α-registres

David Bonnin, Corentin Travers

α-registres

David Bonnin and Corentin Travers

LaBRI, Université de Bordeaux, France
name.surname@labri.fr

On sait que, dans un système distribué asynchrone avec communication par envoi de messages, il est possible de simuler un registre atomique, à condition que la majorité des processus ne tombent pas en panne. À l'inverse, si une majorité des processus peuvent tomber en panne, cette simulation est impossible. Cet article explore des variantes faibles des registres atomiques qui peuvent être simulées en tolérant une majorité de pannes. Plus précisément, cet article introduit une nouvelle classe de registres, appelés α-registres, et montre comment les simuler. Avec les registres atomiques, une lecture retourne la dernière valeur écrite. Les α-registres les généralisent de la façon suivante : pour tout intervalle de temps I ne contenant pas d'écriture, au plus α valeurs distinctes sont retournées par les opérations de lecture ayant lieu pendant I. Une simulation d'un α-registre tolérant f pannes dans un système à n processus est présentée pour $\alpha = 2M-1$, avec $M = max(1, 2f-n+2)$. Cette simulation est optimale à un facteur constant près : les α-registres ne peuvent pas être simulés en tolérant f pannes si $\alpha \leq M$.

Keywords: Envoi de message, tolérance aux pannes, simulation de mémoire partagée.

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 [7]. In their strongest form, registers are *atomic* : each operation appears to take place instantaneously. Registers are useful in distributed computing, because it is often easier to write algorithms and prove results with shared registers than in message-passing systems.

More than 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 [2]. This fundamental result enables shared-memory algorithms to be automatically implemented in message passing environment, and thus, problems solvable with shared-memory are solvable in message passing systems. Furthermore, impossibility results and lower bounds established in the message passing model can directly be translated to shared memory.

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* [2]), 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. 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. Each READ() or WRITE() operation uses a quorum of processes to, respectively, gather and propagate information. However, if less than a majority of the processes are non-faulty, i.e. $f \geq \frac{n}{2}$, two quorums may not intersect, leading to READ() operations returning outdated values, because there is no process in the intersection of READ() and WRITE() quorums that could transmit the information of the last written value. Indeed, simulating atomic registers while tolerating $f \geq \frac{n}{2}$ failures in asynchronous message passing is not possible [2].

A few approaches have been proposed to circumvent this impossibility. Probabilistic quorums systems allow two quorums to be non-intersecting with some small probability [1, 5, 8], leading to a small probabil-

ity that READ() operations return stall values. The approaches [6] is based on stronger model assumptions : in particular on a quorum oracle that forces quorums of consecutive write/read to intersect.

The question addressed in the paper *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.

Contributions of the paper The contribution of the paper is threefold : (1) it introduces α-registers, a new type of register that generalizes atomic registers, where α represents a bound on the number of distinct values that can be read in a period without WRITE() operations (Section 2) ; this property is non-trivial, since the ABD emulation (or simple variants) would lead to an unbounded such number ($\alpha = \infty$) . (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).

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. [3]. Each pair of processes $\{p_i, p_j\}$ is connected by a bi-directional channel. Channels are reliable and asynchronous, meaning that each 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. 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 : WRITE(v), where v is value taken from some set $\mathcal{V}$ and READ(). A WRITE(v) operation returns an acknowledgment *ok* and a READ() returns a value $u \in \mathcal{V} \cup \{\perp\}$, $\perp$ being the initial value of the α-register. In an *admissible execution*, no process starts a WRITE(v) or READ() operation while its previous operation, if any, has not returned. The *execution interval* $I(op)$ of an operation op by process p lasts from the invocation of op until it returns ; if p never returns, $I(op)$ has no end. 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, written $op_1 \prec op_2$. 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 WRITE() operations have the same input value. We will write $op_1 \preceq op_2$ if op_1 precedes or is concurrent to op_2.

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

1. *Termination*. Any READ() or WRITE(v) operation performed by a correct process terminates.
2. *Non-spurious value*. For any terminating READ() operation R that returns u, either $u = \perp$ or there exists a WRITE(u) operation W such that $W \preceq R$.
3. *Chronological read*. Let R, R' be two terminating 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 WRITE(u) $\preceq$ WRITE(u').
4. *Non-triviality*. Let R be a READ() operation by process p, returning value u. If there is a WRITE() operation by p that precedes R, $u \neq \perp$. Moreover, if W is the last WRITE() operation by p that precedes R, WRITE(u) is either W or a WRITE() operation W' by another process such that $W \preceq W'$.
5. *Propagation*. Let $u \in \mathcal{V}$ such that a correct process performed a terminating WRITE(u) or a READ() returning u. Eventually, for every terminating READ() with return value u', then WRITE(u) $\preceq$ WRITE(u').
6. α-*Bounded reads*. Let $R_1, \ldots, R_\ell$ be terminating READ() operations performed in an interval I, returning values $\{u_1, \ldots, u_\ell\}$ = $\mathcal{V}_R$. Let $\mathcal{V}_W$ be the set of values written during I ($\forall v \in \mathcal{V}_W$, some Write($v$) was active during I). Then, $\mathcal{V}_O = \mathcal{V}_R \setminus \mathcal{V}_W$, the set of old values read during I, is of size at most α.

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, with $\alpha = 2M-1$, where $M = 2f-n+2$ if $f \geq \frac{n}{2}$ and $M = 1$ otherwise. The algorithm assumes that channels are FIFO.

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

```
1:  INITIALIZATION
2:      seq_i ← 1 ; ⟨v_i,ts_i⟩ ← ⟨⊥,0⟩ ; ⟨vr_i,tsr_i⟩ ← ⟨⊥,0⟩ ; Qr_i ← ∅ ; Qe_i ← ∅ ; Qw_i ← ∅ ;
3:      Accept_i[1..n] ← [2,...,2] ;                    ▷ array of n integers initialized to 2
4:      for each p_j : 1 ≤ j ≤ n do send UPDATE(seq_i, ⟨v_i,ts_i⟩, 0)
5:  function WRITE(v)
6:      ⟨v_i,ts_i⟩ ← ⟨v,ts_i+1⟩ ; seq_i ← seq_i+1 ; Qw_i ← ∅ ;
7:      wait until |Qw_i| ≥ n−f ;
8:      return ok
9:  function READ( )
10:     n_iter ← 0 ;
11:     repeat
12:         ⟨vr_i,tsr_i⟩ ← ⟨v_i,ts_i⟩ ; seq_i ← seq_i+1 ; Qr_i ← ∅ ; Qe_i ← ∅ ; n_iter ← n_iter+1 ;
13:         wait until |Qr_i ∪ Qe_i| ≥ n−f ;
14:     until (|Qe_i| ≥ n−f) or (n_iter ≥ N)            ▷ N = (4f+2)(⌊n/(n−f)⌋+1)+1
15:     return vr_i
16: WHEN UPDATE(seq, ⟨v,ts⟩, old_seq) FROM PROCESS p_j IS RECEIVED
17:     if old_seq = seq_i then
18:         if ts = ts_i then Qw_i ← Qw_i ∪ {p_j}
19:         if ts > tsr_i then Qr_i ← Qr_i ∪ {p_j}
20:         if ts = tsr_i then Qe_i ← Qe_i ∪ {p_j}
21:     if ts > ts_i then
22:         if Accept_i[j] > 0 then Accept_i[j] ← Accept_i[j] − 1
23:         else ⟨v_i,ts_i⟩ ← ⟨v,ts⟩ ; Accept_i[1..n] ← [2,...,2]   ▷ if Accept_i[j] = 0
24:     send UPDATE(seq_i, ⟨v_i,ts_i⟩, seq) to p_j
```

As in the ABD protocol [2], each time a new value is written it is first associated with a unique timestamp (line 6). As there is a single writer, no two values are associated with the same timestamp. Each process p_i 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 23).

Although the timestamps are unbounded integers in this algorithm, it is possible to have a similar implementation that only uses a finite number of different timestamps. This can be done by using the principles described in [2], but would probably need a larger namespace for timestamps.

Processes constantly exchange messages of type UPDATE containing data of the form $(sq, \langle v,ts \rangle, osq)$, where $\langle v,ts \rangle$ is the local $\langle v_i, ts_i \rangle$ variable of the sender. The sq and osq are sequence numbers such that a process can determine whether this message is related to its current operation or not.

Write() operations The implementation of a WRITE(v) operations is similar to the implementation in the ABD protocol [2]. After a new timestamp t has been associated with v line 6, 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 7–line 8).

Values dissemination 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, every UPDATE sent by p_n contains $\langle v,t \rangle$, until a new WRITE() starts.

We want α to be as small as possible. To that end, the algorithm first bounds the number of messages sent but not delivered. Indeed, at any time, each channel contains at most 2 messages. Hence, at any given

time, the system contains at most $O(n^2)$ distinct values stored either locally by the processes or carried by not yet delivered messages. Of note, α is unbounded for the ABD protocol when used with $f \geq n/2$.

Next, to further decrease α, the algorithm tries to reduce the number of values *accepted* by each process. Each process p_i is endowed with an array $Accept_i[1..n]$ with one entry per process. Initially, $Accept_i[j] = 2$ and at any time, $Accept_i[j] \in \{0,1,2\}$ for any $j, 1 \leq j \leq n$. $Accept_i[j] < 2$ means that p_i knows that its current value is outdated by p_j's current value (line 21 – line 22). However, p_i does not update immediately $\langle v_i, ts_i \rangle$ with the more recent pair $\langle v,t \rangle$ it has just received. Indeed, this pair may no longer be stored by any process, that is $\langle v,t \rangle$ is only contained in messages not yet delivered. Instead, the algorithm ensures that when p_i changes $\langle v_i, ts_i \rangle$ to $\langle v,t \rangle$, $\langle v,t \rangle$ is actually stored locally by some process at some time since p_i discovers that $\langle v_i, ts_i \rangle$ contains an outdated pair (line 21–line 23). Hence, for any given interval I, each process p_i may accept and thus read only *one* value among the values carried by the messages it its input channels at the beginning of I. Therefore, assuming that no WRITE() operations overlap I, $\alpha = O(n)$. A finer analysis, taking into account the fact values read in the absence of concurrent WRITE() operations are accepted by at least $n - f$ processes, leads to $\alpha = 2M - 1 = 4f - 2n + 3$.

Read() operations A READ() operation (line 10–line 15) by process p_i consists in up to $N = O(\frac{fn}{n-f})$ iterations. At the beginning of iteration s, the variable $\langle vr_i, tsr_i \rangle$ is updated, 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 12). An iteration terminates when p_i knows that at least $n - f$ processes store values at least as recent as vr_i (line 13). The READ() operation terminates (1) immediately if $|Qe_i| \geq n - f$, i.e., for at least $n - f$ processes p_j, there is a time at which $v_j = vr_i$ or (2) after N iterations have been performed.

Condition (1) may never be reached in any number of iterations if the READ() operation is concurrent to a large number of WRITE() operations. But, if the READ() operation is performed in the absence of concurrent WRITE() operations, condition (1) will necessarily be reached in less than N iterations, as shown in [4].

The proof of this algorithm, i.e., the fact that this algorithm satisfies the properties of an α-register with $\alpha = 2M - 1$, is presented in [4].

4 Lower bound

In [4], a lower bound has also been proven, presented as the following theorem :

Theorem 4.1. *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, $\alpha \geq M$.*

The idea of the proof is to assume there exists an algorithm implementing an α-register with $\alpha < M$, and then to construct a family of executions leading to M distinct values being returned by READ() operations during a period without WRITE(). Those executions are described both at high level, with the calls of operations by some processes, and lower level, with the forced delay of messages in channels.

Références

[1] I. Abraham and D. Malkhi. Probabilistic quorums for dynamic systems. *Distributed Computing*, 18 :113–124, 2005.

[2] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42 :124–142, 1995.

[3] H. Attiya and J. Welch. *Distributed Computing : Fundamentals, Simulations, and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley, 2004.

[4] D. Bonnin and C. Travers. α-register. In *Principles of Distributed Systems*, volume 8304 of *LNCS*, pages 53–67. Springer, 2013.

[5] R. Friedman, G. Kliot, and C. Avin. Probabilistic quorum systems in wireless ad hoc networks. *ACM Trans. Comput. Syst.*, 28 :7 :1–7 :50, 2008.

[6] R. Friedman, M. Raynal, and C. Travers. Two abstractions for implementing atomic objects in dynamic systems. In *Principles of Distributed Systems*, volume 3974 of *LNCS*, pages 73–87. Springer, 2006.

[7] L. Lamport. On interprocess communication. *Distributed Computing*, 1 :77–85, 1986.

[8] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright. Probabilistic quorum systems. *Inf. Comput.*, 170 :184–206, 2001.