Noname manuscript No.
(will be inserted by the editor)

Tractable low-delay atomic memory*

Antonio Fernández Anta · Theophanis Hadjistasi · Nicolas Nicolaou · Alexandru Popa · Alexander A. Schwarzmann

Received: date / Accepted: date

Abstract 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. Therefore, a more accurate assessment of operation efficiency should account for both *computation* and *communication* costs.

This paper focuses on the efficiency of read and write operations in emulations of *atomic read/write shared memory* in the asynchronous, message-passing, crash-prone environment. The much celebrated work by Dutta et al. presented an implementation in this setting where *all* read and write operations could complete in just a single communication round-trip. Such operations where characterized for the first time as *fast*. At its heart, the work by Dutta et al. used a predicate to achieve that performance. We show that the predicate is computationally *intractable* by defining an equivalent problem and reducing it to *Maximum Biclique*, a known *NP-hard* problem.

We derive a *new, computationally tractable predicate*, and an algorithm to compute it in *linear time*. The proposed predicate is used to develop three algorithms: ccFast, ccHybrid, and OhFast. ccFast is similar to the algorithm of Dutta et al. with the main difference being the use of the new predicate for reduced computational complexity. All operations in ccFast are fast, and particular constraints apply in the number of participants. ccHybrid and OhFast, allow some operations to be "slow", enabling unbounded participants in the service. ccHybrid is a "multi-speed" version of ccFast, where the reader determines when it is not safe to complete a read operation in a single communication round-trip. OhFast, expedites algorithm OhSam of Hadjistasi et al. by placing the developed predicate at *the servers* instead of clients and avoiding excessive server communication when possible. An experimental evaluation using NS3 compares algorithms ccHybrid and OhFast to the classic algorithm ABD of Attiya et al., the algorithm Sf of Georgiou et al. (the first "semifast" algorithm, allowing both fast and slow operations), and algorithm OhSam.

In summary, this work gives the new meaning to the term *fast* by assessing both the *communication* and the *computation* efficiency of each operation.

* This work combines results from [5] and [6], that appeared in OPODIS'15 and OPODIS'16 respectively.

A. Fernández Anta
IMDEA Networks Institute, Madrid, Spain
E-mail: antonio.fernandez@imdea.org

T. Hadjistasi
Algolysis Ltd., Limassol, Cyprus
E-mails: theo@algolysis.com

N. Nicolaou
Algolysis Ltd., Limassol, Cyprus
Email: nicolas@algolysis.com

A. Popa
Department of Computer Science, University of Bucharest, and
the National Institute for Research and Development in Informatics, Bucharest, Romania
Email: alexandru.popa@fmi.unibuc.ro

A. A. Schwarzmann
School of Computer and Cyber Sciences, Augusta University, Augusta, GA, USA. E-mail: aschwarzmann@augusta.edu

1 Introduction

Emulating atomic [15] (linearizable [14]) read/write objects in message-passing systems is one of the fundamental problems in distributed computing. The problem becomes more challenging when participants in the service may fail and the environment is asynchronous, i.e., when there are no time bounds on the delivery of messages and the computation speeds. To cope with failures, distributed object implementations like [2, 17], use *redundancy* by replicating the object at multiple (geographically dispersed) network locations (replica servers). Replication however raises the challenge of consistency, as multiple object replicas can be accessed concurrently by multiple processes and may be inconsistent. To characterize the properties of distributed object implementations, several consistency notions were defined, the strongest of which being *atomicity*. It is the

most intuitive consistency semantic, providing an illusion of a single-copy object that serializes all accesses: each read operation returns the value of the latest preceding write operation, and this value is at least as recent as that returned by any preceding read.

The seminal work of Attiya, Bar-Noy, and Dolev [2] presented an algorithm, commonly known as algorithm ABD, implementing single-writer, multiple-reader (SWMR) atomic objects in message-passing crash-prone asynchronous environments. Each replica of the object is associated with a *timestamp*, and the order write operations is determined by the values of the timestamps. Each operation is guaranteed to terminate as long as some majority of replica servers do not crash. The write protocol involves a single round-trip communication, while the read protocol involves two round-trip stages. Specifically, the writer increments the timestamp for each write and propagates the new value and its timestamp to some majority of replicas. The readers are implemented as a two-phase protocol (each phase incurring one round-trip communication), where the first phase obtains from some majority of the replicas their values and timestamps, and uses the value corresponding to the highest timestamp as the return value. The reader then performs a second phase, in which it propagates this highest timestamp and the associated value to some majority of replica servers, ensuring that any subsequent read will discover the value that is at least as recent. Of note is that the local processing in each operation is very modest: each phase simply needs to ensure that messages from a majority of replicas are received. The algorithm guarantees atomicity in all executions, relying on the fact that any two majorities have a non-empty intersection. Avoidance of the second "write" phase could lead to violations of atomicity. Following this development, a folklore belief persisted that in asynchronous multi-reader atomic memory implementations "reads must write."

The work by Dutta et al. [3] dispelled this belief, by presenting an atomic register implementation where both reads and writes involve only a *single* communication round-trip (or simly *round*). An implementation where *all* read and write operations complete in a single round is called *fast*. It was shown that fast implementations are possible under a constraint on the number of readers. Specifically, if R is the number of readers, S is the number of replicas, and f is the maximum number of crashes in the system, then it must be the case that $R < \frac{S}{f} - 2$. The authors also showed that it is impossible to devise fast multiple-writer, multiple-reader (MWMR) implementations, in which *all* read and write operations complete in a single round.

To circumvent the bounds on the number of readers, researchers started exploring *multi-speed* implementations. We say that an implementation is *multi-speed* if read operations may involve a different number of communication round-trips. For example, works [10,11] proposed SWMR implementations with *two-speed* operations and unbounded number of readers. The Sf algorithm in [11] employs a particular predicate in the readers (similar to the one introduced in [3]) to determine whether it is safe for a read operation to terminate after one round, and allowing arbitrary number of readers. The algorithm in [10] introduced *quorum views* that are the client-side tools that examine the distribution of the latest value among the replicas in order to enable fast read operations. Both [10] and [11] trade communication for the scalability in the number of readers. Under conditions of low concurrency, both algorithms allow most reads to complete in a single communication round-trip; otherwise the reads take two round-trips (similar to ABD). To determine the speed of an operation, both algorithms incur significant computational overheads: in [11] to evaluate the mentioned predicate, and in [10] to examine the distribution of object values in all the possible replica subsets.

Thus, a trade-off emerged in algorithms that implement fast operations: in order to reduce the number of communication rounds the amount of local processing is substantially increased. Georgiou et al. [9] questioned the computational complexities of the "fast" implementations, and presented a study of the effects on the performance of the algorithms. The authors analyzed the computational complexity of the algorithm in [4], the only algorithm that enabled *some* fast writes and reads in the MWMR model, and showed both analytically and experimentally how the communication cost was reduced at the expense of the computational overhead of the algorithm. In particular, the authors expressed the predicate used in the algorithm by a computationally equivalent problem and they showed that this problem is NP-hard. To improve the efficiency of the algorithm from [4], they proposed a polynomial *approximation* algorithm.

Recently, Hadjistasi, Nicolaou and Schwarzmann [13] showed that atomic operations do not necessarily require complete communication round trips. They presented a SWMR algorithm, called OhSam, where each operation involves *three one-way communication exchanges*, in essence taking *one and a half* round-trips to complete. Furthermore, their algorithm performs a very modest amount of local computation, resulting in negligible computation overhead.

Contributions. This paper focuses on the practicality and efficiency of read and write operations in distributed SWMR atomic register implementations. We show that the computational costs of the fast algorithm in [3] are substantial, and we propose a deterministic solution that improves the computational burden and communication bit complexity of the original algorithm while maintaining its fault-tolerance and consistency. Since both implementations impose bounds on the participation in the system, we explore the adoption of "multi-speed" solutions based on computationally-efficient algorithms. Our contributions are as follows:

- We introduce a new problem that is computationally equivalent to the predicate used in [3]. We show that the problem, and thus the computation of the predicate, is NP-hard. The proof is by reduction to the *Maximal Biclique* problem, a known NP-hard problem.
- We then devise a revised *fast* implementation, called ccFast, which uses a new *linear time* predicate to determine the value to be returned by each read operation. 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* processes witnessed the maximum timestamp among those replies. With the new predicate we reduce the size of each message sent by the replicas, and we prove rigorously that atomicity is preserved.
- We analyze the operation complexity of ccFast, in terms of *communication*, *computation*, and *message length*. For the computational complexity, we provide a linear time algorithm for the computation of the new predicate. The algorithm utilizes *buckets* to count the number of appearances of each timestamp in the collected replies. We analyze the complexity of the algorithm and prove that it correctly computes the predicate of ccFast.
- We introduce a new "multi-speed" algorithm, ccHybrid, that allows operations to terminate in *one* or *two* communication round-trips, and that does not impose constraints on the number of readers. ccHybrid uses our linear predicate 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 [11]. However, in contrast to Sf, in which processes rely on an NP-hard predicate, it performs only linear time computation.
- Next we explore the use of our predicate in algorithm OhSam [13]. We obtain a "multi-speed" algorithm, called OhFast, that allows *one* and *one-and-a-half* round-trip operations. In this algorithm the decision of whether a slow read operation is necessary is moved 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 servers need to perform a relay for a single read operation. Some servers may reply directly to the reader, 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 use the NS3 discrete event simulator to obtain experimental results comparing the performance of five algorithms: ABD, OhSam, ccHybrid, OhFast, and Sf. ABD sets the baseline for other algorithms, while OhSam sets the baseline for the operations that use one and a half rounds. Algorithm Sf is used to assess how computation impacts the latency of operations. We evaluate 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 "one speed" algorithms (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 of 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 substantial impact on the algorithms that use *one and a half* round operations.

Our results shed new light on the operation latency in SWMR atomic object implementations in asynchronous message-passing systems, and motivate the need for assessing the performance of *fast* implementations both in terms of communication and computation efficiencies.

2 Model

We assume a system consisting of three distinct sets of processes: a single writer process with identifier w, a set $\mathcal{R}$ of readers, and a set $\mathcal{S}$ of replica servers; we let $\mathcal{I} = \{w\} \cup \mathcal{R} \cup \mathcal{S}$. The values of objects come from the set V. The writer is the only process that is allowed to write a new value of the object, the readers can read the value of an object, and each server maintains a copy of the object value, providing redundancy needed

Algorithm	WE	RE	WC	RC	WM	RM	Participation	Complexity
ABD [2]	2	4	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$4\lvert\mathcal{S}\rvert$	Unbounded	Constant
Fast [3]	2	2	$O(1)$	$O(\lvert\mathcal{S}\rvert^2 \cdot 2^{\lvert\mathcal{S}\rvert})$	$2\lvert\mathcal{S}\rvert$	$2\lvert\mathcal{S}\rvert$	$\lvert\mathcal{R}\rvert < \frac{\lvert\mathcal{S}\rvert}{f} - 2$	NP-Hard
Sf [11]	2	2 or 4	$O(1)$	$O(\lvert\mathcal{S}\rvert^2 \cdot 2^{\lvert\mathcal{S}\rvert})$	$2\lvert\mathcal{S}\rvert$	$O(4\lvert\mathcal{S}\rvert)$	$\lvert\mathcal{V}\rvert < \frac{\lvert\mathcal{S}\rvert}{f} - 1$	NP-Hard
OhSam [13]	2	3	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$2\lvert\mathcal{S}\rvert + \lvert\mathcal{S}\rvert^2$	Unbounded	Constant
ccFast (this paper)	2	2	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$2\lvert\mathcal{S}\rvert$	$\lvert\mathcal{R}\rvert < \frac{\lvert\mathcal{S}\rvert}{f} - 2$	Linear
ccHybrid (this paper)	2	2 or 4	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$O(4\lvert\mathcal{S}\rvert)$	Unbounded	Linear
OhFast (this paper)	2	2 or 3	$O(1)$	$O(\lvert\mathcal{S}\rvert)$	$2\lvert\mathcal{S}\rvert$	$O(\lvert\mathcal{S}\rvert^2)$	Unbounded	Linear

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

to ensure availability in case of failures. The system is asynchronous and processes communicate by exchanging point-to-point messages. The writer, any subset of readers, and up to f servers ($f < |\mathcal{S}|/2$) may *crash* without notice.

An algorithm A is a collection of processes, where process A_p is assigned to processor $p \in \mathcal{I}$. Each process maintains a *state* containing a set of variables. The state of A is a vector of states of all processes. Processes operate by performing *steps* consisting of (i) receiving a message, (ii) performing local computation, (iii) sending a message. Each step by process p causes its state to change from a pre-state σ_p to a post-state σ'_p.

A process p *crashes* in an execution if it stops taking steps; otherwise p is *correct*. Each read or write operation implementation has *invocation* and *response* steps. An operation is *complete* in an execution if both the invocation and the matching response steps occur in the execution; otherwise the operation is *incomplete*. We deal only with *well-formed* executions: any process that invokes an operation does not invoke any other operation before the matching response to the current operation. An operation π *precedes* an operation π' in an execution, denoted by $\pi \to \pi'$, if the response of π appears before the invocation of π'. Two operations are *concurrent* if neither 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. Atomicity of an implementation is defined following [16]. If all the read and write operations that are invoked on an object complete in an execution, then the read and write operations can be partially ordered by an ordering $\prec$, so that the following properties are satisfied:

P1. The partial order is consistent with the external order of invocation and responses, that is, there do not exist operations π_1 and π_2, such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$.

P2. All write operations are totally ordered and every read operation is ordered with respect to all the writes.

P3. Every read operation ordered after any writes returns the value of the last write preceding it in the partial order, and any read operation ordered before all writes returns the initial value of the object.

For the rest of the paper we assume a single object memory system. As atomicity/linearizability is a local property for each process, a complete atomic memory system is obtained by composing several single object implementations [16].

Efficiency Metrics. We measure the efficiency of operations in terms of: (i) *message complexity*, i.e. the worst-case number of messages exchanged during an operation, and (ii) *operation latency*, i.e. the *computation time* and the *communication delays* incurred by an operation. Computation time accounts for the computation steps the algorithm performs in each operation. Communication delays are measured as the number of *communication exchanges*, following [13].

A *communication exchange* during an operation in an execution is defined as follows. Let the operation protocol consist of a sequence of sends (or broadcasts) of typed messages and the corresponding receives. The collection of send events for a specific typed message and the corresponding receive events between the operation invocation and the response constitute a single communication exchange.

Using this definition, implementations such as ABD, are structured in terms of *rounds*, where each round consists of two communication exchanges: a *broadcast*, initiated by the process executing an operation, and a *convergecast* of responses to the initiator. A *fast* operation as in [11,3] involves two communication exchanges (or one round), and a *slow* operation as in [2,10,11] consists of four communication exchanges (or two rounds). A read operation in [13] consists of three communication exchanges (or equivalently one and a half rounds).

3 Fastness in Prior Work and Implications for Atomic Memory Implementations

Algorithm Fast by Dutta et al. in 2004 [3] is the first to present an atomic object implementation for the message-passing environment where all read and write operations take a *single* communication round trip. It is also shown that in any fast implementation the number of readers $|\mathcal{R}|$ must be constrained with respect to the number of servers, $|\mathcal{S}|$, and the number of server failures, f, so that $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$.

Algorithm Fast used timestamps associated with values as in ABD [2] to impose an order on the write operations. The write operation is almost identical to the one-round write in [2]: the writer increments its local timestamp, and broadcasts the timestamp with the new value to the servers. The read operation is very different as it takes a single round to complete. To enable single round 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. Each server records all the processes that witness its local timestamp, in a set called *seen*. This set is reset whenever the server learns a new timestamp. Each reader collects the server replies (in the form $(server, message)$ pair) that contain the maximum timestamp, $maxTs$, in a set $maxAck$, and then examines the following predicate:

$$\exists \alpha \in [1, \ldots, |\mathcal{R}| + 1] \ \wedge \ MS \subseteq \{m.seen : (s, m) \in maxAck\} \ \text{ s.t.} \tag{1}$$

$$|MS| \geq |\mathcal{S}| - \alpha f \ \wedge | \bigcap_{sn \in MS} sn| \geq \alpha \tag{2}$$

In a high level, the idea of the predicate lies on the observation that for any two read operations that witness the same $maxTs$, their respective sets of servers that replied $maxTs$ may differ by at most f members (where f is the maximum number of servers that may crash). In particular, if the writer w completes by sending messages to $|\mathcal{S}|-f$ servers, then a subsequent reader r_1 may only observe $|\mathcal{S}| - 2f$ of them (missing the f servers that may be faulty). In this case each of the $|\mathcal{S}|-2f$ servers could record a $seen = \{w, r_1\}$ and hence the intersection of all those seen sets will have cardinality 2. In the case now where w is incomplete, writing to just $|\mathcal{S}| - 2f$ servers, r_1 may see the same amount of servers replying with the latest write (not distinguish the two executions), while a subsequent read from a reader r_2 may witness only $|\mathcal{S}| - 3f$ servers to have the latest write. All of those servers however will have a seen set equal to $\{w, r_1, r_2\}$ and hence with an intersection of cardinality 3. In general, if a writer “reaches” $|S| - (\alpha - 1)f$, for $2 \leq \alpha \leq |\mathcal{R}|$, servers then a subsequent reader may witness at least $|S| - \alpha f$ servers with the latest value. Thus inductively, a reader may return the latest value when witnessing $|S| - \alpha f$ *seen* sets with intersection of cardinality $\geq \alpha$, hence the predicate follows.

In other words, the reader looks at the *seen* sets of the servers that replied, and tries to determine whether “enough” processes witnessed the maximum timestamp, $maxTs$. If the predicate holds, the reader returns the value associated with the maximum timestamp. Otherwise it returns the value associated with the previous timestamp. Notice here that the predicate examines *which* processes witnessed the latest timestamp as it examines the intersection of the seen sets.

To remove the limitation on the number of readers, Georgiou et al. [11] developed an approach, they call Sf, for grouping the readers into logical sets, called *virtual groups*, and used it in an algorithm that allows some read operations to take two rounds (or 4 communication exchanges). Essentially, for some reads, the algorithm performs the second (“write”) phase as in ABD. In particular, each server records the virtual group of each reader requesting its local value, and attaches to each reply the set of virtual group identifiers he recorded. If the same server is contacted by two readers from the same virtual group, the set will only contain one instance of the group identifier. Upon receiving a reply, the reader applies a similar predicate as Eq. (1) and (2) from [3], on the set of virtual nodes. If $|\bigcap_{sn \in MS} sn| = \alpha$ (from Eq. (2)), then the reader performs the second phase before completing. Otherwise, the read is fast. With this use of the predicate the participation constraint presented in [3] is now imposed on the number of virtual nodes $\mathcal{V}$, that is $|\mathcal{V}| < \frac{|\mathcal{S}|}{f} - 1$. As each virtual group may grow arbitrarily, then Sf supports unbounded number of readers.

Hadjistasi et al. [13] studied the possibility of having read and write operations that take (at most) 3 communication exchanges, or as they term this, “one and a half round.” They present a SWMR algorithm, called OhSam, where writes take just one round (2 communication exchanges), and reads always take one and a half rounds (3 communication exchanges). The main idea here is to have servers to exchange information about the operations prior to replying to the invoking process. OhSam uses negligible local computation at the processors, as each operation performs only basic comparisons. However, the all-to-all server communication in every operation makes the algorithm better suited for environments where the servers are well connected.

Table 1 summarizes the efficiency of these algorithms and the three algorithms presented in this paper.

It also presents any constraints on the participation in the service. Notice that the goal is to minimize communication without incurring high computation overheads in the system. Let us now compare algorithms ABD, Fast, Sf, and OhSam in more detail.

Communication Complexity. Write operations in algorithms ABD, Fast, Sf, and OhSam take 2 communication exchanges, as the sole writer sends messages to all servers and waits for a majority to reply before completing. As previously mentioned, read operations take 2 communication exchanges in Fast, 2 or 4 in Sf, 3 in OhSam, and 4 in ABD.

Computation Complexity. The write operation in all four algorithms terminates once the appropriate number of servers reply, without imposing any additional computation. However, the reduction in the number of communication exchanges has a negative impact on the computational complexity of read operations in Fast and Sf.

In algorithm Fast [3] the computation performed by read operations is prohibitive. If one tries to examine all possible subsets MS of $\mathcal{S}$, then we obtain $2^{|\mathcal{S}|}$ possibilities. If one restricts this space to include only the subsets with size $|MS| = |\mathcal{S}| - \alpha f$ for all $\alpha \in [1, \ldots, |\mathcal{R}|+1]$ (namely $1 \leq |MS| \leq |\mathcal{S}| - f$), then we may examine up to $2^{(|\mathcal{S}|-f)}$ different subsets. Recall also that each *seen* set contains identifiers from the set $\mathcal{R} \cup \{w\}$, and hence at most $|\mathcal{R}|+1$ elements. To compute the intersection we need to check for each element if it belongs in all the *seen* sets. As MS may include $|\mathcal{S}| - f$ servers (and thus as many *seen* sets) the computation of the intersection may take $(|\mathcal{S}| - f)(|\mathcal{R}|+1)$ comparisons. As $|\mathcal{R}|$ is bounded by $|\mathcal{S}|$ then the previous quantity is bounded by $O(|\mathcal{S}|^2)$. So that leads to an upper bound of $O(|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|})$.

Since Sf [11] adopts the same predicate over the virtual nodes, it suffers from the same computation overhead. The main difference in [11] from the work in [3], is that the predicate does not examine all the possible subsets of readers, but rather it examines all the possible subsets of virtual nodes that observed the latest timestamp. Therefore, the *seen* set may contain at most $|\mathcal{V}|+1$ elements, and hence the computation burden in [11] could be improved if readers could be concentrated in as few virtual nodes as possible. However, according to an analysis performed by Georgiou et al. in [8], the fewer the virtual nodes in the service the more read operations would have performed multiple communication rounds. This is due to the fact that fewer virtual nodes will concentrate more readers and therefore more read operations will observe the predicate condition that may lead to a second communication round. Furthermore, fewer virtual nodes will delay the return of a newly written value as the predicate will be validated only when the timestamp is propagated to more servers in the system. For example, consider the case where we place all the readers in a single virtual node. In this case the *seen* set in each server may contain at most 2 elements: the id of the writer and the id of the single virtual node. The computation of the predicate can be done efficiently in this case. However, since the predicate is valid only if the seen sets contain more than $|\mathcal{S}| - \alpha f$ common elements, for $\alpha \in [1, 2]$, then reads may be fast only when a write operation is completed. Any read concurrent with the write operation will be slow, defeating the communication performance of the algorithm. Thus, in order to reduce the communication overhead of the algorithm, more virtual nodes are necessary increasing this way the computational overhead of the predicate.

If indeed the computation in [3,11] is exponential (in the number of servers), then the local computation time will proverbially explode even when the number of servers is modest.

At the same time, the computation complexity at the reader process in both ABD [2] and OhSam [13] is linear in the number of servers, since the reader needs to process at most one reply from each server to compute the maximum timestamp.

Message Complexity. Finally, for each write operation in any algorithm, at most $|\mathcal{S}|$ messages are sent to all servers, and at most $|srvSet|$ messages will be received by the writer as replies. The read operation for algorithms ABD, Fast, and Sf, depends on the number of communication exchanges executed by the algorithm as there exists direct communication between each client and each server. Thus, ABD requires $4|\mathcal{S}|$, Fast requires $2|\mathcal{S}|$, and Sf requires at most $4|\mathcal{S}|$ messages per read operation. Algorithm OhSam includes a server-to-server communication for each read operation, and hence its message complexity rises to $2|\mathcal{S}| + |\mathcal{S}|^2$ messages, including the client-to-server messages ($2|\mathcal{S}|$), and the server-to-server communication ($|\mathcal{S}|^2$).

3.1 Added Knowledge

The works presented thus far [3,11,13], mainly focused on minimizing the communication burden of atomic shared memory emulations as initially proposed by ABD [2]. In this work we examine the problem from a different perspective: except from the communication burden we also consider the computation demands of the more efficient approaches and we examine whether such demands will have a negative impact on the overall performance of these algorithms. Such parameter was overlooked at previous solutions as the main focus was

the reduction of the communication overhead even in the expense of higher computational costs. To this end, we prove in the next section that the predicate used in [3,11] is *NP-complete* potentially increasing the latency of computing the predicate exponentially as the number of replicas and clients increases in the system.

To solve this issue we revise the approach of the previous works, and we examine whether the predicate can be computed efficiently. We devise a new predicate that does not rely on the membership but rather on the size of the *seen* sets. We demonstrate how the predicate can be used in the FAST algorithm [3], without affecting the correctness (atomicity) and the efficiency in terms of the communication burden. We then show the usage of the predicate through the CCHYBRID algorithm to allow some slow read operations while enabling the participation of unbounded readers in the service. Note that this algorithm takes a different approach to the SF [11] algorithm as it does not group the readers into virtual nodes. Using CCHYBRID we wanted to examine how the algorithm performs if the only metric it examines to perform slow reads is the size of the *seen* set. Algorithm SF, should be able to utilize the predicate directly, without any impact on its correctness and the communication efficiency.

Finally we improve the computational efficiency in algorithms that perform three communication exchanges. For this reason we embed our predicate in the OHSAM algorithm presented in [13]. For this algorithm we move our predicate evaluation to the servers instead of the clients, to help them skip an extra message, in case enough processes know about the latest timestamp in the service. This is the first attempt to improve the three communication exchange algorithm.

4 Formulation and Hardness of the Predicate in FAST

We formulate the predicate used in FAST with the following computational problem.

Problem 1 Input: Two sets $U_1 = \{s_1, s_2, \ldots, s_n\}$, $U_2 = \{p_1, p_2, \ldots, p_k\}$, where $\forall s_i \in U_1$, $s_i \subseteq U_2$. Moreover, we are given two integers $\alpha \geq 1$ and $f \geq 0$ such that $n - \alpha f \geq 1$.
Goal: Is there a set $M \subseteq U_1$ such that $|\cap_{s \in M} s| \geq \alpha$ and $|M| \geq n - \alpha f$?

To show the computational equivalence of Problem 1 to the predicate in FAST, we need to provide a polynomial-time reduction of Problem 1 to the predicate in FAST and vice versa. This is captured by the following theorem.

Theorem 1 *Problem 1 is computationally equivalent to the predicate used in* FAST.

Proof The FAST predicate accepts as inputs the set of *seen* sets collected from the replies, say $Seen = \{m.seen : (s, m) \in maxAck\}$, the set of reader and writer processes $\mathcal{R} \cup \{w\}$, the maximum number of server failures f', and the number of servers $|\mathcal{S}|$. For some $\alpha' \in [1, \ldots, |\mathcal{R}| + 1]$, the predicate specifies the set $MS \subseteq Seen$ with properties as defined in formulas (1) and (2).

Given an instance of Problem 1 we can transform it to an instance of the predicate as follows. Let $\mathcal{S} = \{i : s_i \in U_1\}$, $Seen = U_1$, $\mathcal{R} \cup \{w\} = U_2$, $f' = f$ and $\alpha' = \alpha$. Notice that $\alpha \in [1, |U_2| + 1]$ as every $s_i \subseteq U_2$, $M \subseteq U_1$, and $|\cap_{s \in M} s| \geq \alpha$. Thus, by our transformation $\alpha' \in [1, |\mathcal{R}| + 1]$. Solving FAST for α' we get a set of server-message pairs, MS, and we can set $M = MS$ as the solution for Problem 1. Hence, by reduction, if FAST can be solved in polynomial time then so is Problem 1.

In the other direction, given an instance of the predicate in FAST we can transform the input to an instance of Problem 1. We set $f = f'$, $U_2 = \mathcal{R} \cup \{w\}$ and $U_1 = Seen \cup EmptySeen$ where $EmptySeen$ is a set of empty sets, as many as the servers that do not add a seen set in $Seen$. Thus, $EmptySeen$ is used as a padding, so that $n = |U_1|$ becomes equal to $|\mathcal{S}|$. For each $\alpha' \in [1, \ldots, |\mathcal{R}| + 1]$ we run an instance of Problem 1, with $\alpha = \alpha'$. Solving Problem 1, we get a set of subsets M s.t. $|\cap_{s \in M} s| \geq \alpha$ and $|M| > n - \alpha f$. So, if we set $MS = M$ then $|MS| \geq |\mathcal{S}| - \alpha' f'$ and $|\bigcap_{sn \in MS} sn| \geq \alpha'$.

We now prove that the Problem 1 is NP-hard via a reduction from the decision version of the Maximum Edge Biclique problem defined below. The reduction is similar to the one in [18] for showing that the Maximum k-Intersection Problem is NP-hard.

Definition 1 (Maximum Edge Biclique Problem) Given a bipartite graph $G = (X, Y, E)$ a biclique consists of two sets $A \subseteq X$, $B \subseteq Y$ such that $\forall a \in A$, $\forall b \in B$, $(a, b) \in E$. The *size* of a biclique is defined the number of edges in the biclique. The goal is to decide if the given graph G has a biclique of size at least c.

The Maximum Edge Biclique Problem is NP-complete [7].

Theorem 2 *Problem 1 is NP-hard.*

Proof We show that if we can solve Problem 1 in polynomial time, then we can solve the decision version of the Maximum Edge Biclique problem in polynomial time. Given an instance of the biclique problem, i.e.,

a bipartite graph $G = (X, Y, E)$, we construct the following instance of Problem 1. First, let $U_2 = Y$. Then, each element $s_i \in U_1$ corresponds to a vertex $v \in X$ such that $s_i = \{u \in Y : (v, u) \in E\}$. See Figure 1 for an example.

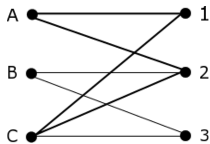

Fig. 1: The left side of the graph (nodes A, B and C) corresponds to the elements of the set U_1 and the right side (nodes 1,2 and 3) corresponds to the elements of the set U_2. Thus, $A = \{1, 2\}$, $B = \{2, 3\}$ and $C = \{1, 2, 3\}$. The maximum edge biclique in this example has two nodes on each side, thus has size 4. In the figure, one of the two maximum edge bicliques is emphasized with bold edges.

In order to decide if a biclique of size at least c exists, we solve $|X|$ instances of Problem 1 where α and f are set such that $\alpha \cdot (n - \alpha f) = c$. If there exists a positive instance of Problem 1 among those $|X|$ checked, then there exists a biclique of size at least c. Otherwise, no such biclique exists.

We focus now on two particular values of α and f such that $\alpha \cdot (n - \alpha f) = c$ and we prove the graph G has a biclique of size c with α vertices in the set X and $n - \alpha f$ vertices on the other side, if and only if a subset M that satisfies the constraints of Problem 1 exists.

First, given a biclique $A \cup B$ of size c with $|B| = \alpha$, then the set $M \subseteq U_1$ contains the elements of U_1 associated with the vertices in A. Since the biclique $A \cup B$ has size c, it follows that the number of the sets in M is larger than $c/\alpha = n - \alpha f$.

Conversely, given a set M of size $n - \alpha f$ whose elements have intersection at least α, we can find a biclique of size $c = \alpha \cdot (n - \alpha f)$. The elements $A \subseteq X$ of the biclique are those corresponding to the elements of the set M. Since the elements in the set M have intersection greater than or equal to α, we have that the common neighborhood of the vertices in A is greater than or equal to $\alpha(n - \alpha f)$. Thus, the size of the biclique is at least $c = \alpha \cdot (n - \alpha f)$.

5 Algorithm ccFast: Refining "Fastness" for Atomic Reads

In this section we modify the algorithm presented in [3] to make it even "faster". Since we allow only single round-trip operations, the new algorithm adheres to the bound presented in [3] and [4] regarding the number of readers. Thus, a solution is possible only if $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$. Also, from the results in [3,11], it follows that such a solution is impossible in the MWMR model. To expedite the calculation of the predicate we aim to eliminate the use of sets in the predicate and we focused on the question: "Can we preserve atomicity if we know *how many* and not *which* processes read the latest value of a server?" An answer to this question could yield two benefits: (i) reduce the size of messages, and (ii) reduce the computation time of the predicate. We provide a positive answer to this question and we present a new algorithm, called ccFast, that preserves communication and is computationally faster than algorithm Fast.

Algorithm 1 presents the formal specification of ccFast. Here we present a high level description of each protocol in the algorithm. The *counter* variables used in the algorithm are solely used by processes to distinguish "fresh" from "stale" messages that may arrive out of order due to asynchrony. In the rest of the description we will not refer to the counters, but rather we assume that the messages received by each process are fresh messages.

Write Protocol. To perform a write operation, the writer w calls function write(val). During the operation the writer stores the value to be written in variable v and the previous written value in variable vp (A1:L8). Then it increments its local timestamp variable ts (A1:L9), and sends a write request along with the triple $\langle ts, v, vp \rangle$ to all servers and waits for $|\mathcal{S}| - f$ replies. Once those replies are received the operation terminates.

Server Protocol. The server protocol contains a recording mechanism which generates information that is used by each read operation to determine the value of the register. Each server $s \in \mathcal{S}$ maintains a timestamp variable along with the values associated with that timestamp. In addition, the server maintains a set of reader and writer identifiers, called *seen*. Initially each server is waiting for read and/or write requests. When a request is received the server examines if the timestamp ts' attached in the request is larger than its local timestamp ts (A1:L41). If $ts' > ts$, the server updates its local timestamp and values to be equal to the ones attached in the received message (A1:L42), and resets its *seen* set to include only the identifier of the process that sent this message (A1:L43); otherwise the server just inserts the identifier of the sender in the *seen* set (A1:L45). Then, the server replies to the sender by sending its local $\langle ts, v, vp \rangle$ triple, and the size of its recording set $|seen|$. This is a departure from the Fast

Algorithm 1 Read, Write and Server protocols of algorithm ccFast

```
1: At 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 : input)
7:     vp ← v
8:     v ← val
9:     ts ← ts + 1
10:    wcounter ← wcounter + 1
11:    wAck ← ∅
12:    broadcast(⟨writeRequest, ts, v, vp, w, wcounter⟩) to S
13:    wait until (|wAck| = |S| − f)
14:    return
15: end function

16: Upon receive m from s
17: if (m.counter = wcounter) then
18:    wAck ← wAck ∪ {(s, m)}
19: end if

20: at each server s_i
21: Components:
22: ts ∈ ℕ+, seen ⊆ R ∪ {w}, v ∈ V, vp ∈ V
23: Counter[1..|R| + 2]: array of int
24: Initialization:
25: ts ← 0, seen ← ∅, v ← ⊥, vp ← ⊥
26: Counter[i] ← 0 for i ∈ R ∪ {w}

27: Upon receive(⟨writeRequest, ts', v', vp', w, wcounter⟩)
28: if (Counter[w] < wcounter) then
29:    Counter[w] ← wcounter
30:    if (ts < ts') then
31:        ⟨ts, v, vp⟩ ← ⟨ts', v', vp'⟩
32:        seen ← {w}
33:    else
34:        seen ← seen ∪ {w}
35:    end if
36:    send(⟨writeAck, Counter[w], s⟩) to w
37: end if
38: Upon receive(⟨readRequest, ts', v', vp', r, rcounter⟩)
39: if  (Counter[r] < rcounter)  then
40:    Counter[r] ← rcounter
41:    if (ts' > ts) then
42:        ⟨ts, v, vp⟩ ← ⟨ts', v', vp'⟩
43:        seen ← {r}
44:    else
45:        seen ← seen ∪ {r}
46:    end if
47:    send(⟨readAck, ts, v, vp, |seen|, prop, Counter[r]⟩) to r
48: end if

49: at each reader r_i
50: Components:
51: ts ∈ ℕ+,  maxTS ∈ ℕ+,  v, vp ∈ V, rcounter ∈ ℕ+
52: srvAck ⊆ S × M,  maxTSmsg ⊆ M
53: Initialization:
54: ts ← 0,  maxTS ← 0, v ← ⊥, vp ← ⊥, rcounter ← 0
55: srvAck ← ∅,  maxTSmsg ← ∅
56: function Read
57:    rcounter ← rcounter + 1
58:    send(⟨readRequest, ts, v, vp, r, rcounter⟩) to S
59:    wait until (|srvAck| = |S| − f)
60:    maxTS ← max({m.ts'|(s, m) ∈ srvAck})
61:    maxAck ← {(s, m)|(s, m) ∈ srvAck ∧ m.ts' = maxTS}
62:    ⟨ts, v, vp⟩ ← m.⟨ts', v', vp'⟩ for (∗, m) ∈ maxAck
63:    if  ∃α ∈ [1, |R| + 1] s.t.
64:        MS = {s : (s, m) ∈ maxAck ∧ m.views ≥ α} ∧
65:        |MS| ≥ |S| − αf then
66:        return(v)
67:    else
68:        return(vp)
69:    end if
70: end function

71: Upon receive m from s
72: if (m.counter = rcounter) then
73:    srvAck ← srvAck ∪ {(s, m)}
74: end if
```

algorithm where the server was attaching the complete *seen* set.

Read Protocol. The read protocol is the most involved. When a reader process invokes a read operation it sends read requests along with its local $\langle ts, v, vp\rangle$ triple to all the servers, and waits for $|\mathcal{S}| - f$ of them to reply. Once the reader receives those replies, it: (i) discovers the maximum timestamp, $maxTS$, among the messages, (ii) collects all the messages that contained $maxTS$ in a set $maxAck$, and (iii) updates its local $\langle ts, v, vp\rangle$ triple to be equal to the triple attached in one of those messages (A1:L60-L62). Then it runs the following predicate on the set $maxAck$ (A1:L65):

$$\exists \alpha \in [1, |\mathcal{R}|+1] \text{ s.t.} \quad (3)$$
$$MS = \{s : (s, m) \in maxAck \ \wedge \ m.views \geq \alpha\} \ \wedge \quad (4)$$
$$|MS| \geq |\mathcal{S}| - \alpha f \quad (5)$$

The predicate examines *how many* processes the maximum timestamp has been sent to. If $|\mathcal{S}| - \alpha f$, or more, servers sent this timestamp to more than α processes, for α between $[1, \ldots, |\mathcal{R}|+1]$, then the predicate is true and the read operation returns the value associated with $maxTS$, namely v; otherwise the read operation returns the value associated with $maxTS - 1$, namely vp.

Idea of the 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, ξ_1. Let the writer perform a write operation ω and receive replies from a set $\mathcal{S}_1$ of $|\mathcal{S}| - f$ servers. Let a reader follow and perform a read operation ρ_1 that receives replies from a set of servers $\mathcal{S}_2$ again of size $|\mathcal{S}| - f$ that *misses* f servers that replied to the write operation. 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 $|\mathcal{S}_1 \cap \mathcal{S}_2| = |\mathcal{S}| - 2f$ servers. Consider now ξ_2 where the write operation ω is not complete and only the servers in $\mathcal{S}_1 \cap \mathcal{S}_2$ receive the write requests. If ρ_1 receive replies from the same set $\mathcal{S}_2$ in ξ_2 then it won't be able to distinguish the two executions. In ξ_1 however the read has to return the value written, as the write in that execution precedes the read

operation. Thus, in ξ_2 the read has to return the value written as well. If we extend ξ_2 by another read operation ρ_2 from a third process, then it may receive replies from a set $\mathcal{S}_3$ missing f servers in $|\mathcal{S}_1 \cap \mathcal{S}_2|$. Thus it may see the value written in $|\mathcal{S}_1 \cap \mathcal{S}_2 \cap \mathcal{S}_3| = |\mathcal{S}| - 3f$ servers. But since there is another read that saw the value from these servers (ρ_1) then ρ_2 has to return the written value to preserve atomicity. Observe now that ρ_1 saw the written value from $|\mathcal{S}| - 2f$ servers and each server replied to both $\{w, \rho_1\}$, and ρ_2 saw the written value from $|\mathcal{S}| - 3f$ and each server replied to all three $\{\omega, \rho_1, \rho_2\}$. By continuing with the same logic, we derive the predicate that if a read sees a value written in $|\mathcal{S}| - \alpha f$ servers and each of those servers sent this value to α 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 $|\mathcal{S}| - \alpha f > f$. Solving this equation results in $\alpha < \frac{|\mathcal{S}|}{f} - 1$. But α 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 $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$.

Algorithm Correctness

To show that the algorithm is correct we need to show that each correct process terminates (liveness) and that the algorithm satisfies the properties *P1 - P3* of atomicity (safety) as presented in Section 2 following [16]. As the main departure of ccFast from Fast is the predicate logic, some of the proofs that follow are very similar to the ones presented in [3]. The lack of knowledge of *which* processes witnessed a particular value, raises challenges in proving that consistency is preserved.

Liveness. Termination is trivially satisfied with respect to our failure model: up to f servers may fail and each operation waits for no more than $|\mathcal{S}| - f$ replies.

Atomicity. In ccFast we associate values with timestamps to order the read and write operations. By the atomicity definition, for each execution of the algorithm there must exist a partial order $\prec$ on the operations that satisfy conditions *P1 - P3*. Let ts_π be the timestamp at the completion of an operation π, when π is a write, and the timestamp associated with the returned value when π is a read. We define the partial order as follows. For two operations π_1 and π_2:

- if π_1 is any operation and π_2 is a write, then $\pi_1 \prec \pi_2$ if $ts_{\pi_1} < ts_{\pi_2}$
- if π_1 is a write and π_2 is a read, then $\pi_1 \prec \pi_2$ if $ts_{\pi_1} \leq ts_{\pi_2}$

The rest of the order is established by transitivity, without ordering the reads with the same timestamps.

Monotonicity allows the ordering of the values according to their associated timestamps. So Lemma 2 shows that the ts variable maintained by each process in the system is monotonically increasing. Let us first make the following observation:

Lemma 1 *In any execution ξ of the algorithm, 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 2 *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 $T' > T$.*

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 π. By Lemma 1 the server 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 3 *In any execution ξ of the algorithm, 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 ρ_1, ρ_2 s.t. $\rho_1 \rightarrow \rho_2$, ρ_2 returns a ts' that is bigger or equal to the timestamp ts returned by ρ_1. This can be easily shown by the fact that ρ_2 attaches the maximum timestamp discovered by the reader before the execution of ρ_2. Say this is ts discovered during ρ_1. By Lemma 2 any server that will receive the message from ρ_2 will reply with a timestamp $ts_s \geq ts$. So ρ_2 will discover a maximum timestamp $ts' \geq ts$. If $ts' = ts$ then the predicate will hold for $\alpha = 1$ for ρ_2 and thus it stores $ts' = ts$. If $ts' > ts$ then ρ_2 stores either ts' or $ts' - 1$. In either case it stores a timestamp greater or equal to ts and the lemma follows.

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.

Lemma 4 *In any execution ξ of the algorithm, if a read ρ by r_1 succeeds a write operation ω by w that writes timestamp ts_ω, i.e. $\omega \to \rho$, and returns a timestamp ts_ρ, then $ts_\rho \geq ts_\omega$.*

Proof According to the algorithm, the write operation ω communicates with a set of $|S_w| = |\mathcal{S}| - f$ servers before completing. Let $|S_1| = |\mathcal{S}| - f$ be the number of servers that replied to the read operation ρ. The intersection of the two sets is $|S_w \cap S_1| \geq |\mathcal{S}| - 2f$ and since $f < |\mathcal{S}|/2$ there exists at least a single server s that replied to both operations. Each server $s \in S_w \cap S_1$ replies to ω before replying to ρ. Thus, by Lemma 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 $ts_s \geq ts_\omega$. Thus there are two cases to investigate on the timestamp: (1) $ts_s > ts_\omega$, and (2) $ts_s = ts_\omega$.

Case 1: In the case where $ts_s > ts_\omega$, ρ will observe a maximum timestamp $maxTS \geq ts_s$. Since ρ returns either $ts_\rho = maxTS$ or $ts_\rho = maxTS - 1$, then $ts_\rho \geq ts_s - 1$. Thus, $ts_\rho \geq ts_\omega$ as desired.

Case 2: In this case every server in $s \in S_w \cap S_1$ replies with a timestamp $ts_s = ts_\omega$. The read ρ 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 ρ returns $ts_\rho \geq ts_\omega$. So it remains to investigate the case where $maxTS = ts_s = ts_\omega$. In this case, at least $|S_w \cap S_1| = |\mathcal{S}| - 2f$ servers replied with $maxTS$ to ρ. Also for each $s \in S_w \cap S_1$, s included both the writer identifier w and r_1 before replying to ω and ρ respectively. So s replied with a size at least $s.views \geq 2$ to ρ. Thus, given that $|\mathcal{R}| \geq 2$, the predicate holds for $\alpha = 2$ and the set $S_w \cap S_1$ for ρ, and hence it returns a timestamp $ts_\rho = ts_\omega$. And the lemma follows.

So now it remains to show that in two succeeding read operations, the latest operation returns a value that is the same or greater than the value returned by the first read. More formally:

Lemma 5 *In any execution ξ of the algorithm, 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 Let the two operations ρ_1 and ρ_2 be executed from the same process, say r_1. As explained in Lemma 3, ρ_2 will discover a maximum timestamp $maxTS \geq ts_{\rho_1}$. If $maxTS > ts_{\rho_1}$, then ρ_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 $\rho_1 \to \rho_2$, then any message sent during ρ_2 contains timestamp ts_{ρ_1}. By Lemma 2, every server s that receives the message from ρ_2 replies with a timestamp $ts_s \geq ts_{\rho_1}$. Since $maxTS = ts_{\rho_1}$, then it follows that all $|\mathcal{S}| - f$ servers that replied to ρ_2, sent the timestamp ts_{ρ_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 ρ_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_{ρ_1}. We have two cases to consider: (1) ρ_1 returns $ts_{\rho_1} = maxTS_1 - 1$, or (2) ρ_1 returns $ts_{\rho_1} = maxTS_1$.

Case 1: In this case ρ_1 returns $ts_{\rho_1} = maxTS_1 - 1$. It follows that there is a server s that replied to ρ_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 ρ_1. Let that write operation be ω. Since, $\rho_1 \to \rho_2$, then it must be the case that $\omega \to \rho_2$ as well. So by Lemma 4, ρ_2 returns a timestamp ts_{ρ_2} greater or equal to the timestamp written by ω, and thus $ts_{\rho_2} \geq maxTS_1 - 1 \Rightarrow ts_{\rho_2} \geq ts_{\rho_1}$.

Case 2: This is the case where ρ_1 returns $ts_{\rho_1} = maxTS_1$. So it follows that the predicate is satisfied for ρ_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 $maxTS_1$ and a seen set size $s.views \geq \alpha$, and $|M_1| \geq |\mathcal{S}| - \alpha f$. We know that ρ_2 receives replies from a set of servers $|S_2| = |\mathcal{S}| - f$ before completing. Let M_2 be the set of servers that replied to ρ_2 with a maximum timestamp $maxTS_2$. Since $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$, then

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

Hence, $S_2 \cap M_1 \neq \emptyset$ and by Lemma 2 every server $s \in S_2 \cap M_1$ replies to ρ_2 with a timestamp $ts_s \geq maxTS_1$. Therefore $maxTS_2 \geq maxTS_1$. If $maxTS_2 > maxTS_1$, then ρ_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 ρ_2 may skip f servers that reply to ρ_1, then $|M_1 \cap M_2| \geq |\mathcal{S}| - (a+1)f$. Recall that for each

server $s \in M_1 \cap M_2$, s replied with a size $s.views \geq a$ to ρ_1. Also s adds r_2 in its seen set before replying to ρ_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 ρ_2' from r_2. Since each process satisfies well-formedness, it must be the case that r_2 completed ρ_2' before invoking ρ_2. All the messages sent by ρ_2 contained $maxTS_1$. So by Lemma 2 any server $s \in S_2$ replies to r_2 with a timestamp $ts_s = maxTS_2 = maxTS_1$. In this case $|\mathcal{S}| - f$ 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 ρ_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 ρ_2. So if the size of the set $s.views = \alpha$ when s replied to ρ_1, s includes a size $s.views \geq a + 1$ when replying to ρ_2. Notice here that if $\alpha = |\mathcal{R}|+1$ for ρ_1, then it means that r_2 was already part of the seen set of s when s replied to ρ_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 $s.views \geq \alpha + 1$ to ρ_2 and since $|M_1 \cap M_2| \geq |\mathcal{S}| - (\alpha+1)f$, then the predicate holds for $\alpha+1 \leq |\mathcal{R}|+1$ and the set $MS = M_1 \cap M_2$ for ρ_2, and thus ρ_2 returns $ts_{\rho_2} = maxTS_2 = maxTS_1 = ts_{\rho_1}$ in this case as well. And this completes our proof.

Theorem 3 *Algorithm* ccFast *implements a SWMR atomic read/write register.*

Proof We now use the lemmas above and the partial order definition to reason about each of the three conditions *P1*, *P2* and *P3*.

P1. 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 it follows that the timestamp of π_2 is no less than the one of π_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 π_2 is a write, the sole writer generates a new timestamp by incrementing the largest timestamp in the system. By well-formedness (see Section 2), any timestamp generated in any write operation that precedes π_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 π_2 is a read and π_1 a write, then by Lemma 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.

P2. 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 ω is greater than the one returned from π, i.e. $ts_\omega > ts_\pi$, then $\pi \prec \omega$ follows directly. Similarly, if $ts_\pi < ts_\omega$ holds, then $\omega \prec \pi$ follows. If $ts_\omega = ts_\pi$, then it must be that π is a read and either discovered ts_ω from a set of servers and the predicate is satisfied, or π discovered $ts_\omega + 1$ but the predicate is not satisfied. Thus, $\omega \prec \pi$ follows.

P3. 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 ω be the last write preceding read ρ. From our definition it follows that $ts_\rho \geq ts_\omega$. If $ts_\rho = ts_\omega$, then ρ either: (i) discovered ts_ω as the maximum timestamp from some servers and their replies satisfied the predicate, or (ii) discovered the value written by some write ω' with timestamp $ts_\omega + 1$ but the replies received did not satisfy the predicate. If (i) holds then it is clear that ω is the last preceding write. If (ii) holds then by Lemma 4, and since $ts_\rho = ts_\omega$, it must be the case that ρ is concurrent with ω' and hence ω is again the last preceding write. Lastly, if ρ discovered $ts = 0$ 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.

6 A Linear Algorithm for the Predicate and Complexity of ccFast

Table 1 presents the comparison of the complexities of ccFast with the rest of the algorithms.
Communication Complexity. The *communication complexity* of ccFast is identical to the communication complexity of Fast: both read and write operations terminate at the end of their first communication round trip.
Computation 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 ccFast we design and analyze an algorithm to compute the predicate during any read operation.

Algorithm 2 presents the formal specification of the algorithm. Briefly, we assume that the input of the al-

Algorithm 2 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:         end if
7:     end for
8:     for α ← |R| + 1 down to 2 do
9:         if buckets[α] ≥ (|S| − αf) then
10:            return(True)
11:        else
12:            buckets[α − 1] ← buckets[α − 1] + buckets[α]
13:        end if
14:    end for
15:    if buckets[1] = (|S| − f) then
16:        return(True)
17:    end if
18:    return(False)
19: end function
```

gorithm 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 $|\mathcal{R}|+1$ "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 (A2:L3-7). 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 $|\mathcal{S}| - kf$. 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 (A2:L8-14). 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*.

Theorem 4 *Algorithm 2 implements the predicate used in every read operation in algorithm* ccFast.

Proof To show that Algorithm 2 correctly implements the predicate used by the read operations in ccFast, we need to show that it returns *True* whenever the predicate holds and returns *False* otherwise. Recall that the predicate is the following:

$$\exists \alpha \in [1, |\mathcal{R}|+1] \text{ s.t.} \quad (6)$$

$$MS = \{s : (s,m) \in maxAck \ \wedge \ m.views \geq \alpha\} \wedge \ |MS| \geq |\mathcal{S}| - \alpha f \quad (7)$$

According to our implementation we have a bucket for each α. For each α the predicate demands that we collect all the servers that replied with $maxTS$ and with $views \geq \alpha$ (set MS). Then we check if these servers are more than $|\mathcal{S}| - \alpha f$. Let $\mathcal{S}_i = \{s : s \in srvAck \ \wedge \ s.ts = maxTS \ \wedge \ s.views = i\}$, for $1 \leq i \leq |\mathcal{R}|+1$, be the set of servers who replied with $views = i$. Since each server includes a single $views$ number, notice that for any $i, j \in [1, |\mathcal{R}|+1]$, $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$.

It is easy to see that initially each bucket k, for $1 \leq k \leq |\mathcal{R}|+1$, holds the number of servers with exactly k views, and hence $bucket[k] = |\mathcal{S}_k|$. Notice that the last bucket $|\mathcal{R}|+1$ collects all the servers that replied to all possible processes (including the writer). Thus, no server may reply with $views > |\mathcal{R}|+1$. So, if the predicate is valid for $\alpha = |\mathcal{R}|+1$, it follows that $MS = S_{|\mathcal{R}|+1}$, and hence $|S_{|\mathcal{R}|+1}| \geq |\mathcal{S}| - (|\mathcal{R}|+1)f$. Since $bucket[|\mathcal{R}|+1] = |S_{|\mathcal{R}|+1}|$, then $bucket[|\mathcal{R}|+1] \geq |\mathcal{S}| - (|\mathcal{R}|+1)f$ and the condition of Algorithm 2 also holds. Thus, the algorithm returns TRUE in this case.

It remains to investigate any case where $\alpha < |\mathcal{R}|+1$. Notice that the MS set in the predicate includes all the servers that replied with $views \geq \alpha$. Thus, for any $\alpha < |\mathcal{R}|+1$,

$$MS = \bigcup_{\alpha \leq i \leq |\mathcal{R}|+1} \mathcal{S}_i$$

Since no two sets $\mathcal{S}_i$ and $\mathcal{S}_j$ intersect, then

$$|MS| = \sum_{\alpha \leq i \leq |\mathcal{R}|+1} |\mathcal{S}_i|$$

When a bucket $k < |\mathcal{R}|+1$ is investigated the value of the bucket becomes

$$bucket[k] = \sum_{k \leq i \leq |\mathcal{R}|+1} bucket[i]$$

where $bucket[i] = |\mathcal{S}_i|$, the initial value of the bucket. Thus, the above summation can be written as

$$bucket[k] = \sum_{k \leq i \leq |\mathcal{R}|+1} |\mathcal{S}_i|$$

Therefore, $bucket[k] = |MS|$, whenever $k = \alpha$. Hence, if $|MS| \geq |\mathcal{S}| - \alpha f$ in the predicate it must be the case that $bucket[\alpha] \geq |\mathcal{S}| - \alpha f$ in the algorithm. It follows that if the predicate is valid the algorithm returns *True*. Similarly, if the condition does not hold for the predicate it does not hold for the algorithm either. If there is no α to satisfy the predicate then there is no k to satisfy the condition in the algorithm. Thus, the algorithm in this case returns *False*, completing the proof.

Finally we can analyze the complexity of Algorithm 2 which in turn specifies the computational complexity of the ccFast. Algorithm 2 traverses once the set

srvAck and once the array of $|\mathcal{R}| + 1$ buckets. Since, *srvAck* contains at most $|\mathcal{S}|$ servers, and $|\mathcal{R}|$ is bounded by $|\mathcal{S}|$, then the complexity of the algorithm is:

Theorem 5 *Algorithm 2 takes $O(|\mathcal{S}|)$ time.*

This shows that we can compute the predicate of algorithm ccFast in *linear* time with respect to the number of servers in the system. This is a huge improvement over the time required by the Fast algorithm, and matches the computational efficiency of the two round ABD algorithm. This result demonstrates that fastness does not necessarily has to sacrifice computation efficiency.

7 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. $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$. In this section we propose a modification to ccFast that removes the bound on the number of readers. To unbound 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) it 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 $\frac{|\mathcal{S}|}{f} - 2$. 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 (second round) to send the latest value to a majority of servers (given that he did not already propagated that value). 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 [11]) is necessary per write operation. In contrast to [11] however, ccHybrid does not examine *which* nodes where recorded in common by the servers (see Section 3), but rather examines *how many* readers observed a value. This enables ccHybrid to reduce computation time.

The code for the reader and server protocols is given in Algorithm 3. Now we give additional details.

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 values and timestamp variables associated with the object, as well as the set variables at the clients, and *seen* sets at the servers, are used as in ccFast. 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. Below we provide a brief description of the predicate and the protocol of each participant of the service.

Writer Protocol. The write protocol remains the same as in both ccFast and ABD (see Algorithm 1): writer w increments its local timestamp and broadcasts a writeRequest message to all the participating servers $\mathcal{S}$. Once the writer receives writeAck messages from $|\mathcal{S}| - f$ servers, the operation completes.

Reader Protocol. 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 $\frac{|\mathcal{S}|}{f} - 2$. More in detail, when a read process r invokes a read operation it sends readRequest messages to all the servers and waits to collect messages from $|\mathcal{S}| - f$ servers (A3:L10-11). When readAck messages are received from a majority of servers, the reader discovers the maximum timestamp, $maxTS$, among the replies (A3:L12), the set of messages $maxAck$ that contained $maxTS$ (A3:L13), and the maximum views reported in those messages (A3:L15). If the maximum views are less than $\frac{|\mathcal{S}|}{f} - 2$ and no reader propagated the maximum timestamp, $propSet = \emptyset$, (A3: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 $|\mathcal{S}| - f$ servers before completion (A3:L18-L23).

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

Algorithm 3 Read and Server protocols of algorithm ccHybrid

1: at each reader r
2: **Components:**
3: $ts \in \mathbb{N}^+, maxTS \in \mathbb{N}^+, v \in V, vp \in V, rcounter \in \mathbb{N}^+$
4: $propSet \subseteq \mathcal{S}$, $srvAck \subseteq \mathcal{S} \times M$, $maxAck \subseteq \mathcal{S} \times M$, $maxViews \in \mathbb{N}^+$
5: **Initialization:**
6: $ts \leftarrow 0$; $maxTS \leftarrow 0$; $v \leftarrow \perp$; $vp \leftarrow \perp, rcounter \leftarrow 0$
7: $propSet \leftarrow \emptyset$, $srvAck \leftarrow \emptyset$, $maxAck \leftarrow \emptyset$, $maxViews \leftarrow 0$
8: **function** Read( )
9: $rcounter \leftarrow rcounter + 1$
10: **send**($\langle$readRequest$, ts, v, vp, r, rcounter\rangle$) to $\mathcal{S}$
11: **wait until** $(|srvAck| = |\mathcal{S}| - f)$
12: $maxTS \leftarrow \max(\{m.ts' | (s, m) \in srvAck\})$
13: $maxAck \leftarrow \{(s, m) | (s, m) \in srvAck \wedge m.ts' = maxTS\}$
14: $\langle ts, v, vp\rangle \leftarrow m.\langle ts', v', vp'\rangle$ for $(*, m) \in maxAck$
15: $maxViews \leftarrow \max(\{m.views | (s, m) \in maxAck\})$
16: $propSet \leftarrow \{s | (s, m) \in maxAck \wedge m.prop = True\}$
17: **if** $(maxViews > \frac{|\mathcal{S}|}{f} - 2) \vee (propSet \neq \emptyset)$ **then**
18: **if** $(|propSet| < f + 1)$ **then**
19: $rcounter \leftarrow rcounter + 1$
20: $srvAck \leftarrow \emptyset$
21: **send**($\langle$writeRequest$, ts, v, vp, r, rcounter\rangle$) to $\mathcal{S}$
22: **wait until** $(|srvAck| = |\mathcal{S}| - f)$
23: **end if**
24: return(v)
25: **else**
26: **if** $\exists \alpha \in [1, \frac{|\mathcal{S}|}{f} - 2]$ s.t.
27: $MS = \{s : (s, m) \in maxAck \;\wedge\; m.views \geq \alpha\} \;\wedge$
28: $|MS| \geq |\mathcal{S}| - \alpha f$ **then**
29: return(v)
30: **else**
31: return(vp)
32: **end if**
33: **end if**
34: **end function**

35: **Upon receive** m from s
36: **if** $(m.counter = rcounter)$ **then**
37: $srvAck \leftarrow srvAck \cup \{(s, m)\}$
38: **end if**

39: at each server s_i
40: **Components:**
41: $ts \in \mathbb{N}^+, seen \subseteq \mathcal{R} \cup \{w\}, v \in V, vp \in V, prop \in \{True, False\}$
42: $Counter[1..|\mathcal{R}| + 2]$: array of int
43: **Initialization:**
44: $ts \leftarrow 0, seen \leftarrow \emptyset, v \leftarrow \perp, vp \leftarrow \perp, prop \leftarrow False$
45: $Counter[i] \leftarrow 0$ for $i \in \mathcal{R} \cup \{w\}$

46: **Upon receive**($\langle$writeRequest$, ts', v', vp', w, wcounter\rangle$)
47: **if** $(Counter[w] < wcounter)$ **then**
48: $Counter[w] \leftarrow wcounter$
49: **if** $(ts < ts')$ **then**
50: $\langle ts, v, vp\rangle \leftarrow \langle ts', v', vp'\rangle$
51: $seen \leftarrow \{w\}$
52: $prop \leftarrow False$
53: **else**
54: $seen \leftarrow seen \cup \{w\}$
55: **end if**
56: **end if**
57: **send**($\langle$writeAck$, Counter[w], s\rangle$) to w

58: **Upon receive**($\langle$readRequest$, ts', v', vp', r, rcounter\rangle$)
59: **if** $(Counter[r] < rcounter)$ **then**
60: $Counter[r] \leftarrow rcounter$
61: **if** $(ts' > ts)$ **then**
62: $\langle ts, v, vp\rangle \leftarrow \langle ts', v', vp'\rangle$
63: $seen \leftarrow \{q\}$
64: $prop \leftarrow False$
65: **else**
66: $seen \leftarrow seen \cup \{r\}$
67: **end if**
68: **if** $(ts' = ts)$ **then**
69: $prop \leftarrow True$
70: **end if**
71: **send**($\langle$readAck$, ts, v, vp, |seen|, prop, Counter[r]\rangle$) to r
72: **end if**

processes that requested this replica, and a flag *prop* that, as we explained earlier, its use is to optimize read operations. Each server $s \in \mathcal{S}$ expects two types of messages.

(1) Upon receiving $\langle$readRequest$, ts', v', vp', r, rcounter\rangle$ message from reader r server 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 *prop* flag to $False$ (A3:L62-63). 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 (A3:L66). 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 (A3:L69). 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. Lastly, reader acknowledges the requesting reader with a readAck message (A3:L57).

(2) Upon receiving $\langle$writeRequest$, ts', v', vp', w, wcounter\rangle$ 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 *prop* flag to $False$ (A3:L49-54). It then acknowledges the requesting writer with a writeAck message (A3:L57).

Algorithm Correctness

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

Liveness. Termination holds with respect to our failure model: $|\mathcal{S}| - f$ servers do not fail and and each operation waits for no more than $|\mathcal{S}| - f$ messages for completion. We now give additional details regarding termination of a read operation.

Read Operation. Each operation ρ sends readRequest messages to all the servers in exchange E1 and waits for $|\mathcal{S}| - f$ readAck messages from exchange E2. According to our failure model $|\mathcal{S}| - f$ 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-value pair before termination, then ρ sends writeRequest messages to all the servers in exchange E3 and waits for $|\mathcal{S}|-f$ writeAck messages from exchange E4. Since at least $|\mathcal{S}|-f$ servers receive the messages from r during exchanges E1 and E3 and at least $|\mathcal{S}|-f$ send an acknowledgment message to r during exchanges E2 and E4, and r awaits for no more than $|\mathcal{S}|-f$ messages, then termination of ρ is always guaranteed.

Atomicity. To prove atomicity, we use the association between the timestamps and the partial order as given in Section 5. Due to the similarity of the writer and server protocols to the ones used in ccFast, we omit some of the proofs. We now state and prove the following lemmas.

Monotonicity allows the ordering of the values according to their associated timestamps. So Lemma 6 shows that the ts variable maintained by each server process in the system is monotonically increasing. Let us first make the following observation:

Lemma 6 *In any execution ξ 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 π. 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.

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 7 *In any execution ξ of* ccHybrid*, 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 $ts' \geq ts$.*

Using the next two lemmas, we show that if a read operation ρ_2 succeeds read operation ρ_1, then ρ_2 always returns a value at least as recent as the one returned by ρ_1.

Lemma 8 *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. $ts_2 \geq ts_1$.*

Proof Since the predicate holds for ρ_1, hence there exists an $\alpha \in [1, \frac{|\mathcal{S}|}{f} - 2]$, and $MS_1 \subseteq \mathcal{S}$ s.t. $|MS_1| = |\mathcal{S}| - \alpha f$, and $\forall s \in MS_1, s.ts = ts_1$ and $s.views \geq \alpha$. Performing the substitutions follows that:

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

Since ρ_2 receives replies from $|S_2| = |\mathcal{S}|-f$ servers, then there exists a server $s \in MS_1 \cap S_2$ that 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 ts_1 to ρ_1, then according to Lemma 6, it replies with a timestamp $ts_s \geq ts_1$ to ρ_2. Thus, ρ_2 observes a timestamp $maxTS \geq ts_1$ and hence $ts_2 \geq ts_1$.

Lemma 9 *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 Lemma 5, 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 6, 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 8, 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 Lemma 7, 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 $\rho_1 \rightarrow \rho_2$, then $\omega_1 \rightarrow \rho_2$. Since ρ_2 is slow, then it returns the maximum timestamp it observes, i.e. $ts_2 = maxTS_2$. Moreover, since $\omega_1 \rightarrow \rho_2$, 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 6, 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 8 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 [2]. 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 $\rho_1 \rightarrow \rho_2$, 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 6, 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 6 *Algorithm* ccHybrid *implements a SWMR atomic read/write register.*

Proof We now use the lemmas stated above and the operations order definition to reason about each of the three *atomicity* conditions *P1*, *P2* and *P3* as given in Section 2 following [16].

P1. 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 9 it follows that the timestamp of π_2 is no less than the one of π_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 π_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 π_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 π_2 is a read and π_1 a write, then by Lemma 7 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.

P2. 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 ω is greater than the one returned from π, 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 π is a read and either (i) discovered ts_ω from a propagation set, *propSet*, written by ω, or (ii) discovered ts_ω from a set of servers and the predicate is satisfied, or (iii) π discovered $ts_\omega + 1$ but the predicate is not satisfied. Thus, $\omega \prec \pi$ follows.

P3. 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 ω be the last write preceding read ρ. From our definition it follows that $ts_\rho \geq ts_\omega$. If $ts_\rho = ts_\omega$, then ρ either: (i) discovered ts_ω from a propagation set, *propSet*, written by ω, or (ii) discovered ts_ω from a set of servers and the predicate is satisfied, or (iii) π discovered $ts_\omega+1$ but the predicate is not satisfied. If case (i) holds then, it is clear that ω is the last preceding write since ρ discovered ts_ω as the maximum timestamp $maxTS$ and

either (a) it was propagated to a set of servers and ρ returns ts_ω without any further actions or (b) ρ propagates ts_ω to a set of servers before completion. When case (ii) holds, then it is clear that ω is the last preceding write. If (iii) holds then by Lemma 7, and since $ts_\rho = ts_\omega$, it must be the case that ρ is concurrent with ω' and hence ω is again the last preceding write. If $ts_\rho > ts_\omega$, then it means that ρ obtained a larger timestamp. However, the larger timestamp can only be originating from a write that succeeds ω, thus ω 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.

Having shown liveness and atomicity of algorithm ccHybrid the result follows.

8 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 *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.

Algorithm 4 provides the formal pseudocode of OhFast. We omit the receipt of a writeRequest message on the server side as it is the same to the one presented for ccFast (see Algorithm 1). We now give additional details.

Again here the variables are used as in algorithms ccFast and ccHybrid. 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. The write protocol is very similar as in both ccFast and ccHybrid (see Algorithm 1): writer w increments its local timestamp and broadcasts a writeRequest message to all the participating servers $\mathcal{S}$. Once the writer receives writeAck messages from $|\mathcal{S}| - f$ servers, the operation completes.

Reader Protocol. The read protocol in OhFast is simpler than the read of ccHybrid. When a read process r invokes a read operation it sends readRequest messages to all the servers and waits to collect messages from $|\mathcal{S}| - f$ servers (A4:L11-12). Once those replies are received the reader discovers the maximum timestamp $maxTS$ among the replies (A4:L13) [1], and collects all the messages that contain $maxTS$ in the set $maxAck$ (A4: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 $maxTS$ (A4:L17-18). Otherwise, the reader evaluates the predicate, that ccFast [5] 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 $maxTS - 1$ (A4:L21-25).

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 expects three types of messages.

(1) Upon receiving $\langle$readRequest$, ts', v', vp', r, rcounter\rangle$ message from reader r, server s_j 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 *securedts* flag to $False$ (A4:L63-64). If not, then the server adds the sender to

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

Algorithm 4 Reader and Server protocols of algorithm OhFast

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

27: **Upon receive** m from s
28: **if** $(m.rcounter = rcounter)$ **then**
29: $\quad srvAck \leftarrow srvAck \cup \{(s,m)\}$
30: **end if**

31: At writer w
32: **Components:**
33: $ts \in \mathbb{N}^+,\ v, vp \in V, wcounter \in \mathbb{N}^+$
34: **Initialization:**
35: $ts \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, wcounter \leftarrow 0$
36: **function** Write($val : input$)
37: $\quad vp \leftarrow v$
38: $\quad v \leftarrow val$
39: $\quad ts \leftarrow ts + 1$
40: $\quad wcounter \leftarrow wcounter + 1$
41: $\quad wAck \leftarrow \emptyset$
42: $\quad$ **broadcast**($\langle$writeRequest, $ts, v, vp, w, wcounter\rangle$) to $\mathcal{S}$
43: $\quad$ **wait until** $(|wAck| = |\mathcal{S}| - f)$
44: $\quad$ **return**
45: **end function**

46: **Upon receive** m from s
47: **if** $(m.wcounter = wcounter)$ **then**
48: $\quad wAck \leftarrow wAck \cup \{(s,m)\}$
49: **end if**

50: at each server s_j
51: **Components:**
52: $ts \in \mathbb{N}^+, v \in V, vp \in V, scounter \in \mathbb{N}^+, securedts \in \{True, False\}$
53: $seen \subseteq \mathcal{R} \cup \{w\}$, $srvRelay \subseteq \mathcal{S}$
54: $Relays[1..|\mathcal{R}|+1]$: array of int, $Counter[1..|\mathcal{R}|+2]$: array of int
55: **Initialization:**
56: $ts \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, scounter \leftarrow 0, securedts \leftarrow False$
57: $seen \leftarrow \emptyset$, $srvRelay \leftarrow \emptyset$
58: $Counter[i] \leftarrow 0$ for $i \in \mathcal{R} \cup \{w\}$, $Relays[i] \leftarrow 0$ for $i \in \mathcal{R}$

59: **Upon receive**($\langle$readRequest, $ts', v', vp', r, rcounter\rangle$)
60: **if** $(Counter[r] < rcounter)$ **then**
61: $\quad Counter[r] \leftarrow rcounter$
62: $\quad$ **if** $(ts < ts')$ **then**
63: $\quad\quad \langle ts, v, vp\rangle \leftarrow \langle ts', v', vp'\rangle$
64: $\quad\quad seen \leftarrow \{r\}$, $securedts \leftarrow False$
65: $\quad$ **else**
66: $\quad\quad seen \leftarrow seen \cup \{r\}$
67: $\quad$ **end if**
68: $\quad$ **if** $(|seen| > \frac{|\mathcal{S}|}{f} - 2) \wedge (securedts = False) \wedge (Relays[r] < ts)$ **then**
69: $\quad\quad scounter \leftarrow scounter + 1$
70: $\quad\quad Relays[r] \leftarrow ts$
71: $\quad\quad srvRelay \leftarrow \emptyset$
72: $\quad\quad$ **send**($\langle$readRelay, $ts, v, vp\rangle, r, s_j, rcounter, scounter$) to $\mathcal{S}$
73: $\quad$ **else**
74: $\quad\quad$ **send**($\langle$readAck, $ts, v, vp\rangle, |seen|, rcounter, securedts$) to r

75: $\quad$ **end if**
76: **end if**

77: **Upon receive**($\langle$readRelay, $ts', v', vp', r, s, c1, c2\rangle$)
78: **if** $(Counter[s] < c2)$ **then**
79: $\quad Counter[s] \leftarrow c2$
80: $\quad$ **if** $(ts' > ts)$ **then**
81: $\quad\quad \langle ts, v, vp\rangle \leftarrow \langle ts', v', vp'\rangle$
82: $\quad\quad seen \leftarrow \{r\}$
83: $\quad$ **else if** $(ts = ts')$ **then**
84: $\quad\quad seen \leftarrow seen \cup \{r\}$
85: $\quad$ **end if**
86: $\quad$ **if** $(Relays[r] = ts')$ **then**
87: $\quad\quad srvRelay \leftarrow srvRelay \cup \{s\}$
88: $\quad\quad$ **if** $(|srvRelay| = |\mathcal{S}| - f)$ **then**
89: $\quad\quad\quad$ **if** $(ts = ts')$ **then**
90: $\quad\quad\quad\quad securedts \leftarrow True$
91: $\quad\quad\quad$ **end if**
92: $\quad\quad\quad$ **send**($\langle$readAck, $ts', v', vp', 0, c1, securedts\rangle$) to r
93: $\quad\quad$ **end if**
94: $\quad$ **else**
95: $\quad\quad scounter \leftarrow scounter + 1$
96: $\quad\quad$ **send**($\langle$readRelay, $ts', v', vp'\rangle, r, s_j, c1, scounter$) to s
97: $\quad$ **end if**
98: **end if**

the seen set (A4:L66). Next, s_j must decide whether to relay the received timestamp or not. In particular, s_j relays a timestamp to all the servers (A4:L72) if : (i) it sent this timestamp to more than $\frac{|\mathcal{S}|}{f} - 2$ 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 A4:L68). Otherwise, if any of these conditions does not hold then s_j just replies to the sender with its local timestamp (A4:L74). 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 $\langle$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 (A4:L80-84). Then s_j checks if it also sent a relay with the same timestamp for the same reader (A4:L86). If this holds, server s_j adds sender s in the servers that received its relay (A4:L87). When the server receives $|\mathcal{S}| - f$ 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 $securedts = True$ (A4:L88-

92). If 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 (A4:L96).

(3) Upon receiving $\langle$writeRequest$, ts', v', vp', w, wcounter\rangle$ 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*. It then acknowledges the requesting writer with a writeAck message.

Algorithm Correctness

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

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

Write Operation. Per algorithm OhFast, writer w creates a writeRequest message and then it broadcasts it to all servers in exchange E1 (A4:L42). Writer w then waits for writeAck messages from $|\mathcal{S}|-f$ servers from E2 (A4:L43). According to our failure model $|\mathcal{S}|-f$ servers do not fail and can receive writeRequest and send writeAck messages to the requesting writer, thus a write operation ω terminates.

Read Operation. Each operation ρ sends readRequest messages to all the servers (A4:L11) and waits for $|\mathcal{S}|-f$ replies before terminating (A4:L12). Thus termination of such process is prevented if less than $|\mathcal{S}|-f$ servers reply to r for operation ρ. 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 $|\mathcal{S}|-f$ 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 (A4:L92), or (b) appends its local *srvRelay* set with the sender if $Relays[r]=ts$ (A4:L87). 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 (A4:L72). 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 $|\mathcal{S}|-f$ readRelay messages that contain ts and thus the check in A4:L88 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 $|\mathcal{S}|-f$ readAck messages and the read operation ρ terminates.

Atomicity. Next it remains to show that atomicity is preserved. We use the association between the timestamps and the partial order as given in Section 5.

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. 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 2. 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 10 *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.

Lemma 11 *In any execution ξ of* OhFast*, if a read ρ from r succeeds a write operation ω that writes timestamp ts_ω from the writer w, i.e. $\omega \rightarrow \rho$, and returns a timestamp ts_ρ, then $ts_\rho \geq ts_\omega$.*

Next, 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 12 *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 $ts' \geq ts$.*

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 $securedts = True$ has received a set $|\mathcal{S}'| \geq |\mathcal{S}|-f$ of replies, and every server $s' \in \mathcal{S}'$ has a timestamp $ts' \geq ts$, by Lemma 10. Thus the lemma follows.

Next, we show that if a read operation ρ_2 succeeds read operation ρ_1, then ρ_2 always returns a value at least as recent as the one returned by ρ_1.

Lemma 13 *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 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 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 [5, 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:

$$\exists \alpha \in [1, \frac{|\mathcal{S}|}{f} - 2] \text{ and } MS \subseteq \mathcal{S} \text{ s.t.} \quad (8)$$

$$MS = \{s : s.ts = maxTS_1 \;\wedge\; s.views \geq \alpha\} \;\wedge\; |MS| \geq |\mathcal{S}| - \alpha f \quad (9)$$

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 (L66), 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 $\rho_1 \to \rho_2$, then a server $s \in \mathcal{S}_2 \cap MS$ replies to ρ_1 with $maxTS_1$ before replying to ρ_2. By Lemma 2, 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 $\rho_1 \to \rho_2$, then it follows that $\omega \to \rho_2$. 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 2, 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 12, 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 2 and 10, 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 (L88). 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 12, 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 10, 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 7 *Algorithm* OhFast *implements a SWMR atomic read/write register.*

Proof We now use the lemmas stated above and the operations order definition to reason about each of the three *atomicity* conditions *P1*, *P2* and *P3* as given in Section 2 following [16].

P1. 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 13 it follows that the timestamp of π_2 is no less than the one of π_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 π_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 π_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 π_2 is a read and π_1 a write, then by Lemma 11 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.

P2. 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 ω is greater than the one returned from π, i.e. $ts_\omega > ts_\pi$, then $\pi \prec \omega$ follows directly. Similartly, if $ts_\omega < ts_\pi$ holds, then $\omega \prec \pi$ follows. If $ts_\omega = ts_\pi$, then it must be that π is a read and either (i) ρ discovered ts_ω from a set of messages that contained ts_ω 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_ω from a set of servers and the predicate is satisfied, or (iii) π discovered $ts_\omega + 1$ but the predicate is not satisfied. Thus, $\omega \prec \pi$ follows.

P3. 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 ω be the last write preceding read ρ. From our definition it follows that $ts_\rho \geq ts_\omega$. If $ts_\rho = ts_\omega$, then ρ either: (i) ρ discovered ts_ω from a set of messages that contained ts_ω 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_ω as the maximum timestamp from some servers and their replies satisfied the predicate, or (iii) discovered the value written by some write ω' with timestamp $ts_\omega + 1$ but the replies received did not satisfy the predicate. If case (i) holds, ω is the last preceding write since ρ discovered ts_ω as the maximum timestamp, $ts_\omega = maxTS$ and it was propagated to a set of servers and ρ returns ts_ω without any further actions. When case (ii) holds, then it is clear that ω is the last preceding write. If (iii) holds then by Lemma 11, and since $ts_\rho = ts_\omega$, it must be the case that ρ is concurrent with ω' and hence ω is again the last preceding write. If $ts_\rho > ts_\omega$, then it means that ρ obtained a larger timestamp. However, the larger timestamp can only be originating from a write that succeeds ω, thus ω 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.

Having shown liveness and atomicity of algorithm OhFast the result follows.

9 Empirical Results

In this section, we present empirical results that we obtained by implementing algorithms ABD [2], OhSam [12], Sf [11], ccHybrid, and OhFast that allow unbounded participation, using the NS3 discrete event simulator [1]. 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. As NS3 introduces some limitations on retrieving the exact performance parameters as we discuss later, we do use these simulations as a *proof-of-concept*. Further experiments in real systems are necessary to reveal more realistic differences between the 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 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 2 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 *Series* topology, Fig.2(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.2(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.

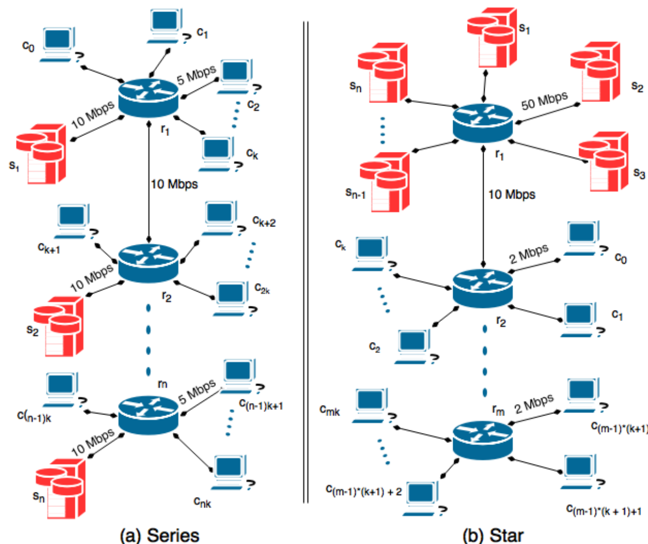

Fig. 2: Simulated topologies.

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. 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 sum of the two yields latency.

Figures	$\|\mathcal{S}\|$	Read Rate (sec)	Inv. Scheme	Topology
3(a)	15	4.6	fix	star
3(b)	15	2.3	fix	star
3(c)	30	2.3	fix	star
3(d)	15	4.6	stochastic	star
3(e)	15	2.3	stochastic	star
3(f)	15	2.3	stochastic	series
3(g)	30	4.6	stochastic	star
3(h)	30	4.6	stochastic	series
3(i)	30	6.9	stochastic	series
3(k)	15	2.3	fix	star
3(l)	15	4.6	fix	star
3(m)	15	4.6	stochastic	star
3(n)	15	4.6	stochastic	series
3(p)	15	0.0 (Pow-law)	fix	series
3(q)	15	0.0 (Pow-law)	fix	star

Table 2: Parameters used for the plots of Fig. 3.

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 $|\mathcal{R}| \in [10, 20, 40, 80, 100, 250]$ 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). As however in practical systems it is unlikely to have all reader participants to choose the same read interval, we also run our simulations with a scenario where the readers are choosing $rInt$ to be between 0 and 15 seconds based on a *power law* distribution. 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; every $rInt$ for reads and $wInt$ for writes. 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. Table 2 presents all the parameters we use for obtaining the results shown in Figure 3. The differences of the scenarios are also apparent from the table.

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

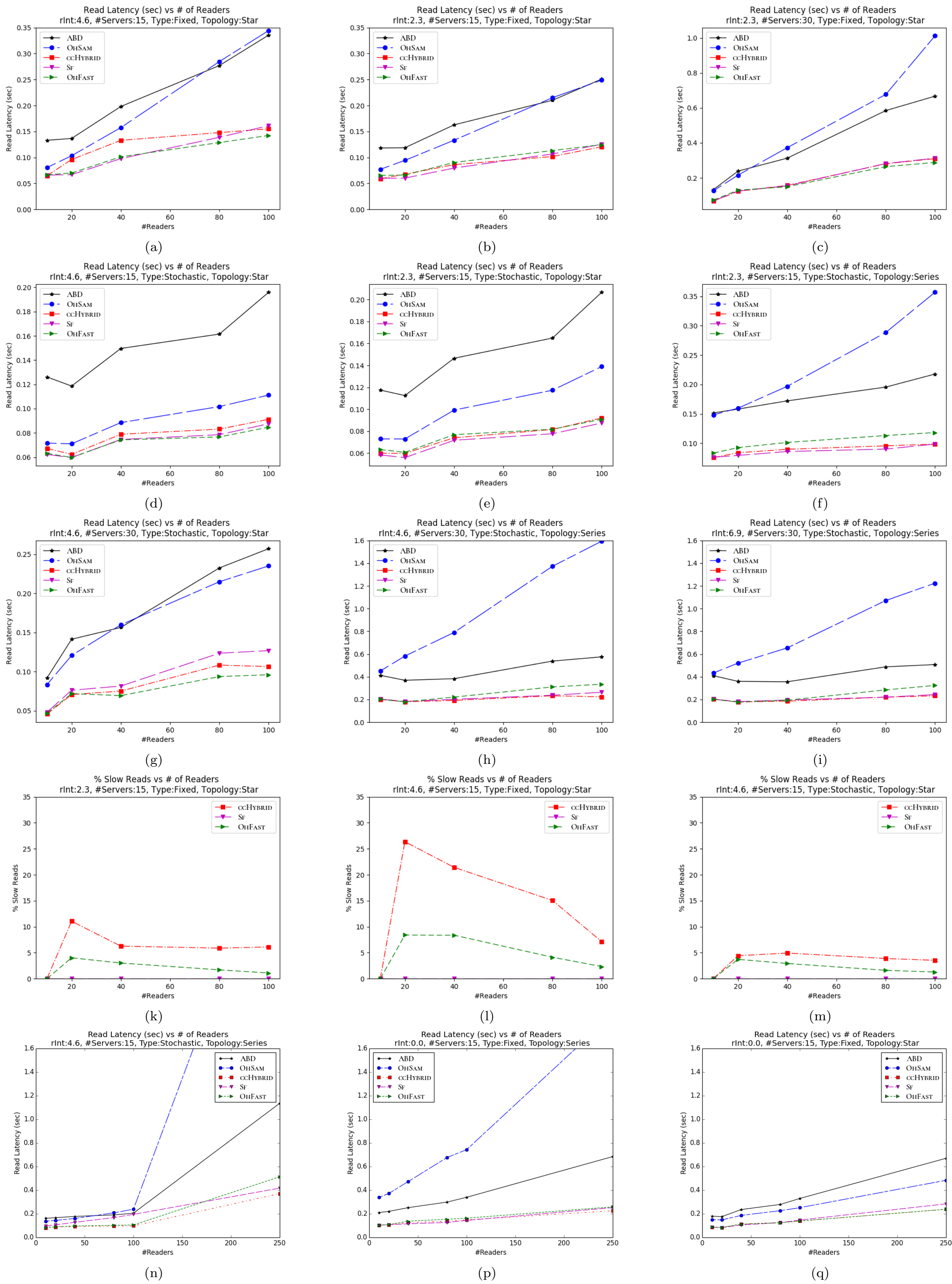

Fig. 3: Experimental Results from NS3 Simulation

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 (as expected according to [8]), 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; algorithm Sf does not have any overhead due to the excessive communication in this case. 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: The increasing number of readers and the servers have a negative impact on all the algorithms. Especially when running our simulations with 250 readers the latency of operations was much greater than when 100 or less readers participated in the service. We believe that this is both due to the excessive communication, but also some overhead might be introduced by the computation in the NS3 simulator. We do present sample results in Fig. 2(n), (p), and (q). However, for ease of comparison between the algorithms, for the rest of the section we do present the results when up to 100 readers are used. Sample plots that present the average latency of read operations for this scenario appear in Fig. 3(b) and (c). In particular, the read latency for algorithms ABD and OhSam increases dramatically even when few servers participate in the system, Fig. 3(b) and the latency becomes even higher when we double the participating servers as shown in Fig. 3(c). On the other hand, we notice algorithms Sf, ccHybrid 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.

Contention: Here we investigate how high concurrency, and thus contention, may affect the efficiency of operations. Contention depends on the following parameters: (i) the duration of each operation (which we will measure as the operation latency), (ii) the latency and symmetry of the network (which we try to capture with our topologies), and (iii) the generation rate of operations (which we try to capture with the operation frequencies and the operation invocation 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. 3(a) and (b). Algorithms ABD and OhSam are not affected (as all of their reads are slow), but the multi-speed algorithms ccHybrid 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. 3(l). This is not observed when $rInt = 2.3$ (or $rInt = 6.9$), see Fig. 3(k). The impact of the slow reads on the operation latency can be also seen in Figs. 3(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 *fix* scheme is used but the readers pick their intervals using a *power-law* distribution or when the *stochastic* scheme is used. The fact that readers pick a different interval using the power-law distribution, allow them to invoke read operations in different points in time in the *fix* scheme, see Figs 3(p) and (q). On the other hand, randomness in the *stochastic* scheme prevents the operations to be invoked at exactly the same time, see Figs. 3(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. 3(m).

Finally, under the same operation frequency, it appears that in the *stochastic* scheme each operation completes almost two times faster than in the *fix* scheme, as shown in Figs. 3(b) and (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 reads 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 3(e)(f) and 3(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. 3(e) OhSam performs better than ABD and OhFast performs similar to ccHybrid and Sf, we notice that in Fig. 3(f) OhSam performs worse than ABD and OhFast worse than ccHybrid and Sf. Same observation can be noticed in Figs. 3(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 than OhSam since operation relays are not performed for every read operation.

10 Conclusions

In this work we consider the complexity of algorithms that implement atomic SWMR registers in the asynchronous, message-passing environment where processes are prone to crashes. We examined the best known (in terms of communication delays) algorithm that implements an atomic SWMR register, Fast, allowing both reads and writes to terminate in a *single* communication round. We showed that the predicate utilized by the Fast to achieve this low latency is NP-hard, and hence the computation performed by the algorithm is not tractable. Next we presented a new predicate that can be computed in linear time, and showed how to use it in a revised atomic SWMR algorithm ccFast that allows operations to complete in a *single* communication round. The efficiency of the newly proposed predicate is demonstrated by presenting a *linear time* algorithm for its computation.

The shortcoming of ccFast is that it guarantees atomicity as long as the number of readers is bounded by $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$ (similar to Fast). To circumvent this bound we presented two new "multi-speed" algorithms, ccHybrid and OhFast, that implement atomic SWMR register. Both algorithms use the improved predicate, to achieve *single round* reads with small computational overhead. However, to remove constraints on the number of readers, both algorithms allow some reads to be *slow*. In ccHybrid the reader decides whether the operation can terminate after one round, or requires the second round. In OhFast the decision of whether operations need to be slow is made at the servers, and the algorithm allows some operations to complete in 1 or 1.5 round. Of interest it is to examine the exact conditions in the participation of the service that may allow all read operations to complete in a single round, using this server-side methodology. Such result will draw the line between on the efficiency of such algorithms. Simulation results show that in realistic settings our algorithms outperform algorithms where all operations are slow, as well as "multi-speed" algorithms that have high computation overheads.

This work shows that practical implementations of atomic registers need to take into account *communication*, *computation*, and *message bit* complexity metrics.

Acknowledgements This work was co-funded by the European Regional Development Fund and the Republic of Cyprus through the Research and Innovation Foundation (Project: POST-DOC/0916/0090), by FP7-PEOPLE-2013-IEF grant ATOMICDFS No:629088, the Spanish grant TIN2017-88749-R (DiscoEdge), the Region of Madrid EdgeData-CM program (P2018/TCS-4499), and the NSF of China grant 61520106005.

References

1. NS3 network simulator. https://www.nsnam.org/.
2. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.
3. 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.
4. Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)*, pages 240–254, 2009.
5. 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.
6. Antonio Fernández Anta, Theophanis Hadjistasi, and Nicolas Nicolaou. Computationally light "multi-speed" atomic memory. In *International Conference on Principles Of Distributed Systems*, OPODIS'16, 2016.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.
8. Chryssis Georgiou, Sotirios Kentros, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Analyzing the number of slow reads for semifast atomic read/write register implementations. In *Proceedings Parallel and Distributed Computing and Systems (PDCS09)*, pages 229–236, 2009.
9. Chryssis Georgiou, Nicolas Nicolaou, Alexander Russel, and Alexander A. Shvartsman. Towards feasible implementations of low-latency multi-writer atomic registers. In *10th Annual IEEE International Symposium on Network Computing and Applications*, August 2011.
10. 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.
11. 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.
12. 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.
13. Theophanis Hadjistasi, Nicolas C. Nicolaou, and Alexander A. Schwarzmann. Oh-ram! one and a half round atomic memory. In *Networked Systems - 5th International*

Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings, pages 117–132, 2017.
14. 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.
15. Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
16. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
17. 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.
18. Eduardo C. Xavier. A note on a maximum k-subset intersection problem. *Information Processing Letters*, 112(12):471 – 472, 2012.