Making "Fast" Atomic Operations Computationally Tractable

Antonio Fernández Anta[*1], **Nicolas Nicolaou**[†2], and **Alexandru Popa**[3]

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

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

3 **Department of Computer Science, Nazarbayev University, Astana, Kazakhstan**
`alexandru.popa@nu.edu.kz`

Abstract

Communication overhead is the most commonly used performance metric for the operation complexity of distributed algorithms in message-passing environments. However, aside with communication, many distributed operations utilize complex computations to reach their desired outcomes. Therefore, a most accurate operation latency measure should account of both *computation* and *communication* metrics.

In this paper we focus on the efficiency of read and write operations in an *atomic read/write shared memory* emulation in the message-passing environment. We examine the operation complexity of the best known atomic register algorithm, presented in [2], that allows all read and write operations to complete in a *single* communication round-trip. Such operations are called *fast*. At its heart, the algorithm utilizes a predicate to allow processes to compute their outcome. We show that the predicate used in [2] is *computationally hard*, by devising a computationally equivalent problem and reducing that to *Maximum Biclique*, a known *NP-hard* problem. To improve the computational complexity of the algorithm we derive a *new predicate* that leads to a new algorithm, we call ccFast, and has the following properties: (i) can be computed in polynomial time, rendering each read operation in ccFast tractable compared to the read operations in the original algorithm, (ii) the messages used in ccFast are reduced in size, compared to the original algorithm, by almost a linear factor, (iii) allows all operations in ccFast to be *fast*, and (iv) allows ccFast to preserve atomicity. A *linear time* algorithm for the computation of the new predicate is presented along with an analysis of the message complexity of the new algorithm. We believe that the new algorithm redefines the term *fast* capturing both the *communication* and the *computation* metrics of each operation.

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

Keywords and phrases atomicity, read/write objects, shared memory, computational complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.19

[*] The work of the first author is supported in part by FP7-PEOPLE-2013-IEF grant ATOMICDFS No:629088, Ministerio de Economia y Competitividad grant TEC2014- 55713-R, Regional Government of Madrid (CM) grant Cloud4BigData (S2013/ICE-2894, co- funded by FSE & FEDER), NSF of China grant 61520106005, and European Commission H2020 grants ReCred and NOTRE.

[†] The work of the second author is fully supported byFP7-PEOPLE-2013-IEF grant ATOMICDFS No:629088.

1 Introduction

Emulating atomic [8] (linearizable [7]) read/write objects in message-passing environments is one of the fundamental problems in distributed computing. The problem becomes more challenging when participants in the service may fail and the environment is asynchronous, i.e. it cannot provide any time guarantees on the delivery of the messages and the computation speeds. To cope with failures, traditional distributed object implementations like [1, 10], use *redundancy* by replicating the object to multiple (possibly geographically dispersed) network locations (replica servers). Replication however raises the challenge of consistency, as multiple object copies can be accessed concurrently by multiple processes. To determine the value of the object when this is accessed concurrently, researchers defined several consistency guarantees, the strongest of those being *atomicity*. Atomicity is the most intuitive consistency semantic as it provides the illusion of a single-copy object that serializes all accesses: each read operation returns the value of the latest 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 [1], was the first to present an algorithm, we refer to as ABD, to implement Single-Writer, Multiple-Reader (SWMR) atomic objects, in message-passing, crash-prone, and asynchronous environments. Here, $\langle timestamp, value\rangle$ pairs are used to order the write operations, and 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. In particular, the writer increments the timestamp for each write and propagates the new value along with its timestamp to some majority of replicas. The readers are implemented as a two-phase protocol, where the first phase obtains from some majority of the replicas their $\langle timestamp, value\rangle$ pairs, and uses the value corresponding to the highest timestamp as the return value. Before returning, each reader performs a second phase, in which it propagates the highest $\langle timestamp, value\rangle$ pair to some majority of replica servers, ensuring that any subsequent read will discover the value that is at least as recent. Here atomicity is guaranteed in all executions relying on the fact that any two majorities have a non-empty intersection. Avoidance of the second round-trip 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. [2] refuted this belief, by presenting atomic register implementations where reads involve only a *single* communication round-trip. Such an implementation is called *fast*. This is shown to be possible whenever the number of readers R is appropriately constrained with respect to the number of replicas S and the maximum number of crashes f in the system; this constraint is expressed by $R < \frac{S}{f} - 2$. In this same work the authors showed that it is not possible to devise fast implementations in the multiple-write and multiple-reader (MWMR) model. Subsequently, works like [5, 6], proposed implementations in the SWMR model where some operations were allowed to perform two communication round-trips, in an attempt to relax the constraint proposed in [2] and allow unbounded number of readers. Such implementations traded communication for scalability. Under conditions of low concurrency, this requirement allowed most reads to complete in a single communication round-trip. Even with this relaxed model [6] showed that MWMR implementations where all write are fast are not possible. Following these developments, [3] provided tight bounds on the efficiency of read/write operations in terms of communication round-trips, and introduced the first algorithm to allow some fast read and write operations in the MWMR model.

A trend appeared in the algorithms that aimed for fast operations: algorithms with lower communication rounds demanded higher computation overhead at the processes. The first work to question the computational complexities of the "fast" implementations, and how that affects the performance of the algorithms, was presented by Georgiou et al. [4]. In that paper the authors analyzed the computational complexity of the algorithm presented in [3], the only algorithm that allows both fast reads and writes in the MWMR model, and showed both theoretically and experimentally that the computational overhead of the algorithm was suppressing the communication costs. In particular, the authors expressed the predicate used in the algorithm by a computationally equivalent problem and they showed that such a problem is NP-hard. To improve the complexity of the algorithm presented in [3], they proposed a polynomial *approximation* algorithm.

Contributions. In this paper, we focus in the efficiency of read and write operations in distributed SWMR atomic read/write register implementations. We show that the computation costs have an impact on the best known atomic register implementation, presented in [2], and we propose a *deterministic* solution to improve both the computational and communication burdens of the original algorithm while maintaining fault-tolerance and consistency. Enumerated our contributions are the following:

- We introduce a new problem that is computationally equivalent to the predicate used in [2]. We show that the new problem, and thus the computation of the predicate, is *NP-hard*. For our proof we reduce the new problem to the *Maximal Biclique* problem, which is known to be an NP-hard problem.
- We then devise a revised *fast* implementation, called ccFast, which uses a new polynomial 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.
- Finally, 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 present a complexity analysis of the proposed algorithm and we prove that it correctly computes the predicate of ccFast.

Our results lower the bar of operation latency in SWMR atomic object implementations in the message-passing, asynchronous environment and redefine the term *fast* to capture both the communication and computation overheads of the proposed algorithms.

2 Model

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

Each process p of the system can be modeled as an I/O Automaton A_p [11]. The automaton A_p of process p is defined over a set of *states*, *states*(A_p), and a set of *actions*,

actions(A). There is a state $\sigma_{0,p} \in states(A_p)$ which is the initial state of automaton A_p. An algorithm A is the automaton obtained from the composition of automata A_p, for $p \in \mathcal{I}$. A state $\sigma \in states(A)$ is a vector containing a state for each process $p \in \mathcal{I}$ and the state $\sigma_0 \in states(A)$ is the initial state of the system that contains $\sigma_{0,p}$ for each process $p \in \mathcal{I}$. The set of actions of A is $actions(A) = \bigcup_{p\in\mathcal{I}} actions(A_p)$. An *execution fragment* ϕ of A is an alternate sequence $\sigma_1, \alpha_1, \sigma_2, \ldots, \sigma_{k-1}, \alpha_{k-1}, \sigma_k$ of *states* and *actions*, s.t. $\sigma_i \in states(A)$ and $\alpha_i \in actions(A)$, for $1 \leq i \leq k$. An *execution* is the execution fragment starting with some initial state σ_0 of A. We say that an execution fragment ϕ' *extends* an execution fragment ϕ (or execution), denoted by $\phi \circ \phi'$, if the last state of ϕ is the first state of ϕ'. A triple $\langle \sigma_i, \alpha_{i+1}, \sigma_{i+1} \rangle$ is called a *step* and denotes the transition from state σ_i to state σ_{i+1} as a result of the execution of action α_{i+1}. A process p *crashes* in an execution ξ if the event fail_p appears in ξ; otherwise p is *correct*. Notice that if a process p crashes, then fail_p is the last action of that process in ξ.

In a read/write atomic object implementation each automaton A contains an *invocation* action $\mathsf{read}_{p,O}$ (or $\mathsf{write}(v)_{p,O}$) to *invoke* a read (resp. write) operation π on an object O. Similarly, $\mathsf{read\text{-}ack}(v)_{p,O}$ and $\mathsf{write\text{-}ack}(v)_{p,O}$ are the *response* actions and return the result of the operation π on O. The steps that contain the invocation and response actions, are called invocation and response steps respectively. An operation π is *complete* in an execution ξ, if ξ contains both the invocation and the *matching* response actions for π; otherwise π is *incomplete.* An execution ξ is *well formed* if any process p that invokes an operation π in ξ does not invoke any other operation π' before the matching response action of π appears in ξ. In other words each operation invokes one operation at a time. Finally we say that an operation π *precedes* an operation π' in an execution ξ, denoted by $\pi \to \pi'$, if the response step of π appears before the invocation step of π' in ξ. The two operations are *concurrent* if none precedes the other.

Correctness of an implementation of an atomic read/write object is defined in terms of the *atomicity* and *termination* properties. The termination property requires that any operation invoked by a correct process eventually completes. Atomicity is defined as follows [9]. For any execution of a memory service, all the completed 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 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 register memory system. By composing multiple single register implementations, one may obtain a complete atomic memory [9]. Thus, we omit further mention of object names.

Efficiency Metrics. We are interested in the *complexity* of each read and write operation. The complexity of each operation π is measured from the invocation step of the π to the response step of π. To measure the complexity of an operation π that is invoked by a process p we use the following three metrics: (i) *communication round-trips* , (ii) *computation steps* taken by p during π, and (iii) *message bit complexity* which measures the length of the messages used during π. A communication round-trip (or simply round) is more formally defined in the following definition that appeared in [2, 6, 5]:

▶ **Definition 1.** Process p performs a communication round during operation π if all of the following hold:

1. p sends request messages that are a part of π to a set of processes,
2. any process q that receives a request message from p for operation π, replies without delay, i.e. without waiting for any other messages before replying to π.
3. when process p receives "enough" replies it terminates the round

At the end of a communication round process p may complete π or start a new round. Operation π is *fast* [2] if it completes after its first communication round; an implementation is fast if in each execution all operations are fast.

3 Fastness and its Implications in Atomic Memory Implementations

The algorithm by Dutta et al. in 2004 [2](we refer to it as FAST) was the first to present an atomic register implementation for the message-passing environment where all read and write operations required just a *single* communication round before completing. The same work showed that for any implementation to be fast it must be the case that the number of readers are constrained with respect to the number of servers and server failures in the service by $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 2$. FAST is using $\langle timestamp, value \rangle$ pairs as in ABD [1] to impose an order on the write operations. The write operation is almost identical to the one round write in [1]: the writer increments its local timestamp, and sends the new timestamp with the value to be written to the majority of the servers. The read operation is much different as it only takes a singe round to complete. To avoid the second round for each read operation, FAST uses two mechanisms: (i) a recording mechanism at the servers, and (ii) a predicate that uses the server recordings at the readers. Essentially each server records all the processes that witness its local timestamp, in a set called *seen*. This set of processes is reset whenever the server learns a new timestamp. The predicate at the readers is the following:

$$\exists \alpha \in [1, \ldots, |\mathcal{R}|+1] \ \wedge \ MS \subset \mathcal{S} \text{ s.t.} \tag{1}$$

$$\forall s \in MS, s.ts = maxTs \ \wedge |MS| \geq |\mathcal{S}| - \alpha f \ \wedge | \bigcap_{s \in MS} s.seen| \geq \alpha \tag{2}$$

Essentially the reader looks at the *seen* sets of the servers that replied, and tries to extract whether "enough" processes witnessed the maximum timestamp. If the predicate holds, the reader returns the value associated with the maximum timestamp. Otherwise it returns the value associated with the previous timestamp. Notice here that the predicate takes in account *which* processes witnessed the latest timestamp as it examines the intersection of the seen sets. Let us now examine what are the complexity costs of FAST in terms of communication, computation and message size. Table 1, presents the comparison of FAST with ABD in all three complexity metrics. Notice that we assume that all three algorithms utilize the same technique to generate timestamps. Thus, we ommit counting the overhead that the timestamp may incur to the complexities presented in Table 1.

Communication Complexity. As previously mentioned, FAST uses one communication round-trip for each read and write operation. That is, each operation sends messages to all the servers and waits replies from a majority. No further communication is required once those replies are received. ABD on the other hand needs one round per write and two rounds per read operation.

Table 1 Communication, Computation, and Message-Bit Complexities of ABD vs FAST vs CCFAST. (WR/RR: write/read-rounds, WC/RC: write/read-computation, WB/RB: write/read-message bits)

Algorithm	WR	RR	WC	RC	WB	RB
ABD	1	2	$O(1)$	$O(\|\mathcal{S}\|)$	$O(\lg \|V\|)$	$O(\lg \|V\|)$
FAST	1	1	$O(1)$	$O(\|\mathcal{S}\|^2 \cdot 2^{\|\mathcal{S}\|})$	$O(\lg \|V\|)$	$\Theta(\|\mathcal{S}\| + \lg \|V\|)$
CCFAST	1	1	$O(1)$	$O(\|\mathcal{S}\|)$	$O(\lg \|V\|)$	$O(\lg \|\mathcal{S}\| + \lg \|V\|)$

Computation Complexity. The reduction on the communication rounds had a negative impact on the computational complexity of FAST. The write operation, as also in ABD, terminates once the appropriate number of servers reply, without imposing any further computation. During the read operation the computation complexity of FAST explodes. If we try to examine all possible subsets MS of $\mathcal{S}$, then we obtain $2^{|\mathcal{S}|}$ possibilities. If we restrict this space to include only the subsets with size $|MS| = |\mathcal{S}| - \alpha f$ for all $\alpha \in [1, \ldots, \mathcal{R}]$ (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 $O(|\mathcal{S}|^2 \cdot 2^{|\mathcal{S}|})$. Such complexity may explode the computation time even when the size of $\mathcal{S}$ is small. As however the communication complexity is linear to the size of $\mathcal{S}$ then small set of servers will keep the communication overhead small. In contrast the computation complexity in ABD is bounded by $O(|\mathcal{S}|)$ as the reader parses the replies of at most $|\mathcal{S}|$ servers to detect the maximum timestamp.

Message Bit Complexity. Finally, for each write operation in both FAST and ABD, all servers may send messages containing a value and a timestamp, thus resulting in a bit complexity of $O(\lg |V|)$ per message. The main difference in the message bit complexity lies in the fact that in FAST servers attach the *seen* set along with the $\langle value, timestamp\rangle$ pair for each read operation. To obtain a tight bound we assume that each server sends a *bitmap* indicating whether each client identifier belongs or not in its *seen* set. As each *seen* set may contain up to $|\mathcal{R}| + 1$ identifiers, and since $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 1$, then the bitmap will contain less than or equal to $|\mathcal{S}|$ bits. Hence the length of each message in FAST is bounded by $\Theta(|\mathcal{S}| + \lg |V|)$.

4 Formulation and Hardness of the Predicate in FAST

We formulate the predicate used in FAST by 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* α *and* f *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* $|M| > n - \alpha f$*?*

It is easy to see the computational equivalence of the above problem and the predicate in FAST: U_1 can be substituted by the set of all the *seen* sets of the servers that replied, M by MS, and U_2 by $\mathcal{R} \cup \{w\}$. We prove that the Problem 1 is NP-hard via a reduction from the decision version of the Maximum Biclique problem defined below. The reduction is similar to the one in [12] for showing that the Maximum k-Intersection Problem is NP-hard.

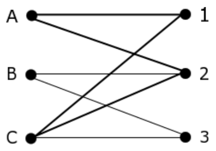

Figure 1 The left side of the graph (nodes A, B and C) corresponds to the elements of the set U_1 and the right side (nodes 1,2 and 3) corresponds to the elements of the set U_2. Thus, $A = \{1, 2\}$, $B = \{2, 3\}$ and $C = \{1, 2, 3\}$. The maximum biclique in this example has two nodes on each side.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

▶ **Definition 2** (Maximum 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 goal is to decide if the given graph G has a biclique of size at least c.

▶ **Theorem 3.** *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 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.

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 [2] to make it even "faster". Since we allow only single round trip operations, the new algorithm adheres to the bound presented in [2] and [3] regarding the possible number of read participants in the service. Thus, the algorithm is possible only if $|\mathcal{R}| < \frac{|\mathcal{S}|}{f} - 1$. Also, from the results in [2, 6], it follows that such algorithm 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, we call ccFast, that is communicationally the same and computationally faster than algorithm Fast.

Algorithm 1 Read, Write and Server protocols of algorithm ccFast

```
 1: at the writer $w$
 2: Components:
 3: $ts \in \mathbb{N}^+,\ v, vp \in V, wcounter \in \mathbb{N}^+$
 4: Initialization:
 5: $ts \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, wcounter \leftarrow 0$
 6: function WRITE($val$)
 7:     $vp \leftarrow v; v \leftarrow val;$
 8:     $ts \leftarrow ts + 1$
 9:     $wcounter \leftarrow wcounter + 1$
10:     send($\langle ts, v, vp\rangle, w, wcounter$) to all servers
11:     wait until $|\mathcal{S}| - f$ servers reply
12:     return(OK)
13: end function

14: at each reader $r_i$
15: Components:
16: $ts \in \mathbb{N}^+,\ maxTS \in \mathbb{N}^+,\ v, vp \in V, rcounter \in \mathbb{N}^+$
17: $srvAck \subseteq \mathcal{S} \times M,\ maxTSmsg \subseteq M$
18: Initialization:
19: $ts \leftarrow 0,\ maxTS \leftarrow 0, v \leftarrow \bot, vp \leftarrow \bot, rcounter \leftarrow 0$
20: $srvAck \leftarrow \emptyset,\ maxTSmsg \leftarrow \emptyset$
21: function READ
22:     $rcounter \leftarrow rcounter + 1$
23:     send($\langle ts, v, vp\rangle, r_i, rcounter$) to all servers
24:     wait until $|srvAck| = |\mathcal{S}| - f$ servers reply //Collect the ($serverid, \langle\langle ts', v', vp'\rangle, views\rangle$) pairs in $srvAck$
25:     $maxTS \leftarrow \max(\{m.ts' | (s, m) \in srvAck\})$
26:     $maxAck \leftarrow \{(s, m) | (s, m) \in srvAck\ \wedge\ m.ts' = maxTS\}$
27:     $\langle ts, v, vp\rangle \leftarrow m.\langle ts', v', vp'\rangle$ for $(*, m) \in maxAck$
28:     if $\exists\alpha \in [1, |\mathcal{R}| + 1]$ s.t. $MS = \{s : (s, m) \in maxAck\ \wedge\ m.views \geq \alpha\}$ and $|MS| \geq |\mathcal{S}| - \alpha f$ then
29:         return($v$)
30:     else
31:         retutn($vp$)
32:     end if
33: end function

34: at each server $s_i$
35: Components:
36: $ts \in \mathbb{N}^+, seen \subseteq \mathcal{R} \cup \{w\},\ v, vp \in V, Counter[1 \ldots |\mathcal{R}| + 1]$
37: Initialization:
38: $ts \leftarrow 0,\ seen \leftarrow \emptyset,\ v, vp \in V, Counter[i] \leftarrow 0$ for $i \in \mathcal{R} \cup \{w\}$
39: function RCV($\langle ts', v', vp'\rangle, q, counter$)          //Called upon reception of a message
40:     if $Counter[q] < counter$ then
41:         if $ts' > ts$ then
42:             $\langle ts, v, vp\rangle \leftarrow \langle ts', v', vp'\rangle$
43:             $seen \leftarrow \{q\}$
44:         else
45:             $seen \leftarrow seen \cup \{q\}$
46:         end if
47:         send($\langle ts, v, vp\rangle,\ |seen|$) to $q$
48:     end if
49: end function
```

The formal specification of the algorithm appears in Figure 1. Here we present a high level description of each protocol in the algorithm. The *counter* variables used throughout the algorithm are solely used to help processes identify “fresh” from “stale” messages 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 process w calls the write(val) function. During the write operation the writer stores the value to be written in a variable v and the previous written value in a variable vp (Line 7). Then it increments its local timestamp variable ts (Line 8), and sends a write request along with the triple $\langle ts, v, vp\rangle$ to all the servers and waits for $|\mathcal{S}| - f$ replies. Once those replies are received the operation terminates.

Server Protocol. We now describe the server protocol before proceeding to the read protocol, as it contains the recording mechanism which generates information that is used by each read to determine the value of the register. Each server 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 (Line 41). If $ts' > ts$, the server updates its local timestamp and values to be equal to the ones attached in the received message (Line 42), and resets its *seen* set to include only the identifier of the process that sent this message (Line 43); otherwise the server just inserts the identifier of the sender in the *seen* set (Line 45). 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 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 (Lines 25-27). Then it runs the following predicate on the set $maxAck$ (Line 28):

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

The predicate examines *how many* processes the maximum timestamp has been sent to. If more than $|\mathcal{S}| - \alpha f$ 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 goal of the predicate is to help 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 proceeds 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 an operation to see the written value it must be the case that there is at least one server that replied with that value, and thus $|\mathcal{S}| - \alpha f \geq f$. Solving this equation results in $\alpha \leq \frac{\mathcal{S}-1}{f}$. But α is the number of processes in the system. As the maximum number of processes is $|\mathcal{R}| + 1$, hence we derive the bound on the number of possible reader participants that $|\mathcal{R}| < \frac{\mathcal{S}-1}{f}$.

5.1 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 of atomicity (safety). 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 [2]. The lack of complete knowledge of *which* processes witnessed a value, introduced challenges in proving that consistency is preserved even when we know *how many* witnessed a value. 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. The atomicity properties can be expressed in terms of timestamps as follows:

A1. For each process p the ts variable is non-negative and monotonically nondecreasing.

A2. If a read ρ succeeds a write operation $\omega(ts)$ and returns a timestamp ts', then $ts' \geq ts$.

A3. If a read ρ returns ts', then either a write $\omega(ts')$ precedes ρ, i.e. $\omega(ts') \rightarrow \rho$, or $\omega(ts')$ is concurrent with ρ.

A4. If ρ_1 and ρ_2 are two read operations such that $\rho_1 \rightarrow \rho_2$ and ρ_1 returns ts_1, then ρ_2 returns $ts_2 \geq ts_1$.

Monotonicity allows the ordering of the values according to their associated timestamps. So Lemma 4 shows that the ts variable maintained by each process in the system is monotonically increasing. Let us first make the following observation:

▶ **Lemma 4.** *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 (Lines 37-38). So the server local timestamp is monotonically non-decreasing and the lemma follows. ◀

The following is also true for a server process.

▶ **Lemma 5.** *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 4 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 6.** *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 read from the same reader, say ρ_1, ρ_2 s.t. $\rho_1 \rightarrow \rho_2$, then 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 5 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 7.** *In any execution ξ of the algorithm, if a read ρ from r_1 succeeds a write operation ω that writes timestamp ts from the writer w , i.e. $\omega \rightarrow \rho$, and returns a timestamp ts', then $ts' \geq ts$.*

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 5 and since s receives the message from ω before replying to any of the two operations, then it replies to ρ with a timestamp $ts_s \geq ts$. Thus there are two cases to investigate on the timestamp: (1) $ts_s > ts$, and (2) $ts_s = ts$.

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

Case 2: In this case all the servers in $S_w \cap S_1$ reply with a timestamp $ts_s = ts$. 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 $ts' \geq ts$. So it remains to investigate the case where $maxTS = ts_s = ts$. 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_2 respectively. So s replied with a size at least $s.views \geq 2$ to ρ_2. 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 $ts' = ts$. 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 8.** *In any execution ξ of the algorithm, if ρ_1 and ρ_2 are two read operations such that $\rho_1 \rightarrow \rho_2$, and ρ_1 returns ts_1, then ρ_2 returns $ts_2 \geq ts_1$.*

Proof. Let the two operations ρ_1 and ρ_2 be executed from the same process, say r_1. As explained in Lemma 6, ρ_2 will discover a maximum timestamp $maxTS \geq ts_1$. If $maxTS > ts_1$, then ρ_2 returns either $ts_2 = maxTS$ or $ts_2 = maxTS - 1$, and thus in both cases

$ts_2 \geq ts_1$. It remains to examine the case where $maxTS = ts_1$. Since $\rho_1 \to \rho_2$, then any message sent during ρ_2 contains timestamp ts_1. By Lemma 5, every server s that receives the message from ρ_2 replies with a timestamp $ts_s \geq ts_1$. Since $maxTS = ts_1$, then it follows that all $|\mathcal{S}| - f$ servers that replied to ρ_2, sent the timestamp ts_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 $ts_2 = maxTS = ts_1$.

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_1. We have two cases to consider: (1) ρ_1 returns $ts_1 = maxTS_1 - 1$, or (2) ρ_1 returns $ts_1 = maxTS_1$.

Case 1: In this case ρ_1 returns $ts_1 = maxTS_1 - 1$. 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 7, ρ_2 returns a timestamp ts_2 greater or equal to the timestamp written by ω, and thus $ts_2 \geq maxTS_1 - 1 \Rightarrow ts_2 \geq ts_1$.

Case 2: This is the case where ρ_1 returns $ts_1 = maxTS_1$. 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 5 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 $ts_2 \geq maxTS_2 - 1 \Rightarrow ts_2 \geq maxTS_1$ and hence $ts_2 \geq ts_1$.

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-formdness, 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 5 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 $ts_2 = maxTS_2 = maxTS_1 = ts_1$.

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 $ts_2 = maxTS_2 = maxTS_1 = ts_1$ in this case as well. And this completes our proof. ◀

6 A Linear Algorithm for the Predicate and Complexity of ccFast

Table 1 presents the comparison of the complexities of ccFast with the complexities of both algorithms ABD and Fast.

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.

Message Bit Complexity. Each message sent in ccFast contains a triple with timestamp and two values. Omitting the timestamp as discussed earlier, then the values alone result in an upper bound of $O(\lg |V|)$ bits. Additionally, each server attaches the size of its *seen* set, which may include $|\mathcal{R}| + 1$ processes. The number of readers however, is bounded by $|\mathcal{S}|$, and hence the size of the seen set can be obtained with $\lg |\mathcal{S}|$ bits. Thus, the size of each message sent in ccFast is bounded by $O(\lg |V| + \lg |\mathcal{S}|)$ bits.

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 algorithm is a set $srvAck$ and a value $maxTS$ which indicate the servers that reply to a read operation and the maximum timestamp discovered among the replies. 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 (Lines 3-7). 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 (Lines 8-14). 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 9.** *Algorithm 2 implements the predicate used in every read operation in algorithm* ccFast.

Algorithm 2 Linear Algorithm for Predicate Computation.

```
function isValidPredicate($srvAck, maxTS$)
    $buckets \leftarrow Array[1 \dots |\mathcal{R}|+1]$, initially $[0, \dots, 0]$
    for all $s \in srvAck$ do
        if $s.ts == maxTS$ then
            $buckets[s.views]++$
        end if
    end for
    for $\alpha = |\mathcal{R}|+1$ to 2 do
        if $buckets[\alpha] \geq (|\mathcal{S}| - \alpha f)$ then
            return TRUE
        else
            $buckets[\alpha-1] \leftarrow buckets[\alpha-1] + buckets[\alpha]$
        end if
    end for
    if $buckets[1] == (|\mathcal{S}| - f)$ then
        return TRUE
    end if
    return FALSE
end function
```

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:

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

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, the set *srvAck* may contain at most $|\mathcal{S}|$ servers, and $|\mathcal{R}|$ is bounded by $|\mathcal{S}|$, then the complexity of the algorithm is:

▶ **Theorem 10.** *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 Conclusions

In this paper we questioned the overall complexity of algorithms that implement atomic SWMR R/W registers in the asynchronous, message-passing environment where processes are prone to crashes. Communication used to be the prominent operation efficiency metric for such implementations. We pick the best known (in terms of communication) algorithm that implements an atomic SWMR R/W register, Fast, that allows both reads and writes to terminate in just a *single* communication round. We show that the predicate utilized by the Fast to achieve such performance is hard to be computed, and hence the problem is not tractable. Next we present a new predicate that provides the following properties: (i) can be computed in polynomial time, (ii) allows operations to complete in a *single* communication round, and (iii) allows algorithm ccFast to preserve atomicity. A rigorous proof of the correctness of the algorithm is presented. Finally we conclude with a *linear time* algorithm to compute the newly proposed predicate. We believe that the new results redefine the term *fast* in atomic register implementations as operation performance accounts of all, *communication*, *computation*, and *message bit* complexity metrics. It is yet to be determined if the new operation efficiency is optimal or can be further improved.

References

1 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.

2 Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)*, pages 236–245, 2004.

3 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.

4 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.

5 Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC'08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 289–304, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-87779-0_20.

6 Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009. doi:10.1016/j.jpdc.2008.05.004.

7 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

8 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Transactions on Computers*, 28(9):690–691, 1979. doi:10.1109/TC.1979.1675439.

9 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

10 Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

11 Nancy A. Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, pages 219–246, 1989.

12 Eduardo C. Xavier. A note on a maximum k-subset intersection problem. *Information Processing Letters*, 112(12):471–472, 2012. doi:10.1016/j.ipl.2012.03.007.