Algorithm CwFr: Using Quorum Views for Fast Reads in the MWMR Setting [*]

Chryssis Georgiou [†] Nicolas C. Nicolaou[†] [‡]

Abstract

This report presents algorithm CwFr, that allows fast read operations in the MWMR setting. The algorithm uses an adjusted version of *Quorum Views*, client side decision tools introduced in [8], to establish whether a read operation can return in a single communication round. The new algorithm does not depend on the morphology of the underlying quorum system. Moreover, it allows fast reads when those are concurrently invoked with write operations. We provide a formal specification of algorithm CwFr and we present its rigorous proof of correctness.

Technical Report TR-10-05
Department of Computer Science
University of Cyprus
December 2010

[*]This work is supported by the Cyprus Research Promotion Foundation's grant ΠΕΝΕΚ/0609/31 and the European Regional Development Fund

[†]Department of Computer Science, University of Cyprus, Cyprus. Email: {chryssis,nicolasn}@cs.ucy.ac.cy.

[‡]Department of Computer Science and Engineering, University of Connecticut, CT, USA.

1 Introduction

Emulating atomic registers in asynchronous, crash-prone, message-passing systems is one of the basic problems in distributed computing. In such settings the register is replicated among a set of replica hosts or servers to provide fault-tolerance and availability. Then read and write operations are implemented as communication protocols that ensure atomic consistency.

Efficiency of register implementations is normally measured in terms of the latency of read and write operations. Two factors affect operation latency: (a) computation, and (b) communication delays. An operation may need to communicate with servers to read or write the register value. This involves at least a single communication round-trip, or *round*, i.e., messages from the invoking process to some servers and then the replies from these servers. Previous works focused on minimizing the number of rounds required by each operation. Dutta et al. [4] developed the first single-writer/multi-reader (SWMR) algorithm, where all operations complete in a single round. Such operations are called *fast*. The authors showed that fast operations are possible only if the number of readers in the system is constrained with respect to the number of servers. They also showed that it is impossible to have multi-writer/multi-reader (MWMR) implementations where *all* operations are fast. To remove the constraint on the number of readers, Georgiou et al. [9] introduced *semifast* implementations where at most one complete two-round read operation is allowed per write operation. They also showed that semifast MWMR implementations are impossible.

As of this writing, algorithm SFW, developed by Englert et al. [5], is the only algorithm that allows both reads and writes to be fast in the MWMR setting. The algorithm uses quorum systems, sets of intersecting subsets of servers, to handle server failures. To decide whether an operation can terminate after its first round, the algorithm employs two *predicates*, one for the write and one for read operations. Validation of both predicates relied on the construction of the underlying quorum system. Predicates hold (and thus operations could be fast) only when every five or more quorums of the quorum system had non empty intersection. Such quorum systems have large intersections. As a result a server needs to belong in more quorums. Thus, the probability that a server is going to be accessed per operation is higher and the load of each server is higher. On the other hand failure of any server in the system may result in the failure of many quorums, decreasing this way the fault tolerance of the quorum system.

Here we explore whether general quorum systems can be used and still allow some operations to be fast in the MWMR setting. We present algorithm CWFR that uses *Quorum Views* [8], client-side decision tools, to allow some fast *read* operations without additional constraints on the quorum system. Write operations in this implementations take two rounds. The greatest challenge of the new algorithm is to handle the multiple concurrent write operations and try to predict the state of each one of those. To handle this problem the algorithm establishes an iterative approach to determine the largest potentially completed write operation.

Backround. Attiya et al. [1] gave a SWMR algorithm that achieves consistency by using intersecting majorities of servers in combination with $\langle timestamp, value \rangle$ pairs. A write operation increments the writer's local timestamp and delivers the new tag-value pair to a majority of servers, taking one round. A read operation obtains tag-value pairs from some majority, then propagates the pair corresponding to the highest timestamp to some majority of servers, thus taking two rounds.

The majority-based approach in [1] is readily generalized to quorum-based approaches in the MWMR setting (e.g., [14, 6, 12, 7, 10]). Such algorithms requires at least two communication rounds for each read and write operation. Both write and read operations query the servers for the latest value of the replica during the first round. In the second round the write operation generates a new tag and propagates the tag along with the new value to a quorum of servers. A read operation propagates to a quorum of servers the largest value it discovers during its first round. Dolev *et al.* [3] and Chockler *et al.* [2], provide MWMR implementations where some reads involve a single communication round when it is confirmed that the value read was already propagated to some quorum.

Dutta et al. [4] present the first *fast* atomic SWMR implementation where all operations take a *single* communication round. They show that fast behavior is achievable only when the number of reader processes R is inferior to $\frac{S}{t} - 2$, where S the number of servers, t of whom may crash. They also showed that fast MWMR implementations are impossible even in the presence of a single server failure. Georgiou et al. [9] introduced the notion of *virtual nodes* that enables an unbounded number of readers. They define the notion of *semifast* implementations where only a single read operation per write needs to be "slow" (take two rounds). They also show the imposibility of semifast MWMR implementations.

Georgiou et al. [8] showed that fast and semifast quorum-based SWMR implementations are possible if and only if a common intersection exists among all quorums. Hence a single point of failure exists in such solutions (i.e., any server in the common intersection), making such implementations not fault-tolerant. To trade efficiency for improved fault-tolerance, *weak-semifast* implementations in [8] require at least one single slow read per write operation, and where all writes are fast. To obtain a weak-semifast implementation they introduced a client-side decision tool called *Quorum Views* that enables fast read operations under read/write concurrency when *general quorum systems* are used.

Recently, Englert *et al.* [5] developed an atomic MWMR register implementation, called algorithm SfW, that allows both reads and writes to complete in a *single round.* To handle server failures, their algortihm uses *n-wise quorum systems*: a set of subsets of servers, such that each n of these subsets intersect. The parameter n is called the *intersection degree* of the quorum system. The algorithm relies on $\langle tag, value \rangle$ pairs to totally order write operations. In contrast with traditional approaches, the algorithm uses the *server side ordering* (SSO) approach that transfers the responsibility of incrementing the tag from the writers to the servers. This way, the *query* round of write operations is eliminated. The authors proved that fast MWMR implementations are possible if and only if they allow not more than $n-1$ successive write operations, where n is the intersection degree of the quorum system. If read operations are also allowed to modify the value of the register then from the provided bound it follows that a fast implementation can accomodate up to $n-1$ readers and writers.

Contributions. Our goal is to provide efficient and practical implementations of atomic MWMR registers. We examined the only known algorithm that allows fast read and write operations, algorithm SfW, and we identified that fast write operations are enabled only if the quorum system satisfies specific quorum intersection properties. If the same properties do not hold, the number of two round read operations may also increase. Moreover, such specifications restrict the flexibility and usability of the proposed algorithm

Motivated by the above observations, we examine whether fast operations can be achieved if one uses *general quorum constructions.* By generalizing the client side decision tools, called Quorum Views, developed for the SWMR setting in [8], we derive algorithm CwFr. The new algorithm uses the conventional two round writes. To allow fast read operations the algorithm analyzes, using quorum views, the distribution of a value within a quorum of replies from servers. As multiple writes can occur concurrently, an iterative technique is used to discover the latest potentially completed write operation.

Paper Organization. In Section 2 we give the model of computation considered in this work. In Section 3 we present algorithm CwFr and in Section 4 we prove its correctness. We conclude in Section 5.

2 Model of Computation

We consider the asynchronous message-passing model. There are three distinct finite sets of crash-prone processors: a set of readers $\mathcal{R}$, a set of writers $\mathcal{W}$, and a set of servers $\mathcal{S}$. The identifiers of all processors are unique and comparable. Communication among the processors is accomplished via reliable communication channels.

Servers and Quorums. Servers are arranged into intersecting sets, or *quorums*, that together form a quorum system $\mathbb{Q}$. For a set of quorums $\mathcal{A} \subseteq \mathbb{Q}$ we denote the intersection of the quorums in $\mathcal{A}$ by $(\bigcap_{Q\in\mathcal{A}} \mathcal{Q}) = \bigcap_{Q\in\mathcal{A}} Q$. A quorum system $\mathbb{Q}$ is called an *n-wise quorum system* if for any $\mathcal{A} \subseteq \mathbb{Q}$, s.t. $|\mathcal{A}| = n$ we have $(\bigcap_{Q\in\mathcal{A}} \mathcal{Q}) \neq \emptyset$. We call n the *intersection degree* of $\mathbb{Q}$. Any quorum system is a *2-wise* (pairwise) quorum system because any two quroums intersect. At the other extreme, a $|\mathbb{Q}|$-*wise* quorum system has a common intersection among all quorums. From the definition it follows that an *n-wise* quorum system is also a *k-wise* quorum system, for $2 \leq k \leq n$.

IO Automata and Executions. An algorithm A is a composition of automata A_i [13, 11], each assigned to some process i. Each A_i is defined in terms of a set of states $states(A_i)$ that includes the initial state σ_0, a signature $sig(A_i)$ that specifies input, output, and internal actions and *transitions*, that for each action ν gives the triple $\langle\sigma, \nu, \sigma'\rangle$ defining the transition of A_i from state σ to state σ'. Such a triple is also called a *step*. An *execution fragment* ϕ of A_i is a finite or an infinite sequence $\sigma_0, \nu_1, \sigma_1, \nu_2, \ldots, \nu_r, \sigma_r, \ldots$ of alternating states and actions, such that every $\sigma_k, \nu_{k+1}, \sigma_{k+1}$ is a step of A_i. If an execution fragment begins with an initial state of A_i then it is called an *execution*.

Our system allows processes to fail by crashing. A process i *crashes* in an execution ϕ if it contains a step $\langle\sigma_k, fail_i, \sigma_{k+1}\rangle$ as the last step of A_i. A process i is *faulty* in an execution if i crashes in the execution; otherwise i is *correct*. A quorum $Q \in \mathbb{Q}$ is non-faulty if $\forall i \in Q$, i is correct; otherwise Q is faulty. We assume that at least one quorum in $\mathbb{Q}$ is non-faulty in any execution.

Atomicity. We study atomic read/write register implementations, where the register is replicated at servers. Reader p requests a read operation ρ on the register using action read_p. Similarly, a write operation is requested using action $\mathsf{write}(*)_p$ at writer p. The steps corresponding to such actions are called *invocation* steps. An operation terminates with the corresponding acknowledgment action; these steps are called *response* steps. An operation π is *incomplete* in an execution when the invocation step of π does not have the associated response step; otherwise we say that π is *complete*. We assume that requests made by read and write processes are *well-formed*: a process does not request a new operation until it receives the response for a previously invoked operation.

In an execution, we say that an operation (read or write) π_1 *precedes* another operation π_2, or π_2 *succeeds* π_1, if the response step for π_1 precedes in real time the invocation step of π_2; this is denoted by $\pi_1 \rightarrow \pi_2$. 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. Assuming the failure model discussed earlier, the termination property requires that any operation invoked by a correct process eventually completes. Atomicity is defined as follows [11]. For any execution if all read and write operations that are invoked complete, then the 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 register.

Efficiency and Fastness. We measure the efficiency of an atomic register implementation in terms of *computation* and *communication round-trips* (or simply rounds). A round is defined as follows [4, 9, 8]:

Definition 2.1 *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.
3. when process p receives enough replies it terminates the round (either completing π or starting new round).

Operation π is *fast* [4] if it completes after its first communication round; an implementation is fast if in each execution all operations are fast. We use quorum systems and tags to maintain, and impose an ordering on, the values written to the register replicas. We say that a quorum $Q \in \mathbb{Q}$, *replies* to a process p for an operation π during a round, if $\forall s \in Q$, s receives a message during the round and replies to this message, and p receives all such replies.

Given that any subset of readers or writers may crash, the termination of an operation cannot depend on the progress of any other operation. Furthermore we guarantee termination only if servers' replies within a round of some operation do not depend on receipt of any message sent by other processes. Thus we can construct executions where only the messages from the invoking processes to the servers, and from the servers to the invoking processes are delivered. Lastly, to guarantee termination under the assumed failure model, no operation can wait for more than a singe quorum to reply within the processing of a single round.

3 Algorithm CwFr

We explored the possibility to introduce fast operations in the MWMR environment by exploiting techniques presented in the SWMR environment. The developments of [3, 2], made an effort to introduce fast read operations in the MWMR environment, but their techniques did not convince that such fast behavior is possible under read and write concurrency. On the other hand the development in [5] showed that it is possible to obtain both fast reads and writes in the MWMR setting but their approach relied on restrictive specifications on the quorum system they deployed.

In this Section we introduce a new algorithm, we call CwFr, which enables fast read operations by adopting the general idea of Quorum Views [8]. The algorithm employs two techniques:

(i) the classic query and propagate technique (two round) for write operations, and

(ii) analysis of Quorum Views for potentially fast (single round) read operations.

The new algorithm can use any general quorum construction and allows read operations to be fast even when they are invoked concurrently with one or multiple write operations. This distinguishes CwFr from previous approaches. To impose a total ordering on the written values, CwFr exploits $\langle tag, value\rangle$ pairs as also used in prior papers (e.g., [2, 3, 12]). A *tag* is a tuple of the form $\langle ts, w\rangle \in \mathbb{N} \times \mathcal{W}$, where ts is the timestamp and w is a writer identifier. Two tags are ordered lexicographically, first by the timestamp, and then by the writer identifier.

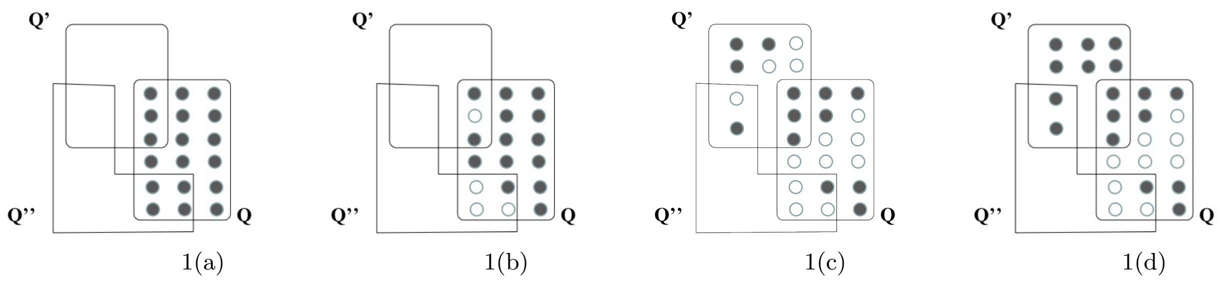

Figure 1: (a) **QV1**, (b) **QV2**, (c) **QV3** with incomplete write, (d) **QV3** with complete write.

3.1 Incorporating Prior Techniques – Quorum Views

To comply with the ordering scheme of CwFr we revised the definition of quorum views as presented in [8], to examine tags instead of timestamps. The revised definition is the following:

Definition 3.1 *Let process p, receive replies from every server s in some quorum $Q \in \mathbb{Q}$ for a read or write operation π. Let a reply from s include a tag $m(\pi)_{s,p}.tag$ and let $maxTag = \max_{s \in Q}(\sigma_{res(\pi)}[s].tag)$. We say that p observes one of the following* ***quorum views*** *for Q:*

- **QV1**: $\forall s \in Q : m(\pi)_{s,p}.tag = maxTag$,
- **QV2**: $\forall Q' \in \mathbb{Q} : \; Q \neq Q' \wedge \exists A \subseteq Q \cap Q'$, *s.t.* $A \neq \emptyset$ *and* $\forall s \in A : m(\pi)_{s,p}.tag < maxTag$,
- **QV3**: $\exists s' \in Q : m(\pi)_{s',p}.tag < maxTag$ *and* $\exists Q' \in \mathbb{Q}$ *s.t.* $Q \neq Q' \wedge \forall s \in Q \cap Q' : m(\pi)_{s,p}.tag = maxTag$

QV1 implies the potential completion of the write operation that wrote a value associated with $maxTag$. **QV2** imposes its non-completion and **QV3** does not reveal any information about the write completion.

Analyzing these three types of quorum views we can derive conclusions on the state of the write operation (complete or incomplete) that tries to propagate a value with $maxTag$ in the system. Figure 1 illustrates those quorum views assuming that a reader r invokes a read operation ρ and receives replies from all the servers in Q_i. The dark nodes maintain the maximum tag of the system and white nodes or "empty" quorums maintain an older tag. Recall that it follows from our failure model that no operation (read or write) can wait for more than one quorum to reply. Thus having a full quorum reporting the same tag, as seen in Figure 1(a), implies the possible completion of the write operation (in the case of Figure 1(a) the complete write operation strictly contacts Q_i).

Observe that if a full quorum contains $maxTag$ then the members of any intersection of that quorum contain $maxTag$. So witnessing a subset of members of each intersection of Q_i (as seen in Fig. 1(b) the representation of **QV2**) to maintain an older timestamp, implies directly that the write operation which propagates $maxTag$ is not yet complete.

Finally, **QV3**, provides insufficient information regarding the state of the write operation. Observe Figures 1(c) and 1(d). In the former an incomplete write operation propagates the $maxTag$ in the dark nodes and in the latter it completes by receiving replies from Q_z. Notice that if a read operation ρ strictly contacts Q_i (i.e., $scnt(\rho, Q_i)_*$) in the two executions, it won't be able to distinguish 1(c) from 1(d). So, more formally, if an operation witnesses some intersection $Q_i \cap Q_z$ that contains $maxTag$ in all of its members, then a write operation might: (i) have been completed and contacted Q_z or (ii) be incomplete and contacted a subset of servers B such that $Q_i \cap Q_z \subseteq B$ and $\forall Q_j \in \mathbb{Q}, Q_j \not\subseteq B$.

3.2 High Level Description of CwFr

The original quorum views algorithm [8] relies on the fact that a single writer is participating in the system. If a quorum view is able to predict the non-completeness of the latest write operation, is immediately understood that – by well-formedness of the single writer – any previous write operation is already completed. Multiple writer participants in the system prohibit such assumption: different values (or tags) may be written concurrently. Hence, the discovery of a write operation that propagates some tag does not imply the completion of the write operations that propagate a smaller tag. To this end, direct adaptation of the quorum view idea form the SWMR model to the MWMR model was impossible. So, algorithm CwFr incorporates an iterative technique around quorum views that not only predicts the completion status of a write operation, but also detects the last potentially completed write operation. Below we provide a high level description of our algorithm and present the main idea behind our technique.

Writers. The write protocol has two rounds. During the first round the writer discovers the maximum tag among the servers: it sends read messages to all servers and waits for replies from all the members of a single quorum. It then discovers the maximum tag among the replies and generates a new tag in which it encloses the incremented timestamp of the maximum tag, and the writer's identifier. In the second round, the writer associates the value to be written with the new tag, it propagates the pair to a complete quorum, and completes the write.

Readers. The read protocol is more involved. When a reader invokes a read operation, it sends a read message to all servers and waits for some quorum to reply. Once a quorum replies, the reader determines the *maxTag*. Then the reader analyzes the distribution of the tag within the responding quorum Q in an attempt to determine the latest, potentially complete, write operation. This is accomplished by determining the quorum view conditions. Detecting conditions of **QV1** and **QV3** are straightforward. When condition for **QV1** is detected, the read completes and the value associated with the discovered *maxTag* is returned. In the case of **QV3** the reader continues into the second round, advertising the latest tag (*maxTag*) and its associated value. When a full quorum replies to the second round, the read returns the value associated with *maxTag*.

Analysis of **QV2** involves discovery of the earliest completed write operation. This is done iteratively by (locally) removing the servers from Q that replied with the largest tags. After each iteration the reader determines the next largest tag in the remaining server set, and then re-examines the quorum views in the next iteration. This process eventually leads to either **QV1** or **QV3** being observed. If **QV1** is observed, then the read completes in a single round by returning the value associated with the maximum tag among the servers that *remain* in Q. If **QV3** is observed, then the reader proceeds to the second round as above, and upon completion it returns the value associated with the maximum tag *maxTag* discovered among the original respondents in Q.

Servers. The servers play a passive role. They receive read or write requests, update their object replica accordingly, and reply to the process that invoked the request. Upon receipt of any message, the server compares its local tag with the tag included in the message. If the tag of the message is higher than its local tag, the server adopts the higher tag along with its corresponding value. Once this is done the server replies to the invoking process.

3.3 Formal Specification of CwFr

We now present the formal specification of CwFr using IOA [13] notation. Our implementation includes four automata: (i) automaton CwFr$_w$ that handles the write operations for each writer $w \in \mathcal{W}$, (ii) automaton CwFr$_r$ that handles the reading for each $r \in \mathcal{R}$, (iii) automaton CwFr$_s$ that

Signature:

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

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

Internal:
$\mathsf{write\text{-}phase1\text{-}fix}_w, w \in \mathcal{W}$
$\mathsf{write\text{-}phase2\text{-}fix}_w, w \in \mathcal{W}$

State:

$tag = \langle ts, w\rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, w\}$
$v \in V$, initially $\perp$
$vp \in V$, initially $\perp$
$maxTS \in \mathbb{N}$, initially 0
$wCounter \in \mathbb{N}^+$, initially 0
$phase \in \{1, 2\}$, initially 1
$status \in \{idle, active, done\}$, initially $idle$
$srvAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$failed$, a Boolean initially **false**

Transitions:

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

Input $\mathsf{rcv}(\langle msgT, t, C\rangle)_{s,w}$
 Effect:
   if $\neg failed$ then
     if $status = active$ and $wCounter = C$ then
       if $(phase = 1 \wedge msgT = \text{ READ-ACK })\vee$
          $(phase = 2 \wedge msgT = \text{ WRITE-ACK })$ then
         $srvAck \leftarrow srvAck \cup \{s, \langle msgT, t, C\rangle\}$

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

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

Internal $\mathsf{write\text{-}phase1\text{-}fix}_w$
 Precondition:
   $\neg failed$
   $status = active$
   $phase = 1$
   $\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
 Effect:
   $maxTS \leftarrow \max_{s \in Q \wedge (s,m) \in srvAck}(m.t.tag.ts)$
   $tag = \langle maxTs + 1, w\rangle$
   $phase \leftarrow 2$
   $srvAck \leftarrow \emptyset$
   $wCounter \leftarrow wCounter + 1$

Internal $\mathsf{write\text{-}phase2\text{-}fix}_w$
 Precondition:
   $\neg failed$
   $status = active$
   $phase = 2$
   $\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s, m) \in srvAck\}$
 Effect:
   $status \leftarrow done$

Input $\mathsf{fail}_w$
 Effect:
   $failed \leftarrow true$
```

Figure 2: $\textsc{CwFr}_w$ Automaton: Signature, State and Transitions

handles the read and write requests on the atomic register for each $s \in \mathcal{S}$, and (iv) $Channel_{p,p'}$ that establish the reliable asynchronous process-to-process communication channels; these are the typical Channel automata, as defined in [11].

Automaton $\textsc{CwFr}_w$**.** The state variables, the signature and the transitions of the $\textsc{CwFr}_w$ are depicted in Figure 2. The state of the $\textsc{CwFr}_w$ automaton includes the following variables:

- $\langle\langle ts, wid\rangle, v\rangle \in \mathbb{N} \times \mathcal{W} \times V$: writer's local tag along with the latest value written by the writer. The tag is composed of a timestamp and the identifier of the writer.
- $vp \in V$: this variable is used to hold the previous value written.
- $maxTS \in \mathbb{N}$: the maximum timestamp discovered during the last write operation.
- $wCounter \in \mathbb{N}$: the number of write requests performed by the writer. Is used by the servers to distinguish fresh from stale messages.
- $phase \in \{1, 2\}$: indicates the active communication round of the write operation.
- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the writer received all the necessary replies to complete its write operation and is ready to respond to the client. This variable also ensures well formedness for the writer, since no request is accepted by the automaton as long as $status \neq idle$.
- $srvAck \subseteq \mathcal{S}$: a set that contains the servers that reply to the write messages as a result of a write request. The set is reinitialized to $\emptyset$ at the response step of every write operation.
- $failed \in \{true, false\}$: indicates whether the process associated with the automaton has failed.

The automaton completes a write operation in two phases. A write operation ω is invoked when the $\mathsf{write}(val)_w$ request is received from the automaton's environment. The *status* variable becomes *active*, the previous value *vp* gets the current value and the variable v gets the requested value *val* to be written. As long as the $status = active$ and $phase = 1$ the automaton sends messages to all server processes and collects the identifiers of the servers that reply to those messages in the *srvAck* set. To avoid adding any delayed message from a previous phase, the writer examines the type of the acknowledgment. The action write-phase1-fix occurs when the replies from the members of a full quorum are received by the writer, i.e., $\exists Q \in \mathbb{Q} : Q \subseteq srvAck$. In the same action the writer discovers the maximum timestamp *maxTS* among the replies and generates the new tag. In particular, it assigns $tag = \langle maxTS + 1, w\rangle$. Once the new tag is generated, the writer changes the *phase* variable to 2, to indicate the start of its second round, and reinitializes the *srvAck* to accept the replies to its new round. When a full quorum replies to w, the *status* of the automaton becomes *done*. This change, and assuming that the writer does not fail, enables the $\mathsf{write\text{-}ack}_w$. Finally, when the action $\mathsf{write\text{-}ack}_w$ occurs, the writer responds to the environment and the *status* variable becomes *idle*.

Automaton $\textsc{CwFr}_\rho$**.** The state variables, the signature and the transitions of the $\textsc{CwFr}_r$ are depicted in Figures 3 and 4. The state of the $\textsc{CwFr}_r$ automaton includes the following variables:

- $\langle\langle ts, wid\rangle, v\rangle \in \mathbb{N} \times \mathcal{W} \times V$: the maximum tag (timestamp and writer identifier pair) discovered during r's last read operation along with its associated value.
- $maxTag \in \mathbb{N} \times \mathcal{W}$, and $retvalue \in V$: the maximum tag discovered and the value that was returned during the last read operation.
- $rCounter \in \mathbb{N}$: read request counter. Used by the servers to distinguish fresh from stale messages.
- $phase \in \{1, 2\}$: indicates the active communication round of the read operation.

Signature:

Input:	Output:	Internal:
$\mathsf{read}_r,\ r \in \mathcal{R}$	$\mathsf{send}(m)_{r,s},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$	$\mathsf{read\text{-}phase1\text{-}fix}_r$
$\mathsf{rcv}(m)_{s,r},\ m \in M,\ r \in \mathcal{R},\ s \in \mathcal{S}$	$\mathsf{read\text{-}ack}(val)_r,\ val \in V,\ r \in \mathcal{R}$	$\mathsf{read\text{-}phase2\text{-}fix}_r$
$\mathsf{fail}_r,\ r \in \mathcal{R}$		

State:

$tag = \langle ts, wid\rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, \min(\mathcal{W})\}$	$status \in \{idle, active, done\}$, initially $idle$
$maxTag = \langle ts, wid\rangle \in \mathbb{N} \times \mathcal{W}$, initially $\{0, \min(\mathcal{W})\}$	$srvAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$v \in V$, initially $\perp$	$maxAck \subseteq M \times \mathcal{S}$, initially $\emptyset$
$retvalue \in V$, initially $\perp$	$maxTagSrv \subseteq \mathcal{S}$, initially $\emptyset$
$phase \in \{1, 2\}$, initially 1	$replyQ \subseteq \mathcal{S}$, initially $\emptyset$
$rCounter \in \mathbb{N}^+$, initially 0	$failed$, a Boolean initially **false**

Figure 3: CwFr$_r$ Automaton: Signature and State

- $status \in \{idle, active, done\}$: specifies whether the automaton is in the middle of an operation ($status = active$) or it is done with any requests ($status = idle$). When $status = done$, it indicates that the reader decided on the value to be returned and is ready to respond to the client. This variable also ensures well formedness for the reader, since no request is accepted by the automaton as long as $status \neq idle$.

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

- $maxAck \subseteq M \times \mathcal{S}$: this set contains the messages (and the servers senders) that contained the maximum tag during r's last read request.

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

- $replyQ \subseteq \mathcal{S}$: The quorum of servers that replied to r during the last read operation.

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

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

A read operation is invoked when the CwFr$_r$ automaton receives a read_r request from its environment. The $status$ of the automaton becomes $active$ preventing the invocation of any other operation until the current operation completes. As long as the $status = active$, the automaton sends messages to each server $s \in \mathcal{S}$ to obtain the value of the atomic object. The $\mathsf{rcv}(m)_{s,r}$ action is triggered when a reply from a server s is received. The reader collects the identifiers of servers that replied to the current operation and their messages, by adding a pair (s, m) in the $srvAck$ set. When the set $srvAck$ contains the members of at least a single quorum Q of the quorum system $\mathbb{Q}$, the set of messages is filtered (during action $\mathsf{read\text{-}phase1\text{-}fix}_r$) to find the messages that contain the maximum tag ($maxTag$). Those messages are placed in $maxTagAck$ set. The servers that belong into the collected quorum and have messages in $maxTagAck$, are placed separately in the $maxTagSrv$ set. Lastly, the $replyQ$ variable becomes equal to the quorum Q and the value v becomes equal to the value assigned to $maxTag$.

Iterative algorithm: From the newly formed sets the reader iteratively analyzes the distribution of the maximum tag on the members of $replyQ$, in an attempt to determine the latest write operation that has potentially completed. This is done by the $\mathsf{read\text{-}qview\text{-}eval}_r$ action. In particular, the iterative

Transitions:

```
Input $\mathsf{read}_r$
 Effect:
   if $\neg failed$ then
     if $status = idle$ then
       $status \leftarrow active$
       $rCounter \leftarrow rCounter + 1$

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

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

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

Internal $\mathsf{read\text{-}phase2\text{-}fix}_r$
 Precondition:
   $\neg failed$
   $status = active$
   $phase = 2$
   $\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s,m) \in srvAck\}$
 Effect:
   $status \leftarrow done$
   $phase \leftarrow 1$

Internal $\mathsf{read\text{-}phase1\text{-}fix}_r$
 Precondition:
   $\neg failed$
   $status = active$
   $phase = 1$
   $\exists Q \in \mathbb{Q} : Q \subseteq \{s : (s,m) \in srvAck\}$
 Effect:
   $replyQ \leftarrow Q$
   $maxTag \leftarrow \max_{s\in replyQ \wedge (s,m)\in srvAck}(m.t.tag)$
   $maxAck \leftarrow \{(s,m) : (s,m) \in srvAck \wedge m.t.tag = maxTag\}$
   $maxTagSrv \leftarrow \{s : s \in replyQ \wedge (s,m) \in maxAck\}$
   $v \leftarrow \{m.t.val : (s,m) \in maxAck\}$

Internal $\mathsf{read\text{-}qview\text{-}eval}_r$
 Precondition:
   $\neg failed$
   $replyQ \neq \emptyset$
 Effect:
   $tag \leftarrow \max_{s\in replyQ \wedge (s,m)\in srvAck}(m.t.tag)$
   $maxAck \leftarrow \{(s,m) : (s,m) \in srvAck \wedge m.t.tag = maxTag\}$
   $maxTagSrv \leftarrow \{s : s \in replyQ \wedge (s,m) \in maxAck\}$
   $retvalue \leftarrow \{m.t.val : (s,m) \in maxAck\}$
   if $replyQ = maxTagSrv$ then
     $status \leftarrow done$
   else
     if $\exists Q' \in \mathbb{Q}, Q' \neq replyQ$ s.t. $replyQ \cap Q' \subseteq maxTagSrv$ then
       $tag \leftarrow maxTag$
       $retvalue \leftarrow v$
       $phase \leftarrow 2$
       $srvAck \leftarrow \emptyset$
       $rCounter \leftarrow rCounter + 1$
     else
       $replyQ \leftarrow replyQ - \{s : s \in maxTagSrv\}$

Input $\mathsf{fail}_r$
 Effect:
   $failed \leftarrow true$
```

Figure 4: CwFr$_r$ Automaton: Transitions

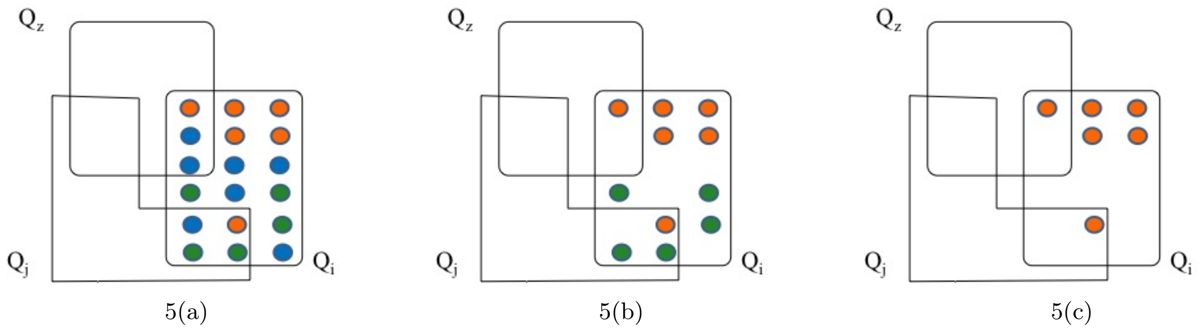

Figure 5: Illustrating the progress of the iterative algorithm.

approach works as follows. Let $maxTag_\ell$ denote the maximum tag in $replyQ$ at iteration ℓ, with $maxTag_0 = maxTag$. Also let $replyQ_\ell$ be the set of servers that the read operation examines during iteration ℓ, with $replyQ_0 = replyQ = Q$. During every iteration ℓ, the reader r proceeds as follows (locally) depending on the quorum view it observes during ρ in $replyQ_\ell$:

Set $\ell = 0$ and $replyQ_0 = replyQ$

Repeat until return:

QV1: Return the value associated with $maxTag_\ell = \max_{s \in replyQ_\ell}(m(\rho)_{s,r}.t.tag)$

QV3: Proceed to a second round, and propagate messages that contain $maxTag_0 = maxTag$ to all servers. Once the read-phase2-fix$_r$ event occurs, return the value associated with $maxTag$.

QV2: Set $replyQ_{\ell+1} = replyQ_\ell - \{s : (s \in replyQ_\ell) \wedge (m(\rho)_{s,r}.t.tag = maxTag_\ell)\}$ and proceed to iteration $\ell + 1$.

Let us discuss the idea behind our proposed technique. Observe that under our failure model, any write operation can expect a response from at least one full quorum. Moreover a write ω distributes its tag tag_ω to some quorum, say Q', before completing.Thus when a read operation ρ, s.t. $\omega \to \rho$, receive replies from some quorum Q, then it will observe one of the following tag distributions: (a) if $Q = Q'$, then $\forall s \in Q, m(\rho)_{s,r} = tag_\omega$ (**QV1**), or (b) if $Q \neq Q'$, then $\forall s \in Q \cap Q', m(\rho)_{s,r} = tag_\omega$ (**QV3**). Hence, if ρ observes a distribution as in **QV1** then it follows that a write operation completed and received replies from the same quorum that replied to ρ. Alternatively, if only an intersection contains a uniform tag (i.e., the case of **QV3**) then there is a possibility that some write completed in an intersecting quorum (in this example Q'). The read operation is fast in **QV1** since it is determinable that the write potentially completed. The read proceeds to the second round in **QV3**, since the completion of the write is indeterminable and it is necessary to ensure that any subsequent operation will observe that tag. If none of the previous quorum views hold (and thus **QV2** holds), then it must be the case that the write that yielded the maximum tag is not yet completed. Hence we try to discover the latest potentially completed write by removing all the servers with the highest tag from Q and repeating the analysis. If at some iteration, **QV1** holds on the remaining tag values, then a potentially completed write – that was overwritten by greater values in the rest of the servers – is discovered and that tag is returned (in a single round). If no iteration is interrupted because of **QV1**, then eventually **QV3** will be observed, in the worst case, when a single server will remain in some intersection of Q. Since a second round cannot be avoided in this case, we take the opportunity to propagate the largest tag observed in Q. At the end of the second round that tag will be written to at least a single complete quorum and thus the reader can safely return it.

Signature:

```
Input:                                           Output:
  rcv$(m)_{p,s}$,  $m \in M$,  $s \in \mathcal{S}$,  $p \in \mathcal{R} \cup \mathcal{W}$        send$(m)_{s,p}$,  $m \in M$,  $s \in \mathcal{S}$,  $p \in \mathcal{R} \cup \mathcal{W}$
  fail$_s$
```

State:

```
$tag = \langle ts, wid \rangle \in \mathbb{N} \times \mathcal{W}$,  initially $\{0, \min(\mathcal{W})\}$
$v \in V$,  initially $\bot$
$Counter(p) \in \mathbb{N}^+$,  $p \in \mathcal{R} \cup \mathcal{W}$,  initially 0
$msgType \in \{$WRITEACK,READACK,INFOACK$\}$
$status \in \{idle, active\}$,  initially $idle$
$failed$,  a Boolean initially **false**
```

Transitions:

```
Input rcv$(\langle msgT, t, C \rangle)_{p,s}$
  Effect:
    if $\neg failed$ then
      if $status = idle$ and $C > Counter(p)$ then
        $status \leftarrow active$
        $Counter(p) \leftarrow C$
        if $tag < t.tag$ then
          $(tag.ts, tag.wid, v) \leftarrow (t.tag.ts, t.tag.wid, t.val)$

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

Input fail$_s$
  Effect:
    $failed \leftarrow true$
```

Figure 6: CwFr$_s$ Automaton: Signature, State and Transitions

In Figure 5 we present an example that illustrates the stages of our iterative algorithm. Assume that a reader r invokes a read operation ρ that receive replies from all the servers in Q_i. Let the reader observe three different values in that quorum as shown in Figure 5(a). Assume that the blue value is associated with a higher tag than the green value which in turn is associated with a higher tag than the orange value. We refer the writes that propagate these values as the blue, green and orange write. In the first iteration of our algorithm we investigate the distribution of the higher tag, in this case the blue value. We observe that in every intersection there is a server that maintains a smaller tag. Since a server updates its local tag whenever it receives a higher tag, it follows that those servers did not receive message from the writer that invoked the write of the blue value. Thus, the blue write should not be completed as **QV2** is satisfied. By our algorithm we remove the servers that replied with a blue value and we re-evaluate the quorum views on the remaining values. A **QV2** holds for the green value (see Figure 5(b)) as every intersection contains a smaller value. When we remove the servers that replied with a green value we observe that the remaining servers replied with the same, orange value (see Figure 5(c)). So re-evaluating the quorum views we obtain a **QV1**. Thus, the read operation will return the orange value in a single round. So this example implies that the orange write may have completed by receiving replies from Q_i, and the orange value was replaced by a higher value in the servers of Q_i that replied with a blue or orange value.

Automaton CwFr$_s$**.** The server automaton has relatively simple actions. The signature, state and transitions of the CwFr$_s$ are depicted in Figure 6. The state of the CwFr$_s$ contains the following variables:

- $\langle\langle ts, wid \rangle, v \rangle \in \mathbb{N} \times \mathcal{W} \times V$: the maximum tag (timestamp, writer identifier pair) reported to s along with its associated value. This is the value of the register replica of s.
- $Counter(p) \in \mathbb{N}$: this array maintains the latest request index of each client (reader or writer).

It helps s to distinguish fresh from stale messages.

- $status \in \{idle, active\}$: specifies whether the automaton is processing a request received ($status = active$) or it can accept new requests ($status = idle$).
- $msgType \in \{\text{WRITEACK,READACK,INFOACK}\}$: Type of the acknowledgment depending on the type of the received message.
- $failed \in \{true, false\}$: indicates whether the server associated with the automaton has failed.

Each server replies to a message without waiting to receive any other messages from any process. Thus, the status of the server automaton determines whether the server is busy processing a message ($status = active$) or if it is able to accept new messages ($status = idle$). When a new message arrives, the $\mathsf{rcv}(m)_{p,s}$ event is responsible to process the incoming message. If the $status$ is equal to idle and this is a fresh message from process p then the $status$ becomes active. The $Counter(p)$ for the specific process becomes equal to the counter included in the message. The the server checks if $m(\pi)_{p,s}.t.tag > tag_s$. The comparison is validated if either:

- the timestamp of the received tag is greater than the timestamp in the local tag of the server (i.e., $m(\pi)_{p,s}.t.tag.ts > tag_s.ts$), or
- $m(\pi)_{p,s}.t.tag.ts = tag_s.ts$ and the writer identifier included in the tag of the received message is greater than the writer identified included in the local tag of the server (i.e., $m(\pi)_{p,s}.t.tag.wid > tag_s.wid$).

If any of the above cases hold, the server updates its tag and v variables to be equal to the ones included in the received message. The type of the received message specifies the type of the acknowledgment.

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

4 Correcntess

We proceed to show that algorithm CwFr satisfies both termination and atomicity properties as discussed in Section 2.

Termination. A read/write operation terminates each phase when it receives replies from at least a single quorum. As, according to our failure model, any adversary can fail all but one quorums, then any correct process eventually receives replies from at least the correct quorum. Thus, every operation from a correct process terminates; therefore, the termination condition is satisfied.

Atomicity. We now show that algorithm CwFr satisfies all atomicity properties. In particular, we use var_p to refer to the variable var of the automaton A_p. To access the value of a variable var of A_p in a state σ of an execution ξ, we use $\sigma[p].var$. Also, by $m(\pi)_{p,p'}$ we denote the message sent from p to p' as a result of operation π. Any variable var enclosed in such a message is accessed by $m(\pi)_{p,p'}.var$. We refer to a step $\langle\sigma, \mathsf{read\text{-}qview\text{-}eval}_r, \sigma'\rangle$, where $\sigma'[r].status = done$ or $\sigma'[r].phase = 2$, as the *read-fix step* of a read operation ρ invoked by reader r. Similarly we refer to a step $\langle\sigma, \mathsf{write\text{-}phase2\text{-}fix}_w, \sigma'\rangle$ as the *write-fix step* of a write operation ω invoked by w. We use the notation $\sigma_{fix(\pi)}$, to capture the final state of a read or write fix step (i.e., σ' in the previous examples) for an operation π. Finally, for an operation π, $\sigma_{inv(\pi)}$ and $\sigma_{res(\pi)}$ denote the system state before the invocation and after the response of operation π respectively.

Given this notation, the value of the maximum tag observed during a read operation ρ from a reader r is $\sigma_{fix(\rho)}[r].maxTag$. As a shorthand we use $maxTag_\rho = \sigma_{fix(\rho)}[r].maxTag$ to denote the maximum tag witnessed by ρ. Similarly, we use $minTag_\rho$ to denote the minimum tag witnessed by ρ. For a write operation we use $maxTag_\omega = \sigma_{rfix(\omega)}[w].maxTag$ to denote the maximum tag witnessed during the read phase. The state $\sigma_{rfix(\omega)}$ is the state of the system after the $\mathsf{write\text{-}phase1\text{-}fix}_w$ event occurs during operation ω. Note that $\sigma_{res(\pi)}[p].tag$ is the tag returned if π is a read operation. Lastly given tag' and a set of servers Q that replied to some operation π from p, let $(Q)^{>tag'} = \{s : s \in Q \wedge m(\pi)_{s,p}.tag > tag'\}$ be the set of servers in Q that replied with a tag greater than tag'.

We first provide an alternative definition to atomicity, to express the three atomicity properties based on the tags returned. Notice that for ease of analysis we split property P1 of definition of atomicity in two properties that capture the relation between reads and writes separately. So, the following must hold for every finite or infinite execution ξ of our implementation:

1. For each process p the tag_p variable is alphanumerically monotonically nondecreasing and it contains a non-negative timestamp.

2. If the read_r event of a read operation ρ from reader r succeeds the write-fix step of a write operation ω in ξ then, $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$.

3. If ω and ω' are two write operations from the writers w and w' respectively, such that $\omega \rightarrow \omega'$ in ξ, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$.

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

First we need to ensure that any process in the system maintains only monotonically nondecreasing tags. Hence, once some process p sets its tag_p variable to a value k at a state σ in an execution ξ, then $tag_p \neq \ell$ such that $\ell \leq k$ at a state σ' that appears after σ in ξ.

Lemma 4.1 *In any execution ξ of* CwFr*, $\sigma'[s].tag \geq \sigma[s].tag$ for any server $s \in \mathcal{S}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ.*

Proof. It is easy to see that a server s modifies its tag variable when the step $\langle \sigma, \mathsf{rcv}(m)_{p,s}, \sigma' \rangle$. From that step, $\sigma[s].tag \neq \sigma'[s].tag$ only if s receives a message during $\mathsf{rcv}_{p,s}$ such that $m(\pi)_{p,s}.tag > \sigma[s].tag$. This means that either: a) $m(\pi)_{p,s}.tag.ts > \sigma[s].tag.ts$ or b) $m(\pi)_{p,s}.tag.ts = \sigma[s].tag.ts$ and $m(\pi)_{p,s}.tag.wid > \sigma[s].tag.wid$. So, if $\sigma[s].tag \neq \sigma'[s].tag$, then $\sigma[s].tag < \sigma'[s].tag$ and the tag is monotonically incrementing. Furthermore, since the initial tag of the server is set to $\langle 0, \min(wid) \rangle$ and the tag is updated only if $m(\pi)_{p,s}.tag.ts \geq \sigma[s].tag.ts$, then for any state σ'' $\sigma''[s].tag.ts$ is always greater than 0. $\square$

Lemma 4.2 *In any execution ξ of* CwFr*, if a server s receives a message $m(\pi)_{p,s}$ from a process p, for operation π, then s replies to p with $m(\pi)_{s,p}.tag \geq m(\pi)_{p,s}.tag$.*

Proof. When the server receives the message from processor p it first compares $m(\pi)_{p,s}.tag$ with its local tag tag_s. If $m(\pi)_{p,s}.tag > tag_s$ then the server sets $tag_s = m(\pi)_{p,s}.tag$. From this it follows that the tag of the server at the state σ' after $\mathsf{rcv}_{p,s}$ is $\sigma'[s].tag \geq m(\pi)_{p,s}.tag$. Since by Lemma 4.1 the tag of the server is monotonically nondecreasing, then when the $\mathsf{send}_{s,p}$ event occurs, the server replies to p with a tag $m(\pi)_{s,p}.tag \geq \sigma'[s].tag \geq m(\pi)_{p,s}.tag$. Hence, the lemma follows. $\square$

Lemma 4.3 *In any execution ξ of* CwFr*, $\sigma'[w].tag \geq \sigma[w].tag$ for any writer $w \in \mathcal{W}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ. Also, for any state σ in ξ, $\sigma[w].tag.ts \geq 0$.*

Proof. Each writer process w modifies its local tag during its first communication round. In particular when the write-phase1-fix$_w$ event happens for a write operation ω, then the tag of the writer becomes equal to $tag_w = \langle maxTag_\omega.ts + 1, w\rangle$. So, it suffice to show that $\sigma_{inv(\omega)}[w].maxTS \leq maxTag_\omega$. Suppose that all the servers of a quorum $Q \in \mathbb{Q}$, received messages and replied to w, for ω. Every message sent from w to any server $s \in Q_j$ (when send$_{w,s}$ occurs), contains a tag $m(\omega)_{w,s}.tag = \sigma_{inv(\omega)}[w].maxTS$. By Lemma 4.2, any $s \in Q$ replies with a tag $m(\omega)_{s,w}.tag \geq m(\omega)_{w,s}.tag \geq \sigma_{inv(\omega)}[w].maxTS$. Thus, $\forall s \in Q$, $m(\omega)_{s,w}.tag \geq \sigma_{inv(\omega)}[w].maxTS$ and it follows that $m(\omega)_{s,w}.tag.ts \geq \sigma_{inv(\omega)}[w].maxTS.tag.ts$. Since $maxTag_\omega.ts = \max(m(\omega)_{s,w}.tag.ts)$ then $maxTag_\omega.ts \geq \sigma_{inv(\omega)}[w].maxTS.tag.ts$ and hence, $\sigma_{res(\omega)}[w].tag = \langle maxTag_\omega.ts + 1, w\rangle > \sigma_{inv(\omega)}[w].maxTS$. Therefore not only the tag of a writer is nondecreasing but we show explicitly that the writer's tag is monotonically increasing. Furthermore since the writer adopts the maximum tag sent from the servers, and since by Lemma 4.1 the servers tags contain non-negative timestamps, then it follows that the writer contains non-negative timestamps as well. □

Lemma 4.4 *In any execution ξ of* CwFr*, $\sigma'[r].tag \geq \sigma[r].tag$ for any reader $r \in \mathcal{R}$ and any σ, σ' in ξ, such that σ appears before σ' in ξ. Also, for any state σ in ξ, $\sigma[r].tag.ts \geq 0$.*

Proof. Notice that the tag variable of a reader is $\sigma_{inv(\rho)}[r].tag \leq \sigma_{inv(\rho)}[r].maxTS$ when the read$_r$ event occurs and becomes $\sigma_{res(\rho)}[r].tag = \sigma_{res(\rho)}[r].tag$ at the end of the operation. So, it suffices to show that $\sigma_{res(\rho)}[r].tag \geq \sigma_{inv(\rho)}[r].maxTS$. With similar arguments to Lemma 4.3 it can be shown that for every $s \in Q$ that replies to an operation ρ invoked by r, $m(\rho)_{s,r}.tag \geq \sigma_{inv(\rho)}[r].maxTS$. Since $maxTag_\rho = \max(m(\rho)_{s,r}.tag)$ and $minTag_\rho = \min(m(\rho)_{s,r}.tag)$ then it follows that both $maxTag_\rho, minTag_\rho \geq \sigma_{inv(\rho)}[r].maxTS$. By the algorithm the tag returned by the read operation is $minTag_\rho \leq \sigma_{res(\rho)}[r].tag \leq maxTag_\rho$. Hence, $\sigma_{res(\rho)}[r].tag \geq \sigma_{inv(\rho)}[r].maxTS$. Thus, no matter which of the tags is chosen to be returned at the end of the read operation nondecreasing monotonicity is preserved. Also since by Lemma 4.1 all the servers reply with a non negative timestamp, then it follows that r contains non-negative timestamps as well. □

Lemma 4.5 *For each process $p \in \mathcal{R} \cup \mathcal{W} \cup \mathcal{S}$ the tag variable is monotonically nondecreasing and contains a non-negative timestamp.*

Proof. Follows from Lemmas 4.1, 4.3 and 4.4 □

The following lemma states that if a read operation returns a tag $\tau < maxTag$ it must be the case that any pairwise intersection of the replied quorum contains a server s such that $tag_s = \tau$.

Lemma 4.6 *In any execution ξ of* CwFr*, if a read operation ρ from r receives replies from the members of quorum Q and returns a tag $\sigma_{res(\rho)}[r].tag < maxTag_\rho$, then $\forall Q' \in \mathbb{Q}, Q' \neq Q$, $(Q \cap Q') - (Q)^{>\sigma_{res(\rho)}[r].tag} \neq \emptyset$.*

Proof. By definition the intersection of two quorums $Q, Q' \in \mathbb{Q}$ is not empty. Let us assume to derive contradiction that a read operation ρ may return a tag $\sigma_{res(\rho)}[r].tag < maxTag_\rho$ and may exist $(Q \cap Q') - (Q)^{>\sigma_{res(\rho)}[r].tag} = \emptyset$. According to our algorithm, when read-qview-eval event occurs, we first check if either **QV1** or **QV3** is observed in Q. If neither of those quorum views is observed then we remove all the servers with the current maximum tag from Q and we repeat the check on the remaining servers. It follows that since all the servers $s' \in Q \cap Q'$ were removed from Q then it must be the case that $m(\rho)_{s',r}.tag > \sigma_{res(\rho)}[r].tag$. So there must be a tag $\tau' > \sigma_{res(\rho)}[r].tag$ s.t. $A = (Q \cap Q') - (Q)^{>\tau'} \neq \emptyset$ and all servers $s' \in A$ replied with $m(\rho)_{s',r}.tag = \tau'$. If this happens there are two cases for the reader:

a) $\forall s' \in (Q)-(Q)^{>\tau'}, m(\rho)_{s',r}.tag = \tau'$ and thus **QV1** is observed and the reader returns $\sigma_{res(\rho)}[r].tag' = \tau'$, or

b) $\forall s' \in A, m(\rho)_{s',r}.tag = \tau'$ and thus, **QV3** is observed and the reader returns $\sigma_{res(\rho)}[r].tag' = maxTag_{\rho}$.

Since $maxTag_{\rho} \geq \tau'$, then in any case the read operation ρ would return a tag $\sigma_{res(\rho)}[r].tag' > \sigma_{res(\rho)}[r].tag$ and that contradicts our assumption. $\square$

Derived from the above lemma, the next lemma states that a read operation basically returns either the *maxTag* or the maximum of the minimum tags of all the pairwise intersections of the replied quorum.

Lemma 4.7 *If a read operation ρ from r receives replies from a quorum Q in an execution ξ of* CwFr*, then $\forall Q' \in \mathbb{Q}, Q' \neq Q$, $\sigma_{res(\rho)}[r].tag \geq \min(m(\rho)_{s,r})$ for $s \in Q \cap Q'$.*

Proof. This lemma follows directly from Lemma 4.6. Let a subset of servers in $Q \cap Q'$ replied to ρ with the minimum tag among all the servers of that intersection, say τ. If the iteration of the read-eval-qview$_r$ event of ρ reaches tag τ then either ρ observes **QV1** and returns $\sigma_{res(\rho)}[r].tag = \tau$ or it observes **QV3** and returns $\sigma_{res(\rho)}[r].tag = maxTag_{\rho} \geq \tau$. This is true for all the intersections $Q \cap Q'$, for $Q \neq Q'$. And the lemma follows. $\square$

Lemma 4.8 *If the invocation step of a read operation ρ from reader r succeeds the write-fix step of a write operation ω from w in an execution ξ of* CwFr *then, $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$.*

Proof. Assume w.l.o.g. that the write operation receives messages from two, not necessarily different, quorums Q and Q' during its first and second communication rounds respectively. Furthermore, let us assume that the read operation receives replies from a quorum Q'', not necessarily different from Q or Q', during its first communication round. According to the algorithm the write operation ω detects the maximum tag from Q, increments that and propagates the new tag to Q'. Since $\forall s \in Q, maxTag_{\omega} \geq m(\omega)_{s,w}.tag$ then from the intersection property of a quorum system it follows that $\forall s' \in (Q \cap Q') \cup (Q \cap Q''), \sigma_{res(\omega)}[w].tag > maxTag_{\omega} \geq m(\omega)_{s',w}.tag$. From the fact that w propagates $\sigma_{res(\omega)}[w].tag$ in ω's second communication round and from Lemma 4.2 it follows that every $s \in (Q' \cap Q'')$ contains a tag $m(\omega)_{s,w}.tag \geq \sigma_{res(\omega)}[w].tag$.

Since the read$_r$ operation succeeds the write-fix step of ω, then from Lemma 4.2 the read operation will obtain a tag $m(\rho)_{s,r}.tag \geq m(\omega)_{s,w}.tag \geq \sigma_{res(\omega)}[w].tag$, from every server $s \in Q' \cap Q''$. So, $\min(m(\rho)_{s,r}.tag) \geq \sigma_{res(\omega)}[w].tag$. Thus from Lemma 4.7 $\sigma_{res(\rho)}[r].tag \geq m(\rho)_{s,r}$ for $s \in Q' \cap Q''$ and hence $\sigma_{res(\rho)}[r].tag \geq \sigma_{res(\omega)}[w].tag$ completing the proof. $\square$

Lemma 4.9 *If ω and ω' are two write operations from the writers w and w' respectively, such that $\omega \rightarrow \omega'$ in ξ, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega')}[w].tag$*

Proof. From the precedence relation of the two write operations it follows that the write-fix step occurs before the write$_{w'}$ event of ω'. Recall that for a write operation ω, $\sigma_{res(\omega)}[w].tag = \langle maxTag_{\omega}.ts{+}1, w\rangle$. So, it suffices to show here that $maxTag_{\omega'} > maxTag_{\omega}$. This however is straightforward from Lemma 4.2 and the value propagated during the second communication round of ω. In particular let ω propagate $\sigma_{res(\omega)}[w].tag > maxTag_{\omega}$ to a quorum Q. Notice that every $s \in Q$ replies with $m(\omega)_{s,w}.tag \geq \sigma_{res(\omega)}[w].tag$ to the second communication round of ω. Furthermore, let the write operation ω' receive replies from a quorum Q', not necessarily different than Q, during its first communication round. Since the write-fix step of ω occurs before the write$_{w'}$ event of ω' then, by Lemmas 4.1 and 4.2, $\forall s' \in Q \cap Q'$ $m(\omega')_{s',w'} \geq m(\omega)_{s,w} \geq \sigma_{res(\omega)}[w].tag$. Thus, $maxTag_{\omega'} \geq m(\omega')_{s',w'} \geq \sigma_{res(\omega)}[w].tag$ and hence, since $\sigma_{res(\omega)}[w].tag = \langle maxTag_{\omega'}.ts + 1, w'\rangle > maxTag_{\omega'}$, then $\sigma_{res(\omega')}[w'].tag > \sigma_{res(\omega)}[w].tag$. $\square$

Lemma 4.10 *If ρ and ρ' are two read operations from the readers r and r' respectively, such that $\rho \rightarrow \rho'$ in ξ, then $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.*

Proof. Since $\rho \rightarrow \rho'$ in ξ, then the read-ack_r event of ρ occurs before the $\text{read}_{r'}$ event of ρ'. Lets consider that both read operations are invoked from the same reader $r = r'$. It follows from Lemma 4.4 that $\sigma_{res(\rho)}[r].tag \leq \sigma_{res(\rho')}[r].tag$ because the *tag* variable is monotonically non-decrementing. So it remains to investigate what happens when the two read operations are invoked by two different processes, r and r' respectively. Suppose that every server $s \in Q$ receives the messages of operation ρ with an event $\text{rcv}(m)_{r,s}$, and replies with a tag $m(\rho)_{s,r}.tag$ with an event $\text{send}(m)_{s,r}$ to r. Notice that for every server that reply, as mentioned in Lemma 4.2, $m(\rho)_{s,r}.tag \geq \sigma_{inv(\rho)}[r].maxTS$. Let the members of the quorum Q' (not necessarily different than Q) receive messages and reply to ρ'. Again for every $s' \in Q'$, $m(\rho')_{s'} \geq \sigma_{inv(\rho')}[r'].maxTS$. We know that the tag of the read operation ρ after the read-qview-eval_r event of ρ may take a value between $maxTag_\rho \geq \sigma_{res(\rho)}[r].tag \geq minTag_\rho$. It suffice to examine the two extreme cases and every intermediate value can be proved similarly. So we have two cases to examine: (1) $\sigma_{res(\rho)}[r].tag = minTag_\rho$, and (2) $\sigma_{res(\rho)}[r].tag = maxTag_\rho$.

Case 1: Consider the case where $\sigma_{res(\rho)}[r].tag = minTag_\rho$, including the case where $minTag_\rho = maxTag_\rho$. This may happen only if the read-qview-eval_r event reaches an iteration with tag $\tau = minTag_\rho$ and observes **QV1**. In other words all the servers $s \in Q - (Q)^{>\tau}$ replied with $m(\rho)_{s,r}.tag = minTag_\rho$. By Lemma 4.6 it follows that $(Q \cap Q') - (Q \cap Q')^{>\tau} \neq \emptyset$ and thus every server $s' \in Q \cap Q'$ replied to ρ with a tag $m(\rho)_{s,r}.tag \geq minTag_\rho$. By Lemma 4.1 it follows that every server $s' \in Q \cap Q$, replies with a tag $m(\rho')_{s',r'}.tag \geq m(\rho)_{s',r}.tag \geq minTag_\rho$. The read operation ρ' may return a value within the interval $minTag_\rho \leq \sigma_{res(\rho')}[r'].tag \leq maxTag_\rho$. Since for every server $s' \in Q \cap Q'$, $m(\rho')_{s',r'}.tag \geq minTag_\rho = \sigma_{res(\rho)}[r].tag$ then $maxTag_{\rho'} \geq m(\rho')_{s',r'}.tag \geq \sigma_{res(\rho)}[r].tag$. Hence, if $\sigma_{res(\rho)}[r'].tag = maxTag_{\rho'}$ it follows that $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$. On the other hand, if $\sigma_{res(\rho')}[r'].tag = minTag_{\rho'}$ we need to consider two cases: a) $minTag_{\rho'} \geq minTag_\rho$ and b) $minTag_{\rho'} < minTag_\rho$. If the first case is valid then it follows immediately that $\sigma_{res(\rho')}[r'].tag \geq minTag_\rho$ and thus $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$. If case b) is valid then it follows that the iteration reached a tag equal to $minTag_{\rho'}$. Since however every server $s' \in Q \cap Q'$, replied with $m(\rho')_{s',r'}.tag \geq minTag_\rho$, then $m(\rho')_{s',r'}.tag \geq minTag_{\rho'}$ as well and thus all these servers should be removed by iteration where tag is equal to $minTag_{\rho'}$. But this means that $(Q \cap Q') - (Q')^{>minTag_{\rho'}} = \emptyset$ and that contradicts Lemma 4.6. So such a case is impossible.

Case 2: Here we examine the case where $\sigma_{res(\rho)}[r].tag = maxTag_\rho$. This may happen after the read-qview-eval_r of ρ if either observes a quorum view **QV1** or a quorum view **QV3**. Let us examine the two cases separately.

Case 2a: In this case ρ witnessed a **QV1**.

Therefore it must be the case that $\forall s \in Q, s$ replied with $m(\rho)_{s,r}.tag = maxTag_\rho = minTag_\rho = \sigma_{res(\rho)}[r].tag$. Thus by Lemma 4.1 $\forall s \in Q \cap Q'$, s replies with a tag $m(\rho')_{s,r'}.tag \geq m(\rho)_{s,r}.tag$ to ρ', and hence, ρ' witnesses a maximum tag

$$maxTag_{\rho'} \geq maxTag_\rho \Rightarrow maxTag_{\rho'} \geq \sigma_{res(\rho)}[r].tag \tag{1}$$

Recall that $minTag_{\rho'} \leq \sigma_{res(\rho')}[r'].tag \leq maxTag_{\rho'}$. Clearly if $\sigma_{res(\rho')}[r'].tag = maxTag_{\rho'}$ then $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$. So it remains to examine the case where $\sigma_{res(\rho')}[r'].tag < maxTag_{\rho'}$. By Lemma 4.7, $\sigma_{res(\rho')}[r'].tag$ must be greater or equal to the minimum tag of any intersection of Q'. Since $\min(m(\rho')_{s',r'}.tag) \geq \sigma_{res(\rho)}[r].tag$, for every $s' \in Q \cap Q'$, then by that lemma $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$.

Case 2b: This is the case where $\sigma_{res(\rho)}[r].tag = maxTag_\rho$, because r witnessed a quorum view **QV3**. In this case ρ proceeds in phase 2 before completing. Since $\rho \rightarrow \rho'$ and since ρ' happens after the read-ack_r [1] action of ρ, it means that ρ' happens after the read-phase2-fix_r action of ρ as well. However ρ proceeds to phase 2 only after the read-phase1-fix_r and read-qview-eval_r actions. In the latter action

[1] read-ack_r occurs only if all phases reach a fix point and the *status* variable becomes equal to *done*

ρ fixes the $maxTag$ variable to be equal to the $maxTag_\rho$. Once in phase 2, ρ sends inform messages with $maxTag_\rho$ to a complete quorum, say Q''. By Lemma 4.5, every server $s \in Q''$ replies with a tag

$$m(\rho)_{s,r}.tag \geq maxTag_\rho \Rightarrow m(\rho)_{s,r}.tag \geq \sigma_{res(\rho)}[r].tag \tag{2}$$

So ρ' will observe (by Lemma 4.1) that at least $\forall s' \in Q' \cap Q''$, $m(\rho')_{s',r'}.tag \geq \sigma_{res(\rho)}[r].tag$. Hence by Lemma 4.7 ρ' returns a tag $\sigma_{res(\rho')}[r'].tag \geq \min(m(\rho')_{s',r'}.tag)$ and thus, $\sigma_{res(\rho')}[r'].tag \geq \sigma_{res(\rho)}[r].tag$ and this completes our proof. □

Lastly the following lemma states that if two read operations return two different tags then the values that correspond to these tags are also different.

Lemma 4.11 *If ρ and ρ' two read operations from readers r and r' respectively, such that ρ (resp. ρ') returns the value written by ω (resp. ω'), then if $\sigma_{res(\rho)}[r].tag \neq \sigma_{res(\rho')}[r'].tag$ then ω is different than ω' otherwise they are the same write.*

Proof. This lemma is ensured because a unique tag is associated to each written value by the writers. So it cannot be the case that two readers such that $\sigma_{res(\rho)}[r].tag \neq \sigma_{res(\rho')}[r'].tag$ returned the same value. □

Using the above lemmas we can obtain:

Theorem 4.12 *Algorithm* CwFr *implements a MWMR atomic read/write register.*

5 Conclusions

We presented a new algorithm, called CwFr, that implements an atomic read/write register in the MWMR model and allows some fast read operations. To achieve this, the algorithm incorporates Quorum Views, client side decision tools used to identify the last completed write. CwFr does not depend in any constraints on the system: it allows unbounded number of participants (readers or writers) and can utilize any general quorum system. Thus, it overcomes the restrictive requirements of algorithm SfW [5] on the quorum construction by sacrificing the speed of write operations. We believe that our algorithm will allow more fast read operations than SfW when deploying quorum systems with small intersection degree ($n < 5$). A comparable number of fast reads is expected when using quorum systems with higher intersection degree. The computation cost of the iterative procedure used by CwFr is polynomial on the size of the quorums and appears to be faster than the computation of the predicates in SfW. We plan to examine the computation burden of the predicates in SfW and a more precise comparison will be obtained.

Acknowledgments. We would like to thank Alexander A. Shvartsman and Alexander C. Russell for several discussions.

References

[1] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message passing systems. *Journal of the ACM 42(1)* (1996), 124–142.

[2] Chockler, G., Gilbert, S., Gramoli, V., Musial, P. M., and Shvartsman, A. A. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 100–116.

[3] Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proceedings of 17th International Symposium on Distributed Computing (DISC)* (2003).

[4] Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.

[5] Englert, B., Georgiou, C., Musial, P. M., Nicolaou, N., and Shvartsman, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of DIstributed Systems (OPODIS 09)* (2009), pp. 240–254.

[6] Englert, B., and Shvartsman, A. A. Graceful quorum reconfiguration in a robust emulation of shared memory. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)* (2000), pp. 454–463.

[7] Fan, R., and Lynch, N. Efficient replication of large data objects. In *Distributed algorithms* (Oct 2003), F. E. Fich, Ed., vol. 2848/2003 of *Lecture Notes in Computer Science*, pp. 75–91.

[8] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.

[9] Georgiou, C., Nicolaou, N. C., and Shvartsman, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 62–79. A preliminary version of this work appeared in the proceedings 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'06).

[10] Gramoli, V., Anceaume, E., and Virgillito, A. SQUARE: scalable quorum-based atomic memory with local reconfiguration. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), ACM, pp. 574–579.

[11] Lynch, N. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[12] Lynch, N., and Shvartsman, A. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)* (2002), pp. 173–190.

[13] Lynch, N., and Tuttle, M. An introduction to input/output automata. *CWI-Quarterly* (1989), 219–246.

[14] Lynch, N. A., and Shvartsman, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.