α-Register

David Bonnin, Corentin Travers

α-register[⋆]

David Bonnin and Corentin Travers

LaBRI, University Bordeaux 1, France
name.surname@labri.fr

Abstract. It is well known that in an asynchronous message-passing system, one can emulate an atomic register providing that more than half of the processes are non-faulty. By contrast, when a majority of the processes may fail, simulating atomic register is not possible. This paper investigates weak variants of atomic registers that can be simulated tolerating a majority of processes failures. Specifically, the paper introduces a new class of registers, called α-register and shows how to emulate them. For atomic registers, a read operation returns the last written value when there is no concurrent write operations. α-registers generalize atomic registers in the following sense: In any interval I, at most α values written before I are returned by the read operations in I. A simulation of an α-register tolerating f failures in a n-processes system is presented for $\alpha = 2M-1$, where $M = \max(1, 2f-n+2)$. The simulation is optimal up to a constant multiplicative factor: the paper establishes that α-registers cannot be simulated tolerating f failures if $\alpha \leq M$.

Keywords: Message passing, fault-tolerance, shared-memory simulation.

1 Introduction

Registers A *register* is a basic shared object that allows processes to store and retrieve values. The state of a register consists in a value in some set $\mathcal{V}$; it supports two operation: WRITE(v), that changes its state to v and READ() that returns the value stored in the register. Several consistency conditions have been defined that specify correct responses for READ() operations overlapping concurrent WRITE() operations [22]. In their strongest form, registers are *atomic*: each operation appears to take place instantaneously at some point between its invocation and its response.

Twenty years ago, Attiya, Bar-Noy and Dolev showed that atomic registers can be emulated in asynchronous, crash prone message passing systems provided that a *majority* of the processes do not fail [5]. This fundamental result enables shared-memory algorithms to be automatically implemented in message passing environment. Furthermore, impossibility results and lower bounds established in the shared memory model can directly be translated to message passing. For example, the asynchronous computability theorem that characterizes tasks wait-free solvable in shared memory [21] and its extensions to the t-resilient case [18]

[⋆] This work is supported in part by the ANR project DISPLEXITY.

apply as well to the asynchronous message passing model with a majority of non-faulty processes.

Beyond the majority barrier A key ingredient of the simulation of registers in message passing is a *quorum system*, that is a collection of sets of processes such that any two sets intersect. In Attiya, Bar-Noy and Dolev protocol (*ABD protocol* [5]), a quorum is any set of $n-f$ processes, where n is the total number of processes in the system and $f < \frac{n}{2}$ an upper bound on the number of failures. Intuitively, a WRITE(v) involves communicating v to a quorum while a READ() returns the most recent value in a quorum. By the intersection property, some process participate in both operations, allowing the READ() to return an up to date value. Quorums defined as set of $n - f$ processes are *live*, in the sense that any process can broadcast a request and eventually receives replies from $n - f$ processes. However, if less than a majority of the processes are non-faulty, i.e. $f \geq \frac{n}{2}$, contacting $n - f$ processes in a READ() operation may not ensure that the value returned by that operation is up to date. Indeed, simulating atomic registers while tolerating $f \geq \frac{n}{2}$ failures in asynchronous message passing is not possible [5].

A few approaches has been proposed to circumvent this impossibility. Probabilistic quorums systems allow two quorums to be non-intersecting with some small probability [1,16,23], leading to a small probability that READ() operations return stall values. Dynamic atomic storage systems, such as RAMBO [20] and DynaStore [3] emulate atomic registers in dynamic environments. They support a reconfiguration operation for adding or removing processes. Reconfiguration may thus be used to replace failed processes. However, failures are typically assumed to be limited when reconfigurations take place. The approaches [17,25] are also based on stronger model assumptions.

Another approach consists in relaxing the consistency guarantees of the implementation. *Eventual consistency* [15,24] essentially only requires that if finitely many WRITE() operations are performed, eventually every READ() operation returns the last written value. When availability is a primary concern, eventually consistent services has been implemented and deployed for large-scale, geo-replicated systems (e.g.,[10,13]). In this settings, network partitions may occur but operation must complete even in the case of such events.

The question addressed in the paper The paper investigates the following question:

> *Given n and $\frac{n}{2} \leq f < n$, what type of (weak) register can be simulated in an n-processes asynchronous message passing system tolerating f failures?*

By the ABD emulation, shared memory may be seen as an high-level language to design message passing algorithms tolerating a minority of failures. The question above thus amounts to finding an equivalent high level construct for the case in which a majority of the processes may fail.

Moreover, recently, non-trivial asynchronous algorithms for k-set agreement and k-parallel consensus[1] that tolerate $f(k) \geq \{\frac{n}{2}, k\}$ failures have been designed for message passing systems [8,9]. While the liveness of these algorithms depends on some additional assumption (such as, e.g., an eventual, non-faulty leader), the safety part relies solely on the bound $f(k)$ on the number of failures. As $f(k) \geq \frac{n}{2}$, the existence of those algorithms cannot be inferred from shared memory results. Identifying weak types of registers, that one can simulate when a majority of the processes could fail, might help understanding what can be computed in such systems.

Contributions of the paper The paper introduces a new type of registers, called *α-register* and shows an implementation in message passing systems tolerating a majority of faulty processes. Implementations of α-registers are required to be available, that is any WRITE() or READ() request must eventually complete, and partition-tolerant. Indeed, in an asynchronous system in which $f \geq \frac{n}{2}$ processes may fail, processes can be partitioned in two or more sets of at least $n - f$ processes, and messages exchanged between partitions may be arbitrarily delayed. Hence, according to the CAP theorem ([19], Corollary 1.1), it is unavoidable that some READ() operations return outdated values. The parameter α specifies how many distinct outdated values can be read in any interval I, that is values that have been written before I. When $\alpha = 1$, the definition boils down to atomic register.

In more detail, the contribution of the paper is threefold: (1) it introduces α-registers, a new type of register that generalizes atomic registers (Section 2); (2) for $f \geq \frac{n}{2}$ and $M = 2f - n + 2$, it presents a f-resilient message passing implementation of a single-writer multi-reader α-register with $\alpha = 2M - 1$ (Section 3); (3) finally, the paper establishes a lower bound linking f, n and α, namely there is no n-processes, f-resilient implementation of an α-register for $\alpha \leq M$ (Section 4). This lower bound implies that our α-register implementation is within an additive term of at most $\frac{\alpha}{4}$ of the maximal number of failures that can be tolerated.

2 Computational Model and Definition of α-Registers

Message passing asynchronous distributed system We consider a distributed system made of a set Π of n asynchronous processes $\{p_1, \ldots, p_n\}$, as described in e.g. [6,11]. Each process runs at its own speed, independently of the other processes.

Processes communicate by sending and receiving messages over a reliable but asynchronous network. Each pair of processes $\{p_i, p_j\}$ is connected by a bidirectional channel. Channels are reliable and asynchronous, meaning that each

[1] k-set agreement [12] and k-parallel consensus [2] generalize the consensus problem. In k-set agreement, at most k distinct values may be decided. k-parallel consensus consider k instances of consensus and requires each non-faulty process to decide in at least one of them.

message sent by p_i to p_j is received by p_j after some finite, but unknown, time; there is no global upper bound on message transfer delays. The algorithm in Section 3 assumes *FIFO* channels, that is for any pair of processes p_i, p_j, the order in which the messages sent by p_i to p_j are received is the same as the order in which they are sent.

The system is equipped with a global clock whose ticks range $\mathbb{T}$ is the set of positive integers. This clock is not available to the processes, it is used from an external point of view to state and prove properties about executions.

In a *step*, a process may send a message to some other process, performs arbitrary local computation and receives a message that has been previously sent to it but has not been already received. An *execution* is a possibly infinite sequence of steps. Processes may fail by *crashing.* A process that crashes prematurely halts and never recovers. In an execution, a process is *faulty* if it fails and *correct* otherwise. f denote an upper bound on the maximal number of processes that may fail.

Definition of α-registers As classical read/write registers, an α-register supports two operations: $\textsc{write}(v)$, where v is a value taken from some set $\mathcal{V}$, and $\textsc{read}()$. A $\textsc{write}(v)$ operation returns an acknowledgment *ok* and a $\textsc{read}()$ returns a value $u \in \mathcal{V} \cup \{\perp\}$ where u is the input of a $\textsc{write}()$ operation or the initial value $\perp$ of the α-register. In an *admissible execution*, no process starts a $\textsc{write}(v)$ or $\textsc{read}()$ operation while its previous operation, if any, has not returned. The *execution interval* $I(op)$ of an operation instance op by process p begins when p calls $\textsc{write}()$ or $\textsc{read}()$ and ends when p returns from that call; if p never returns, $I(op)$ has no end. We sometimes simply say operation instead of operation instance. Two operations op_1 and op_2 are *concurrent* if $I(op_1) \cap I(op_2) \neq \emptyset$. A terminating operation op_1 *precedes* operation op_2 if $I(op_1) \cap I(op_2) = \emptyset$ and $I(op_1)$ ends before $I(op_2)$ begins. An operation op is *active* in an interval I if $I \cap I(op) \neq \emptyset$. To simplify the exposition, we assume without loss of generality that no two distinct $\textsc{write}()$ operations have the same input value[2].

In any admissible execution e, a α-register satisfies the following properties.

1. *Termination.* Any $\textsc{read}()$ or $\textsc{write}(v)$ operation performed by a correct process terminates.
2. *Non-spurious value.* For any terminating $\textsc{read}()$ operation R that returns u, either $u = \perp$ or there exists a $\textsc{write}(u)$ operation that precedes or is concurrent with R.
3. *Chronological read.* Let R, R' be two terminating $\textsc{read}()$ operations performed by the same process in that order and let u, u' be the values returned. If $u \neq \perp$, then $u' \neq \perp$ and $\textsc{write}(u)$ precedes or is concurrent with $\textsc{write}(u')$.
4. *Non-triviality.* Let R be a $\textsc{read}()$ operation by process p and let u be the value returned by R. If there is a $\textsc{write}()$ operation by p that precedes R, $u \neq \perp$. Moreover, if W is the last $\textsc{write}()$ operation by p that precedes

[2] This can be enforced by appending a sequence number and the id of the writer to each value to be written.

R, WRITE(u) is either W or a WRITE() operation by another process that is concurrent with or is preceded by W.

5. *Propagation.* Let u be the input of a terminating WRITE() or the value returned by a READ() performed by a correct process. Eventually, for every terminating READ() operation R' with return value u' either $u = u'$ or WRITE(u) is concurrent with or precedes WRITE(u').
6. α-*Bounded reads.* In any interval I, the set of values that have been written by WRITE() operations that terminate before I and returned by the READ() operations whose execution interval is contained in I is of size at most α.

The termination property implies that an α-register is always available. In particular, any READ() operation by a non-faulty process always returns a value. The properties chronological read and non-triviality express consistency requirements in the context of a single process. Chronological read requires that successive READ() by the same process p_i do not return older values. Non-triviality intuitively requires that p_i "sees" its writes. After a WRITE(u) operation, every subsequent READ() by p_i returns a value as least as recent as u. The propagation properties implies that α-register are eventually consistent. If after some time no WRITE() operations are performed, eventually every READ() operation returns the last value written.

Since $f \geq \frac{n}{2}$, it can be shown by a partition argument that READ() may return arbitrary old values. Consider two sets Q_1, Q_2 of $n - f$ processes that do not intersect and suppose that every process not in $Q_1 \cup Q_2$ initially fails. As communication is asynchronous, messages exchanged between Q_1 and Q_2 may be delayed during an arbitrary long interval I. For some process $p_i \in Q_i, 1 \leq i \leq 2$, the operations by p_i may thus return after messages have been exchanged only with the processes in Q_i (this is indistinguishable for p_i from an execution in which every process not in Q_i fail before I.). Therefore, if p_1 performs WRITE() operations, the values it writes are not seen by p_2. Thus READ() operations by p_2 may return values that have written before any WRITE() operations by p_2.

Rather that bounding the staleness of values returned by READ() operation, which is impossible if asynchrony and a majority of failures have to be tolerated, the α-bounded read property imposes that not too many stale values, namely no more that α, are returned by READ() operations.

3 Single-writer Multiple-reader α-register

This section presents a protocol (Algorithm 3.1) that implements a single-writer multiple-readers (SWMR) α-register in an asynchronous system in which up to $f \leq n - 1$ processes may fail. The value of α depends on the number of failures the protocol tolerates, namely $\alpha = 2M - 1$, where $M = 2f - n + 2$ if $f \geq \frac{n}{2}$ and $M = 1$ otherwise. The algorithm assumes that channels are FIFO.

The algorithm is similar to the ABD protocol [5]. Each time a new value is written it is first associated with a unique timestamp (line 7). Timestamps are increasing so that more recent values get larger timestamps. As there is a single writer, no two values are associated with the same timestamp. Each process p_i

Algorithm 3.1 SWMR α-register (code for process p_i)

```
1: INITIALIZATION
2:     $seq_i \leftarrow 1$; $\langle v_i, ts_i \rangle \leftarrow \langle \perp, 0 \rangle$; $\langle vr_i, tsr_i \rangle \leftarrow \langle \perp, 0 \rangle$;
3:     $Qr_i \leftarrow \emptyset$; $Qe_i \leftarrow \emptyset$; $Qw_i \leftarrow \emptyset$;
4:     $Accept_i[1..n] \leftarrow [2, \ldots, 2]$        ▷ array of $n$ integers initialized to 2
5:     for each $p_j : 1 \leq j \leq n$ do send UPDATE($seq_i$, $\langle v_i, ts_i \rangle$, 0)
6: function WRITE($v$)
7:     $\langle v_i, ts_i \rangle \leftarrow \langle v, ts_i + 1 \rangle$; $seq_i \leftarrow seq_i + 1$; $Qw_i \leftarrow \emptyset$;
8:     wait until $|Qw_i| \geq n - f$;
9:     return ok
10: function READ( )
11:     $n\_iter \leftarrow 0$;
12:     repeat
13:         $\langle vr_i, tsr_i \rangle \leftarrow \langle v_i, ts_i \rangle$; $seq_i \leftarrow seq_i + 1$; $Qr_i \leftarrow \emptyset$; $Qe_i \leftarrow \emptyset$;
14:         wait until $|Qr_i \cup Qe_i| \geq n - f$;
15:         $n\_iter \leftarrow n\_iter + 1$
16:     until ($|Qe_i| \geq n - f$) or ($n\_iter \geq N$)        ▷ $N = 2(2f+1)(\lfloor \frac{n}{n-f} \rfloor + 1) + 1$
17:     return $vr_i$
18: WHEN UPDATE($seq$, $\langle v, ts \rangle$, $old\_seq$) FROM PROCESS $p_j$ IS RECEIVED
19:     if $old\_seq = seq_i$ then
20:         if $ts = ts_i$ then $Qw_i \leftarrow Qw_i \cup \{p_j\}$
21:         if $ts > tsr_i$ then $Qr_i \leftarrow Qr_i \cup \{p_j\}$
22:         if $ts = tsr_i$ then $Qe_i \leftarrow Qe_i \cup \{p_j\}$
23:     if $ts > ts_i$ then
24:         if $Accept_i[j] > 0$ then $Accept_i[j] \leftarrow Accept_i[j] - 1$
25:         else $\langle v_i, ts_i \rangle \leftarrow \langle v, ts \rangle$; $Accept_i[1..n] \leftarrow [2, \ldots, 2]$        ▷ $Accept_i[j] = 0$
26:     send UPDATE($seq_i$, $\langle v_i, ts_i \rangle$, $seq$) to $p_j$
```

maintains a pair of local variables $\langle v_i, ts_i \rangle$ which store the most recent value p_i knows of together with its timestamp. We say that p_i *accepts* a pair $\langle v, t \rangle$ when p_i changes $\langle v_i, ts_i \rangle$ to $\langle v, t \rangle$ (line 25).

Processes constantly exchange messages of type UPDATE that contain the most recent value and its timestamp known by the message's sender[3]. For any pair of processes p_i, p_j, UPDATE messages are exchanged between p_i and p_j following a "ping-pong" pattern. Initially, p_i and p_j send UPDATE messages to each other, and each time p_i (resp. p_j) receives UPDATE from p_i (resp. p_j), it replies with an UPDATE message. Each such message contains a triple $(sq, \langle v, ts \rangle, osq)$, where sq and osq are sequence numbers, and $\langle v, ts \rangle$ is the current value and timestamp of the process sending the message. p_i maintains a sequence number that is incremented each time p_i starts a new WRITE() operation or a new phase in a READ() operation (see below). If message m =

[3] The algorithm is not quiescent: processes keep sending and receiving messages even if no WRITE() or READ() operations are performed. The algorithm can be made quiescent at the price of an increasing complexity in the pseudo-code. We choose to ignore this issue to keep the pseudo-code simple.

UPDATE$(sq, \langle v, ts\rangle, osq)$ is sent by p_i in reply to a message UPDATE$(sq', \langle v', ts'\rangle, osq')$ from p_j (see line 18–line 26), then $osq = sq'$ and sq is the current sequence number of p_i. Thus, by comparing osq with its current sequence number, process p_j can determine whether m is related to its current operation or to a previous operation.

Write() operations The implementation of WRITE(v) is similar to the implementation in the ABD protocol. After a new timestamp t has been associated with v on line 7, the writer p_n changes its local variable $\langle v_n, ts_n\rangle$ to $\langle v, t\rangle$. It then waits until each process in a quorum of $(n - f)$ processes have accepted $\langle v, t\rangle$, and the operation then returns (line 8–line 9). In more detail, the local variable Qw_n, intended to contain a set of processes ids, is emptied at the beginning of the operation (line 7). Then, each time, a message UPDATE containing the pair $\langle v, t\rangle$ from a process p_j is received, p_j is added to Qw_n (line 20). The operation returns when $|Q_w| \geq n - f$.

The new pair $\langle v, t\rangle$ is disseminated by the UPDATE messages sent by the writer: once $\langle v_n, ts_n\rangle$ has been changed to $\langle v, t\rangle$, and until a new WRITE() operation is initiated, every UPDATE sent by p_n contains $\langle v, t\rangle$.

Value dissemination As newly written values are propagated asynchronously, at any point in time there might be pending UPDATE that have been sent to p_i by the processes in some set S, but not yet received by p_i. Each of these messages may contain a distinct pair $\langle \text{value}, \text{timestamp}\rangle$ from some set $\{\langle w_1, t_1\rangle, \ldots, \langle w_m, t_m\rangle\}$. If $\langle v_i, ts_i\rangle$ changes each time p_i receives a newer value, the successive values of v_i might be $w_1, \ldots, w_m$. Furthermore, if the same happens at each process in set of size at least $n - f$, it could be the case that each value $w_1, \ldots, w_m$ is returned by a READ() operation. Instead, to avoid that READ() operations return many old values, we ensure that if $\langle v_i, ts_i\rangle$ changes from $\langle w, t\rangle$ to $\langle w', t'\rangle$, some process p_j stores $\langle w', t'\rangle$ after $\langle v_i, ts_i\rangle$ is set to $\langle w, t\rangle$ and before it is changed to $\langle w', t'\rangle$.

Due to the "ping-pong" pattern followed by messages exchanged between processes p_i and p_j, there are at most two messages that have been sent by p_j but have not yet been received by p_i at any point in time. Hence, if p_i receives three UPDATE from p_j in some interval I, the last one of these messages has been sent during I. The array $Accept_i$ is used to keep track of how many consecutive messages from the same process carrying new values have been received. Initially, $Accept_i[j] = 2$ and at any time, $Accept_i[j] \in \{0, 1, 2\}$ for any $j, 1 \leq j \leq n$. $Accept_i[j]$ is decremented each time p_i receives an UPDATE from p_j (line 24) carrying a value newer than p_i's current value. The array is also reset to $[2, \ldots, 2]$ when $\langle v_i, ts_i\rangle$ changes (line 25). Hence, $Accept_i[j] < 2$ means that p_i knows that its current value is outdated by p_j's current value (line 23 – line 24). If UPDATE$(*, \langle w, t\rangle, *)$ where $\langle w, t\rangle$ is newer than $\langle v_i, ts_i\rangle$ is received from p_j when $Accept_i[j] = 0$, $\langle v_i, ts_i\rangle$ is changed to $\langle w, t\rangle$ (line 25).

Suppose that at some point a set X of messages have not been yet received by p_i. Note that if, when some message $m \in X$ is received, $\langle v_i, ts_i\rangle$ changes, then no other message in X updates $\langle v_i, ts_i\rangle$. This is because $Accept_i$ is reset to $[2, \ldots, 2]$ each time $\langle v_i, ts_i\rangle$ changes, messages are received in FIFO order and at any point time and for any process p_j, no more than two messages sent by p_j have not been received by p_i.

Read() operations A READ() operation (line 11–line 17) by process p_i consists in up to $N = O(\frac{fn}{n-f})$ iterations. Each iteration is identified by an increasing sequence number seq_i. At the beginning of iteration s, the pair $\langle v, ts\rangle$ currently hold by p_i is stored in $\langle vr_i, tsr_i\rangle$ and the two sets Qe_i and Qr_i, intended to contain processes that hold a pair equal to or more recent than, respectively, $\langle vr_i, tsr_i\rangle$ are emptied (line 13). An iteration terminates when p_i knows that at least $n - f$ processes store values as least as recent than vr_i, i.e., when $|Qr_i \cup Qe_i| \geq n-f$. The READ() operation terminates (1) immediately if $|Qe_i| \geq n-f$, i.e., for each process $p_j \in Qe_i$, there is a time at which $v_j = vr_i$ or (2) after N iterations have been performed. The value returned is then vr_i, the value of v_i at the beginning of the last iteration. We show in the proof that for every READ() operation *op* that returns a value w written before the operation starts, the operation terminates by condition (1) above. That is, each process p_j in a set Q of size $n - f$ stores v at some point during the interval of *op*. Intuitively, in each iteration for which condition (1) is not satisfied, p_i learns a newer value. This value is propagated to at least $n - f$ processes in the following iterations. Since the number of values that have been written before *op* starts and that can be learned and propagated is bounded by a function of f and n, every process knows the last value written before *op* starts after some constant number of iterations or new values are written concurrently with *op*.

Consider an interval I. Let w_ℓ be the value written by the last WRITE() that terminates before I. When WRITE(w_ℓ) returns, each process in a set Q_ℓ of size $n - f$ stores w_ℓ. Since a process can only replace its value with a newer one, the value stored by any process of Q_ℓ at any point in I is w_ℓ or a more recent value. Therefore, any value older than w_ℓ present in the system at the beginning of I is stored or contained in a message not yet received by a processes of $\Pi \setminus Q_\ell$. Let L, T be respectively the values stored by the processes of $\Pi \setminus Q_\ell$ and the values contained in the messages not yet received by the processes of $\Pi \setminus Q_\ell$.

In the worst case, $|L| = f$. The protocol ensures that if $v \in L$ is returned by a READ() performed during I, then at least $n - f$ processes stores v at some point during I. Since no process changes its value for an older one, the $n - f - 1$ oldest values in L cannot be returned by READ() operations. Hence, at most $f - (n - f - 1) = 2f - n + 1 = M - 1$ distinct values of L can be read in I.

As previously explained, for each process $p_i \in \Pi \setminus Q_\ell$, at most one message not yet received by p_i at the beginning of I may change the value v_i stored by p_i. Therefore, at most f values of T may be stored by the processes and thus be returned by READ() operations. As in the case of the values of L, the $n - f - 1$ oldest values cannot be returned by READ() operations. Therefore, at most $f - (n - f - 1) = 2f - n + 1 = M - 1$ distinct values of T can be read in I. In addition, w_ℓ may be read in I. It thus follows that the number of values written before I and returned by READ() operations during this interval is at most $2(M - 1) + 1 = 2M - 1 = \alpha$.

3.1 Proof of the protocol

We consider an arbitrary infinite admissible execution α in which the unique writer is the process p_n. var_i denote the local variable var of process p_i and var_i^τ its value at time τ. Due to space constraints, some proofs are omitted. They can be found in [7].

Whenever the writer initiates a new WRITE() operation, it increases a counter whose value ts is assigned as a timestamp to the value v being written (line 7). This timestamp is unique and no other timestamp is ever associated to v. That is, for any process p_i, whenever the local variable v_i is changed to v, ts_i is changed accordingly to the timestamp ts associated with v (line 25). Values can thus be totally ordered according to their timestamp. In particular we say that value v is *newer* than or *more recent* than value v' is the timestamp ts assigned to v is larger than or equal to the timestamp ts' assigned to v' and we note $\langle v', ts' \rangle \preceq \langle v, ts \rangle$. Note that, for any process p_i, whenever the pair $\langle v_i, ts_i \rangle$ is modified (line 7 or line 25), it is replaced by a more recent value. That is,

Observation 1. *For every process p_i, and every times $\tau < \tau'$, $\langle v_i^\tau, ts_i^\tau \rangle \preceq \langle v_i^{\tau'}, ts_i^{\tau'} \rangle$.*

Each time a process p_i receives a message UPDATE from a process p_j, it sends back an UPDATE to process p_j (line 18 and line 26). Moreover, initially each process sends an UPDATE message to every process (line 5). It thus follows that messages UPDATE are perpetually exchanged between p_i and p_j if both processes are correct:

Lemma 1. *Let p_i, p_j be two correct processes. p_i receives infinitely many messages UPDATE from p_j.*

Proof. Initially, p_i sends an UPDATE message to p_j (initialization, line 5). By the code (line 18 and line 26), each time a correct process p receives a message UPDATE from a process q, p sends a message UPDATE to q. Since p_i and p_j are two correct processes, p_i receives infinitely many messages UPDATE from p_j. □

Next Lemma shows that whenever a correct process learns a new value, every other correct eventually learn that value or a more recent one. It forms the basis to show that READ() (Lemma 3) and WRITE() (Lemma 4) operations performed by correct processes terminate.

Lemma 2. *Let p_i, p_j be two correct processes. If at some time, $\langle v_i, ts_i \rangle = \langle v, ts \rangle \neq \langle \perp, 0 \rangle$, then eventually $\langle v, ts \rangle \preceq \langle v_j, ts_j \rangle$.*

Lemma 3. *Let p_i be a correct process. Every invocation of* READ*() by p_i returns.*

Lemma 4. *Assume that the writer p_n is a correct process. Every invocation of* WRITE*() by p_n returns.*

Proof of the bounded reads property Let $\mathcal{I}$ be an arbitrary interval. Let $\mathcal{R} = \{R_1, \ldots, R_m\}$ be a set of READ() operations whose execution intervals are contained in $\mathcal{I}$. Let w_i be the value returned by operation R_i and let $\mathcal{V}_R = \{w_1, \ldots, w_m\}$.

The remainder of this section is devoted to the proof of the following Lemma:

Lemma 5. $|\mathcal{V}_R \setminus \mathcal{V}_W| \leq 2M - 1$ *where* $M = \max(1, 2f - n + 2)$*, where* $\mathcal{V}_W$ *is the set of values written during* $\mathcal{I}$*.*

The values returned by each READ() of $\mathcal{R}$ is a value that is either written by one of the Write operation active during $\mathcal{I}$, or a value present in the system at the beginning of $\mathcal{I}$. A value v with timestamp ts is present in the system at the beginning of $\mathcal{I}$ if it is the value locally stored by a process, i.e., $\langle v, ts\rangle = \langle v_i, ts_i\rangle$ for some process p_i or $\langle v, ts\rangle$ is carried by a message UPDATE that has not yet been delivered. Let τ_b be the time at which $\mathcal{I}$ starts. We define:

- $\mathcal{V}_W$ the set of the values written during $\mathcal{I}$. That is, $w \in \mathcal{V}_W$ if $I(\text{WRITE}(w)) \cap \mathcal{I} \neq \emptyset$.
- $Inc_{j\to i}$ as the set of incoming messages UPDATE that have been sent by p_j but has not yet been received by p_i by time τ_b ;
- $\mathcal{V}_L = \{v : v \notin \mathcal{V}_W \text{ and } \exists p_i, \langle v_i, ts_i\rangle = \langle v, ts\rangle \text{ at time } \tau_b\ \}$;
- $\mathcal{V}_I = \{v : v \notin \mathcal{V}_L \cup \mathcal{V}_W \text{ and } \exists p_i, p_j, \text{UPDATE}(*, \langle v, ts\rangle, *) \in Inc_{j\to i}\}$;
- $vlast$, the value written by the last WRITE() operation that precedes $\mathcal{I}$.

That is, $\mathcal{V}_L$ is the set of values locally stored by the processes at the beginning of $\mathcal{I}$, while $\mathcal{V}_I$ is the set of values that are not locally stored, but part of the content of some messages still in transit. Note that $\mathcal{V}_R \setminus \mathcal{V}_W \subseteq \mathcal{V}_L \cup \mathcal{V}_I$.

We first observe that $|\mathcal{V}_L| \leq f + 1$ (Corollary 1). Essentially, this follows from the fact that a quorum $Qlast$ of at least $n - f$ processes must have accepted the value written by the last WRITE() operation $Wlast$ preceding $\mathcal{I}$ in order for that operation to return (Lemma 6). As a process replace the value it stores locally only with a more recent one (Observation 1), the value locally stored by each process $p_i \in Qlast$ is newer than or equal to $vlast$, at any time in $\mathcal{I}$. In other words, the value stored by p_i during $\mathcal{I}$ belongs to $\{vlast\} \cup \mathcal{V}_W$.

Lemma 6. *Let* $vlast$ *be the value written by the last* WRITE*() operation preceding* $\mathcal{I}$ *and let* $tslast$ *denote the timestamp associated with it. There is a set* $Qlast$ *of at least* $n - f$ *processes such that at any time in* $\mathcal{I}$, $v_i \in \{vlast\} \cup \mathcal{V}_W$ *and* $ts_i \geq tslast$*.*

Proof. Let $Wlast$ be the last WRITE() that precedes $\mathcal{I}$. When this operation returns, p_n has received a message UPDATE$(*, \langle vlast, tslast\rangle, *)$ from each process in a set Q of size at least $n - f$ (line 8). This means that for each $p_i \in Q$, we have at some time $\langle v_i, ts_i\rangle = \langle vlast, tslast\rangle$ (line 26).

Moreover, by the code each value v written by the single writer p_n before $vlast$ is associated with a timestamp strictly smaller than $tslast$. As each time the value stored locally (in v_i for process p_i) is modified, it is replaced by a more recent value, i.e., a value associated with a larger timestamp (Observation 1), it follows from the fact that $Wlast$ is the last write operation preceding $\mathcal{I}$ that for every process $p_j \in Q$, $ts_j \geq tslast$ and $v_j \in \{vlast\} \cup \mathcal{V}_W$ at any time in $\mathcal{I}$. □

Corollary 1. $|\mathcal{V}_L| \leq f + 1$

Proof. By Lemma 6, at the beginning of $\mathcal{I}$, for each process $p_i \in Qlast$, $v_i \in \{vlast\} \cup \mathcal{V}_W$. As $\mathcal{V}_W \cap \mathcal{V}_L = \emptyset$ and as $|Qlast| \geq n - f$, $|\mathcal{V}_L| \leq f + 1$. □

Consider two processes p_j and p_i. By the code p_j sends a message UPDATE to p_i each time it receives a message UPDATE from that process (line 18–line 26). Since initially both p_i and p_j send a message UPDATE to each other, it follows that at any time at most two messages UPDATE have been sent by p_j to p_i and has not yet been received by the latter:

Observation 2. *For every pair of processes* p_i, p_j, $|Inc_{j \to i}| \leq 2$.

Let $\mathcal{U} \subseteq \mathcal{V}_I$ be the set of values that are not locally stored by any process at the beginning of $\mathcal{I}$, but later stored by at least one process at some time in $\mathcal{I}$. That is,

$$u \in \mathcal{U} \iff u \in \mathcal{V}_I \text{ and at some time in } \mathcal{I}\ , v_i = u \text{ for some process } p_i$$

We upper-bound the size of $\mathcal{U}$ (Lemma 7) by f. This upper bound is a key ingredient in establishing that, for any value v read during $\mathcal{I}$, there is set of at least $(n - f)$ processes p_j that hold v at some point in $\mathcal{I}$, that is $v_j = v$ at some time in $\mathcal{I}$ (Lemma 8).

Lemma 7. $|\mathcal{U}| \leq f$

Proof. Let $u \in \mathcal{V}_I$ be a value and let ts denote the timestamp associated with it. Suppose that at some time τ in $\mathcal{I}$, $\langle v_x, ts_x \rangle = \langle u, ts \rangle$ for some process p_x. At the beginning of $\mathcal{I}$, no process stores locally u (for every process p_j, $v_j \neq u$) but a message UPDATE whose content contains u has been sent to some process but has not yet been received by that process.

Let p_i be the first process that, during $\mathcal{I}$ change its pair ⟨local value, timestamp⟩ to $\langle u, ts \rangle$ (at line 25). Since $u \notin \mathcal{V}_L$, this occurs when p_i receives a message $m =$ UPDATE$(*, \langle u, ts \rangle, *)$ sent to it before $\mathcal{I}$, that is there exists a process p_j such that $m \in Inc_{j \to i}$.

Consider another value $u' \neq u$ that similarly to u (1) is contained in $\mathcal{V}_I$ and (2) at some time τ' in $\mathcal{I}$ is stored locally by some process $p_{x'}$ (i.e., at time τ', $v_{x'} = u'$). Let $p_{i'}$ be the first process that changes during $\mathcal{I}$ its local value $v_{i'}$ to u'. As explained above, this occurs when $p_{i'}$ receives a message $m' \in Inc_{j' \to i'}$.

Suppose for contradiction that $p_i = p_i'$. Assume without loss of generality that p_i first changes v_i to u and then later to u'. By the code, immediately after $\langle v_i, ts_i \rangle$ has been modified, the array $Accept_i$ is reset to $[2, \ldots, 2]$ (line 25). As the channels are FIFO, any message received from $p_{j'}$ by p_i during $\mathcal{I}$ and before m' is received are contained in $Inc_{j' \to i}$. Hence, after m has been received and before the reception of m', p_i has received at most $|Inc_{j' \to i}| - 1$ messages from $p_{j'}$. As $|Inc_{j' \to i}| \leq 2$ (Observation 2), it thus follows that $Accept_i[j'] > 0$ when m' is received by p_i (Recall that the counter $Accept_i[j']$ is decremented at most once each time a message from $p_{j'}$ is received line 23–line 24). Therefore (line 24–line 25), v_i remains unchanged when m' is received: a contradiction.

Finally, note that neither p_i nor $p_{i'}$ are contained in $Qlast$ since for each process p_j in this set, v_j is $vlast$ or a more recent value (Lemma 6). As $|Qlast| \geq n-f$, we conclude that $|\mathcal{U}| \leq f$. □

Lemma 8. *Let $R \in \mathcal{R}$ be a* READ*() operation and let v be the value returned by R. Either $v \in \mathcal{V}_W$, or there is a set Q_R of at least $n-f$ processes such that for each $p_i \in Q_R$, there is a time in $I(R)$ at which $v_i = v$.*

Finally, we bound the number of values that are read and, on one hand, stored by at least one process (Lemma 9), or, on the other hand, only contained in messages that have not yet been delivered at the beginning of $\mathcal{I}$ (Lemma 10). As any old value (i.e., a value not in $\mathcal{V}_W$) that is read during $\mathcal{I}$ is either stored locally by some process or contains in message not yet delivered at the beginning of $\mathcal{I}$, the bound on the number of values read during $\mathcal{I}$ follows.

Lemma 9. $|\mathcal{V}_L \cap \mathcal{V}_R| \leq M = 2f - n + 2$

Lemma 10. $|\mathcal{V}_I \cap \mathcal{V}_R| \leq M - 1 = 2f - n + 1$

Proof of Lemma 5. Any value that is returned by a READ() operation during $\mathcal{I}$ is either contained in $\mathcal{V}_L$ or $\mathcal{V}_I$ or $\mathcal{V}_W$. As $|\mathcal{V}_L \cap \mathcal{V}_R| \leq 2f-n+2$ (Lemma 9), and $|\mathcal{V}_I \cap \mathcal{V}_R| \leq 2f-n+1$ (Lemma 10), $|\mathcal{V}_R \setminus \mathcal{V}_W| = |\mathcal{V}_R \cap (\mathcal{V}_L \cup \mathcal{V}_I)| \leq (2f-n+2)+(2f-n+1) = 2M-1$ □

Finally, the correctness of Algorithm 3.1 is implied by the following theorem.

Theorem 3. *Algorithm 3.1 implements a SWMR $(2M-1)$-register, where $M = 2f-n+2$.*

Proof. Consider an admissible execution of Algorithm 3.1. The *termination* property of α-registers immediately follows from Lemma 3 and Lemma 4. For the *Non-spurious value*, the value returned by a READ() operation by process p_i is the value of the variable v_i at some time in the execution interval of the operation. At any point in the execution, v_i stores $\perp$ or a value that has been introduced by the writer.

For the *Chronological read* property, consider u, u' two values returned in that order by READ() operations performed by the same process and let t, t' be the timestamp associated with u, u' respectively. By Observation 1, $\langle u, t\rangle \preceq \langle u', t'\rangle$. Henceforth, WRITE(u) precedes WRITE(u') since there is a single writer. The *Non-triviality* property immediately follows from Observation 1: For the single writer, every READ() operation returns the input of its last preceding WRITE() or $\perp$ if there is no preceding WRITE().

To see why the *propagation* property is satisfied, let u be the input of a terminating WRITE() or the value returned by a READ() performed by a correct process p_i and let t denote its timestamp. By the code, at some point $\langle v_i, ts_i\rangle = \langle u, t\rangle$. Then, by Lemma 2, for every non-faulty process p_j, eventually $\langle u, t\rangle \preceq \langle v_j, ts_j\rangle$. Hence, eventually every value u' returned by READ() operations is either u or a value written after u. Finally, the *α-Bounded reads* property with $\alpha = 2M-1$ follows immediately from Lemma 5. □

4 Lower bound

This section presents a lower bound on α for any implementation of an α-register. More precisely, it proves the following theorem:

Theorem 4. *Let n, f such that $f \geq \frac{n}{2}$. For any implementation of a SWMR α-register for n processes that tolerates f failures, $\alpha \geq M$.*

Proof. (Sketch) Without loss of generality, assume that A is a full information f-resilient protocol that implements a SWMR α-register. That is, the state of each process consists in its initial state and all its history and each time a process sends a message, it sends its entire state. The single writer is the process p_n.

We construct a family of executions of A. Each execution is parametrized by M integers $k_1, \ldots, k_M$. We show that for some values of $k_1, \ldots, k_M$, M distinct values are returned by READ() operations in an interval in which no WRITE() operation is active. Each execution is divided into two phases, each phase consisting in M sequential rounds.

Recall that $M = 2f - n + 2$. Let $\mathcal{V} = \{v_1, \ldots, v_M\}$ be a set of M distinct values. $k = (k_1, \ldots, k_M)$ is a M-tuple of positive integers. For $i, 1 \leq i \leq M$, let Q_i, Q'_i denote the sets of $n - f$ processes $Q_i = \{p_i\} \cup \{p_{f+2}, \ldots, p_n\}$ and $Q'_i = \{p_i\} \cup \{p_{M+1}, \ldots, p_{M+(n-f)-1}\}$, respectively. Observe that $Q_i \cap Q'_i = \{p_i\}$ since $M + (n - f) - 1 = f + 1$. Execution $\mathcal{E}_k$ is defined as follows:
First phase. This phase consists in M rounds $r_1, \ldots, r_n$. For each $i, 1 \leq i < M$, round r_{i+1} begins after the end of round r_i. Only processes in Q_i take steps in round r_i. We first let every message that have been sent to processes in Q_i during the previous rounds (if any) to be received. Then, $k_i + 2$ operations on the α-register implemented by A are performed sequentially, in that order:

1. WRITE(v_i) is performed by process p_n;
2. Process p_n performs READ();
3. k_i READ() operations are performed by process p_i.

The messages sent to processes $p_j \notin Q_i$ are delayed until some time specified later. As $|Q_i| = n - f$, the execution is indistinguishable by processes in Q_i from an execution that is the same as $\mathcal{E}_k$ until the end of round r_{i-1} and in which the f processes $\notin Q_i$ fail at the beginning of r_i. As A tolerates f failures, every operation performed during round r_i terminates.

Note that the READ() operation performed by p_n return v_i by the non-triviality property of α-registers. Moreover, observe that if k_i is chosen large enough, the last READ() operation performed by p_i returns also v_i by the propagation property.
Second phase. This phase consists also in M rounds $r'_1, \ldots r'_M$. In round r'_i, which begins after round r'_{i-1} has ended, only processes in Q'_i take steps. We first let the messages that have been sent to the processes in Q'_i during previous rounds $r'_j, 1 \leq j < i$ to be delivered. Then, process p_i performs a READ() operation. As in round r_i, the execution is indistinguishable to the processes in Q'_i from an

execution that is the same until the end of round r_{i-1} and in which the f processes $\notin Q_i$ fail at the beginning of r'_i. Since A tolerates f failures, the READ() operation terminates.

Assuming that k_i has been chosen large enough, the previous READ() operation performed by p_i (in round r_i) returns v_i. By the chronological read property of α-registers, the READ() by p_i in r'_i must return v_i or more recent value, that is a value v_j with $j > i$.

Consider the rounds $r_{i+1}, \ldots, r_M$. The set of processes that take steps in these rounds is $P = \{p_{i+1}, \ldots, p_M\} \cup \{p_{f+2}, \ldots, p_n\}$. On the other hand, the set of processes that takes steps during rounds $r'_1, \ldots, r'_i$ is $P' = \{p_1, \ldots, p_i\} \cup \{p_{M+1}, \ldots, p_{M+(n-f)-1}\}$. Note that $P \cap P' = \emptyset$. Moreover, by construction, for every pair of processes $p \in P, p' \in P'$, every message sent by p (if any) to p' during any round $r_{i+1}, \ldots, r_M$ has not been received by p' by the end of r'_i. Therefore, until the end of r'_i the execution is indistinguishable to the processes in P' from an execution $\mathcal{E}'$ that is the same except that rounds $r_{i+1}, \ldots, r_M$ do not occur in $\mathcal{E}'$. Therefore, the READ() performed by p_i in r'_i cannot return v_j, for any $j > i$. That is, this operation returns v_i.

In the second phase, no WRITE() operation is active. M distinct values are returned by READ() operation performed during this phase. Hence $\alpha \geq M$. □

Remark. The lower bound can be slightly improved by a similar, though more involved, argument to yield $\alpha \geq M + 1$. See [7].

5 Conclusion

The paper has introduced α-registers. For n processes and at most $f \geq \frac{n}{2}$ failures, an implementation of a SWMR $(2M - 1)$-register is presented, where $M = 2f - n + 2$. The implementation is complemented by a lower bound stating that f-resilient simulation of an α-register for $\alpha < M + 1$ is impossible.

Many questions remain open for future research including closing the gap between the implementation and the lower bound, designing a multi-writer multi-reader implementation and understanding the computing power of α-registers. Another challenging direction is to generalize the bounded version of the ABD simulation [5]. Doing so may entail solving problems similar to the ones encountered in the design of fault-tolerant and self-stabilizing atomic registers [4,14].

References

1. Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. *Distributed Computing*, 18(2):113–124, 2005.
2. Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The k-simultaneous consensus problem. *Distributed Computing*, 22(3):185–195, 2010.
3. Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7, 2011.

4. Noga Alon, Hagit Attiya, Shlomi Dolev, Swan Dubois, Maria Potop-Butucaru, and Sébastien Tixeuil. Pragmatic self-stabilization of atomic memory in message-passing systems. In *SSS*, lncs #6976, pages 19–31. Springer, 2011.
5. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.
6. Hagit Attiya and Jennifer Welch. *Distributed Computing.* Wiley, 2004.
7. David Bonnin and Corentin Travers. α-register. Technical report hal#00863060, 2013. `http://hal.inria.fr/hal-00863060/PDF/`
8. Zohir Bouzid and Corentin Travers. $(\omega^x \times \sigma_z)$-based k-set agreement algorithms. In *OPODIS*, lncs#6490, pages 189–204. Springer, 2010.
9. Zohir Bouzid and Corentin Travers. Parallel consensus is harder than set agreement in message passing. In *ICDCS*. IEEE Computer Society, 2013.
10. Apache cassandra `http://cassandra.apache.org/`.
11. Tushar Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.
12. Soma Chaudhuri. More choices allow more faults: set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.
13. Giuseppe DeCandia et al. Dynamo: amazon's highly available key-value store. In SOSP, pages 205–220. ACM, 2007.
14. Shlomi Dolev, Swan Dubois, Maria Gradinariu Potop-Butucaru, and Sébastien Tixeuil. Crash resilient and pseudo-stabilizing atomic registers. In *OPODIS*, lncs#7702, pages 135–150. Springer, 2012.
15. Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. Eventually-serializable data services. *Theor. Comput. Sci.*, 220(1):113–156, 1999.
16. Roy Friedman, Gabriel Kliot, and Chen Avin. Probabilistic quorum systems in wireless ad hoc networks. *ACM Trans. Comput. Syst.*, 28(3), 2010.
17. Roy Friedman, Michel Raynal, and Corentin Travers. Two abstractions for implementing atomic objects in dynamic systems. In *OPODIS*, lncs#3974, pages 73–87. Springer, 2006.
18. Eli Gafni. The extended bg-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92. ACM, 2009.
19. Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
20. Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, 2010.
21. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.
22. Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–101, 1986.
23. Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems. *Inf. Comput.*, 170(2):184–206, 2001.
24. Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, pages 140–149. IEEE Computer Society, 1994.
25. Haifeng Yu. Overcoming the majority barrier in large-scale systems. In *DISC*, lncs#2848, pages 352–366. Springer, 2003.