



**HAL**  
open science

## Randomized Message-Passing Test-and-Set

Emmanuelle Anceaume, François Castella, Achour Mostefaoui, Bruno Sericola

► **To cite this version:**

Emmanuelle Anceaume, François Castella, Achour Mostefaoui, Bruno Sericola. Randomized Message-Passing Test-and-Set. 2014. hal-01075650v1

**HAL Id: hal-01075650**

**<https://hal.science/hal-01075650v1>**

Preprint submitted on 19 Oct 2014 (v1), last revised 13 Aug 2015 (v3)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Randomized Message-Passing Test-and-Set

Emmanuelle Anceaume<sup>\*</sup>, François Castella<sup>†</sup>, Achour Mostéfaoui<sup>‡</sup> and Bruno Sericola<sup>§</sup>

<sup>\*</sup>IRISA / CNRS Rennes (France), emmanuelle.anceaume@irisa.fr

<sup>†</sup>IRMAR / Université de Rennes 1 (France), francois.castella@univ-rennes1.fr

<sup>‡</sup>LINA / Université de Nantes (France), achour.mostefaoui@univ-nantes.fr

<sup>§</sup>INRIA Rennes – Bretagne Atlantique (France), bruno.sericola@inria.fr

**Abstract**—This paper presents a solution to the well-known Test&Set operation in an asynchronous system prone to process crashes. Test&Set is a synchronization operation that, when invoked by a set of processes, returns *yes* to a unique process and returns *no* to all the others. Recently many advances in implementing Test&Set objects have been achieved, however all of them target the shared memory model. In this paper we propose an implementation of a Test&Set object in the message passing model. This implementation can be invoked by any number  $p \leq n$  of processes where  $n$  is the total number of processes in the system. It has an expected individual step complexity in  $O(\log p)$  against an oblivious adversary, and an expected individual message complexity in  $O(n)$ . The proposed Test&Set object is built atop a new basic building block, called selector, that allows to select a winning group among two groups of processes. We propose a message-passing implementation of the selector whose step complexity is constant. We are not aware of any other implementation of the Test&Set operation in the message passing model.

**Keywords**-Test&Set, agreement problem, asynchronous message-passing system, crash failures, randomized algorithm, synchronization.

## I. INTRODUCTION

The Test&Set problem is a classical synchronization service in shared-memory centralized systems classically provided by a unique hardware atomic instruction. It allows us to solve competition problems. When invoked by a set of processes, it returns *yes* to a unique process (the winner) and returns *no* to all the others (the losers). According to the hierarchy of agreement problems based on consensus numbers given by Herlihy in [11], consensus is harder to solve than Test&Set, that is a solution to Test&Set does not allow us to solve consensus, but Test&Set is still too hard to be solved in a pure asynchronous system [4]. Indeed, Test&Set has a consensus number equal to two, just like renaming and queues for example, whereas the consensus number of the consensus problem is infinite [11].

**Contributions** : In this paper we propose a randomized implementation of the Test&Set operation. Indeed, Herlihy [11] has shown that this operation does not have a deterministic implementation as soon as one crash may occur, and thus in order to implement it in an asynchronous system, it is necessary to add synchrony assumptions or to use randomization. We focus on the latter option. For randomized solutions, the relations between random decisions

and the scheduling of processes (*i.e.*, read/write operations in the shared memory model, and send/receive operations on messages in the message passing model) are taken into account through the definition of the adversary. In this paper we consider the *oblivious adversary model*, that is, the model in which the adversary makes all its scheduling decisions at the beginning of the execution independently of the random values tossed by the processes in the course of the execution. In contrast, the *adaptive adversary model* supposes that the adversary makes its decisions based on the full history of the events. This model, while stronger than the oblivious one, has so far led to space expensive known implementations [3] or non adaptive ones [1] in the shared memory model.

In this paper, we propose to implement the Test&Set operation in the message-passing model. So far all the efficient solutions have been implemented with shared registers [1]–[3], [9]. Our implementation can be invoked by any number  $p \leq n$  of processes where  $n$  is the total number of processes in the system. This implementation has an expected individual step complexity in  $O(\log p)$  and an expected individual message complexity in  $O(n)$  against an oblivious adversary. The *expected step complexity* (respectively the *expected message complexity*) represents the maximum number of steps (respectively the maximum number of messages) needed by any process in expectation to complete its execution, assuming the scheduling of the worst adversary taken from its family (*i.e.*, the oblivious or adaptive adversary family). Having a step or message complexity that depends on  $p$  and not on the number of processes  $n$  of the system makes our solution *adaptive*. We are not aware of any adaptive implementation of the Test&Set object. This makes our solution interesting from both the theoretical aspect but also from the practical one, as the cost of the implementation depends uniquely on the number of processes that concurrently invoke this operation.

Finally, the proposed Test&Set object is built atop a new basic building block, called selector. A selector is a distributed service, invoked by a set of processes, that allows to select a winning group among at most two competing ones. We propose a message-passing implementation of the selector whose step complexity is constant.

**Road map**: In the remaining of the paper, Section II presents the underlying computation model and specifies

the Test&Set problem. Section III presents a randomized implementation to this problem. Section IV shows the correctness of the implementation, and both its message and step complexity. Section V proposes a randomized solution to the selector operation and shows the correction of this operation. Finally Section VI concludes.

## II. COMPUTATION MODEL AND PROBLEM DEFINITION

### A. Computation Model

We consider an asynchronous system consisting of a set  $\Pi$  of  $n$  processes, namely,  $\Pi = \{p_1, p_2, \dots, p_n\}$ . A process can fail prematurely by crashing. A process behaves according to its specification until it (possibly) crashes. After it has crashed a process executes no step. A process that never crashes is said to be correct; otherwise it is faulty. Let  $t$  denote the maximum number of processes that may crash. We assume that a majority of processes is correct, namely  $t < n/2$ . We focus on a message-passing solution, that is processes communicate and synchronize by sending and receiving messages through reliable but not necessarily FIFO channels. Specifically, every pair of processes is connected by a channel that cannot create, alter, or lose messages however can duplicate them. Hence if both the sender and the receiver are correct then any sent message is eventually received but no spurious messages are generated. As the system is asynchronous, there are no assumption regarding the relative speed of processes nor the message transfer delays. The communication system offers two types of communication primitives. A broadcast primitive *bcast* that allows a process to send a same message to all the processes. This operation is not atomic, it can be implemented as a multi-send statement; if the sender of a message is faulty some processes can receive it and others not. Moreover, if a process knows the identity of another process with which it wants to communicate, it can send a message directly to it using a point-to-point communication primitive *send*.

### B. Problem Definition

Test&Set is usually a hardware operation offered by the processor. In the case of distributed computing, the Test&Set problem is a coordination problem where a set of processes invoke Test&Set and return a binary value **yes** or **no** such that exactly one returns **yes** (the winner) and all the others return **no** (the losers). From an operational point of view, the Test&Set operation is attached to distributed objects. Let  $o$  be a Test&Set object that can be accessed through the method `t&s`, which can be invoked by any process  $p_i$  using `o.t&s()`. An invocation returns a binary result **yes** or **no**. A protocol that solves the randomized Test&Set problem must satisfy the following four properties:

- **TS-Validity:** A process, invoking the `o.t&s` primitive, that returns a value must return either **yes** or **no**.
- **TS-Obligation:** If no process crashes then, exactly one process returns **yes**.

Output	Meaning
(yes,yes)	The group of $p_i$ wins and $p_i$ is the winner in the group
(yes,no)	The group of $p_i$ wins but there is no winner in the group
(no,no)	Either the group of $p_i$ loses or there is a winner in the group of $p_i$ and the winner is not $p_i$
(no,yes)	Impossible, $p_i$ cannot be a winner if its group does not win

Table I  
THE POSSIBLE OUTPUTS OF A SELECTOR OBJECT INVOKED BY A SET OF  $p \geq 1$  PROCESSES,  $p_i$  BELONGS TO.

- **TS-Agreement:** At most one process returns **yes** and in this case, all the other returning processes return **no**.
- **TS-Termination:** An invocation by a correct process of the `o.t&s` primitive terminates with probability 1.

Moreover, the different calls to the Test&Set operation need to be linearisable. It has been proved in [10] that any object that satisfies the properties cited above can be used together to implement a linearisable Test&Set object. Consequently, we will not worry about linearisability.

## III. A RANDOMIZED TEST&SET OBJECT

This section presents a Test&Set object that can be invoked by any number of competing processes. We implement this object by having competing processes go through a series of accesses to a basic agreement service object called *selector*.

### A. A selector Object

A selector object proposes a unique access primitive `play()` that is invoked with a boolean parameter  $g$  (0 or 1). Each of the two binary values represents a group. A process randomly chooses its group (0 or 1) before invoking `play()`. This basic object is in charge of selecting the winning group  $g'$ , and the winning process within this group, if any. Consequently the primitive `play()` returns two boolean values to each invoking process. The first one says if the group of the invoking process is the winner one, and the second one indicates whether the invoking process is also the winner in its group. Table I shows the three possible responses process  $p_i$  can receive upon invocation of primitive `play`. More formally, let  $s$  be a selector object, invoked by any process  $p_i$  using `s.play(g)` with  $g$  equal to 0 or 1. A protocol that implements such an object must satisfy the following six properties:

- **S-Validity:** A process, invoking the `s.play` primitive, that does not crash must return either (yes,yes), (yes,no) or (no,no).
- **S-Obligation-solo:** If a process invokes `s.play` alone (solo execution) and does not crash then, it returns (yes,yes).

```

Function  $o.$ Test&Set()
(1)  $step_i = 1;$ 
(2) repeat forever
(3)    $(b_1, b_2) \leftarrow selector[step_i].play(rnd(0, 1));$ 
(4)   if  $(b_1 = \text{yes} \wedge b_2 = \text{yes}) \vee (b_1 = \text{no})$  then
(5)     return  $(b_1 = \text{yes} \wedge b_2 = \text{yes})$ 
(6)   endif;
(7)    $step_i \leftarrow step_i + 1;$ 
(8) endrepeat;

```

Figure 2. Randomized Test&Set Object

- **S-Obligation:** If no process crashes then, at least one process returns (yes,-).
- **S-Agreement:** At most one process returns (yes,yes), and in this case, all the other returning processes return (no,no).
- **S-Exclusion:** If an invocation of  $s.play$  with parameter  $g$  returns (yes,-) then, no invocation with parameter  $\neg g$  can return (yes,-).
- **S-Termination:** An invocation of  $s.play$  by a correct process terminates with probability 1.

Section III-A proposes a message-passing implementation of the selector object.

### B. Implementing the Randomized Test&Set Object

As aforementioned, implementation of the Test&Set object relies on instances of the selector object as illustrated in Figure 1.

Pseudo-code of the Test&Set algorithm, given in Figure 2, can be seen as a process elimination by dichotomy. At the first step, each of the  $p$  competing processes  $p_i$  flips a local coin (we suppose that each process uses an unbiased local coin) and invokes the first instance of the selector object with this coin as parameter. This parameter represents  $p_i$  group for this instance of the selector object. The selector object selects the winning group (set of processes) allowed to continue the competition, and eliminates all the processes of the other group (if any). Specifically, any process  $p_i$  that exits with (yes,no) from the current instance of the selector object triggers a new step of the Test&Set algorithm by invoking the next instance of the selector object by flipping again a local coin. On the other hand, any process  $p_i$  that exits from the current instance of the selector object with (no,no) also exits from the Test&Set invocation with no. The last step of the Test&Set algorithm occurs when one of the remaining competing processes  $p_i$  exits from the selector object invocation with (yes,yes). This winning process exits with yes from the Test&Set invocation. As it will be proven in the sequel, any invocation by a process that does not crash terminates with probability 1. As said in the introduction, our algorithm does not need to know how many processes access the Test&Set object. Remaining of the paper will clarify all these points.

## IV. ANALYSIS OF THE TEST&SET IMPLEMENTATION

This section proves the correction of the algorithm of Figure 2 by proving the four properties of a Test&Set object, namely the TS-Validity, TS-Obligation, TS-Agreement and TS-Termination properties and then shows the complexity both in terms of steps and messages of the algorithm.

### A. Proof of Correctness of the Test&Set Implementation

The TS-Validity property is a direct consequence of line 4 of the algorithm, while the TS-Obligation property is a consequence of the S-Obligation of the selector underlying object. Indeed, if none of the processes that execute  $o.$ Test&Set() crash, then necessarily they execute line 3 of the algorithm. By the S-Obligation property of the selector object, at least some process will exit with (yes,-). If some process returns (yes,yes) then by the S-Agreement of the selector object, all the other processes will return (no,no). Now if only (yes,no) is returned, then all these processes will execute a new instance of the selector. The same argument applies until exactly one process executes a solo execution in which case it will return (yes,yes), and by the S-Agreement of the selector object, all the other processes will return (no,no).

To prove the TS-Agreement property, let us consider the first process that exits with yes at line 5 at some step  $step_i$ . Necessarily this process invoked  $selector[step_i].play()$  and this invocation returned (yes,yes). Consequently, by the S-Agreement property of a selector, all the other processes that invoke the selector will exit with (no,no) and consequently, these processes will return no at line 5 at the same step  $step_i$ .

Property TS-Termination is more tricky to prove. By the S-Validity of a selector, returned values are pairs of boolean, consequently, the Test&Set algorithm is properly executed (no type errors). Moreover, by the S-Termination property of a selector, a correct process terminates the call of line 3 with probability 1. Saying this, we conclude that if the Test&Set algorithm does not terminate, it will execute an infinity of times the repeat loop. Section IV-B proves that this loop terminates after no more than  $2 \log_2(p)$  invocations of the selector object in expectation for large values of the contention  $p$  of the Test&Set execution (see Theorem 2). Moreover, the average number of selector invocations done by any of the competing processes during a Test&Set execution is constant (2 invocations per process as shown in Corollary 1).

### B. Complexity Analysis of the Test&Set Implementation

We now analyse the complexity of our implementation with respect to both the number of exchanged messages and the number of execution steps.

**Theorem 1 (Message complexity of Test&Set()):** The total number of messages exchanged by the randomized

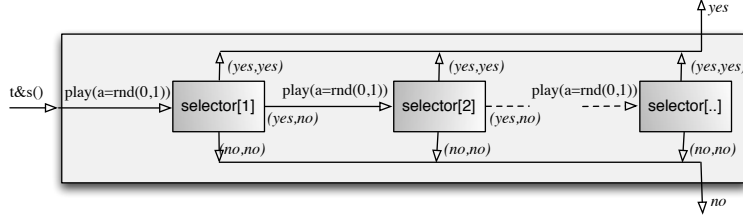


Figure 1. Test&Set object using selector objects as building blocks.

implementation of the Test&Set object when concurrently invoked by  $p$  processes is  $O(np)$ .

*Proof:* Consider a Test&Set execution with contention  $p$ . From Theorem 4 (see Section V-C), the message complexity of each invocation of the selector object requires  $O(np)$  messages. By Corollary 1 (see below) each competing process invokes  $s.play()$  twice in expectation. Consequently, the expected total number of messages exchanged by the Test&Set algorithm with contention  $p$  is  $O(np)$ , and thus  $O(n)$  messages are needed per competing process. ■

To carry out the step complexity analysis, we consider the worst-case execution, namely that the Test&Set protocol terminates at the latest when there is only one process executing the protocol. Indeed, the protocol may terminate before, that is, as soon as a process succeeds in being the winner of the winning group. However, as this situation may or may not happen, the analysis supposes the worst case execution, where the Test&Set protocol executes until there is a unique competing process. Consequently, for the worst case analysis, only the first boolean returned by the selector object is relevant. Recall that this boolean indicates whether the invoking process belongs to the winning group or not.

Now, when competing processes invoke a selector, each one chooses a group at random (line 3 of Figure 2). By the S-Exclusion property of a selector, only one group will win. The identity of the winning group (0 or 1) depends on the actual scheduling and the adversary. Hence, as the choice of the group is done at random and this is not a priori known by the oblivious adversary, we assume that the two events "group 0 wins" and "group 1 wins" occur with the same probability  $1/2$  and that the behaviours of the processes at each instant are independent of each other.

We suppose that  $p \leq n$  processes concurrently access the Test&Set object. The behaviour of the algorithm can be modelled by a Markov chain  $X = \{X_\ell, \ell \geq 1\}$ , where  $X_\ell$  represents the number of processes in competition at the  $\ell$ -th transition, *i.e.*, the number of processes that execute the  $\ell$ -th step. Hence, the state of the Markov chain is an integer value  $i$  ( $1 \leq i \leq p$ ). The initial state of  $X$  is state  $p$ , with probability 1, that is  $\mathbb{P}\{X_0 = p\} = 1$  and we denote by  $P$  the transition probability matrix of  $X$ . The probability  $P_{i,j}$  to go from state  $i$  to state  $j$  in one transition is equal to 0

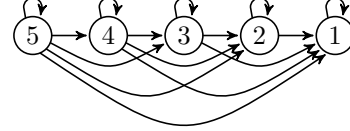


Figure 3. Transition graph of Markov chain  $X$  when  $p = 5$

if  $i < j$ . Indeed, a process that returns  $b_1=no$  cannot any more continue the competition (see line 5 in Figure 2). Now, when all the  $i$  competing processes choose the same group (either 0 or 1) then they all restart the competition in the same state. It follows that, for  $i = 1, \dots, p$ ,

$$P_{i,i} = \frac{1}{2^i} + \frac{1}{2^i} = \frac{1}{2^{i-1}}.$$

Finally, for  $1 \leq j < i \leq p$ ,  $P_{i,j}$  is the probability that exactly  $j$  processes among  $i$  choose the same group and that this group wins. We thus have, in this case,

$$P_{i,j} = \frac{1}{2} \left[ \frac{1}{2^i} \binom{i}{j} + \frac{1}{2^i} \binom{i}{i-j} \right] = \frac{1}{2^i} \binom{i}{j}.$$

The states  $2, 3, \dots, p$  are thus transient states and state 1 is absorbing since  $P_{1,1} = 1$ . Figure 3 shows the graph of  $X$  in the case where  $p = 5$ .

In the following we evaluate the average number of steps to reach state 1, and the average total contention before termination, *i.e.*, before reaching state 1. By total contention we mean the following: Let us consider the series  $n_1, n_2, n_3, \dots, n_k$  where  $n_\ell$  represents the number of processes that execute step  $\ell$  of the Test&Set protocol (contention on step  $\ell$ ). By assumption  $n_1 = p$ . We call the total contention on the whole selector objects the sum  $n_1 + n_2 + n_3 + \dots + n_k$ . The evaluation shows that the average total contention is linear in  $p$ .

When  $p$  processes are initially competing, the worst case time needed by the Test&Set protocol to terminate is the hitting time of state 1 by Markov chain  $X$ . If we denote by  $T_p$  such a time, we have

$$T_p = \inf\{\ell \geq 0 \mid X_\ell = 1\}.$$

It is well-known, see for instance [13], that the expected value of  $T_p$  is given by

$$\mathbb{E}\{T_p\} = \alpha(I - Q)^{-1}\mathbb{1},$$

where  $Q$  is the matrix of dimension  $p - 1$  obtained from  $P$  by deleting the row and the column corresponding to absorbing state 1,  $\alpha$  is the row vector containing the initial probabilities of the transient states, that is  $\alpha_p = 1$  and  $\alpha_i = 0$  for  $i = 2, \dots, p - 1$ , and  $\mathbb{1}$  is the column vector of dimension  $p - 1$  with all its entries equal to 1. The expected value  $\mathbb{E}\{T_p \mid X_0 = p\}$  can also be evaluated using the well-known recurrence relation, see for instance [13],

$$\mathbb{E}\{T_p \mid X_0 = p\} = 1 + \sum_{k=2}^p P_{p,k} \mathbb{E}\{T_k \mid X_0 = k\}. \quad (1)$$

**Theorem 2 (Step Complexity of Test&Set()):** The expected time  $\mathbb{E}\{T_p \mid X_0 = p\}$  needed to terminate the Test&Set protocol when  $p$  processors are initially competing satisfies

$$\mathbb{E}\{T_p \mid X_0 = p\} = O(\log(p)).$$

More precisely, there exists a positive integer  $p_0$  such that, for every  $p \geq p_0$ , we have

$$\mathbb{E}\{T_p \mid X_0 = p\} \leq 2 \log(p),$$

where  $\log$  denote the logarithm function to the base 2.

*Proof: (Sketch.)* Introducing the notation  $u_p = \mathbb{E}\{T_p \mid X_0 = p\}$  and replacing  $P_{p,k}$  by its value, Formula 1 can be written as

$$u_p = 1 + \sum_{k=2}^{p-1} 2^{-p} \binom{p}{k} u_k + O(2^{-p}).$$

The key idea lies in the fact that  $\binom{p}{k}$  is maximal when  $k = p/2$ , and decreases rapidly away from the value  $k = p/2$ , so that the above recursion formula for  $u_p$  very roughly asserts that  $u_p \approx 1 + u_{p/2}$ . Would this simplified recursion formula hold true *exactly*, the bound  $u_p = O(\log(p))$  would be obvious. Based on this rough idea, the proof is split into three main steps.

First, given a small  $\alpha > 0$ , Stirling formula implies  $2^{-p} \binom{p}{k} = O(\exp(-2p^{2\alpha}))$  uniformly in  $k$  whenever  $|k - p/2| \geq p^{1/2+\alpha}$ . This provides the simplified recursion formula

$$u_p = 1 + \sum_{k: |k-p/2| \leq p^{1/2+\alpha}} 2^{-p} \binom{p}{k} u_k + O(2^{-2p^\alpha}).$$

The second step consists in introducing a dyadic partition, so we define  $U_j = \max_{2^j \leq k \leq 2^{j+1}} u_k$ . A detailed analysis of the above recursion formula provides, roughly

$$U_{j+1} \leq 1 + \frac{U_j + U_{j+1}}{2} + O(2^{-2p^\alpha}),$$

where  $p = 2^j$ .

The last argument consists in proving that the above bound provides  $U_j \leq 2j + C$ , for some constant  $C$  that does not depend on  $j$ . This completes the proof. For space reasons, the full proof appears in Appendix A. ■

Using this result and the Markov inequality, we obtain, for the positive integer  $p_0$  of Theorem 2, for every  $m \geq 1$  and  $p \geq p_0$ ,

$$\mathbb{P}\{T_p > 2m \log(p)\} \leq 1/m.$$

We consider now the total contention before termination. For  $\ell \geq 0$ , we denote by  $W_\ell(p)$  the number of processes that executed step  $\ell$  of the protocol when  $p$  processes are initially competing. This random variable is defined by  $W_\ell(p) = \sum_{i=2}^p i \mathbb{1}_{\{X_\ell=i\}}$ . Since the initial state is state  $p$ , we have  $W_0(p) = p$  with probability 1.  $W_0(p)$  represents the contention of the Test&Set and also the contention of the first invocation of the selector object. The total contention before termination is denoted by  $N(p)$  and given by  $N(p) = \sum_{\ell=0}^{\infty} W_\ell(p)$ . Note that  $N(p)$  is also the total contention of the whole invocations of the selector object. The next theorem gives the expectation of  $N(p)$ .

**Theorem 3 (Total Contention):** For every  $p \geq 2$  and  $\ell \geq 0$ , we have

$$\mathbb{E}\{W_\ell(p)\} = \frac{p}{2^\ell} \text{ and } \mathbb{E}\{N(p)\} = 2p.$$

*Proof:* Since the initial state is state  $p$ , we have, for  $\ell \geq 0$ ,  $\mathbb{E}\{W_\ell(p)\} = \sum_{i=2}^p i \mathbb{P}\{X_\ell = i \mid X_0 = p\} = \sum_{i=2}^p i (Q^\ell)_{p,i}$ . For  $\ell = 0$ , we have  $\mathbb{E}\{W_0(p)\} = p$ . For  $\ell \geq 1$ ,

$$\begin{aligned} \mathbb{E}\{W_\ell(p)\} &= \sum_{i=2}^p i \sum_{j=i}^p Q_{p,j} (Q^{\ell-1})_{j,i} \\ &= \sum_{j=2}^p Q_{p,j} \sum_{i=2}^j i (Q^{\ell-1})_{j,i} \\ &= \sum_{j=2}^n Q_{p,j} \mathbb{E}\{W_{\ell-1}(j)\}. \end{aligned}$$

We pursue by recurrence over index  $\ell$ . The result being true for  $\ell = 0$ , suppose that for every  $j \geq 2$ , we have  $\mathbb{E}\{W_{\ell-1}(j)\} = j/2^{\ell-1}$ .

Then, for every  $p \geq 2$ ,

$$\begin{aligned}
\mathbb{E}\{W_\ell(p)\} &= Q_{p,p}\mathbb{E}\{W_{\ell-1}(p)\} + \sum_{j=2}^{p-1} Q_{p,j}\mathbb{E}\{W_{\ell-1}(j)\} \\
&= \frac{1}{2^{p-1}} \frac{p}{2^{\ell-1}} + \frac{1}{2^p} \sum_{j=2}^{p-1} \binom{p}{j} \frac{j}{2^{\ell-1}} \\
&= \frac{p}{2^p 2^{\ell-1}} + \frac{1}{2^p} \sum_{j=2}^p \binom{p}{j} \frac{j}{2^{\ell-1}} \\
&= \frac{1}{2^p 2^{\ell-1}} \sum_{j=1}^p j \binom{p}{j} \\
&= \frac{p}{2^p 2^{\ell-1}} \sum_{j=1}^p \binom{p-1}{j-1} \\
&= \frac{p}{2^\ell}.
\end{aligned}$$

We then have  $\mathbb{E}\{N(p)\} = \sum_{\ell=0}^{\infty} \mathbb{E}\{W_\ell(p)\} = 2p$ , which completes the proof.  $\blacksquare$

**Corollary 1:** Each process competing for the Test&Set object invokes 2 instances of the selector object in expectation.

*Proof:* Straightforward from Theorem 3 as  $\mathbb{E}\{N(p)\}/p = 2$ .  $\blacksquare$

## V. MESSAGE-PASSING IMPLEMENTATION OF THE SELECTOR OBJECT

The implementation of the selector object, whose properties appear in Section III, falls under the impossibility result of many agreement problems in the context of asynchronous distributed systems prone to process failures [8]. We thus consider an asynchronous message-passing distributed system augmented with a random oracle (see Section V-A) to circumvent the impossibility result. Note that the problem we want to solve is an adaptation of Ben-Or consensus algorithm [5], and is close to the one solved by Tromp and Vitány [14] in the context of shared memory systems. Tromp and Vitány's problem is the same as the classical Test&Set problem except that it can be invoked by at most two different processes. Operationally, the selector is attached to distributed objects. Let us consider a selector  $s$ . As previously described in Section III, selector  $s$  can be concurrently invoked by  $p$  processes,  $1 \leq p \leq n$ , however all the  $n$  processes of the system have to participate. Indeed, as there is no shared memory and processes may fail by crashing, the participation of all processes is required to serve as arbiters and as collective memory [4].

### A. Common Coin

In the implementation of the selector object, we assume that processes have access to a function `common_coin()` which provides all the processes the same value (0

or 1) with probability 1/2. Each process invokes function `common_coin()` at the beginning of each round of the protocol. Thus the values returned by function `common_coin()` to all the processes are identical.

### B. Implementing the Selector Object

The algorithm is round based. It is presented in Figure 4. It goes through a series of rounds each one composed of two communication phases. The algorithm is divided into two parts. The first one is executed by the invoking processes (that is the processes that have invoked method `play` on object  $s$ ), while the second part (called Relay Task in Figure 4) is executed by all processes including the invoking ones. This is done for generality since the messages sent by a process to itself are directly delivered to it. The Relay Task serves as a relay to the messages sent by the competing processes and implements some kind of shared memory. We will respectively call these two groups of processes the *invoking* processes and the *relaying* processes.

The goal of the method `play` is to determine the winning value among the proposed ones and the winner of the competition, if any. A process  $p_i$  wins the competition if either  $p_i$  is the only process that invoked the primitive `play` or  $p_i$  has invoked the primitive `play` with the winning boolean value  $g'$  and  $p_i$  has no evidence that another process did the same. If none of both conditions hold, then all the processes that have proposed the winning value will compete in a new execution of primitive `play`, while all the other ones stop the competition. This is achieved as follows. Each invoking process  $p_i$  handles a variable  $g\_est_i$  representing its estimation of the winning group (0 or 1). Variable  $g\_est_i$  is initially set to the value  $g_i$  that  $p_i$  proposed when it invoked the method `play`. Then, this estimate will evolve according to what  $p_i$  will learn during the protocol. Similarly each invoking process  $p_i$  manages a variable  $id\_est_i$  representing its estimation of the possible winning process inside the winning group (initially  $id\_est_i$  is set to  $p_i$ ).

At the beginning of each round, each invoking process  $p_i$  tosses a common coin  $c$  (as said in Section V-A this value is common to all invoking processes in the current round). During the first phase of the current round,  $p_i$  broadcasts its estimates  $g\_est_i$  and  $id\_est_i$  in a PHASE message to all processes, and waits for their echo (Line 5 in Figure 4). As several processes may play during a same round, some relaying processes may first receive the PHASE message from some invoking process  $p_i$  and thus will only echo  $p_i$  estimate while other relaying processes may first receive a PHASE message from another invoking process  $p_j$  possibly endorsing the group  $\neg g$  and will echo  $p_j$  estimate. Each relaying process manages two variables  $g[r, x]$ ,  $id[r, x]$  for each of the two phases  $x$  of each round  $r$ . They are used to store the estimates ( $g$  and  $id$ ) received in the PHASE message received from an invoking process. Thereafter, these same

estimates are echoed to all invoking processes from which a PHASE message was received for the same phase of the same round.

Each invoking process  $p_i$  collects in set  $G_i$  the echoed values received from a majority of processes including itself. Upon receipt of a majority of echoes, if  $G_i$  contains a single value then  $p_i$  keeps this value in  $g\_aux_i$  otherwise, it knows that there is contention between two groups of processes each one championing for the two possible values (0 and 1). Process  $p_i$  sets variable  $g\_aux_i$  to a value  $\perp$  reflecting such a contention. Process  $p_i$  applies the same argument for the echoed identifiers.

To summarize, the first phase ensures that for any pair of invoking processes  $p_i$  and  $p_j$ , if  $g\_aux_i$  and  $g\_aux_j$  are both different from  $\perp$  then they necessarily contain the same value  $g$  and if  $id\_aux_i$  and  $id\_aux_j$  are both different from  $\perp$  then they necessarily contain the same process identity  $id$ .

During the second phase of the round,  $p_i$  broadcasts both  $g\_aux_i$  and  $id\_aux_i$  and collects in  $G_i$  and  $Id_i$  the echoes from a majority of processes. By construction of the first phase, if  $G_i$  contains a value  $g$  and possibly  $\perp$  then  $p_i$  is sure that any other invoking process  $p_j$  will receive either  $g$  or  $\perp$  values. Moreover, if  $G_i$  contains a unique value,  $p_i$  is certain that any other invoking process  $p_j$  will receive at least this value (two majorities always intersect). In particular, if  $G_i$  contains only the  $\perp$  value,  $p_i$  knows that no winning values has been exhibited during the round, thus  $p_i$  triggers a new round by setting its estimate to the random value  $c$  picked at the beginning of the round, and  $id\_aux_i$  to  $\perp$  (Line 14). Now, if  $G_i$  only contains a non bottom value  $g$  then  $g$  is the *winning value* of the round. Process  $p_i$  must then determine whether the echoed values it has received reflect a contention among the potential winners or not. Such a contention exists if  $Id_i$  contains at least the bottom value. If  $p_i$  does not observe such a contention and if it considers itself as the winner of the competition (*i.e.*,  $Id_i = \{p_i\}$ ) then it successfully leaves the competition by returning (yes,yes), see Line 15. Meanwhile, for any *other* process  $p_j$ , if  $p_j$  suspects that  $p_i$  may have won the competition (Lines 15, 17 and 19) then  $p_j$  abandons the competition by returning (no,no) (Lines 15, 18 and 20). Now, if  $p_i$  observes a contention among the potential winners but there is no hint of the potential winner, *i.e.*,  $Id_i = \{\perp\}$  (Line 16), then if  $p_i$  is among the processes that initially proposed  $g$  it starts a new competition by returning (yes,no). It abandons the competition otherwise. If, on the other hand,  $p_i$  knows that a majority of processes have seen its estimate in the first Phase ( $id = p_i$  at Line 19) but not necessarily in the second Phase ( $\perp \in Id_i$ ), then  $p_i$  triggers a new round by specifying that there is a winning group value, but there is no hint on the potential winner. This will allow all the processes involved in this new round to return (yes,no). Finally, the last possible scenario occurs when  $p_i$  sees a contention on the group value

(*i.e.*,  $\perp \in G_i$ ) but one group  $g$  has nevertheless been seen by a majority of processes (Line 19 and 21). If  $p_i$  knows that a majority of processes have seen its estimate in the first Phase ( $id = p_i$  at Line 19) but not necessarily in the second Phase, then it triggers a new round by specifying that there is a winning group value, but there is no hint on the potential winner. This will allow all the processes involved in this new round to return (yes,no). On the other hand, in Line 21, there is a hint on the potential winning group but no hint about the winning process thus  $p_i$  triggers a new round with  $g\_est_i = g$  and  $id\_est_i = \perp$ .

It is easy to see that if there is a unique invoking process  $p_i$  during some round  $r$ , it will return (yes,yes) at line 15 of round  $r$  as  $p_i$  can only received echoes of its own value.

### C. Correctness proofs of the Selector object implementation

In this section, we show that the randomized implementation of the selector object presented in Figure 4 is correct, that is guarantees the properties given in Section III-A. We start by showing the following lemmata prior to prove the properties of the Selector implementation. Let  $s$  be a selector object.

**Lemma 1 (Non-blocking):** No correct process blocks forever in a round.

*Proof:* Let us first note that no relaying process can block forever at lines 24 or 25 and will respond to any message sent by any invoking process. By assumption, there is a majority of correct processes. Thus any invoking process that broadcasts a message at lines 4 or 10 will receive at least a majority of associated echoes (*i.e.*, PHASE messages). Consequently, no invoking process can remain blocked forever at lines 5 or 11. ■

**Lemma 2:** If all the invoking processes start a round  $r$  with the same estimate  $g$ , all the invoking processes that do not crash return either in round  $r$  or in round  $r + 1$ .

*Proof:* Let  $g$  be the winning group estimate proposed by all invoking processes at the beginning of round  $r$ . The invoking processes will broadcast the same value  $g$  at line 4, and thus will get only value  $g$  in their buffer  $G$ . Consequently, each invoking process  $p_i$  executes Line 8 by setting  $g\_aux_i$  to  $g$  and will receive (a majority of) RELAY messages with  $g\_aux = g$ . Thus each of the three cases at Lines 15, 16, and 17 need to be considered. When considering the code of these lines, one can see that whatever is the returned value, the executing process always returns. ■

**Lemma 3 (S-Validity):** A process, invoking the `s.play` primitive, that returns a value, must return (yes,yes), (yes,no) or (no,no).

*Proof:* Straightforward from Lines 15, 16, 18, and 20. ■

**Lemma 4 (S-Obligation-solo):** If a process invokes the `s.play` primitive alone and does not crash then, it returns (yes,yes).



```

Function  $s.\text{play}(g_i)$ 
(1)  $r_i \leftarrow 0; g\_est_i \leftarrow g_i; id\_est_i \leftarrow p_i;$ 
(2) while true do // Sequence of rounds
(3)  $r_i \leftarrow r_i + 1; c \leftarrow \text{common\_coin}();$ 
    ----- Phase 1 of round  $r_i$  -----
(4) bcast PHASE( $r_i, 1, g\_est_i, id\_est_i$ );
(5) wait until (PHASE( $r_i, 1, g\_est, id\_est$ ) messages have been received from a majority of
    processes);
(6) let  $G_i$  be the set of  $g\_est$  values received at line 5;
(7) let  $Id_i$  be the set of  $id\_est$  values received at line 5;
(8) if ( $G_i = \{g\}$  then  $g\_aux_i \leftarrow g$  else  $g\_aux_i \leftarrow \perp$  endif;
(9) if ( $Id_i = \{id\}$  then  $id\_aux_i \leftarrow id$  else  $id\_aux_i \leftarrow \perp$  endif;
    ----- Phase 2 of round  $r_i$  -----
(10) bcast PHASE( $r_i, 2, g\_aux_i, id\_aux_i$ );
(11) wait until (PHASE( $r_i, 2, g\_aux, id\_aux$ ) messages have been received from a majority of
    processes);
(12) let  $G_i$  be the set of  $g\_aux$  values received at line 11;
(13) let  $Id_i$  be the set of  $id\_aux$  values received at line 11;
(14) case ( $G_i = \{\perp\}$ ):  $g\_est_i \leftarrow c; id\_est_i \leftarrow \perp;$ 
(15) ( $G_i = \{g\} \wedge Id_i = \{id\}$ ): if ( $id = p_i$ ) then return (yes,yes) else return (no,no);
(16) ( $G_i = \{g\} \wedge Id_i = \{\perp\}$ ): if ( $g = g_i$ ) then return (yes,no) else return (no,no);
(17) ( $G_i = \{g\} \wedge Id_i = \{id, \perp\}$ ): if ( $id = p_i$ ) then  $g\_est_i \leftarrow g; id\_est_i \leftarrow \perp;$ 
(18) else return (no,no);
(19) ( $G_i = \{g, \perp\} \wedge Id_i = \{id, \perp\}$ ): if ( $id = p_i$ ) then  $g\_est_i \leftarrow g; id\_est_i \leftarrow \perp;$ 
(20) else return (no,no);
(21) ( $G_i = \{g, \perp\} \wedge Id_i = \{\perp\}$ ):  $g\_est_i \leftarrow g; id\_est_i \leftarrow \perp;$ 
(22) endcase;
(23) endwhile
    ----- Relay Task -----
Task  $s.\text{relay}$  // Launched by any process  $p_i$  that receives the first message related to object  $s$ 
    // It maintains four variables  $g[r, 1], g[r, 2], id[r, 1]$  and  $id[r, 2]$  per round  $r$ 
    upon the reception of PHASE( $r, x, g, id$ ) from  $p_j$ 
(24) if (this message is the first for round  $r$ , phase  $x$ ) then  $g[r, x] \leftarrow g; id[r, x] \leftarrow id$ ; endif;
(25) send PHASE( $r, x, g[r, x], id[r, x]$ ) to  $p_j$ ;

```

Figure 4. A randomized protocol implementing a selector run by process  $p_i$  ( $t < n/2$ )

*Proof:* If an invoking process  $p_i$  executes alone then necessarily the echoes it will receive during the first round at lines 5 and 11 contain a single value  $g$  and a single identifier  $p_i$  both broadcast by itself. Consequently,  $G_i$  and  $Id_i$  will contain one non- $\perp$  value each, respectively the group and the id of  $p_i$ , leading process  $p_i$  to decide (yes,yes). ■

**Lemma 5 (S-Agreement):** At most one process returns (yes,yes), and in this case, all the other returning processes return (no,no).

*Proof:* Let  $p_i$  be the first process that returns (yes,yes) at round  $r$ . By construction of the algorithm, this can only occur at Line 15, that is  $G_i = \{g\}$  and  $Id_i = \{id\}$ , with  $id = p_i$ . Thus, by the majority argument, for any other process  $p_k$ , variables  $G_k$  and  $Id_k$  must respectively contain at least  $g$  and  $id$  at round  $r$ . Consequently, process  $p_k$  necessarily executes one of the three cases at lines 15, 17 and 19, and in all three cases  $p_k$  returns (no,no) at round  $r$  as  $id \neq p_k$ . ■

**Lemma 6 (S-Exclusion):** If an invocation of  $s.\text{play}$  with parameter  $g$  returns (yes,-) then, no invocation with parameter  $\neg g$  can return (yes,-).

*Proof:* Let  $r$  be the smallest round at which some invoking process  $p_i$  returns (yes,-). By construction, it can only happen at Line 15 or 16. If  $p_i$  returns (yes,yes) (Line 15) then by Lemma 5 all the other processes, that

return a value, return (no,no) whatever is their group. Now, suppose that  $p_i$  returns (yes,no) at Line 16, and  $g_i = g$ . By construction, this means that for any other processes  $p_k$ ,  $G_k$  must contain at least  $g$  at round  $r$  (two majorities always intersect) and consequently if  $p_k$  does not crash, it will execute one of the lines 16, 17, 19 and 21. Process  $p_k$  returns (yes,no) (at Line 16) if  $g_k = g$  and thus the lemma holds. If  $p_k$  executes lines 17 or 19 it abandons the execution if  $id \neq p_k$  and thus  $g$  is not the group of  $p_k$  and the lemma also holds. In the last case,  $p_k$  triggers round  $r + 1$  with  $g\_est_k = g$ . This applies for all processes executing round  $r + 1$ . By Lemma 2, for all these processes that return a value, they return, at round  $r + 1$ . The only group estimate value being  $g$  and hence  $p_k$  returns (yes,-) only if  $g_k = g$  ( $g$  being the group of  $p_i$ , the process that returns (yes,no)) proving the lemma. ■

**Lemma 7 (S-Obligation):** If no process crashes then, at least one process returns (yes,-).

*Proof:* By contradiction. Suppose that no correct process returns (yes,-). By Lemma 1, no correct process blocks forever in a round. By Lemma 4, at least two processes must invoke  $\text{play}$  otherwise the solo execution returns (yes,yes). Let  $p_i$  and  $p_k$  be any two of these processes such that  $p_i$  invokes  $\text{play}(g)$  and  $p_k$  invokes  $\text{play}(g')$  with  $g, g' \in \{0, 1\}$ . Two cases need to be considered.

1) Suppose first that for all the invoking processes, we have  $g = g'$ . Thus all the processes execute either Line 15, Line 16, or Line 17 in Phase 2 of the protocol. By assumption, all the processes have initially proposed the same value  $g$ . Thus, none of them can return (no,no) at line 16. Now, not all of them can return (no,no) at line 18 because the process whose id matches  $id$  should trigger round 2, in which case it would execute Line 16 and return (yes,no). In all the other cases, processes return (yes,no) or (yes,yes) in round 1, which also contradicts the assumption of the lemma.

2) Suppose now that  $g = \neg g'$ .

- a) If some process  $p_k$  returns (no,no) at Line 16, then its initial value  $g_k = g'$  is not the potential winning value  $g$  and none of the other processes can execute line 14 (by the majority argument,  $g$  must belong to all the  $G_j$ s). Thus all the processes that have invoked `play` with  $g'$  return (no,no). Now, all the processes that have invoked `play` with  $g$  (by construction there must be at least one such process) cannot all return (no,no) at Lines 18 or 20 because the process whose id matches  $id$  must necessarily have executed any of the cases (except the one at Line 14 by assumption of the case). Thus such a process must have either returned (yes,-) in this round or triggered the next round with  $g\_est_i = g$  and  $id\_est_i = \perp$ , and thus shall return (yes,no) in round 2. This contradicts the assumption of the lemma.
- b) If some process  $p_k$  returns (no,no) at Line 15 or Line 18, then a similar argument as above applies.
- c) If some process  $p_k$  returns (no,no) during round  $r$  at Lines 20, then a similar argument as the above one also applies, although the case at Line 14 may also hold. Thus, as previously, all the processes that have invoked `play` with  $g$  and that trigger round  $r + 1$  do it with either  $g\_est = g$  and  $id\_est = \perp$ , or  $g\_est = \neg g$  and  $id\_est = \perp$  if  $c = \neg g$ . Suppose that at least two processes  $p_i$  and  $p_j$  execute round  $r + 1$  and one does it with  $g\_est = g$  and  $id\_est = \perp$  and the other one does it with  $g\_est = \neg g$  and  $id\_est = \perp$  (the case where a single process executes round  $r + 1$  trivially returns (yes,no)). By construction, only cases at Line 14, 16 and 21 may hold. If all these processes execute Line 14 then they all trigger a new round with the same estimate during which they will return (yes,no). Suppose now, that at Lines 16 and 21, the potential winning value is

$\neg g$  (this may happen because some process has triggered round  $r + 1$  with  $\neg g$ ). Note however that all the processes involved in round  $r + 1$  have initially invoked `play` with  $g$ . Thus all the processes that run Line 16 will return (no,no), and those that execute Line 21 will trigger a new round with  $g\_est = g$ , and will finally return in Line 16 with (yes,no). All the scenario contradict the assumption of the lemma, which completes its proof. ■

**Lemma 8 (S-Termination):** An invocation of `s.play` by a correct process terminates with probability 1.

*Proof:* Suppose by contradiction that some correct process  $p_i$  does not terminate. It must be the case that either  $p_i$  blocks forever in an execution or  $p_i$  never stops from triggering new rounds. By Lemma 1,  $p_i$  cannot block forever. Now, by Lemma 2, if all the competing processes in a given execution start a round  $r$  with the same estimate  $g$ , all the invoking processes that do not crash return either in round  $r$  or in round  $r + 1$ . By the proof of Lemma 5, if some process wins the competition at round  $r$ , that is returns (yes,yes), then all the other processes stop the execution, by returning (no,no), at round  $r$ . By Lemma 7 if all processes are correct then at least one returns (yes,-). Thus it must be the case that  $p_i$  and possibly some other processes end Phase 2 of some round  $r$  by either executing Lines 14, 17, 19 or 21. Once again, by Lemma 2, if Line 14 is executed and  $c = g$  then all processes return either in round  $r$  or in round  $r + 1$ . Thus let  $p'$  be the number of processes that trigger round  $r + 1$ , such that some of them propose  $g$  and the other ones propose  $\neg g$ . In this last case, there is a probability  $p = 1/2$  that the value kept by the process that executes Line 3 is equal to  $g$ . So, there is a probability  $p_\ell \geq 1 - 1/2^\ell$  that all non-crashed processes have the same estimate after at most  $\ell$  rounds. As  $\lim_{\alpha \rightarrow \infty} p_\ell = 1$ , it follows that, with probability 1, both invoking processes will start a round with the same estimate. Then, according to Lemma 2, they will return. ■

**Theorem 4 (Message complexity of `play()`):** The total number of messages exchanged by the randomized implementation of the selector object when concurrently invoked by  $p$  processes is  $O(np)$ .

*Proof:* As explained above, if there is a unique process that invokes the selector, it will return within a unique round. The number of messages needed is at most  $2(n + 1)$  ( $n + 1$  messages for each phase). If  $p$  processes invoke the selector, they will go through a constant number of rounds as it is the case for randomized consensus [6], [7]. During a phase, each invoking process broadcasts a message and each process responds once to each of the invoking processes. Thus, during one phase a maximum of  $p(n + 1)$  messages are exchanged. As there are two phases per round and the

total number of rounds is constant, the message complexity is  $O(np)$ . ■

## VI. CONCLUSION

This paper has presented a randomized solution to the Test&Set operation in fully asynchronous systems. This solution is built using a basic building block, called selector, whose specification, message-passing implementation and analysis are presented in this paper. The Test&Set implementation has a step complexity of  $O(\log p)$  which makes it adaptive, that is our implementation does not require to know at any time the number of processes that concurrently invoke the Test&Set object. We are not aware of any such work.

## REFERENCES

- [1] Afek Y., Gafni E., Tromp J. and Vitnyi P., Wait-free Test-and-Set. *Proc. 6th Workshop on Distributed Algorithms (WDAG, now DISC)*, pages 85-94, 1992.
- [2] Alistarh D. and Aspnes J., Sub-Logarithmic Test-and-Set Against a Weak Adversary. *Proc. 25th Int. Symposium on Distributed Computing (DISC'11)*, Springer Verlag LNCS, pp.97-109, 2011.
- [3] Alistarh D., Attiya H., Gilbert S., Giurgiu A., Guerraoui R., Fast Randomized Test-and-Set and Renaming. *24th Int. Symposium DISC*, LNCS 6343, pp. 94-108, September, 2010.
- [4] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.
- [5] Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symp. PODC*, pp. 27-30, 1983.
- [6] Chor B., Merritt M. and Shmoys D.B., Simple Constant-Time Consensus Protocols in Realistic Failure Models. *Journal of the ACM*, 36(3):591-614, 1989.
- [7] Canetti R. and Rabin T., Fast Asynchronous Byzantine Agreement with Optimal Resilience. *Proc. 25th ACM Symposium STOC*, pp. 42-51, 1993.
- [8] Fischer M., Lynch N. and Paterson M., Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(5):374-382, 1985.
- [9] Giakkoupis G., Woelfel P., On the time and space complexity of randomized test-and-set. *PODC 2012: 19-28 Proc. 31st ACM Symposium on Principles of Distributed Computing (PODC'12)*, ACM Press, pp. 19-28, 2012.
- [10] Golab W.M., Hendler D. and Woelfel P., An  $O(1)$  RMRs Leader Election Algorithm. *SIAM J. Comput.* vol. 39(7):2726-2760, 2010.
- [11] Herlihy M., Wait-free Synchronization. *ACM TOPLAS*, 13 (1):124-149, 1991.
- [12] Mostéfaoui A., Raynal M. and Tronel F., From Binary Consensus to Multivalued Consensus in asynchronous message-passing systems. *Information Processing Letters*, vol 73(5-6): 207-212, 2000.
- [13] Sericola, B. *Markov Chains: Theory, Algorithms and Applications*. Iste Series, Wiley, 2013
- [14] Tromp J. and Vitanyi P., A Protocol for Randomized Anonymous Two-process Wait-free Test-and-Set with Finite-state Verification. *Proc. 9th Int. Colloquium SIROCCO*, pp.275-291, 2002.

APPENDIX A

**Theorem 2** The expected time  $\mathbb{E}\{T_p \mid X_0 = p\}$  needed to terminate the Test&Set protocol when  $p$  processors are initially competing satisfies

$$\mathbb{E}\{T_p \mid X_0 = p\} = O(\log(p)).$$

More precisely, there exists a positive integer  $p_0$  such that, for every  $p \geq p_0$ , we have

$$\mathbb{E}\{T_p \mid X_0 = p\} \leq 2 \log(p).$$

*Proof:* Introducing the notation  $u_p = \mathbb{E}\{T_p \mid X_0 = p\}$  and  $\gamma_{p,k} = \binom{p}{k}/2^p$ , Relation (1) becomes

$$\begin{cases} u_2 = 2 \\ u_p = \frac{2^{p-1}}{2^{p-1} - 1} \left( 1 + \sum_{k=2}^{p-1} \gamma_{p,k} u_k \right), \text{ for } p \geq 3. \end{cases} \quad (2)$$

Consider first the coefficients  $\gamma_{p,k}$ . Using the Stirling formula, we have for all  $p \geq 1$ ,

$$\sqrt{2\pi} \sqrt{p} p^p e^{-p} \leq p! \leq e \sqrt{p} p^p e^{-p}.$$

We then have, for  $1 \leq k \leq p$ ,

$$\gamma_{p,k} = 2^{-p} \binom{p}{k} \leq e 2^{-p} \frac{\sqrt{p} p^p e^{-p}}{\sqrt{k} k^k e^{-k} \sqrt{p-k} (p-k)^{(p-k)} e^{-(p-k)}} = e \delta_{p,k},$$

where

$$\begin{aligned} \delta_{p,k} &= \frac{\sqrt{p}}{\sqrt{k(p-k)}} 2^{-p} p^p k^{-k} (p-k)^{-(p-k)} \\ &= \frac{\sqrt{p}}{\sqrt{k(p-k)}} \exp(p \ln(p/2) - k \ln(k) - (p-k) \ln(p-k)). \end{aligned}$$

Now, taking any fixed value of  $p \geq 3$ , the two functions  $\phi_p$  and  $\psi_p$ , defined by

$$\begin{aligned} x \in [1, p-1] &\mapsto \phi_p(x) = \sqrt{x(p-x)}, \\ x \in [1, p-1] &\mapsto \psi_p(x) = p \ln(p/2) - x \ln(x) - (p-x) \ln(p-x) \end{aligned}$$

both are increasing on  $[1, p/2]$  and decreasing on  $[p/2, p-1]$  (this is obvious for function  $\phi_p$ , while the derivative of  $\psi_p$  with respect to  $x$  is  $\psi'_p(x) = -\ln(x) + \ln(p-x) = -\ln(x/(p-x))$  which is  $\geq 0$  when  $1 \leq x \leq p/2$ , and  $\leq 0$  when  $p/2 \leq x \leq p-1$ ).

We now take a small  $\alpha$ ,  $0 < \alpha < 1/2$ , and we estimate  $\gamma_{p,k}$  for the  $k$ 's belonging to the set  $E = [2, p/2 - p^{\alpha+1/2}] \cup [p/2 + p^{\alpha+1/2}, p-1]$ . Taking  $0 < \alpha < 1/2$ , there exists a  $p_1$  such that for  $p \geq p_1$ , the two intervals forming  $E$  are non empty. For this  $\alpha$  and  $p \geq p_1$ , we have for all  $x \in E$ ,

$$\phi_p(x) \geq \sqrt{p-1} \text{ and } \psi_p(x) \leq \psi_p(p/2 - p^{\alpha+1/2}) = \psi_p(p/2 + p^{\alpha+1/2}),$$

with

$$\begin{aligned} &\psi_p(p/2 + p^{\alpha+1/2}) \\ &= p \ln(p/2) - (p/2 - p^{\alpha+1/2}) \ln(p/2 - p^{\alpha+1/2}) - (p/2 + p^{\alpha+1/2}) \ln(p/2 + p^{\alpha+1/2}) \\ &= -(p/2 - p^{\alpha+1/2}) \ln(1 - 2p^{\alpha-1/2}) - (p/2 + p^{\alpha+1/2}) \ln(1 + 2p^{\alpha-1/2}). \end{aligned}$$

Using the Taylor-Lagrange formula, we get

$$\begin{aligned} -\ln(1 + 2p^{\alpha-1/2}) &= -2p^{\alpha-1/2} + K_1 \left( 2p^{\alpha-1/2} \right)^2, \text{ with } 0 \leq K_1 \leq 1/2 \text{ for } p \geq p_2. \\ -\ln(1 - 2p^{\alpha-1/2}) &= 2p^{\alpha-1/2} + K_2 \left( 2p^{\alpha-1/2} \right)^2, \text{ with } 0 \leq K_2 \leq 1 \text{ for } p \geq p_3, \end{aligned}$$

This leads, for  $p \geq \max(p_1, p_2, p_3)$ , to

$$\begin{aligned}
& \psi_p \left( p/2 + p^{\alpha+1/2} \right) \\
&= (p/2 - p^{\alpha+1/2}) \left( 2p^{\alpha-1/2} + 4K_2 p^{2\alpha-1} \right) - (p/2 + p^{\alpha+1/2}) \left( 2p^{\alpha-1/2} + 4K_1 p^{2\alpha-1} \right) \\
&= -4p^{2\alpha} + 2(K_2 - K_1)p^{2\alpha} - (K_2 + K_1)p^{3\alpha-1} \\
&\leq -4p^{2\alpha} + 2(K_2 - K_1)p^{2\alpha} \\
&\leq -2p^{2\alpha}.
\end{aligned}$$

We thus obtain, for all  $k \in E$ ,

$$\gamma_{p,k} \leq \frac{e\sqrt{p} \exp(\psi_p(k))}{\phi_p(k)} \leq e \frac{\sqrt{p}}{\sqrt{p-1}} \exp(-2p^{2\alpha}) \leq e\sqrt{2} \exp(-2p^{2\alpha}).$$

Using this bound and introducing

$$M_p = \max_{2 \leq k \leq p} u_k,$$

we obtain from Relation (2),

$$u_p \leq \frac{1}{1 - 2^{-(p-1)}} \left( 1 + e\sqrt{2} p e^{-2p^{2\alpha}} M_p + \sum_{k: |k-p/2| \leq p^{\alpha+1/2}} \gamma_{p,k} u_k \right).$$

Since  $1/(1-x) \leq 1+2x$ , for  $0 < x \leq 1/2$  and since  $e\sqrt{2}(1+2^{-(p-2)})p e^{-p^{2\alpha}} \leq 1$ , for  $p$  large enough (i.e.  $p \geq p_4$  for some  $p_4$  whose precise value is irrelevant), we obtain

$$u_p \leq \left( 1 + 2^{-(p-2)} \right) \left( 1 + \sum_{k: |k-p/2| \leq p^{\alpha+1/2}} \gamma_{p,k} u_k \right) + e^{-p^\alpha} M_p. \quad (3)$$

We introduce a dyadic partition of the indices  $p$ , and set, for any  $j \geq 1$ , the notation

$$U_j = \max_{2 \leq k \leq 2^j} u_k.$$

We take a fixed index  $j \geq 2$ , or  $j \geq j_0$  for some  $j_0$  whose value is irrelevant, and estimate  $U_{j+1}$  as a function of  $U_j$ . To do so, we take  $p$  such that  $2^j < p \leq 2^{j+1}$  and we write

$$\begin{aligned}
\sum_{k: |k-p/2| \leq p^{\alpha+1/2}} \gamma_{p,k} u_k &= \sum_{\substack{k: |k-p/2| \leq p^{\alpha+1/2} \\ k \leq 2^j}} \gamma_{p,k} u_k + \sum_{\substack{k: |k-p/2| \leq p^{\alpha+1/2} \\ 2^j < k < p}} \gamma_{p,k} u_k \\
&\leq U_j \left( \sum_{0 \leq k \leq 2^j} \gamma_{p,k} \right) + U_{j+1} \left( \sum_{2^j < k \leq p} \gamma_{p,k} \right) \\
&= U_j s_{p,j} + U_{j+1} (1 - s_{p,j}),
\end{aligned} \quad (4)$$

where  $s_{p,j}$  is defined by

$$s_{p,j} = \sum_{0 \leq k \leq 2^j} \gamma_{p,k},$$

and we have used the fact that  $\sum_{k=0}^p \gamma_{p,k} = 1$ . Note the obvious estimate  $0 \leq s_{p,j} \leq 1$  and note also that, since  $p \in (2^j, 2^{j+1}]$ , we have

$$s_{p,j} = \sum_{0 \leq k \leq 2^j} \gamma_{p,k} \geq \sum_{0 \leq k \leq p/2} \gamma_{p,k},$$

while

$$1 = \sum_{k=0}^p \gamma_{p,k} = \sum_{0 \leq k \leq p/2} \gamma_{p,k} + \sum_{p/2 < k \leq p} \gamma_{p,k} = \sum_{0 \leq k \leq p/2} \gamma_{p,k} + \sum_{0 \leq k < p/2} \gamma_{p,k} \leq 2 \sum_{0 \leq k \leq p/2} \gamma_{p,k},$$

from which it comes

$$s_{p,j} \geq 1/2.$$

Relations (3) and (4) eventually provide

$$\begin{aligned} U_{j+1} &\leq \left(1 + 2^{-(p-2)}\right) (1 + U_j s_{p,j} + U_{j+1} (1 - s_{p,j})) + e^{-p\alpha} M_p \\ &\leq \left(1 + 2^{-(p-2)}\right) \left(1 + \frac{U_j + U_{j+1}}{2}\right) + e^{-p\alpha} M_p, \end{aligned}$$

where we have used  $s_{p,j} \geq 1/2$  together with  $U_j \leq U_{j+1}$ . In other words, and taking into account  $p \in (2^j, 2^{j+1}]$ , we have

$$U_{j+1} \leq \left(1 + 2^{-(2^j-2)}\right) \left(1 + \frac{U_j + U_{j+1}}{2}\right) + e^{-2^j\alpha} U_{j+1}.$$

Hence we arrived at

$$U_{j+1} \leq \frac{1 + 2^{-(2^j-2)}}{1 - 2^{-(2^j-2)} - 2e^{-2^j\alpha}} (2 + U_j). \quad (5)$$

For later convenience, we rewrite (5) in a more convenient form. To do so, we fix some  $\beta$  such that  $0 < \beta < \alpha$  and we introduce

$$\beta_j = e^{-2^j\beta}.$$

It is clear that for  $j$  large enough ( $j \geq j_1$  for some  $j_1$ ), Relation (5) implies that

$$U_{j+1} \leq (1 + \beta_j)(2 + U_j). \quad (6)$$

We are now in position to conclude. Formula (6) implies that

$$\begin{aligned} U_{j+1} &\leq 2 \left[ (1 + \beta_j) + (1 + \beta_j)(1 + \beta_{j-1}) + \cdots + (1 + \beta_j)(1 + \beta_{j-1}) \cdots (1 + \beta_{j_1}) \right] \\ &\quad + (1 + \beta_j)(1 + \beta_{j-1}) \cdots (1 + \beta_{j_1}) U_{j_1}. \end{aligned}$$

Introducing the quantities

$$\Pi_j := \prod_{k=j_0-1}^j (1 + \beta_k),$$

the above bound rewrites

$$U_{j+1} \leq 2 \left[ \frac{\Pi_j}{\Pi_{j-1}} + \frac{\Pi_j}{\Pi_{j-2}} + \cdots + \frac{\Pi_j}{\Pi_{j_1}} \right] + \frac{\Pi_j}{\Pi_{j_1}} U_{j_1}.$$

Hence, since the infinite product  $\prod_{j \geq j_1} (1 + \beta_j)$  clearly converges, we have  $\Pi_j \rightarrow \Pi > 0$  as  $j \rightarrow \infty$  and we may write,

$$U_{j+1} \leq 2\Pi_j \sum_{\ell=j_1}^{j-1} \frac{1}{\Pi_\ell} + \frac{\Pi_j}{\Pi_{j_1}} U_{j_1}.$$

We denote by  $\tilde{U}_j$  the right hand side of this last inequality. It is clear, using a standard fact about diverging series, that

$$\begin{aligned} \Pi_j \sum_{\ell=j_1}^{j-1} \frac{1}{\Pi_\ell} &\underset{j \rightarrow \infty}{\sim} \Pi \sum_{\ell=j_1}^{j-1} \frac{1}{\Pi} \underset{j \rightarrow \infty}{\sim} j, \\ \frac{\Pi_j}{\Pi_{j_1}} U_{j_1} &\underset{j \rightarrow \infty}{\sim} \frac{\Pi}{\Pi_{j_1}} U_{j_1}. \end{aligned}$$

We deduce that

$$U_{j+1} \leq \tilde{U}_j \quad \text{with } \tilde{U}_j \underset{j \rightarrow \infty}{\sim} 2j. \quad (7)$$

In particular, defining  $\tilde{u}_p = \tilde{U}_j$ , for  $2^j < p \leq 2^{j+1}$ , we easily deduce that

$$u_p \leq \tilde{u}_p \quad \text{with } \tilde{u}_p \underset{p \rightarrow \infty}{\sim} 2 \log(p). \quad (8)$$

This completes the proof. ■