Towards Feasible Implementations of Low-Latency Multi-Writer Atomic Registers

Chryssis Georgiou
Univ. of Cyprus
chryssis@cs.ucy.ac.cy

Nicolas Nicolaou
Univ. of Cyprus & Univ. of Connecticut
nicolas@engr.uconn.edu

Alexander C. Russell
Univ. of Connecticut
acr@cse.uconn.edu

Alexander A. Shvartsman
Univ. of Connecticut
aas@cse.uconn.edu

***Abstract*—This work explores implementations of multi-writer/multi-reader (MWMR) atomic registers in asynchronous, crash-prone, message-passing systems with the focus on low latency and computational feasibility. The efficiency of atomic read/write register implementations is traditionally measured in terms of the latency of read and write operations. To reduce operation latency researchers focused on the communication costs, expressed as the number of communication round-trips (or rounds), often ignoring the computation costs.**

In this paper we consider efficiency of a register implementation in terms of both communication and computation costs. As of this writing, algorithm SFW is the sole known MWMR algorithm that allows single round read and write operations. The algorithm uses collections of intersecting sets (quorums), and to enable single round operations, SFW relies on the evaluation of certain predicates. We formulate a new combinatorial problem that captures the computational burden of evaluating the predicates in algorithm SFW and we show that it is *NP-Complete*. To make the evaluation of the predicates feasible, we present a *polynomial* $\log$*-approximation* algorithm for this problem and we show how to use it with algorithm SFW. Then we present a new algorithm, called CWFR, that allows fast operations *independently* of the underlying quorum system construction. The algorithm implements two-round writes and allows reads to complete in a single round. We conclude with experimental evaluations of our algorithms obtained from simulations in NS2.

I. 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. [6] developed the first single-writer/multi-reader (SWMR) algorithm, where all operations complete in a single round. Such operations are called *fast*. They also showed that it is impossible to have multi-writer/multi-reader (MWMR) implementations where *all* operations are fast.

As of this writing, algorithm SFW of Englert et al. [7], is the only MWMR algorithm that enables some reads and writes to be fast. 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 specialized *predicates*. The main drawbacks of this algorithm is that it contains evaluations of the predicates that require substantial computational effort, and that it relies on very specialized quorum constructions. Thus this algorithm is primarily of theoretical importance and it is not sufficiently practical.

Contributions: Our goal is to provide efficient and practical implementations of atomic MWMR registers. We examined algorithm SFW [7], and we identified two weaknesses with respect to its practicality: (1) the algorithm uses two computationally hard predicates to decide on the value of the register, and (2) fast write operations are enabled only if the quorum system satisfies specific quorum intersection properties. Motivated by these observations, our contributions are as follows:

(1) We define a new combinatorial problem, called K-SET-INTERSECTION, that represents both predicates used in algorithm SFW. We prove that the problem, and hence the evaluation of the predicates, are *NP-Complete* by reduction from the 3-SAT problem. We present a polynomial time approximation algorithm that uses as its core a greedy approximation algorithm for the SET COVER problem. Our approximation provides a $\log u$-approximation for the number of sets included in the solution, where u is the size of the set given as the input; for algorithm SFW, u is the number of severs. We derive a new atomic register algorithm, called APRX-SFW, by embedding our approximation algorithm to evaluate the predicates in algorithm SFW. For $O(\log u)$ predicate evaluations, the approximation used by algorithm APRX-SFW may yield false negatives, however this is a performance, not a correctness, issue.

(2) 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 [9], 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.

(3) We obtained experimental results by simulating our algorithms on the NS2 simulator. In particular, we first compare algorithms SFW and APRX-SFW in terms of the number of second communication rounds and show that the experimental results are within the theoretical approximation bounds. Furthermore, the hardness of the predicate evaluation computation is made evident from the observed operation latency (as the number of servers increases). We then compare the operation latency of algorithms APRX-SFW, CWFR, and a traditional two-round algorithm that incurs a low computational overhead. We observe that the first two algorithms achieve lower latency despite the computational burden. Finally, we compare the operation latency and the percentage of fast reads of algorithms CWFR and APRX-SFW. We observe that in quorum systems with small intersection degree, CWFR seems to perform better than APRX-SFW; in quorums with large intersection degree APRX-SFW performs better.

Background and prior work: Attiya et al. [2] gave a SWMR algorithm that achieves consistency by using intersecting majorities of servers in combination with $\langle timestamp, value\rangle$ value tags. 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 [2] is readily generalized to quorum-based approaches in the MWMR setting (e.g., [14], [13], [8], [11]). 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.* [5] and Chockler *et al.* [3], 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. [6] 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. [10] 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 impossibility of semifast MWMR implementations.

Georgiou et al. [9] showed that fast and semifast quorum-based SWMR implementations are possible iff 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 [9] 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.* [7] 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 algorithm 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 accommodate up to $n-1$ readers and writers.

Paper organization: In Section II we give the model of computation and the notation we use throughout. In Section III we overview algorithm SFW. Section IV introduces the new combinatorial problem, its analysis, and the approximation algorithm. Algorithm CWFR is presented in Section V. Simulation results and comparisons of algorithms are in Section VI. We conclude in Section VII. *Omitted discussion and proofs are found in [1].*

II. MODEL AND DEFINITIONS

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 $I_{\mathcal{A}} = \bigcap_{Q\in\mathcal{A}} Q$. 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 $I_{\mathcal{A}} \neq \emptyset$. We call n the *intersection degree* of $\mathbb{Q}$. Any quorum system is a *2-wise* (pairwise)

quorum system because any two quorums intersect. At the other extreme, a $|\mathbb{Q}|$-*wise* quorum system has a common intersection among all quorums. Note that an *n-wise* quorum system is also a *k-wise* quorum system, for $2 \leq k \leq n$.

Our system allows processes to fail by crashing. A process i is *faulty* in an execution if i crashes in the execution **(i is not allowed to recover)**; 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*. 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 [12]. 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 $\pi_1 \rightarrow \pi_2$, 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 [6], [10], [9]:

Definition 2.1: Process p performs a communication round during operation π if all of the following hold:

1. p sends request messages for 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* [6] 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.

III. Brief Description of Algorithm SFW

Algortihm SFW assumes that the servers are arranged in an n-*wise* quorum system. To order the written values the algorithm uses $\langle tag, value \rangle$ pairs. To enable fast writes the algorithm assigns partial responsibility to the servers for the ordering of the values written. If a server receives a write request it generates a new tag, larger than any of the tags it witnessed, and assigns it to the value enclosed in the write message. The server records a generated tag, along with the write operation it was created for, in a set called $inprogress$. The set holds only the latest tag generated for each writer.

Each reader or writer must communicate with a quorum of servers, say Q, during the first round of each read/write operation. Due to concurrency different servers can receive messages from write operations in different order, thus an operation may witness different tags assigned to a single write operation. To deal with this algorithm SFW uses two *predicates* to determine whether "enough" servers in the replying quorum assigned the same tag to a particular write operation. Let n be the intersection degree of the quorum system, and $inprogress_s(\omega)$ be the $inprogress$ set that server s enclosed in the message it sent to the writer that invoked ω. The write and read predicates are:

PW: Writer predicate for a write ω: $\exists\ \tau, A, MS$ where: $\tau \in \{\langle ., \omega \rangle : \langle ., \omega \rangle \in inprogress_s(\omega) \ \wedge \ s \in Q\}$, $A \subseteq \mathbb{Q}, 0 \leq |A| \leq \frac{n}{2} - 1$, and $MS = \{s : s \in Q \ \wedge \ \tau \in inprogress_s(\omega)\}$, s.t. either $|A| \neq 0$ and $I_A \cap Q \subseteq MS$ or $|A| = 0$ and $Q = MS$.

PR: Reader predicate for a read ρ: $\exists\ \tau, B, MS$, where: $\max(\tau) \in \bigcup_{s \in Q} inprogress_s(\rho)$, $B \subseteq \mathbb{Q}, 0 \leq |B| \leq \frac{n}{2} - 2$, and $MS = \{s : s \in Q \ \wedge \ \tau \in inprogress_s(\rho)\}$, s.t. either $|B| \neq 0$ and $I_B \cap Q \subseteq MS$ or $|B| = 0$ and $Q = MS$.

The predicates examine whether the same tag for a write

operation is contained in the replies of all servers in the intersection among the replying quorum and $\frac{n}{2}-1$ for PW (resp. $\frac{n}{2}-2$ for PR) of other replying quorums. Satisfaction of the predicates for a tag τ guarantees that any subsequent operation will also determine that the write operation is assigned tag τ. If the predicates hold with $|A| \geq \frac{n}{2}-1$ or $|B| = \frac{n}{2}-2$ then the write or read operation respectively needs to proceed to a second round. A write operation can only be fast if **PW** holds. A read operations can be fast even if **PR** does not hold, but the read observed enough $confirmed$ tags with the same value. Confirmed tags are maintained in the servers and they indicate that either the write of the value with that tag is complete, or the tag was returned by some read operation. See [7] for full details.

IV. NP-Completeness and Approximation

The complexity of the predicates raises the question whether they can be computed efficiently. The two predicates can be captured by a decision problem that we formalize as follows:

Definition 4.1 (k-Set-Intersection): Given a set of elements U, a subset of those elements $M \subseteq U$ and a set of subsets $\mathbb{Q} = \{Q_1, \ldots, Q_n\}$ s.t. $Q_i \subseteq U$, a set I is an intersecting set if $I \subseteq \mathbb{Q}$, $\bigcap_{Q \in I} Q \neq \emptyset$, and $\bigcap_{Q \in I} Q \subseteq M$. If $|I| = k$ then I is a k intersecting set.

To the best of our knowledge this is a new combinatorial problem and it is similar to the open problem stated in [4]. In the context of [7], the universe of elements U is the set of servers, and the set of subsets of U is the deployed quorum system. Clearly k-Set-Intersection is in NP: given $(U, M, \mathbb{Q})$ and a set $I \subset \mathbb{Q}$, s.t. $|I| = k$, we can verify in polynomial time **(with respect to $|\mathbb{Q}|$)** if $\bigcap_{Q \in I} Q \subseteq M$.

A. Polynomial Reduction from 3-Sat

We now show that the k-Set-Intersection problem is *NP-Complete* by providing a polynomial reduction from the 3-Sat problem. The reduction involves a polynomial transformation of the input to 3-Sat to an instance of k-Set-Intersection. We first provide the definition of 3-Sat [15]:

Definition 4.2 (3-Sat): Let $X = \{x_1, \ldots, x_n\}$ be a set of variables and Φ a boolean formula in CNF (Conjuctive Normal Form) where each clause contains at most three literals (variable or its negation). Is there a truth assignment to every $x_i \in X$ s.t. Φ becomes true?

Construction: We transform an instance of the 3-Sat problem to an instance $(U, M, \mathbb{Q}, k)$ of k-Set-Intersection as follows. Let $k = n$ the total number of variables. The universe consists of an element for each variable, the negation of each variable and an element for each clause C_i of 3-Sat. It also includes n elements which will ensure that each variable is chosen at least once:

$$U = \{x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n, C_1, \ldots, C_m, \ell_1, \ldots, \ell_n\}$$

The set $M \subseteq U$ contains all the elements that appear in the clauses. Both the variable x_i and its negation $\overline{x}_i$ may appear in M, if they appear is some clause of the boolean formula. Thus the set M is constructed in $O(2nm)$ time as follows:

$$M = \{x_i : \exists C_j, x_i \in C_j\} \cup \{\overline{x}_i : \exists C_j, \overline{x}_i \in C_j\}$$

Lastly we construct the set of subsets $\mathbb{Q}$. For each variable $x_i \in M$ we construct a subset, Q_i and for each variable $\overline{x}_i \in M$ we construct a subset Q'_i. Every Q_i contains the variable x_i, the variables x_j for $j \neq i$ and their negations, and the clauses that do not contain x_i or contain $\overline{x}_i$. Intuitively, those are the clauses that are not directly satisfied if we set $x_i = true$. Finally, we include one element ℓ_j for each $j \neq i$. These elements will ensure that for a variable x_i we choose either Q_i or Q'_i but not both. We construct Q'_i similarly for $\overline{x}_i$. More formally the sets we obtain are the following:

$$\begin{aligned} Q_i &= \{x_i : x_i \in M\} \cup \{x_j, \overline{x}_j : j \neq i\} \\ &\quad \cup \{C_j : x_i \notin C_j \text{ or } \overline{x}_i \in C_j\} \cup \{\ell_j : j \neq i\} \\ Q'_i &= \{\overline{x}_i : \overline{x}_i \in M\} \cup \{x_j, \overline{x}_j : j \neq i\} \\ &\quad \cup \{C_j : \overline{x}_i \notin C_j \text{ or } x_i \in C_j\} \cup \{\ell_j : j \neq i\} \end{aligned}$$

Given the above sets, the set of subsets $\mathbb{Q}$ is: $\mathbb{Q} = \left\{\bigcup_{x_i \in M}\{Q_i\}\right\} \cup \left\{\bigcup_{\overline{x}_i \in M}\{Q'_i\}\right\}$. The construction of all sets Q_i and Q'_i takes at most $O(2n^2m)$.

The idea of this construction is to find a set of subsets such that their intersection contains positive and negative variables and no clauses or elements ℓ_j. In our construction this implies that setting the variables of the intersection to true satisfies all clauses. In addition, the elimination of the elements ℓ_j, in combination with k being equal to n, implies that we choose either Q_i or Q'_i but not both. Therefore, the intersection of n subsets implies that we chose a single truth value for every variable. With this construction we formally show that 3-Sat $\leq_p$ k-Set-Intersection, obtaining the following theorem:

Theorem 4.3: k-Set-Intersection is *NP-Complete*.

B. Approximation Algorithm

Here we provide a polynomial time algorithm that yields an approximate solution to the problem given in Definition 4.1. As a part of our algorithm we use the standard Set-Cover greedy $\log$-approximation algorithm (cf. [15]). The set cover problem is defined as follows [15]:

Definition 4.4 (Set-Cover): Given a universe U of elements, a collection of subsets of U, $\mathcal{S} = \{S_1, \ldots, S_z\}$, and a number k, find at most k sets of $\mathcal{S}$ such that their union covers all elements in U.

We now present the steps of the algorithm in Figure 1 and provide an explanation of the algorithm's rationale.

Every T_m contains the complements of the quorums that contain m. Let $R_{m,i} = (U - M) - (Q_i - M)$ for $m \in Q_i$. Given the sets $R_{m,i}$ if we can find k of those that $R_{m,1} \cup \ldots \cup R_{m,k} = U - M$, then by de Morgan's Law it follows that $\overline{R_{m,1}} \cap \ldots \cap \overline{R_{m,k}} = \emptyset$. Since, $R_{m,i} = (U-M)-(Q_i-M)$, then $\overline{R_{m,i}} = (Q_i - M)$ and

$$\overline{R_{m,1}} \cap \ldots \cap \overline{R_{m,k}} = (Q_i - M) \cap \ldots \cap (Q_k - M) = \emptyset \quad (1)$$

For an instance $(U, M, \mathbb{Q}, k)$ of k-SET-INTERSECTION do:
Step 1: $\forall m \in M$
let $T_m = \{(U - M) - (Q_i - M) : m \in Q_i\}$
Step 2: Run SET-COVER greedy algorithm on
the instance $\{U - M, T_m, k\}$ for every $m \in M$:
Step 2a: Pick the set $R_i \in T_m$ with
the maximum uncovered elements
Step 2b: Take the union of every $R \in T_m$
picked in Step 2a (incl. R_i)
Step 2c: If the union equals $U - M$ go to Step 3;
else if there are more sets in T_m go to Step 2a
else repeat for another $m \in M$
Step 3: For any set $(U - M) - (Q_i - M)$ in the solution
of set cover, add Q_i in the intersecting set.

Fig. 1. Polynomial approximation algorithm for k-SET-INTERSECTION.

By construction $\forall R_{m,i} \in T_m$, $m \in Q_i$, and thus $\{m\} \subseteq Q_i \cap \ldots \cap Q_k$. From this and (1) it follows that $Q_i \cap \ldots \cap Q_k$ is a non-trivial subset of M.

It is known [15] that SET-COVER greedy algorithm is a $\log u$-approximation algorithm, where $u = |U|$. That is, if k is the optimal solution, then the greedy algorithm will include at most $k \log u$ sets in its solution. As the number of subsets in the solution of k-SET-INTERSECTION is the same as the number of subsets in the solution of SET-COVER, we obtain the following lemma:

Lemma 4.5: The algorithm in Figure 1 is a $\log u$-approximation algorithm for the k-SET-INTERSECTION problem, where $u = |U|$.

If we use the above algorithm to evaluate the predicates of algorithm SFW, the resulting implementation yields a logarithmic in the number of servers increase in the number of second communication rounds. This is a modest price to pay in exchange for substantial reduction in the computation overhead of algorithm SFW. In Section VI we present an empirical evaluation of the approximate algorithm SFW comparing it to the original algorithm SFW.

V. ALGORITHM CWFR

In this section we explore the possibility of introducing fast operations in the MWMR setting when servers are organized as an arbitrary quorum system. We introduce a new algorithm, called algorithm CWFR, that enables fast read operations by adopting the general idea of Quorum Views [9]. The algorithm employs two techniques:

(i) the typical query and propagate approach (two rounds) for write operations, and
(ii) analysis of Quorum Views [9] for potentially fast (single round) read operations.

Read operations can be fast in algorithm CWFR even when they are invoked concurrently with write operations. This distinguishes algorithm CWFR from previous approaches [5], [3]. To impose a total ordering on the written values, algorithm CWFR uses $\langle tag, value \rangle$ pairs. A tag is a tuple of the form $\langle \tau, w \rangle \in \mathbb{N} \times \mathcal{W}$, where τ is the timestamp and w is a writer identifier. Such tags are compared lexicographically.

A. Quorum Views

We generalize the definition of *quorum views* from [9] for use with structured tags:

Definition 5.1: 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 $tag_s(\pi)$ and let $maxTag = \max_{s \in Q}(tag_s(\pi))$. We say that p observes one of the following **quorum views** for Q:

- $qView(1)$: $\forall s \in Q : tag_s(\pi) = maxTag$,
- $qView(2)$: $\forall Q' \in \mathbb{Q} : \ Q \neq Q' \wedge \exists A \subseteq Q \cap Q'$, s.t. $A \neq \emptyset$ and $\forall s \in A : tag_s(\pi) < maxTag$,
- $qView(3)$: $\exists s' \in Q : tag_{s'}(\pi) < maxTag$ and $\exists Q' \in \mathbb{Q}$ s.t. $Q \neq Q' \wedge \forall s \in Q \cap Q' : tag_s(\pi) = maxTag$

Restating the above definition, $qView(1)$ requires that all servers in some quorum reply with the same tag. $qView(3)$ reveals that some servers in the quorum contain an older value, but there exists an intersection where all of its servers contain the new value. Finally $qView(2)$ is the negation of the other two views, revealing a quorum where the new value is neither distributed to the full quorum nor distributed fully in any of its intersections.

B. Description of CWFR

The original quorum views algorithm [9] relies on the fact that there is a single writer. If a quorum view is able to predict the non-completeness of the latest write operation, it is immediately understood that – by the well-formedness of the single writer – any previous write operation is already complete. Multiple writers invalidate such a conclusion: different values (and 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. Thus a direct adaptation of the quorum view idea from the SWMR model to the MWMR model is not possible. Consequently, 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 complete write operation. Below we provide a description of our algorithm and present the main idea behind our technique. The pseudocode of the algorithm appears in Figure 2.

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 members of some 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 some quorum, and completes the write.

Readers: The read protocol is more involved. The reader sends a read message to all servers and waits for some quorum to reply. Once a quorum replies, the reader determines $maxTag$. Then the reader analyzes the distribution of the tag within the responding quorum Q in an attempt to

write(val):
init: tag=$\langle 0, wid\rangle$, v=$\perp$, $wcounter$=0
1: $wcounter$++
2: send $\langle READ, \langle tag, v\rangle, wcounter\rangle$ to all servers
3: wait for the servers of a quorum Q to reply
4: /* find maximum tag among the replies */
5: $tag = \max_{s\in Q}(s.tag)$
6: /* increment the maximum tag and generate a new tag */
7: $tag = \langle tag.ts + 1, wid\rangle$
8: $v = val$
9: $wcounter$++
10: send $\langle WRITE, \langle tag, v\rangle, wcounter\rangle$ to all servers
11: wait for the servers of a quorum Q to reply
12: return OK

read():
init: tag=$maxTag$=$\langle 0, 0\rangle$, v=$\perp$, $rcounter$=0
1: $rcounter$++
2: send $\langle READ, \langle tag, v\rangle, wcounter\rangle$ to all servers
3: wait for the servers of a quorum Q to reply
4: **while** $(Q \neq \emptyset)$ **do**
5: $\langle maxTag, v\rangle = \max_{s\in Q}(\langle s.tag, s.v\rangle)$
6: **if** $(\forall s \in Q : s.tag = maxTag)$ **then**
7: /* qView(1) */
8: $tag = maxTag$
9: return tag
10: **end if**
11: /* qView(3) */
12: **if** $\exists Q' : Q' \neq Q \wedge \forall s \in Q' \cap Q, s.tag = maxTag$ **then**
13: $tag = maxTag$
14: send $\langle WRITE, \langle tag, v\rangle, wcounter\rangle$ to all servers
15: wait for the servers of a quorum Q to reply
16: return tag
17: **end if**
18: /* qView(2) */
19: **if** $\forall Q' : Q' \neq Q \wedge \exists s \in Q' \cap Q, s.tag < maxTag$ **then**
20: $Q = Q - \{s : s \in Q \wedge s.tag = maxTag\}$
21: **end if**
22: **end while**

serve():
init: tag=$\langle 0, 0\rangle$, v=$\perp$, $pCounter[]$=0
1: upon receipt of $\langle msgType, \langle t, val\rangle, counter\rangle$ from process p
2: /* check message freshness */
3: **if** $counter > pCounter[p]$ **then**
4: **if** $t > tag$ **then**
5: $\langle tag, v\rangle = \langle t, val\rangle$
6: **end if**
7: **if** $msgType = WRITE$ **then**
8: send $\langle WRITEACK, \langle tag, v\rangle, pCounter[p]\rangle$ to p
9: **else**
10: send $\langle READACK, \langle tag, v\rangle, pCounter[p]\rangle$ to p
11: **end if**
12: **end if**

Fig. 2. Pseudocode for Writer, Reader and Server of algorithm CWFR.

determine the latest, potentially complete, write operation. This is accomplished by determining the quorum view conditions. Detecting conditions of $qView(1)$ and $qView(3)$ are straightforward. When condition for $qView(1)$ is detected, the read completes and the value associated with the discovered $maxTag$ is returned. In the case of $qView(3)$ the reader continues to the second round, advertising the latest tag ($maxTag$) and its associated value. When a full quorum replies in the second round, the read returns the value associated with $maxTag$.

Analysis of $qView(2)$ involves the 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 $qView(1)$ or $qView(3)$ being observed. If $qView(1)$ 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 $qView(3)$ 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.

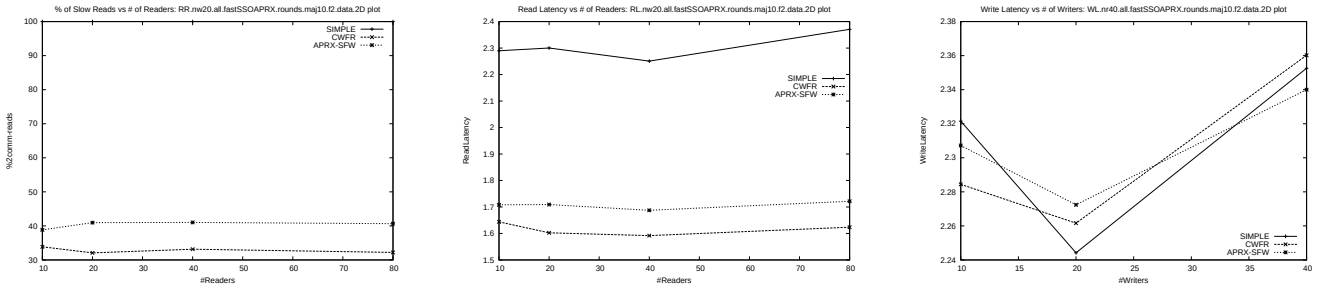
Servers: The servers play a passive role. They receive read or write requests, update their object replica accordingly, and reply to the process that invoked the operation. Upon receipt of any message, the server compares its local tag with the tag included in the message. If the tag of the message is 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.

Main Idea: We now explain the idea behind our 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_i, before completing. Thus, when a read operation ρ, s.t. $\omega \rightarrow \rho$, receives replies from some quorum Q_j, then observes one of the following tag distributions: (a) if $Q_j = Q_i$, then $\forall s \in Q_j, tag_s = tag_\omega$ ($qView(1)$), or (b) if $Q_j \neq Q_i$, then $\forall s \in Q_i \cap Q_j, tag_s = tag_\omega$ ($qView(3)$). Hence, if ρ observes a distribution as in $qView(1)$ then the 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 $qView(3)$) then there is a possibility that some write completed in an intersecting quorum (in this example Q_i). The read operation is fast in $qView(1)$ since it is determinable that the write potentially completed. The read proceeds to the second round in $qView(3)$, since the completion of the write is indeterminable and it is necessary to ensure that any subsequent operation observes that tag. If neither $qView(1)$ nor $qView(3)$ hold, then $qView(2)$ holds, and it must be the case that the write that yields the maximum tag is not yet complete. Hence we try to discover the latest potentially complete write by removing all servers with the highest tag from Q_j and repeating the analysis. If at some iteration, $qView(1)$ holds on the remaining tag values, then a potentially complete write (that was overwritten by greater tags in the rest of the servers) is discovered and that tag is returned. If no iteration is interrupted because of $qView(1)$, then eventually $qView(3)$ is observed, in the worst case, when a single server remains in some intersection of Q_j. Since a second round cannot be avoided in this case, we take the opportunity to propagate the largest tag observed in Q_j. At the end of the second round that tag is written to at least one complete quorum and thus the reader can safely return the corresponding value.

Theorem 5.2: Algorithm CWFR implements an atomic MWMR register.

VI. EMPIRICAL RESULTS: SIMULATIONS

We now present experimental evaluations of our algorithms, obtained by using the NS-2 network simulator.



(a) (b) (c)

Fig. 3. 4-wise quorum system ($|\mathcal{S} = 10$, $f = 2$): (a) Percentage of slow reads, (b) Latency of read operations, and (c) Latency of write operations.

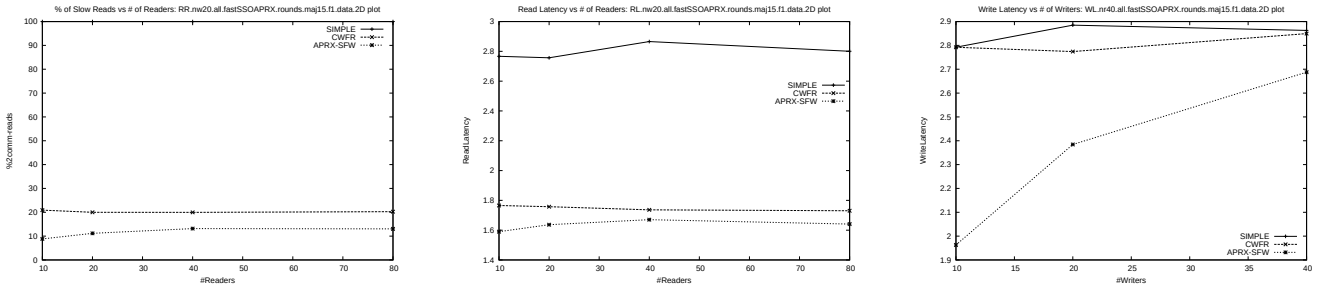
Experimentation Platform: Our test environment consists of a set of writers, readers, and servers. We use bidirectional links between the communicating nodes, with 1Mb bandwidth, latency of $10ms$, and a DropTail queue. To model asynchrony, the processes send messages after a random delay between 0 and 0.3 sec. **The NS2 was running in Ubuntu, on a Centrino 1.8GHz processor. The average of 5 samples per scenario provided operation latencies.**

We have evaluated the algorithms with majority quorums. As discussed in [7], assuming $|\mathcal{S}|$ servers out of which f can crash, we can construct an $(\frac{|\mathcal{S}|}{f} - 1)$-wise quorum system $\mathbb{Q}$. Each quorum Q of $\mathbb{Q}$ has size $|Q| = |\mathcal{S}| - f$. The processes are not aware of f. The quorum system is generated *a priori* and is distributed to each participant node via an external service (out of the scope of this work). We model server failures by selecting some quorum of servers (unknown to the participants) to be correct and allowing any other server to crash. The positive time parameter $cInt$ is used to model the failure frequency or reliability of every server s. We use the positive time parameters $rInt = 5sec$ and $wInt = 10sec$ to model operation frequency. Readers and writers pick a uniformly at random time between $[0 \dots rInt]$ and $[0 \dots wInt]$, respectively, to invoke their next read (resp. write) operation.

Algorithm SFW ***vs.*** APRX-SFW***:*** First we compare algorithms SFW and APRX-SFW. We examine a specific scenario where the number of readers is fixed at 40 and the number of writers is fixed at 20 (other scenarios can be found in [1]). By assuming a single server failure and increasing the number of servers in the system, we evaluate the two algorithms using quorum systems with different intersection degrees. In particular, we run the scenario using 10, 15, and 25 servers that, with a single failure, yield a 9-wise, 14-wise, and 24-wise quorum system respectively. Examining the latency of the two algorithms, including both communication and computation costs, provides evidence of the heavy computational burden of algorithm SFW. In particular, we obtained the following numbers for the average read latency: (i) $|\mathcal{S}| = 10$, SFW $RL = 1.72s$, APRX-SFW $RL = 1.56s$, (ii) $|\mathcal{S}| = 15$, SFW $RL = 10.72s$, APRX-SFW $RL = 1.67s$, and (iii) $|\mathcal{S}| = 25$, SFW $RL = 45min$, APRX-SFW $RL = 1.23s$. It appears that the latency of algorithm SFW grows exponentially, whereas the latency of APRX-SFW can even improve when using quorum systems with large intersection degree (due to the larger number of fast reads). The exceedingly large delay of SFW in the scenario where $|\mathcal{S}| = 25$, forced us to terminate the simulation prior to its completion. The results presented above were obtained by examining the log files and taking an average of the time over all the completed read operations. We then examine the number of two-round writes. A writer performs two rounds only when the predicate does not hold. Thus, counting the number of two-round writes reveals how many times the predicate does not hold for an algorithm. Below we present the number of two round writes, out of a total 900 writes, that each algorithm performed in two different scenarios: (i) $|\mathcal{S}| = 10$, SFW $\#2comm = 545$, APRX-SFW $\#2comm = 593$, (ii) $|\mathcal{S}| = 15$, SFW $\#2comm = 428$, APRX-SFW $\#2comm = 592$. According to our theoretical findings, algorithm APRX-SFW should allow no more than $\log|\mathcal{S}| \cdot RR$ two-round reads or $\log|\mathcal{S}| \cdot WR$ two-round writes in each scenario, where RR and WR are the number of two-round reads and writes allowed by the algorithm, respectively. Our experimental results are within the theoretical upper bound, illustrating the fact that algorithm APRX-SFW implements a $\log|\mathcal{S}|$-approximation relative to algorithm SFW. These scenarios demonstrate the performance benefit of using algorithm APRX-SFW over algorithm SFW.

Algorithm CWFR ***vs.*** APRX-SFW***:*** We now proceed to compare Algorithm APRX-SFW with the new algorithm CWFR. To examine the impact of computation on the operation latency, we also compare these algorithms to algorithm SIMPLE. This is a standard two-round read and write protocol. Both read and write operations involve a query phase to discover the maximum tag in the system; then the write operation increments the maximum tag and propagates the new tag along with the value to be written to some quorum, whereas the read operation just propagates the maximum tag to some quorum. Note that algorithm SIMPLE requires insignificant computation, and thus the latency of an operation in this algorithm directly reflects four communication delays (i.e., two rounds).

To evaluate the efficiency of the algorithms we use several scenarios. For reasons of space we present only two scenarios (all scenarios can be found in [1]). The first uses a quorum system with a small intersection degree and the second uses a quorum system with a large intersection degree: (i) $|\mathcal{S}| = 10, f = 2$, thus $n = 4$, and (ii) $|\mathcal{S}| = 15, f = 1$, thus $n = 14$. In the scenarios we use



(a) (b) (c)

Fig. 4. 14-wise quorum system ($|\mathcal{S} = 15,\ f = 1$): (a) Percentage of slow reads, (b) Latency of read operations, and (c) Latency of write operations.

10, 20, 40 and 80 readers, combined with 10, 20, and 40 writers respectively. We observe that in all scenarios algorithms APRX-SFW and CWFR exhibit better read and sometimes better write latency than algorithm SIMPLE. This suggests that the additional computation incurred in these two algorithms does not exceed the delay associated with a second communication round. Figures 3 and 4 depict two specific scenarios that we explain further below.

Scenario 1: In this scenario we consider a system with $|\mathcal{S}| = 10$ servers where 2 of them may crash, resulting in a 4-wise quorum system. Using a small intersection degree none of the predicates used in algorithm APRX-SFW can be satisfied. Reads may be fast even if the predicate does not hold. Figure 3 illustrates the run where the number of writers is fixed to 20 in (a) and (b) and the number of readers is fixed to 40 in (c). Observe from Figure 3(a) that algorithm CWFR requires fewer two-round reads than APRX-SFW. For this reason, in Figure 3(b), we observe that the average read latency of CWFR is overall lower. Since the write predicate does not hold when assuming small intersection degree, the three algorithms require all write operations to perform two rounds. The extra computation required by algorithms CWFR and APRX-SFW explains why the write latency of these algorithms is slightly higher than the write latency of algorithm SIMPLE; see Figure 3(c).

Scenario 2: In this scenario we consider a system with $|\mathcal{S}| = 15$ where a single server may crash. This scenario is designed to test the performance of the algorithms when quorum systems with large intersection degree are used. The scenario yields a 14-wise quorum system and contains 15 quorums. Figure 4 depicts the results obtained for a specific run of this scenario where the number of writers is fixed to 20 in (a) and (b) and the number of readers is fixed to 40 in (c). Due to the large intersection degree, algorithm APRX-SFW allows more fast reads than CWFR (see Figure 4(a)). Consequently, as it can be seen in Figure 4(b), algorithm APRX-SFW achieves better read latency than CWFR. Moreover, from Figure 4(c) it can be observed that APRX-SFW allows some write operations to be fast and thus, its average write latency is better than in the other approaches.

It is worth mentioning that from other scenarios we have run (see [1]), we observed that when the intersection degree of the deployed quorum system is of "medium" size, algorithms APRX-SFW and CWFR incur very similar operation latencies.

A general observation is that the performance of algorithm APRX-SFW is affected by both the number of writers and the intersection degree of the underlying quorum system; algorithm CWFR appears to have more stable performance in the scenarios we tested.

VII. CONCLUSIONS

We explored the feasibility of implementing multi-writer atomic registers that enable fast, single round operations. We determined that the only such previously known algorithm incorporates a decision problem that we showed to be *NP-Complete*, making the algorithm not practical. We presented more practical algorithms, one of which uses a log-approximation to speed up its computation. Simulation results illustrate the advantages of our approach. We intend to explore next whether there are specialized quorum constructions that improve the logarithmic approximation factor.

REFERENCES

[1] Technical Report of this work, `http://www.cs.ucy.ac.cy/fastMWMR/MWMR-TR.pdf`.

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

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

[4] R. Clifford and A. Popa. Maximum subset intersection. *Inf. Process. Lett.*, 111:323–325, March 2011.

[5] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. Geoquorums: Implementing atomic memory in mobile ad hoc networks. In *Proc. of 17th Int'l Symp. on Distrib. Comp. (DISC)*, 2003.

[6] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proc. of the 23rd ACM Symp. on Principles of Distr. Computing (PODC)*, pages 236–245, 2004.

[7] B. Englert, C. Georgiou, P. M. Musial, N. Nicolaou, and A. A. Shvartsman. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proc. 13th Int'l Conf. On Principle Of DIstributed Systems (OPODIS)*, pages 240–254, 2009.

[8] R. Fan and N. Lynch. Efficient replication of large data objects. In *Distributed Algorithms*, volume LNCS 2848, pages 75–91, 2003.

[9] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *Proc. of 22nd Int'l Symp. on Distributed Computing (DISC)*, pages 289–304, 2008.

[10] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009.

[11] V. Gramoli, E. Anceaume, and A. Virgillito. SQUARE: scalable quorum-based atomic memory with local reconfiguration. In *Proc. of ACM Symp. on Applied Computing*, pages 574–579, 2007.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Pub., 1996.

[13] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of 16th Int'l Symp. on Distributed Computing (DISC)*, pages 173–190, 2002.

[14] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.
[15] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.