A Message-Passing and Adaptive Implementation of the Randomized Test-and-Set Object

Emmanuelle Anceaume*, François Castella†, Achour Mostéfaoui‡ and Bruno Sericola§
*IRISA / CNRS (France), emmanuelle.anceaume@irisa.fr
†IRMAR / Université de Rennes 1 (France), francois.castella@univ-rennes1.fr
‡LINA / Université de Nantes (France), achour.mostefaoui@univ-nantes.fr
§INRIA Rennes Bretagne Atlantique (France), bruno.sericola@inria.fr

***Abstract*—This paper presents a solution to the well-known Test-and-Set operation in asynchronous systems prone to process crashes. Test-and-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 and Set objects have been achieved, however all of them uniquely target the shared memory model. In this paper we propose an implementation of a Test-and-Set object for message passing distributed systems. This implementation can be invoked by any number p of processes. It has an expected step complexity in O(p) and an expected message complexity in O(np), where n is the total number of processes in the system. The proposed Test and Set object is built atop a new basic building block that allows to select a winning group among two groups of processes.**

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

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 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 [13], consensus[1] is harder to solve than Test&Set, that is a solution to Test&Set does not allow to solve consensus, but Test&Set is still too hard to be solved in a pure asynchronous system [6]. 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 [13]. For instance, a simple way to solve the Test&Set problem is to use a multivalued consensus service, such that each process proposes its identity to the consensus and decides a value. If the decided identity is its own identity it returns yes otherwise, it returns no.

This work was partially funded by the French ANR project AMORES (ANR-11-INSE-010).

[1]In the consensus problem, each process proposes a value and every correct processes eventually decides a value, such that a unique value is decided and this value has been proposed by at least one process.

Contributions: In this paper we propose a randomized and distributed message-passing implementation of the Test&Set operation. Regarding randomization, Herlihy [13] has shown that Test&Set 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 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. The adaptive adversary is stronger than the oblivious one and has led to space expensive known implementations [3] or non adaptive ones [1] in the shared memory model. Regarding communication, all the efficient solutions have, so far, been implemented with shared registered [1]–[3], [11] arguing that one can automatically transform shared-memory based algorithms to message-passing ones [5]. Of course, this automatic transformation does not necessarily keep the efficiency property of the shared-memory based algorithms. Table I proposes a summary of the important results of the Test&Set object implemented in the shared memory model. The first column gives the *step complexity*. For shared memory systems, it represents the maximum number of steps (read/write) needed by any process in expectation to complete its execution. The step complexity, for message-passing systems, is the size of the longest causal sequence of messages that is needed for a process to complete its execution. A shared register can be implemented in a message-passing system [5]. A read/write operation on a shared register needs $\Omega(n)$ messages and a constant number of communication steps. The step complexity is thus the same for both systems. In Table I, the *space complexity* refers to the number of

shared multi-writer/multi-reader atomic registers used in the implementation of the Test&Set object. It is interesting to notice that all the best solutions [1]–[3], [11] implemented in the shared memory model require at least a number of atomic registers linear in the total number of processes n in the system. Actually Giakkoupis and Woelfel [11] have proven that at least $\log n$ atomic registers are needed for any randomized register-based Test&Set implementation. The *message complexity* column represents the number of messages needed in expectation to complete the execution of all none crashed processes. Note that it is possible to give the message complexity for the shared-memory solutions assuming the shared registers are implemented over a message-passing system [5]. To obtain this message complexity, we need the total number of read/write operations executed in expectation (the column "total step complexity") knowing that each read/write operation needs $O(n)$ messages.

In this work, and in contrast to the aforementioned solutions, we propose a message-passing implementation of the Test&Set operation. Our implementation can be invoked by any number $p \leq n$ of processes. It has an expected step complexity in $O(\log p)$ and an expected message complexity in $O(np)$ against an oblivious adversary. These complexities assume the scheduling of the worst adversary taken from the oblivious family. Having a step complexity that depends on p and not on the number of processes n of the system makes our solution *adaptive*. The implementation we propose of the Test&Set object goes through a series of calls to a basic building block that we call in the following *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 in presence of an oblivious adversary. The step complexity of the selector implementation is constant. A variant of the *GroupElect* object proposed by Woelfel and Giakouppis [11] would provide a shared memory implementation of the *selector* object in presence of an oblivious adversary.

Road map: In the remaining of the paper, Section II presents the underlying model and specifies the Test&Set problem. Section III presents the selector object, proposes a randomized implementation of this object whose correctness is demonstrated, and derives its message complexity. Section IV presents a randomized implementation of the Test&Set object, demonstrates the correctness of this implementation, and presents both its message and step complexity. Finally Section V concludes.

II. Computation Model and Problem Definition

A. Computation Model

We consider an asynchronous system consisting of a set Π of n processes, namely, $\Pi = \{p_1, p_2, \ldots, p_n\}$. 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. 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 point-to-point communication primitive *send*, and 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. Finally, 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.

B. The Randomized Test&Set Problem

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 `Test&Set`, which can be invoked by any process p_i using o.`Test&Set`(). 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.`Test&Set` primitive, that returns a value must return either `yes` or `no`.
- **TS-Obligation**: If no process crashes then, exactly one process returns `yes`.
- **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.`Test&Set` primitive terminates with probability 1.

Moreover, the different calls to the Test&Set operations need to be linearizable. It has been proved in [12] that any object that satisfies the properties cited above can be used together to implement a linearizable Test&Set object. Consequently, we will not worry about linearizability.

III. A New Construction: the Selector Object

The key technical idea of our work relies on the *selector*, a new distributed structure that is used as a building block

Test&Set Protocol	Step complexity	Total step complexity	Message complexity	Space complexity	Adversary	Adaptive	
						step	space
Afek et al. 1992 [1]	$O(\log n)$	$O(n \log n)$	$O(n^2 \log n)$	$O(n)$ registers	adaptive	no	no
Alistarh et al. 2010 [3]	$O(\log p)$	$O(p \log^2 p)$	$O(np \log^2 p)$	$\Theta(n^3)$ registers	adaptive	yes	no
Giakkoupis and Woelfel 2012 [11]	$O(\log p)$	$O(p \log^2 p)$	$O(np \log^2 p)$	$O(n)$ registers	adaptive	yes	no
Alistarh and Aspnes 2011 [2]	$O(\log \log n)$	$O(n)$	$O(n^2)$	$\Theta(n^3)$ registers	oblivious	no	no
Giakkoupis and Woelfel 2012 [11]	$O(\log^* p)$	$O(p \log^* p)$	$O(np \log^* p)$	$O(n)$ registers	oblivious	yes	no
This paper	$O(\log p)$	$O(p)$	$O(np)$ messages	-	oblivious	yes	-

Table I
COMPLEXITIES OF TEST&SET OBJECTS IN BOTH SHARED MEMORY AND MESSAGE-PASSING MODELS

for the implementation of the randomized Test&Set object. As will be shown in the sequel, the message complexity of the selector object invoked by p processes requires $O(np)$ messages, and has a constant step complexity, *i.e.*, in average the number of round executed by each competing process is 2. The following section presents this new construction.

A. Specification of the Selector Object

The selector object proposes a unique access primitive `play()`, which is invoked with a Boolean parameter g ($g = 0$ or $g = 1$). Each of the two binary values represents a group, *i.e.*, group 0 or 1. A process randomly chooses its group 0 or 1 each time it invokes primitive `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 winning group, and the second one indicates whether the invoking process is also the winner in the group. Table II shows the four 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 five properties.

- **S-Validity**: If a process invokes the s.`play` primitive and returns then, it returns either `(yes,yes)`, `(yes,no)` or `(no,no)`.
- **S-Obligation-solo**: If a correct process invokes s.`play` alone (solo execution) then, it returns `(yes,yes)`.
- **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.

B. A Message-Passing Implementation of the Selector

The implementation of the selector object falls under the impossibility result of many agreement problems in the context of asynchronous distributed systems prone to process failures [10]. We thus consider an asynchronous message-passing distributed system augmented with a random oracle to circumvent the impossibility result. Specifically, processes have access to a function `common_coin()`, which provides to all the invoking 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, and thus all the processes get the same value. Our solution is an adaptation of Ben-Or consensus algorithm [7] and can be seen as a variant of the commit/abort mechanism introduced in the message-passing model in [14] coupled with a random generator to circumvent the impossibility result. The same approach has also been followed in the Test&Set algorithm by Tromp and Vitany [16] in the context of shared memory systems, except that their algorithm can be invoked by at most two processes. Woelfel and Giakouppis [11] propose the *GroupElect* object in the context of shared-memory. Similarly to our solution, the GroupElect object supposes an oblivious adversary and uses random numbers, however, and in contrast to our solution, a process can never know whether it is the only winner of the election or not.

Operationally, the selector is attached to distributed objects. Let us consider a selector s. As previously described, 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 [6].

The algorithm is round based. It is presented in Figure 1. 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 1) 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 collective memory [6]. 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. 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 1). 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. 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 groups (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

Output	Meaning
(yes,yes)	p_i's group wins and p_i is the winner in the group
(yes,no)	p_i's group wins and there is no winner in the group
(no,no)	either the group of p_i looses or there is a winner in p_i's group but the winner is not p_i
(no,yes)	Impossible, p_i cannot be a winner if its group does not win

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

Function s.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) $r_i \leftarrow r_i + 1$; $c \leftarrow$ 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 were 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** $g\_aux_i \leftarrow \perp$;
(9) **if** $(Id_i = \{id\}$ **then** $id\_aux_i \leftarrow id$ **else** $id\_aux_i \leftarrow \perp$;
——— 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 were 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**
(15) $(G_i = \{\perp\})$: $g\_est_i \leftarrow c$; $id\_est_i \leftarrow \perp$;
(16) $(G_i = \{g\}) \wedge (Id_i = \{id\})$: **if** $(id = p_i)$ **then**
(17) **return** (yes,yes);
(18) $(G_i = \{g\}) \wedge (Id_i = \{\perp\})$: **if** $(g = g_i)$ **then**
(19) **return** (yes,no)
(20) **else return** (no,no);
(21) $(G_i = \{g\}) \wedge (Id_i = \{id, \perp\})$: **if** $(id = p_i)$ **then**
(22) $g\_est_i \leftarrow g$; $id\_est_i \leftarrow \perp$;
(23) **else return** (no,no);
(24) $(G_i = \{g, \perp\}) \wedge (Id_i = \{id, \perp\})$: **if** $(id = p_i)$ **then**
(25) $g\_est_i \leftarrow g$; $id\_est_i \leftarrow \perp$;
(26) **else return** (no,no);
(27) $(G_i = \{g, \perp\}) \wedge (Id_i = \{\perp\})$: $g\_est_i \leftarrow \perp$; $id\_est_i \leftarrow \perp$;
(28) **endcase**;
(29) **endwhile**
——— Relay Task ———
Task s.relay // *Launched by any process* p_i, $1 \leq i \leq n$. p_i *maintains four variables* $g[r, 1]$, $g[r, 2]$, $id[r, 1]$ *and* $id[r, 2]$ *per round* r, *Initially initialized to* $\perp$ //
(30) **upon receipt of** PHASE(r, x, g, id) message **from** p_j
(31) **if** $(g[r, x] = \perp$ and $id[r, x] = \perp)$ **then**
(32) $g[r, x] \leftarrow g$; $id[r, x] \leftarrow id$;
(33) *send* PHASE$(r, x, g[r, x], id[r, x])$ to p_j;

Figure 1. A randomized protocol implementing the selector object run by process p_i ($t < n/2$)

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_est_i to $\perp$ (Line 15). 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 p_i is actually the winner of the competition (*i.e.*, $Id_i = \{p_i\}$) then it successfully leaves the competition by returning (`yes,yes`), see Line 17. Meanwhile, for any *other* process p_j, if p_j suspects that p_i may have won the competition (Lines 21 and 24) then p_j abandons the competition by returning (`no,no`) (Lines 23 and 26). 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 18), then if p_i is among the processes that initially proposed g it starts a new competition by returning (`yes,no`), see Line 19. It abandons the competition otherwise, see Line 20. 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 24) 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`). 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 24 and 27). If p_i knows that a majority of processes have seen its estimate in the first Phase ($id = p_i$ at Line 24) 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 the new round to return (`yes,no`). On the other hand, in Line 27, there is no hint on the potential winning process thus p_i triggers a new round with g_est_i and id_est_i both equal to $\perp$. Finally, it is easy to see that if there is a unique invoking p_i during some round r, p_i will return (`yes,yes`) at line 17 of round r as p_i can only received echoes from its own value.

C. Correctness of the Selector Implementation

We now show that the randomized implementation of the selector object presented in Figure 1 is correct, that is guarantees the properties given in Section III-A. 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 line 31 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 echo (*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 then, all the invoking processes that do not crash return either in round r or in round $r+1$.

Proof: Let g be the 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 affecting g_aux_i to g and will receive (a majority of) PHASE messages with $g\_aux = g$. Thus each of the three cases at Lines 16, 18, and 21 need to be considered. Let us examine the two former ones: p_i returns (`yes,yes`) if it is the only invoking process seen by a majority of processes and, returns (`yes,no`) if the contention between the invoking processes has been detected (that is not all the relaying processes have received the same estimations). Now consider the case at Line 21. If p_i is not the invoking process seen by a majority of relays during the first phase of the current round, then p_i returns (`no,no`), otherwise p_i triggers round $r+1$ with $g\_est_i = g$ and $id\_est = \perp$, and will exclusively execute Line 18. Consequently, p_i will return (`yes,no`) in Phase 2 (of round $r+1$). ■

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 16, 18, 23, and 26. ■

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

Proof: If an invoking process p_i executes alone a given round then necessarily the echoes it will receive at lines 5 and 11 contain a single value g and its identifier p_i. Consequently, G_i will always contains a single non-$\perp$ value 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 17, 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 two cases at Lines 21 and 24, and in both cases p_k returns (`no,no`) at round r.

■

Lemma 6 (S-Exclusion): If an invocation of s.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 happens at Line 17 or 19. If p_i returns (yes,yes) (Line 17) then by Lemma 5 all the other processes, that return a value, return (no,no). Now, suppose that p_i returns (yes,no) at Line 19, 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 majority always intersect). Thus, if p_k returns (yes,no) then necessarily $g_k = g$ and thus the lemma holds. Now, if p_k does not abandon the execution, then 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 $r+1$, (yes,−) only if $g_k = g$. ■

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

Proof: For space reasons, proof of the lemma is presented in the companion paper [4]. ■

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 15, 21, 24 or 27. Once again, by Lemma 2, if Line 15 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 none crashed processes have the same estimate after at most ℓ rounds. As $\lim_{\ell\to\infty} p_\ell = 1$, 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. ■

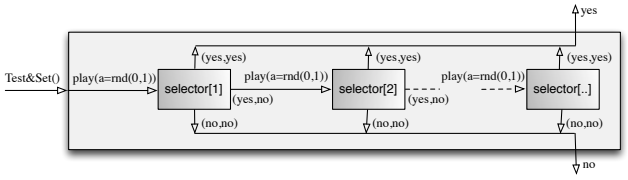
D. Complexity Analysis of the Selector Implementation

Theorem 1 (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: 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 [8], [9]. During a phase, each invoking process broadcasts a message and each process responds once to each of the invoking processes. Thus, during a 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, message complexity is $O(np)$. ■

IV. A Message-Passing Implementation of the Randomized Test&Set Object

We now present the implementation of the Test&Set object. Recall that it can be invoked by any number $p \leq n$ of competing processes. As aforementioned, implementation of the Test&Set object relies on instances of the selector object as illustrated in Figure 2(a). Correctness of the implementation is presented in Section IV-B, and its complexity is derived in Section IV-C.



(a) Randomized Test&Set object using selector objects as building blocks

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** $(b_1 =$ yes $\wedge\ b_2 =$ yes$) \vee (b_1 =$ no$)$ **then**
(5) **return**$(b_1 =$ yes $\wedge\ b_2 =$ yes$)$
(6) **endif**;
(7) $step_i \leftarrow step_i + 1$;
(8) **endrepeat**;

(b) Randomized Test&Set Object Algorithm

Figure 2. Randomized Test&Set Object

A. The Randomized Test&Set Algorithm

Pseudo-code of the Test&Set algorithm, given in Figure 2(b), 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 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 above, our algorithm does not need to know how many processes access the Test&Set object. Remaining of the paper will clarify all these points.

B. Correctness of the Test&Set Implementation

This section proves the correction of the algorithm of Figure 2(b) 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.

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, at least some process will exit with `(yes,-)`. If only `(yes,no)` is returned, then all these processes will execute a new instance of the selector until possibly exactly one process execute a solo execution in which case it will return `(yes,yes)`. 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-C 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).

C. Complexity Analysis of the Test&Set Implementation

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

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 the analysis supposes the worst case execution, where the Test&Set protocol executes until there is a unique competing process. Consequently, for the purpose of this worst case analysis, only the first boolean returned by the selector object is relevant. Recall that this boolean indicates if the invoking process belongs to the winning group or not.

When competing processes invoke a selector instance, each one chooses a group at random (line 3 of Figure 2(b)). 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, we assume that the two events "group 0 wins" and "group 1 wins" occur with the same probability $1/2$ and that the behaviors 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 behavior of the algorithm can be modeled by a Markov chain $X = \{X_\ell, \ell \geq 1\}$, where X_ℓ represents the number of processes in competition at the ℓ-th transition, *i.e.*, the number of processes that execute the ℓ-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 if $i < j$. Indeed, a process that returns b_1=`no` cannot any more continue the competition (see line 5 in Figure 2(b)). 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 $i = 1, \ldots, p$,

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

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 $2, 3 \ldots, p$ are thus transient states and state 1 is absorbing since $P_{1,1} = 1$.

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 sequence $n_1, n_2, \ldots, n_{p-1}$ where n_ℓ represents the number of processes that execute step ℓ of the Test&Set protocol (contention on step ℓ). By assumption $n_1 = p$. We call the total contention on the whole selector objects the sum $n_1 + n_2 + \cdots + n_{p-1}$. We show 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 this time, we have

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

It is well-known, see for instance [15], 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, α is the row vector containing the initial probabilities of the transient states, that is $\alpha_p = 1$ and $\alpha_i = 0$ for $i = 2, \ldots, p-1$, 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 [15],

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

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 an integer $p_0 > 0$ such that, for all $p \geq p_0$, we have

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

where log denotes the logarithm function to the base 2.

Proof: Sketch of the proof. For space reasons the full proof appears in the companion paper [4]. 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

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

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

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

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. ■

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 ℓ of the protocol when p processes are initially competing. This random variable is defined by $W_\ell(p) = \sum_{i=2}^{p} i1_{\{X_\ell = i\}}$. 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

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

Proof: Since $X_0 = p$, we have, for $\ell \geq 0$,

$$\mathbb{E}\{W_\ell(p)\} = \sum_{i=2}^{p} i\mathbb{P}\{X_\ell = i \mid X_0 = p\} = \sum_{i=2}^{p} i\left(Q^\ell\right)_{p,i}.$$

For $\ell = 0$, we have $\mathbb{E}\{W_0(p)\} = p$. For $\ell \geq 1$,

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

We pursue by recurrence over index ℓ. 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 $p \geq 2$,

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

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

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

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

Theorem 4 (Message complexity of Test&Set()): The total number of messages exchanged by the randomized implementation of the Test&Set object when concurrently invoked by $p \leq n$ processes is $O(np)$.

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

V. Conclusion

In this paper we have presented a randomized solution to the Test&Set operation in fully asynchronous systems prone to crash failures. This solution is built using a new building block, called selector. This solution has an adaptive step complexity. From a practical point of view, this property is very important as it guarantees that the Test&Set operation on the attached distributed object solely depends on the number of processes p that concurrently want to access this object, and not on the size n of the system. Finally, the total number of messages involved by this operation is $O(np)$, which improves upon all the existing adaptive implementations.

References

[1] Y. Afek, E. Gafni, J. Tromp, and P. Vitnyi. Wait-free test-and-set. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG, now DISC)*, pages 85–94, 1992.

[2] D. Alistarh and J. Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2011.

[3] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2010.

[4] E. Anceaume, F. Castella, A. Mostefaoui, and B. Sericola. A message-passing and adaptive implementation of the randomized test-and-set object. https://hal.archives-ouvertes.fr/hal-01075650, 2015.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[6] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

[7] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

[8] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 42–51, 1993.

[9] B. Chor, M. Merritt, and D.B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, 1989.

[10] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(5):374–382, 1985.

[11] G. Giakkoupis and P. Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2012.

[12] W.M. Golab, D. Hendler, and P. Woelfel. An $o(1)$ rmrs leader election algorithm. *SIAM Journal on Computing*, 39(7):2726–2760, 2010.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[14] A. Mostéfaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: a general quorum-based approach. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 49–63, 1999.

[15] B. Sericola. *Markov Chains: Theory, Algorithms and Applications*. Iste Series, Wiley, 2013.

[16] J. Tromp and P. Vitanyi. A protocol for randomized anonymous two-process wait-free test-and-set with finite-state verification. In *Proceedings of the ACM International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 275–291, 2002.