair in every phase. So, s may set *prop* during the first or second phase of a read.

The main departure of ccHybrid from ccFast lies in the read protocol. A reader behaves as in ccFast as long as the maximum number of *views* reported by the servers remains below $\frac{|\mathcal{S}|}{f} - 2$. In particular, a reader sends read messages to all the servers and waits from $|\mathcal{S}| - f$ to reply (L22). When those replies are received, the reader discovers the maximum timestamp ($maxTS$) among the replies (L24), the set of messages that contained $maxTS$ (L25), and the maximum views reported in those messages (L27). If the maximum views are less than $\frac{|\mathcal{S}|}{f} - 2$ and no reader propagated $maxTS$ (L29), then the reader evaluates the predicate as in ccFast to decide which value to return; otherwise the reader returns the value associated with the $maxTS$. If at least $f + 1$ of the messages that contain $maxTS$, also contain $prop = True$, the reader returns without further action. If this is not the case then the reader performs a second phase propagating the maximum *timestamp-value* pair to $|\mathcal{S}| - f$ servers (L30–33). Notice that ccHybrid 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 [5]) is necessary per write operation. 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 $|\mathcal{S}| - f$ servers,
(ii) a succeeding read receives replies from $|\mathcal{S}| - f$ servers, and
(iii) the read discovers $prop = True$ for $maxTS$ in more than $|\mathcal{S}| - 2f > f + 1$ servers.

4.1 Algorithm Correctness

Our algorithm is correct if it can satisfy Termination (liveness condition) and Atomicity (safety condition). It is trivial to see that termination is satisfied given that the system respects our failure model. To proof atomicity we are going to express atomicity in terms of timestamps written and returned in a SWMR model, as also presented in [3]:

A1. For each process p the ts variable is non-negative and monotonically nondecreasing.
A2. If a read ρ succeeds a write operation $\omega(ts)$ and returns a timestamp ts', then $ts' \geq ts$.
A3. If a read ρ returns ts', then either a write $\omega(ts')$ precedes ρ, i.e. $\omega(ts') \to \rho$, or $\omega(ts')$ is concurrent with ρ.
A4. If ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$ and ρ_1 returns ts_1, then ρ_2 returns $ts_2 \geq ts_1$.

Due to space limitations and due to the similarity of the writer and server protocols to the ones used in ccFast, we omit some of the proofs and we refer the reader to specific lemmas presented in [3]. Properties **A1** and **A3** can be extracted easily from the algorithm. Now let us proof an important lemma about the timestamp returned by a server process:

▶ **Lemma 1.** *In any execution ξ of the algorithm, 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 $T' > T$.*

To following lemma shows **A2**, after which we show that property **A4** holds.

▶ **Lemma 2.** *In any execution ξ of the algorithm, if a read ρ from r_1 succeeds a write operation ω that writes timestamp ts from the writer w , i.e. $\omega \to \rho$, and returns a timestamp ts', then $ts' \geq ts$.*

▶ **Lemma 3.** *In any execution ξ of* ccHybrid*, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$, ρ_1 is fast satisfying the predicate for $maxTS = ts_1$, then ρ_2 receives a $maxTS = ts_2$ s.t. $ts_2 \geq ts_1$.*

▶ **Lemma 4.** *In any execution ξ of* ccHybrid*, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$, and ρ_1 returns ts_1, then ρ_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 ρ_1 and ρ_2. There are four cases to investigate:
(a) ρ_1 is *fast*, and ρ_2 is *fast*,
(b) ρ_1 is *fast*, and ρ_2 is *slow*,
(c) ρ_1 is *slow*, and ρ_2 is *slow*, and
(d) ρ_1 is *slow*, and ρ_2 is *fast*.
Let $maxTS_i$ be the maximum timestamp observed by a read ρ_i, for $i \in \{1, 2\}$, during its first phase.

Case a: In case both operations are *fast* then, according to ccHybrid, either they observe $maxViews \leq \frac{|\mathcal{S}|}{f} - 2$ and $propSet = \emptyset$, or they observe an $|propSet| \geq f + 1$. If both observe $maxViews \leq \frac{|\mathcal{S}|}{f} - 2$ and check the predicate, then with the same reasoning as in [3, Lemma 8], it follows that $ts_2 \geq ts_1$.

If ρ_1 observes $|propSet| \geq f + 1$ then since ρ_2 receives replies from $|\mathcal{S}_2| = |\mathcal{S}| - f$ servers, then there exists a server $s \in propSet \cap \mathcal{S}_2$ such that s replies to both ρ_1 and ρ_2. Since $\rho_1 \to \rho_2$, then s replies to ρ_1 before replying to ρ_2. Since s replies with $maxTS_1$ to ρ_1, then by Lemma 1, s replies with a timestamp $ts_s \geq maxTS_1$ to ρ_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 ρ_2 will return $ts_2 = maxTS_1 = ts_1$. If $maxTS_2 > maxTS_1$ then ρ_2 returns either $maxTS_2$ or $maxTS_2 - 1$ and thus $ts_2 \geq ts_1$.

It remains to examine the case where ρ_1 observes $maxViews \leq \frac{|\mathcal{S}|}{f} - 2$ and $propSet = \emptyset$, and ρ_2 observes $|propSet| \geq f + 1$. If the predicate holds for ρ_1 then by Lemma 3, ρ_2 observes $maxTS_2 \geq maxTS_1$. Since ρ_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 ρ_1 then we know that the write operation propagating $maxTS_1 - 1$ completed before or during ρ_1. Since $\rho_1 \to \rho_2$ then this write completed before ρ_2 as well. Thus, by **A2**, ρ_2 observes $maxTS_2 \geq maxTS_1 - 1$. Since ρ_2 observes $|propSet| \geq f+1$, then it returns $ts_2 = maxTS_2 \Rightarrow ts_2 \geq maxTS_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Case b: Since ρ_1 in this case is *fast* then ρ_1 returns either: (i) $maxTS_1 - 1$, or (ii) $maxTS_1$.

In (i), since ρ_1 observed $maxTS_1$ and since we have a single writer, it follows that the write operation that wrote timestamp $maxTS_1 - 1$, say ω_1, proceeds or is concurrent to ρ_1, and completes before the response step of ρ_1. Since $\rho_1 \to \rho_2$, then $\omega_1 \to \rho_2$. Since ρ_2 is slow, then it returns the maximum timestamp it observes, i.e. $ts_2 = maxTS_2$. Moreover, since $\omega_1 \to \rho_2$, and since both operations wait for $|\mathcal{S}| - f$ replies, then according to our failure model, there exist at least a single server s that replies to both operations, first to ω_1 and then to ρ_2. According to Lemma 1, s sends a timestamp $ts_s \geq maxTS_1 - 1$ to ρ_2. Thus, $maxTS_2 \geq maxTS_1 - 1$, and therefore $ts_2 \geq ts_1$.

In (ii) it follows that either the predicate holds for ρ_1, or ρ_1 observes $|propSet| \geq f+1$. Since ρ_2 is slow and returns $ts_2 = maxTS_2$, then by Lemma 3 and with similar reasoning as in Case (a) for when ρ_1 observes $|propSet| \geq f+1$, we can show that $maxTS_2 \geq maxTS_1$ and hence $ts_2 \geq ts_1$.

Case c: The case where both reads are slow is simple and resembles the behavior of the reads in ABD [1]. Here each read ρ_i, for $i \in [1,2]$, returns $maxTS_i$ and before completing it propagates $maxTS_i$ to $|\mathcal{S}| - f$ servers. Thus, ρ_1 returns $ts_1 = maxTS_1$, and before completing propagates $maxTS_1$ to $|P_1| = |\mathcal{S}| - f$ servers. Since $\rho_1 \to \rho_2$, and since ρ_2 receives $|S_2| = |\mathcal{S}| - f$ replies, then it is going to receive a timestamp $ts_s \geq maxTS_1$ from at least a single server $s \in P_1 \cap S_2$. Thus, ρ_2 returns $ts_2 = maxTS_2 \geq maxTS_1$, and $ts_2 \geq ts_1$.

Case d: So it remains to investigate the case where ρ_1 is *slow* and ρ_2 is *fast*. Observe that this case is possible when a server s is "saturated" by concurrent reads (more than $\frac{|\mathcal{S}|}{f} - 2$) and s replies to ρ_1 but does not reply to ρ_2. Now we have two cases to investigate: either ρ_2 observes $maxTS_2 \geq maxTS_1$, or $maxTS_2 = maxTS_1 - 1$. If ρ_2 observes a $maxTS_2 \geq maxTS_1$, it may either return $ts_2 = maxTS_2$ or $ts_2 = maxTS_2 - 1$. In either case $ts_2 \geq maxTS_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Let us examine now the case where $maxTS_2 = maxTS_1 - 1$. Since ρ_1 is slow and returns $maxTS_1 - 1$, then before completing it propagates $maxTS_1 - 1$ to $|\mathcal{S}| - f$ servers. Let P_1 be the set of servers that received the messages and replied to the second phase of ρ_1. Moreover, $|S_2| = |\mathcal{S}| - f$ are the servers that received messages and replied to ρ_2. So by Lemma 1, every server $s \in P_1 \cap S_2$ replies to both ρ_1 and then to ρ_2, with a timestamp $ts_s \geq maxTS_1 - 1$. In addition s sets $prop = True$ before replying to ρ_1. Since $maxTS_2 = maxTS_1 - 1$, then s replies with $ts_s = maxTS_1 - 1$ to ρ_2, and thus the *propSet* contains at least s in ρ_2. According to the algorithm ρ_2 returns $ts_2 = maxTS_2$ in this case and hence $ts_2 \geq ts_1$. ◀

▶ **Theorem 5.** *Algorithm* ccHybrid *implements a SWMR atomic read/write register.*

5 Algorithm OhFast: Switching from One to One and a Half Rounds

Similar to algorithm ccHybrid, OhFast aims to allow unbounded number of readers to participate in the service while allowing operations to complete in one round. In contrast to the classic approach of the two rounds per read operation, OhFast tries to further reduce the communication required by *slow* reads. Thus OhFast combines ideas from ccFast and the

Algorithm 2 Read protocol of algorithm OhFast.

1: at each reader r_i
2: **Components:**
3: $ts \in \mathbb{N}^+;\ maxTS \in \mathbb{N}^+;\ v, vp \in V; rcounter \in \mathbb{N}^+$
4: $srvAck \subseteq \mathcal{S} \times M$
5: **Initialization:**
6: $ts \leftarrow 0,\ maxTS \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot; rcounter \leftarrow 0$
7: **function** READ( )
8: $rcounter \leftarrow rcounter + 1$
9: **send**($\langle ts, v, vp\rangle, r_i, rcounter$) to all servers
10: **wait until** $|\mathcal{S}| - f$ servers reply
▷ Collect the $(sid, \langle\langle ts', v', vp'\rangle, views, secured\rangle)$ msgs in $srvAck$
11: $maxTS \leftarrow \max\{m.ts' | (s, m) \in srvAck\}$
12: $maxAck \leftarrow \{(s, m) | (s, m) \in srvAck \wedge m.ts' = maxTS\}$
13: $\langle ts, v, vp\rangle \leftarrow m.\langle ts', v', vp'\rangle$ for $(*, m) \in maxAck$
14: $maxViews \leftarrow \max\{m.views | (s, m) \in maxAck\}$
15: **if** $\exists (s, m) \in maxAck$ s.t. $m.secured = True$ **then**
16: return(v)
17: **else if** $\exists \alpha \in [1, \frac{|\mathcal{S}|}{f} - 2]$ s.t.
18: $MS = \{s : (s, m) \in maxAck\ \wedge\ m.views \geq \alpha\}$ and
19: $|MS| \geq |\mathcal{S}| - \alpha f$ **then**
20: return(v)
21: **else**
22: return(vp)
23: **end if**
24: **end function**

one and a half round approach suggested by OhSam. With server to server communication, OhFast is expected to perform better in environments where the servers communicate via high capacity links, e.g., data centers.

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 on a slow read 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. 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. So, 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 $|\mathcal{S}| - f$ 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.

Algorithms 2 and 3 provide the formal pseudocode of OhFast. We omit the write protocol as it is the same to the one presented for ccHybrid. The read protocol in OhFast (Algorithm 2) is simpler than the read of ccHybrid. The reader sends messages to all the servers and waits for $|\mathcal{S}| - f$ of them to reply (L9). Once those replies are received the reader discovers the maximum timestamp $maxTS$ among the replies (L11), and collects the messages that contain $maxTS$ (L12) [1] in the set *maxAck*. If some message in *maxAck* indicates that $maxTS$ is secured, i.e. it contains $secured = True$ (L15), then the reader returns $maxTS$. Otherwise, it evaluates the predicate on the messages in *maxAck* (L19) to determine which timestamp to return.

The server protocol (Algorithm 3) is the most involved in OhFast. 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* list storing the latest timestamp the server relayed for each reader. A server s waits for read/write and relay requests. When s receives a read/write request it updates its local replica state and *seen* set appropriately (L13-14). In case the timestamp in the request is higher than its local timestamp it also sets *securedts* flag to $False$. Then, s decides whether to relay the received timestamp or not. In particular, s relays a timestamp if (L19):

[1] Notice that this is another departure from OhSam as each reader in OhSam returns the smallest discovered timestamp.

Algorithm 3 Server protocol of algorithm OhFast.

```
 1: at each server s_j
 2: Components:
 3: ts ∈ ℕ+; seen ⊆ R ∪ {w}; v, vp ∈ V; Counter[|R|+1] ∈ ℕ+
 4: scounter ∈ ℕ+; securedts ∈ {True, False}
 5: Relays[|R|] ∈ ℕ+
 6: Initialization:
 7: ts ← 0; seen ← ∅; v, vp ← ⊥; prop ← False
 8: Counter[i] ← 0 for i ∈ R ∪ {w}; scounter ← 0
 9: Relays[i] ← 0; securedts ← False
10: function rcv(⟨ts', v', vp'⟩, q, counter)
              ▷ Called upon reception of a READ/WRITE message
11:   if Counter[q] < counter then
12:     if ts' > ts then
13:       ⟨ts, v, vp⟩ ← ⟨ts', v', vp'⟩; seen ← {q}
14:       securedts ← False
15:     else
16:       seen ← seen ∪ {q}
17:     end if
18:     if q ∈ R and |seen| > |S|/f − 2 and
19:       securedts = False and Relays[q] < ts then
20:       scounter ← scounter + 1
21:       sendRelay(⟨ts, v, vp⟩, q, s_j, counter, scounter)
22:          to all the servers
23:       Relays[q] ← ts; srvRelay ← ∅
24:     else
25:       send(⟨ts, v, vp⟩, |seen|, counter, securedts) to q
26:     end if
27:   end if
28: end function
29: function rcvRelay(⟨ts', v', vp'⟩, q, s, c1, c2)
              ▷ Called upon reception of a RELAY message
30:   if Counter[s] < c2 then
31:     if ts' > ts then
32:       ⟨ts, v, vp⟩ ← ⟨ts', v', vp'⟩
33:       seen ← {q}
34:     else if ts = ts' then
35:       seen ← seen ∪ {q}
36:     end if
37:     if Relays[q] = ts' then
38:       srvRelay ← srvRelay ∪ {s}
39:       if |srvRelay| = |S| − f then
40:         if ts = ts' then
41:           securedts ← True
42:         end if
43:         send(⟨ts', v', vp'⟩, 0, c1, True) to q
44:       end if
45:     else
                                   ▷ reply back to the sender
46:       scounter ← scounter + 1
47:       sendRelay(⟨ts', v', vp'⟩, q, s_j, scounter) to s
48:     end if
49:   end if
50: end function
```

(i) the sender is a reader,
(ii) it sent this timestamp to more than $\frac{|\mathcal{S}|}{f} - 2$ processes,
(iii) the timestamp has not already being relayed (i.e. $securedts = False$) and
(iv) the server has not yet relayed this timestamp for the same reader.

If some of these conditions does not hold then s just replies to the sender with its local timestamp (L25). Notice here that servers only relay for the readers and do not relay for the writer, as the sole writer always has the latest timestamp. In a relay message s includes its local replica state, the id of the reader that initiated the relay, and its own id. When a server s' receives a relay message from s, it first updates its local replica and *seen* set appropriately (L32-33, L35). Then s' checks if it also sent a relay with the same timestamp for the same reader (L37). If not then s' bounces the relay to s and completes (L47); otherwise s' adds s in the servers that received its relay 38). When it receives $|\mathcal{S}| - f$ relays, s' replies to the reader that initiated the relay along with the timestamp that it initially relayed (not its local timestamp) (L43). Finally, if its local timestamp is the same as the relayed timestamp, then s' also sets $securedts = True$ (L41).

5.1 Algorithm Correctness

In order to show that OhFast is correct we have to prove that it satisfies both termination (liveness) and atomicity (safety) properties. Termination of the write operation is easy to see as according to our failure model $|\mathcal{S}| - f$ servers do not fail and can receive and reply to the write request. However, termination of the read protocol is not straightforward: a server may communicate with other servers before responding to a reader. The next lemma shows that all the read operations terminate.

▶ **Lemma 6.** *In any execution ξ of* OhFast*, every read operation ρ invoked by a correct process r eventually terminates.*

Next it remains to show that atomicity is preserved. To prove atomicity we are going to use the four properties that express atomicity in terms of timestamps written and returned, as presented in Section 4.1. It is easy to see from the algorithm, that every process updates its local replica only when a value with a higher timestamp is received. Thus, it can be easily seen that the algorithm satisfies properties **A1** and **A3**. Notice also that when a server

receives a timestamp ts then it attaches a timestamp $ts_s \geq ts$ to any message it sends from that point onward. This can be shown with similar statements as in Lemma 1. We need to show that when a server receives a *relay* that contains a timestamp ts then it sends a timestamp $ts_s \geq ts$ from that point onward.

▶ **Lemma 7.** *In any execution ξ of* OhFast*, if a server s receives a relay 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 $T' > T$.*

Now we can show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written. This shows the validity of property **A2**.

▶ **Lemma 8.** *In any execution ξ of the algorithm, if a read ρ from r succeeds a write operation ω that writes timestamp ts_ω from the writer w, i.e. $\omega \to \rho$, and returns a timestamp ts_ρ, then $ts_\rho \geq ts_\omega$.*

Finally, it remains to investigate if property **A4** holds. Before we do so, we prove a lemma showing that if a timestamp ts is secured from a server s, then at least $|\mathcal{S}| - f$ servers have a timestamp $ts' > ts$.

▶ **Lemma 9.** *In any execution ξ of* OhFast*, if a server s sets $securedts = True$ for a timestamp ts at time T then $\exists \mathcal{S}' \subseteq \mathcal{S}$ at T, s.t. $|\mathcal{S}'| \geq |\mathcal{S}| - f$ and $\forall s' \in \mathcal{S}'$, the local timestamp of s' is $ts' \geq ts$.*

▶ **Lemma 10.** *In any execution ξ of* OhFast*, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \to \rho_2$, and ρ_1 returns ts_{ρ_1}, then ρ_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 ρ_2, r_1 includes a timestamp $ts_{r_1} \geq ts_{\rho_1}$ in every message it sends to servers. According to Lemma 1 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 ρ_2. So the predicate is valid for $|MS| \geq |\mathcal{S}| - f$ and $\alpha = 1$. Hence, ρ_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 ρ_1 and ρ_2 when $r_1 \neq r_2$. If both read operations examine the predicate to decide on the value to return (i.e., they do not receive a secured timestamp), then with same reasoning as in [3, Lemma 8] we can show that atomicity is preserved. So it remains to examine the following three cases:

1. ρ_1 evaluates the predicate, and ρ_2 receives a secured $maxTS_2$,
2. ρ_1 receives a secured $maxTS_1$, and ρ_2 evaluates the predicate, and
3. ρ_1 receives a secured $maxTS_1$, and ρ_2 receives a secured $maxTS_2$.

Case 1: In this case, ρ_1 evaluates the predicate, and ρ_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) ρ_1 returns $maxTS_1$, and (b) ρ_1 returns $maxTS_1 - 1$.

Case 1a: If ρ_1 returns $maxTS_1$ it follows that the predicate is valid for ρ_1. Hence:

$$\exists \alpha \in [1, \frac{|\mathcal{S}|}{f} - 2] \text{ and}$$

$$MS \subseteq \mathcal{S} \text{ s.t. } MS = \{s : s.ts = maxTS_1 \ \wedge \ s.views \geq \alpha\} \ \wedge \ |MS| \geq |\mathcal{S}| - \alpha f\,.$$

Moreover, since ρ_1 examines the predicate, then none of the servers that replied with $maxTS_1$ sends $secured = True$. Therefore, $\forall s \in MS$, it must be true that $s.views \leq \frac{\mathcal{S}}{f} - 2$ before replying to ρ_1 (L16), otherwise s would proceed to relay and secure $maxTS_1$. Since every $s.views \leq \frac{\mathcal{S}}{f} - 2$, then it must be the case that $\alpha \leq \frac{\mathcal{S}}{f} - 2$ as well. Thus substituting:

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

Since ρ_2 receives replies from $|\mathcal{S}_2| = |\mathcal{S}| - f$ servers then $\mathcal{S}_2 \cap MS \neq \emptyset$. Also notice that since $\rho_1 \rightarrow \rho_2$, then a server $s \in \mathcal{S}_2 \cap MS$ replies to ρ_1 with $maxTS_1$ before replying to ρ_2. By Lemma 1, s replies to ρ_2 with a timestamp $ts_s \geq maxTS_1$. Thus, $maxTS_2 \geq ts_s \Rightarrow maxTS_2 \geq maxTS_1$ and ρ_2 returns $ts_{\rho_2} \geq maxTS_1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$.

Case 1b: Assume now the case where ρ_1 returns $maxTS_1 - 1$. Since ρ_1 received $maxTS_1$, and since the sole writer invokes one operation at a time, then it follows that the write operation that wrote $maxTS_1 - 1$, say ω, completed during or before ρ_1. Since though $\rho_1 \rightarrow \rho_2$, then it follows that $\omega \rightarrow \rho_2$. Since ω communicates with $|\mathcal{S}| - f$ servers before completing, and since ρ_2 waits for $|\mathcal{S}| - f$ replies, then there is a server s that replies to ω before replying to ρ_2. By Lemma 1, s replies with a timestamp $ts_s \geq maxTS_1 - 1$ to ρ_2. Thus ρ_2 observes a $maxTS_2 \geq maxTS_1 - 1$, and hence $ts_{\rho_2} \geq maxTS_1 - 1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$ in this case as well.

Case 2: Here, ρ_1 returns $ts_{\rho_1} = maxTS_1$ as it received a message that contained $maxTS_1$ and $secured = True$. Read ρ_2 evaluates the predicate to decide on the value to return. We have two subcases to examine again: (a) ρ_2 returns $maxTS_2$, or (b) ρ_2 returns $maxTS_2 - 1$. Since ρ_1 returned a secured timestamp, then it received $maxTS_1$ and $secured = True$ from some server s. By Lemma 9, a set $|\mathcal{S}'| \geq |\mathcal{S}| - f$ of servers have a timestamp $ts' \geq maxTS_1$ before s replies to ρ_1. Since ρ_2 receives replies from $|\mathcal{S}_2| = |\mathcal{S}| - f$ servers, then $\mathcal{S}' \cap \mathcal{S}_2 \neq \emptyset$. Then by Lemmas 1 and 7, any server in $s' \in \mathcal{S}' \cap \mathcal{S}_2$ replies to ρ_2 with a timestamp $ts_{s'} \geq maxTS_1$. Thus, ρ_2 observes a $maxTS_2 \geq maxTS_1$. If $maxTS_2 > maxTS_1$ and since ρ_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 ρ_2 returns $ts_{\rho_2} = maxTS_2$ then $ts_{\rho_2} \geq ts_{\rho_1}$. Let us examine now if ρ_2 may return $maxTS_2 - 1$. As we said before every server s' in $\mathcal{S}' \cap \mathcal{S}_2$ replies with $ts_{s'} \geq maxTS_1$ to ρ_2. Since $|\mathcal{S}'| \geq |\mathcal{S}| - f$ and $|\mathcal{S}_2| \geq |\mathcal{S}| - f$ then $|\mathcal{S}' \cap \mathcal{S}_2| \geq |\mathcal{S}| - 2f$. Also by the algorithm, every server in $\mathcal{S}'$ adds r_1 in its *seen* set before replying to the relay message from s (L39). Furthermore, every server in $\mathcal{S}_2$ adds r_2 in its *seen* set before replying to ρ_2. So every server $s' \in \mathcal{S}' \cap \mathcal{S}_2$ replies with a $s.views \geq 2$. Thus, the predicate holds for at least $|MS| = |\mathcal{S}' \cap \mathcal{S}_2| \geq |\mathcal{S}| - 2f$ and $\alpha = 2$. Hence ρ_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 ρ_1 and ρ_2 return a secured timestamp. Let s_1 be the server that send $maxTS_1$ and $secured = True$ to ρ_1, and s_2 (not necessarily different than s_1) be the server that sent $maxTS_2$ and $secured = True$ to ρ_2. By Lemma 9, there exists a set $\mathcal{S}'$ s.t. every server $s \in \mathcal{S}'$ has a timestamp $ts_s \geq maxTS_1$ before s_1 replies to ρ_1. As explained in Case 2, $\mathcal{S}' \cap \mathcal{S}_2 \neq \emptyset$. Hence there exists a server that replied both to the relay message of s_1 and to ρ_2. By Lemma 7, each server $s' \in \mathcal{S}' \cap \mathcal{S}_2$ replies to ρ_2 with a timestamp $ts_{s'} \geq maxTS_1$. Hence, $maxTS_2 \geq maxTS_1$. Since ρ_2 returns a secured timestamp, then it returns $maxTS_2$. Therefore, $ts_{\rho_2} = maxTS_2 \Rightarrow ts_{\rho_2} \geq maxTS_1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$. ◀

▶ **Theorem 11.** *Algorithm* OhFast *implements a SWMR atomic read/write register.*

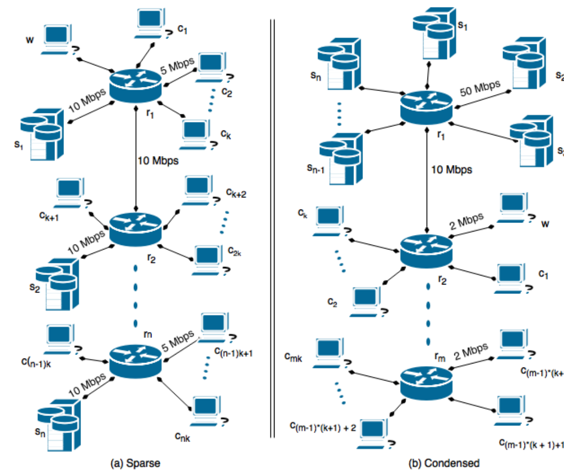

Figure 1 Simulated topologies.

6 Empirical Results

In this section, we present empirical results that we obtained by implementing algorithms ABD [1], OhSam [6], Sf [5], ccHybrid, and OhFast, using the NS3 discrete event simulator [11]. 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 exact parameters that may affect the performance of our algorithms.

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 so as every operation would wait for all but one servers to reply, inflicting that way high concurrency and potentially inconsistency in our system. 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, *Sparse* and *Condensed*, which mainly differ on the deployment of server nodes.

Figure 1 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 10Mpbs links and 4ms delay. In the *Sparse* topology (Figure 1(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 *Condensed* topology (Figure 1(b)) all the servers are connected to a single router with 50Mbps links and 2ms delay, simulating a network where servers are connected in close proximity and with high bandwidth links (e.g., a datacenter).

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

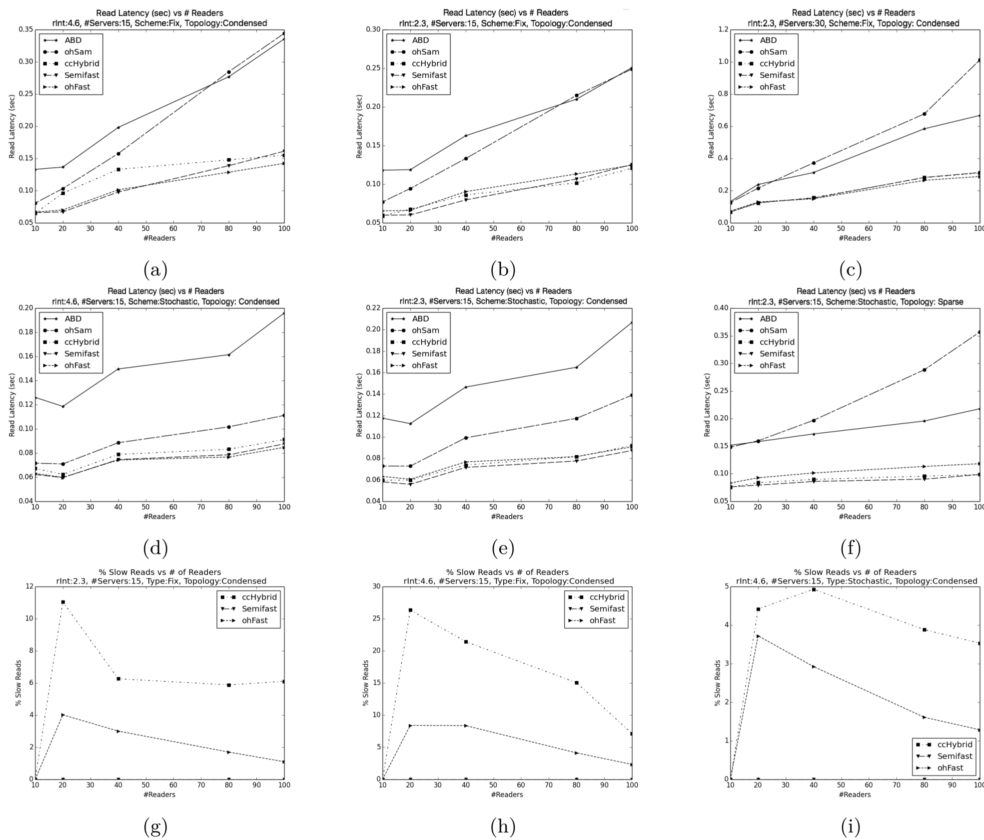

Figure 2 Experimental Results from NS3 Simulation.

communication and computation latencies. As NS3 only provides simulated time events and omits any computation, we combined two clocks: (a) the simulation clock, and (b) a real time clock. The simulation clock was able to estimate the communication time, while the real clock allowed us obtain the time taken by the computation at each operation. The latency is calculated adding both times.

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. In addition, we define two read invocation schemes: (i) *fix* and (ii) *stochastic*. In the fix scheme all the read operations are scheduled periodically at the read interval. In the stochastic scheme each operation is scheduled at random between $1s$ and $rInt$ seconds in each read interval. Finally, to test topological effects we run our algorithms using both the *Sparse* and *Condensed* topologies.

Results

As a general observation, the new algorithms outperform all the other algorithms in most scenarios. In particular, it is clear that ccHybrid 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 ccHybrid 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, taking our tests one by one we conclude to the following observations:

Scalability: As can be seen in Figures 2(b) and (c), the increasing number of readers and the servers have a negative impact on all the algorithms. The impact is higher on ABD and OhSam, and lower for the rest of the algorithms.

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. This can be seen in Figure 2(a) and (b). Algorithms ABD and OhSam are not affected (as all of their reads are slow), but the multi-speed algorithms Sf, ccHybrid and OhFast, are affected negatively. This behaviour 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 (Figure 2(h)). This is not the case when $rInt = 2.3$ (Figure 2(g)). Notice that the same behavior is not being observed when a *stochastic* scheme is used, as radomness prevents the operations to be invoked at exactly the same time (Figure 2(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, as shown in Figure 2(i).

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 (Figure 2(b) and 2(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: Plots 2(e) and 2(f) 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 (e) OhSam performs better than ABD and OhFast performs similar to ccHybrid and Sf we notice that in (f) OhSam performs worse than ABD and OhFast worse than the 2 others. This behaviour is expected as both OhSam and OhFast need to exchange messages between the servers during a relay phase. However, notice that OhFast performs much better since operation relays are not performed for every read operation.

7 Conclusions

In this paper we present two new algorithms ccHybrid and OhFast that implement atomic SWMR register in a message-passing, asynchronous environment. Both algorithms use the predicate introduced in [3], to achieve *single round* reads with small computational footprint. However, to avoid constraints in reader participation both algorithms allow some reads to be *slow*. In ccHybrid the reader decides on the speed of its read operation, reasulting in operations that perform *1 or 2 rounds*. OhFast moves the decision of slow operations to the servers, enabling *1 or* 1.5 *round* operations. Simulation results show that our algorithms outperform all slow operation algorithms, as well as "multi-speed" implementations that have high computation demands. We claim that our developments take us closer to *practical* implementations of atomic read/write objects in the message-passing environment.

References

1. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.
2. P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC04)*, pages 236–245, 2004.
3. A. Fernández Anta, N. Nicolaou, and A. Popa. Making "fast" atomic operations computationally tractable. In *Proceedings 19th International Conference On Principle Of DIstributed Systems (OPODIS 15)*, 2015.
4. Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC'08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 289–304, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87779-0_20.
5. Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009. doi:10.1016/j.jpdc.2008.05.004.
6. T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann. Brief announcement: Oh-ram! one and a half round read/write atomic memory. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC'16, pages 353–355, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933073.
7. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
8. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE Transactions on Computers*, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.
9. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
10. Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.
11. NS3 network simulator. URL: https://www.nsnam.org/.