Long-Lived Test-And-Set Using Bounded Space*

Jaap-Henk Hoepman[1]

Department of Computer Science, University of Twente, the Netherlands
hoepman@cs.utwente.nl

Abstract This paper studies the problem of implementing a shared memory test-and-set object using only shared registers. Our contribution is threefold. First, we present a general framework to allow reasoning about reusing one-shot shared memory objects in the construction of bounded space long-lived objects. Then we derive general theorems about test-and-set objects that simplifies reasoning about their implementation. Finally we show the validity of our approach by constructing an n process long-lived test-and-set object from $n + 1$ one-shot test-and-set objects, and proving this construction formally correct.

1 Introduction

A test-and-set object is a shared memory synchronisation primitive that shares a single token among a collection of n concurrent processors. Processors can request the token by invoking the test-and-set operation on the object. The result (either *won* or *lost*) indicates whether the process was successful in acquiring the token. A successful processor must return the token by invoking the reset operation. This allows other processors to acquire the token. The object guarantees that at any time only one processor holds the token. This processor can then execute a critical section in which it is free from interference by other processors that are waiting to obtain the token. Fairness is not guaranteed however: a single processor may always obtain the token while another processor fails to do so infinitely often.

We are interested in implementing such a test-and-set object efficiently, both in time and space, in the shared memory model using only read-write registers. The implementation should be crash-failure resilient, i.e., should be *wait-free* [Her91]. Because 2-processor test-and-set can be used to implement 2-processor consensus, deterministic implementations of a wait-free test-and-set object using only read-write registers are known not to exist [LAA87, Her91]. We therefore focus on randomised solutions.

Afek *et al.* [AGTV92] were the first to present a direct implementation of n-processor test-and-set using read-write registers. Previous implementations go by way of a general construction of a concurrent object [Her91], and therefore are less efficient.

* Id: test-and-set.tex,v 1.16 1999/09/10 13:03:22 hoepman Exp hoepman

Actually, Afek *et al.* construct the n-processor test-and-set in three stages. The first stage is a randomised implementation of a 2-processor test-and-set using 2 single-writer shared registers due to Tromp and Vitányi [TV91a, TV91b]. The second stage is the (deterministic) construction of a n-processor *one-shot* test-and-set (which can give the token only once to a single processor, and which cannot be reset) using a tournament-tree of 2-processor test-and-set objects as a primitive. The third and last stage builds a multi-use, long-lived, test-and-set object from a one-shot test-and-set object by adding fields to the shared registers. Their construction is unbounded, but it is claimed that the space can be bounded using a (modified version of a) *sequential time stamping scheme* [IL87]. These use $\Omega(2^n)$ labels, and hence each single-writer multi-reader register contains $\Omega(n)$ bits. The protocol uses n of those registers. None of the, quite complex, constructions are formally proven correct.

Our results are as follows.

- We present a general framework to allow reasoning about reusing one-shot shared memory objects in the construction of bounded space long-lived objects.
- We derive general theorems about test-and-set objects that simplifies reasoning about their implementation.
- We show the validity of our approach by constructing an n process long-lived test-and-set object from $n+1$ one-shot test-and-set objects, and proving this construction formally correct.

We use multi-writer registers, which add a logarithmic factor to the space complexity when compared with single-writer multi-reader registers [IS92]. Reading a multi-writer register takes a linear factor more time than reading a single-writer register. Writing takes the same amount of time for both types of registers.

The time complexity of our test-and-set operation is an expected $O(\log n)$ operations on multi-writer registers. The time complexity of our reset operation is $5n$ operations on multi-writer registers. Our construction uses $4n^2+4$ (multi-writer) registers of 2 bits each, plus n single writer registers and 1 (multi-writer) register containing $\lceil \log n + 1 \rceil$ bits each. Both time and space complexity are comparable to the results of Afek *et al.* . Our construction is formally proven correct.

The paper is organised as follows. General theorems about test-and-set objects, how to re-use one-shot objects and the concept of linearizability [HW90] are discussed in Section 2. We then show, in Section 3 how to build a long-lived test-and-set object using $n+1$ one-shot test-and-set objects. In the appendix we show how a simple tournament can be used to implement n-process one-shot test-and-set using 2-process one-shot test-and-set objects.

2 Preliminaries

2.1 Specifying Test-And-Set Objects

We assume familiarity with the concept of *linearizability* as defined by Herlihy [HW90, Her91]. We summarise the main concepts for the sake of self containment.

A shared memory object is a data-structure stored in shared memory that may be accessed by several, sequential, processors concurrently. Let P be the set of processors sharing the object. Such an object defines a set of *operations* $\mathcal{O}$ which provide the only means for a processor to modify or inquire the state of the object. Each operation $O \in \mathcal{O}$ takes zero or more parameters p on its invocation and returns a value r as its response ($r = O(p)$). To make the object $\mathcal{X}$ on which O operates explicit, we may occasionally write $\mathcal{X}.O(p)$. Each execution of an operation is called an *action*. For an action A, we denote by $t_I(A) \geq 0$ its invocation time and by $t_R(A) > t_I(A)$ its response time. For incomplete actions, $t_R(A) = \infty$ and the response of the action is unknown. We write $A \in \mathcal{X}$ if action A involves an operation on $\mathcal{X}$.

For test-and-set objects, $\mathcal{O}$ contains, for each processor $p \in P$, the parameterless function $\mathit{TestAndSet}_p()$ returning either *won* or *lost*, and the parameterless procedure $\mathit{Reset}_p()$ (not returning any value), which can only be called by a process whose previous operation was a won test and set. Single shot test-and-set objects do not have the reset operation, and allow each processor p to execute $\mathit{TestAndSet}_p()$ at most once.

A *run* over the object is a tuple $\langle \mathcal{A}, \rightarrow \rangle$ with actions $\mathcal{A}$ and partial order $\rightarrow$ such that for $A, B \in \mathcal{A}$, $A \rightarrow B$ iff $t_R(A) < t_I(B)$.[1] Two actions A, B *overlap* $(A \| B)$ if neither $A \rightarrow B$ nor $B \rightarrow A$. We write $A \not\leftarrow B$ for $\neg(B \rightarrow A)$. Runs start in some initial state where no processor is accessing the object, and can have infinite length[2]. A run captures the externally observable behaviour of the object. A run is *wellformed* if processors invoke actions sequentially.

Objects may impose extra restrictions, e.g. on when certain operations can be called, before their runs are called well-formed. Test-and-set objects, for instance, require that the $\mathit{Reset}()$ operation is called only by the winner of the test-and-set.

A wellformed run is called *complete*, if all actions that start in it have finished in it, i.e., if no crashes occurred. A non-complete run can be completed by including an arbitrary response for each pending action, where the inserted responses are ordered after all other actions in the run, and ordered arbitrarily among each other[3]. This complete run is called the *completion* of the run. A

[1] Hence $\rightarrow$ is an interval order, i.e., a transitive binary relation where $A \rightarrow B$ and $C \rightarrow D$ implies $A \rightarrow D$ or $C \rightarrow B$.

[2] We do not consider runs that start with one or more processors allready accessing the object.

[3] Pending actions may have had an effect on other concurrent actions, and therefore should be ‘serialised’ in time before these actions in the corresponding sequential

run may have many completions (by using all possible values for the unknown responses).

The *sequential specification* S describes the desired behaviour of an object. This specifies the set of possible states of the object, its initial state $\mathit{init}(S)$, and for each operation its effect on the state and its (optional) response. We write $(s, r = O(p), s') \in S$ if invoking O with parameters p in state s changes the state of the object to s' and returns r as its response.

Consider, as an example, the sequential specification S_{TI} of a test and set object. The state of a test-and-set object is a single variable *owner* whose value is either $\perp$ or a processor number. Initially $\mathit{owner} = \perp$. Furthermore, for all processors p, q with $p \neq q$,

$$(\mathit{owner} = \perp, \mathit{TestAndSet}_p() = \mathit{won}, \mathit{owner} = p) \in S_{TS} \ , \tag{TS1}$$

$$(\mathit{owner} = q, \mathit{TestAndSet}_p() = \mathit{lost}, \mathit{owner} = q) \in S_{TS} \ \text{, and} \tag{TS2}$$

$$(\mathit{owner} = p, \mathit{Reset}_p(), \mathit{owner} = \perp) \in S_{TS} \ . \tag{TS3}$$

We will write $\mathit{owner}(TS)$ for the state of test-and-set object TS. One-shot test-and-set objects do not define a *Reset* operation. Their sequential specification is described by $(TS1)$ and $(TS2)$ only.

A *sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$ *corresponding* to a complete run $\langle \mathcal{A}, \rightarrow \rangle$ is an infinite sequence $s_1 A_1 s_2 A_2 \ldots$, where $\bigcup_i \{A_i\} = \mathcal{A}$, where s_i a state of the object as in the sequential specification of the object, and $\Rightarrow$ a total order over $\mathcal{A}$ extending $\rightarrow$ (i.e., $A \rightarrow B$ implies $A \Rightarrow B$) defined by $A_i \Rightarrow A_j$ if and only if $i \leq j$. A sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$ *satisfies* a sequential specification S if and only if for all $i \geq 1$, $(s_i, A_i, s_{i+1}) \in S$ and $s_1 = \mathit{init}(S)$. In other words, the sequential execution corresponding to a run is a run in which no two actions are concurrent but in which the 'observable' order of actions in the run is preserved.

Definition 2.1. *A wellformed run* $\langle \mathcal{A}, \rightarrow \rangle$ *over an object is* linearisable w.r.t. sequential specification S*, if for at least one of its completions, there exists a corresponding sequential execution* $\langle \mathcal{A}, \Rightarrow \rangle$ *satisfying* S*.*

An object is linearisable w.r.t. its sequential specification S if all well-formed runs over the object started in a properly initialised state are linearisable w.r.t. S.

We have the following lemma

Lemma 2.2. *An object is linearizable w.r.t. its sequential specification* S *if all* complete *runs over the object started in a properly initialized state are linearizable w.r.t.* S*.*

Proof. Each incomplete run is a prefix of a complete run over the object where all crashed processors were actually delayed until the end of the incomplete run. □

This allows us to assume w.l.o.g. that all runs are complete.

execution. In order to verify that the sequential specification is satisfied, the response of this action must be known.

2.2 Implementing Objects

The implementation $\mathcal{I}$ of a compound object from lower level ones implements each of the compound operations by a sequential procedure, that calls low level operations or does some local computations. For example, one can implement a multi writer register using only single writer registers by specifying for both the read and the write operation on the multi writer register a sequential procedure calling reads and writes on the single writer registers. Let $init(\mathcal{I})$ denote the necessary initial state of these lower level objects. Such an implementation is *wait-free* if each operation completes in a priori bounded number of steps, irrespective of the other processors (which may all have crashed).

Another way to view an implementation $\mathcal{I}$ of an object X is the following. An implementation of X restricts (using the properties of the low level objects it uses, their initial states described by $init(\mathcal{I})$, and the 'implementation' of each operation by a sequential procedure operating on these low level objects) the externally observable behaviour of X (i.e., the runs over X) to (a subset of)[4] all possible runs over the object. These runs are called the runs over $\mathcal{I}$.

Lamport [Lam86] presents a much more thorough and formal approach to implementing objects from lower level ones. For our purposes the above informal discussion and the following definition of a correct implementation suffice.

Definition 2.3. *The implementation $\mathcal{I}$ of an object X is* correct *if all wellformed runs over $\mathcal{I}$ are linearizable w.r.t. the sequential specification S_X of the object X.*

Note that we only require the wellformed runs to be linearizable. This is because wellformedness cannot be guaranteed by the implementation: only the user of an object controls when and where to invoke a certain operation on an object.

2.3 Re-using Short-Lived Objects

Straightforward implementation of a long-lived object (that can be accessed repeatedly) from short-lived ones (that can be accessed only a few times) generally withdraws fresh objects from an infinite pool of short-lived objects. Such an implementation uses unbounded space.

Intuitively, one can bound the space by re-using the short-lived objects as follows (cf. [KST91, MA94]). The infinite pool of fresh objects is replaced by a finite pool of clean objects, that gets replenished with objects that are no longer used. Once the implementation can guarantee that it is done with one of the withdrawn, now dirty, objects[5], it must somehow reset, or *wash*, the object and insert it back into the pool. This washing, or resetting, operation is added to the implementation of the object later, and may not be atomic. We therefore have to ensure that the washing operation is not interfered by any other activity. To operations performed after the reset, such a cleaned object looks exactly the same as a truly new object.

[4] We will not discuss lifeness in this paper.

[5] Some guarantee like this must be met because the objects are short-lived anyway.

Let $\mathcal{I}$ be an implementation of X, then we write $\mathcal{I} + \mathcal{I}(O)$ for the implementation of object $X + O$, i.e. the object X with a new operation O.

The following theorem formalises the above discussion.

Theorem 2.4. *Let X have sequential specification S. Let $\mathcal{I}$ be a correct implementation of X. Let Wash be a new operation for X with implementation $\mathcal{I}(Wash)$ that, when executed interference free in an arbitrary state of $\mathcal{I}$, exits leaving the implementation in state $init(\mathcal{I})$.*

Consider a run over $\mathcal{I}+\mathcal{I}(Wash)$, where for all actions A on X (including washing actions) and washing actions $W \neq A$ in this run, $\neg(A \| W)$ holds. Then each wellformed subrun of this run that lies exactly inbetween two succesive washing actions is linearizable w.r.t. S. If all such subruns of the run are wellformed, all actions A in the run can be assumed to be atomic and satisfying S.

Proof. If for a run over $\mathcal{I} + \mathcal{I}(Wash)$, for all actions A on X and washing actions W in this run, $\neg(A\|W)$ holds then the runs over X inbetween succesive calls to *Wash* actually form a partition of the full run (i.e. each action occurs in exactly one of these runs). Moreover, by the properties of $\mathcal{I}(Wash)$ and the fact that each washing action is indeed executed interference free, each of these runs on X start in the initial state $init(\mathcal{I})$. Because $\mathcal{I}$ is correct, each of these runs is linearizable w.r.t. S if it is wellformed. □

This scheme can only be applied if used objects can easily be washed, and if one can guarantee never to run out of clean objects before a dirty object can be washed. We will show that this technique is applicable in the implementation of long-lived test-and-set objects.

2.4 Properties of Test-And-Set Objects

Let x, y, z denote *TestAndSet* actions, subscripted by p when executed by processor p. Write $res(x)$ for the result returned by x, and $R(x)$ for the *Reset* action corresponding to x (i.e., executed after x to reset the test-and-set object). For one-shot test-and-set objects, that do not have a *Reset* operation, define $R(x)$ as a 'virtual action' and add it to the run ordered after all regular actions in it.

Definition 2.5. *For a wellformed run $\langle \mathcal{A}, \rightarrow \rangle$ or sequential execution $\langle \mathcal{A}, \Rightarrow \rangle$, define*

$$\mathcal{W} = \{x \in \mathcal{A} \mid res(x) = won\}$$
$$\mathcal{L} = \{x \in \mathcal{A} \mid res(x) = lost\}$$

Because we only consider complete runs where all actions return a response, this is well defined. We have the following general theorem

Theorem 2.6. *Let S_{TS} be the sequential specification of a test-and-set object. Let $\langle \mathcal{A}, \Rightarrow \rangle$ correspond to a wellformed run $\langle \mathcal{A}, \rightarrow \rangle$, satisfying S_{TS} and let $x, y \in \mathcal{W}$ be arbitrary. Then*

$$x \Rightarrow y \quad \text{if and only if} \quad x \rightarrow R(y) \ .$$

Proof. For $x = y$ the theorem trivially holds. So assume $x \neq y$. We consider the if and only-if part separately.

(if, $\Leftarrow$) If $x \to R(y)$ then $x \Rightarrow R(y)$. Now if $y \Rightarrow x$, then $y \Rightarrow x \Rightarrow R(y)$ with $x \neq y$ and hence $res(x) = lost$ by (TS2), contrary to assumption. Therefore $x \Rightarrow y$.

(only if, $\Rightarrow$) Suppose $x \Rightarrow y$. If $y \Rightarrow R(x)$, then by similar reasoning as above, $res(y) = lost$. Hence $R(x) \Rightarrow y$ and so $R(x) \not\to y$. With $x \to R(x)$ and $y \to R(y)$ we conclude $x \to R(y)$.

This completes the proof. □

This theorem motivates the following definition of $\Rightarrow'$ among all $x, y \in \mathcal{W}$ in a run $\langle \mathcal{A}, \to \rangle$.

$$x \Rightarrow' y \quad \text{if and only if} \quad x \to R(y) \ . \tag{D1}$$

Clearly $x \to R(x)$ (and likewise for y). Because $\to$ is an interval order, $x \to R(y)$ or $y \to R(x)$. Hence $\Rightarrow'$ orders each pair $x, y \in \mathcal{W}$. Also note that we have $x \Rightarrow' x$ for all $x \in \mathcal{W}$. Define the following property.

(T1) For all $x, y \in \mathcal{W}$, $x \neq y$, not both $x \to R(y)$ and $y \to R(x)$.

The next lemma shows that if (T1) holds for a run $\langle \mathcal{A}, \to \rangle$, then $\Rightarrow'$ as defined by (D1) is acyclic[6].

Lemma 2.7. *Let $\langle \mathcal{A}, \to \rangle$ be a wellformed run over a test-and-set object, and let $\Rightarrow'$ be defined by (D1). If (T1) holds, then $\Rightarrow'$ is a transitive acyclic order, in fact a total order over $\mathcal{W}$, extending $\to$.*

Proof. $\Rightarrow'$ extends $\to$ among all such $z \in \mathcal{W}$, because if $x \to y$, then surely $x \to R(y)$ and hence, using (D1), $x \Rightarrow' y$. If $x \Rightarrow' y$ and $y \Rightarrow' z$, then by (D1) and (T1) $x \to R(y)$ and $z \not\to R(y)$. Because $z \to R(z)$ and $\to$ is an interval order, $x \to R(z)$ and hence by (D1) also $x \Rightarrow' z$. This shows that $\Rightarrow'$ is transitive. Now $\Rightarrow'$ is easily seen to be acyclic. If there were a cycle of length larger than 1, by transitivity it could be reduced to a cycle of length 2: $x \Rightarrow' y$ and $y \Rightarrow' x$ (with $x \neq y$). But together with (D1), this would contradict (T1). □

Define the following property.

(T2) For each $x \in \mathcal{L}$ there exists an $y \in \mathcal{W}$ such that $y \not\to x$ and $x \not\to R(y)$, with for all $z \in \mathcal{W}$,
1. if $R(z) \to x$, then $z \to R(y)$, and
2. if $x \to z$ then $y \to R(z)$.

Broadly speaking, (T2) states that if a test-and-set is lost, it must have competed with another test-and-set operation that won. Some technicalities make (T2) slightly more complicated. Requiring that a test-and-set that is lost must overlap with a test-and-set that won is not enough, as witnessed by the scenario depicted in Figure 1. Here, $y, z \in \mathcal{W}$. $x \in \mathcal{L}$ overlaps with $R(y)$, but because x

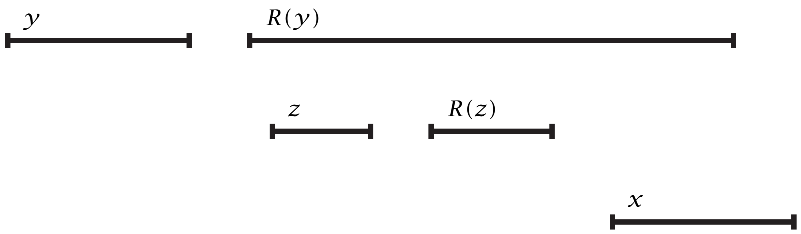

Figure1. Overlapping test-and-set can win

starts after $R(z)$, it cannot have been beaten by y, and therefore x should win as well.

In the proof of Theorem 2.10 we need the following definition, inspired by [AKKV88]. To avoid awkward limiting conditions we postulate two extra actions $\perp$ and $R(\perp)$ with $\perp \to R(\perp)$, $\perp \to x$ for all $x \in \mathcal{A}$, and $R(\perp) \to x$ for all $x \in \mathcal{A}$. We add $\perp$ to $\mathcal{W}$.

Definition 2.8. *Suppose (T1) and (T2) hold. For $x \in \mathcal{W}$, define*

$$Clan(x) = Clan(R(x)) = \text{the index of } x \text{ in } \Rightarrow', 1 \text{ being the first.}$$

For $x \in \mathcal{L}$, define $C(x)$ to be the last $y \in \mathcal{W}$ (according to $\Rightarrow'$, i.e., with the highest clan number) such that $y \to x$. Then define

$$Clan(x) = \begin{cases} 1 + Clan(C(x)) & \text{if } R(C(x)) \to x, \\ Clan(C(x)) & \text{otherwise} \end{cases}$$

Because (T1) holds, $\Rightarrow'$ is acyclic by Lemma 2.7, and so $Clan(x)$ is properly defined for all $x \in \mathcal{W}$. In fact each clan contains exactly one $z \in \mathcal{W}$, and for $x, y \in \mathcal{W}$, with $x \neq y$, we have $x \Rightarrow' y$ iff $Clan(x) < Clan(y)$.

Since $\perp \in \mathcal{W}$, $C(x)$ exists for all $x \in \mathcal{L}$. Therefore, $Clan(x)$ is also properly defined for all $x \in \mathcal{L}$.

Lemma 2.9. *Let $\langle \mathcal{A}, \to \rangle$ be a wellformed run over a test-and-set object satisfying (T1) and (T2). Let $x \in \mathcal{L}$ and $y \in \mathcal{W}$ be arbitrary. Then*

1. *If $x \to R(y)$, then $Clan(x) \leq Clan(y)$.*
2. *If $x \to y$, then $Clan(x) < Clan(y)$.*

Proof. Suppose $x \to R(y)$. Because by definition $C(x) \to x$ we have $C(x) \to R(y)$ and hence by (D1) $C(x) \Rightarrow' R(y)$ and so $Clan(C(x)) \leq Clan(R(y))$. Recall that $Clan(y) = Clan(R(y))$ for all $y \in \mathcal{W}$.

There are two cases.

[6] We allow 1-cycles, i.e., $x \Rightarrow' x$.

1. $R(C(x)) \not\to x$: Then $\mathit{Clan}(x) = \mathit{Clan}(C(x))$ and so $\mathit{Clan}(x) \leq \mathit{Clan}(y)$. Moreover, if $x \to y$, then $C(x) \to y$ so $C(x) \neq y$ and hence $\mathit{Clan}(x) < \mathit{Clan}(y)$.
2. $R(C(x)) \to x$: Then $\mathit{Clan}(x) = 1 + \mathit{Clan}(C(x))$. As by assumption $x \to R(y)$, we have $R(C(x)) \neq R(y)$, so $C(x) \neq y$ which implies $\mathit{Clan}(x) - 1 = \mathit{Clan}(C(x)) < \mathit{Clan}(y)$. Hence $\mathit{Clan}(x) \leq \mathit{Clan}(y)$.
   Moreover, if $x \to y$, then by property (T2) there is a $z \in \mathcal{W}$ such that $z \not\to x$ and $x \not\to R(z)$ with $C(x) \to R(z)$ [because we have $R(C(x)) \to x$], and $z \to R(y)$ [because $x \to y$]. Hence $C(x) \Rightarrow' z$ and $z \Rightarrow' y$ by (D1). Since $z \neq y$ [by $x \to y$ and $z \not\to x$], and $z \neq C(x)$ [by $R(C(x)) \to x$ and $x \not\to R(z)$], we have $\mathit{Clan}(C(x)) < \mathit{Clan}(z) < \mathit{Clan}(y)$. We conclude $\mathit{Clan}(x) < \mathit{Clan}(y)$.

This completes the proof. □

The next theorem proves that if (T1) and (T2) both hold for a run $\langle \mathcal{A}, \to \rangle$, then this run is a valid run for a test-and-set object.

Theorem 2.10. *Let $\langle \mathcal{A}, \to \rangle$ be a wellformed run over a test-and-set object satisfying (T1) and (T2). Then $\langle \mathcal{A}, \to \rangle$ is linearisable w.r.t. S_{TS}.*

Proof. We construct the corresponding sequential order $\Rightarrow$ in phases. First let $\Rightarrow$ be $\Rightarrow'$ as in (D1). Then $\Rightarrow$ is a total acyclic order among all $x \in \mathcal{W}$ by Lemma 2.7.

Now insert for each $x \in \mathcal{W}$, $R(x)$ immediately after x in $\Rightarrow$. Similarly, for all $y \in \mathcal{L}$, insert y between x and $R(x)$ in $\Rightarrow$ for which $\mathit{Clan}(x) = \mathit{Clan}(y)$. Such x exists by (T2) and the definition of $C(y)$: it either equals $C(y)$ or the next write after $C(y)$ which exists due to (T2).

Arbitrarily order all $y \in \mathcal{L}$ with the same clan number consistent with $\to$, and take the transitive closure. Note that this does not introduce cycles.

The resulting sequential execution is easily verified to satisfy (TS1), (TS2) and (TS3). It remains to verify that $\Rightarrow$ extends $\to$. We show this by case analysis.

1. $x \to y$ with $x = a$ or $x = R(a)$ for $a \in \mathcal{W}$ and $y = b$ or $y = R(b)$ for $b \in \mathcal{W}$: then $a \to R(b)$ and hence $a \Rightarrow' b$ by (D1) and therefore $x \Rightarrow y$ by construction of $\Rightarrow$.
2. $R(x) \to y$, $y \in \mathcal{L}$: Then $x \to y$, hence $x \Rightarrow' C(y)$. If $x \neq C(y)$, then $\mathit{Clan}(x) < \mathit{Clan}(C(y)) \leq \mathit{Clan}(y)$. If $x = C(y)$, then by $R(x) \to y$, $\mathit{Clan}(y) = 1 + \mathit{Clan}(C(y))$. Again $\mathit{Clan}(x) < \mathit{Clan}(y)$. So in both cases, by construction, $x \Rightarrow y$.
3. $x \to R(y)$, $x \in \mathcal{L}$: By Lemma 2.9, $\mathit{Clan}(x) \leq \mathit{Clan}(y)$. By construction $x \Rightarrow R(y)$.
4. $x \to y$:
   (a) $x \in \mathcal{W}$ and $y \in \mathcal{L}$: Then $x \Rightarrow' C(y)$ so $\mathit{Clan}(x) \leq \mathit{Clan}(C(y)) \leq \mathit{Clan}(y)$. By construction $x \Rightarrow y$.
   (b) $x \in \mathcal{L}$ and $y \in \mathcal{W}$: By Lemma 2.9, $\mathit{Clan}(x) < \mathit{Clan}(y)$. By construction $x \Rightarrow y$.
   (c) $x \in \mathcal{L}$ and $y \in \mathcal{L}$: If $z \to x$ then $z \to y$. Therefore $C(x) \Rightarrow' C(y)$. If $R(C(x)) \to x$ then $R(C(x)) \to y$. We conclude $\mathit{Clan}(x) \leq \mathit{Clan}(y)$. By construction $x \Rightarrow y$.

This completes the proof. □

3 Implementing Long-Lived Test-And-Set

Protocol 3.1 implements an n-processor long-lived test-and-set object from $n+1$ one-shot test-and-set objects for the same number of processors.

The notational conventions are as follows. Shared registers are written SHARED. SHARED denotes a multi-writer multi-reader register. When indexed by a processor name (e.g. SHARED_p), this denotes a single-writer multi-reader register owned (i.e. written) by p. Local variables of process p are written as var_p. Ghost variables (used for the proof of correctness but not influencing the flow of control of the program) are denoted by $GHOST$. Assignments to them appear indented on the following line in square brackets. Each numbered line contains at most one read from or one write to a single shared variable, or calls the single shot test-and-set operation. We assume that each of these operations is atomic (including any assignments to ghost variables immediately following that statement).

In the protocol we assume a bounded set of n-processor one-shot test-and-set objects $\mathsf{OneShot}[\,]$, indexed by elements from a set I. We will fix the size of I later. A shared register INDEX maintains the index of the current one-shot test-and-set object to play in. The winner of this one-shot object wins the long-lived test-and-set. When resetting, it is responsible for finding a new, empty, test-and-set object, to clean it, and finally let INDEX point to this object. If the object we implement is indeed a test-and-set object, resets occur sequentially. Therefore there is no interference during a reset and hence finding an empty object and resetting it is relatively straightforward.

Before entering a single shot test-and-set object, a processor writes the index of this object to shared register CHOOSE_p, and checks whether the global index changed before actually entering. Therefore, while INDEX is being updated, only processors that have revealed to enter a now old and already abandoned object (i.e. won by another processor) will possibly be using this old object.

The argument to see why this is the case runs as follows. While the winner p is in the reset phase, INDEX does not change. Therefore, a processor q is either busy in an old test-and-set object, and then the index of this object is stored in CHOOSE_q, or q is busy in or about to enter test-and-set object INDEX. Hence, the winner can easily find a free object, that is guaranteed not to be used or about to be used, by eliminating INDEX and all indices found in CHOOSE_q for all $q \neq p$ from the set I of all indices. This removes at most n indices. We conclude that $|I| \geq n+1$ and take $I = \{0, \dots, n\}$.

Initially, INDEX and CHOOSE_p equal an arbitrary element from I. Moreover[7], $owner(i) = \perp$ for all $i \in I$, $ROUND = 1$ and $OWNER = \perp$.

3.1 Proof of Correctness

We will use the following additional notation. The predicate $p@x$ is true if the next step of process p is to execute line x of Protocol 3.1. If x is subdivided

[7] We write $owner(x)$ for $owner(\mathsf{OneShot}[x])$.

Registers

$\mathsf{INDEX} \in I$, a multi-writer register, initially 0.

$\mathsf{CHOOSE}_p \in I$, single writer registers for each $p \in \{1, \dots, n\}$, initially arbitrary.

$\mathsf{OneShot}[\,]$, an array of one-shot test-and-set objects, indexed by $i \in I$, initially $owner(i) = \perp$.

Ghost variables

$OWNER \in \{1, \dots, n\}$, initially $\perp$.

$ROUND, round_p \in \mathbb{N}$, initially 1.

$choose_p[\,] \in I$, an array indexed by $\{1, \dots, n\}$, initially arbitrary.

$Q_p \in 2^{\{1,\dots,n\}}$, initially arbitrary.

Local variables of processor p

$index_p, free_p \in I$, initially 0.

$res_p \in \{won, lost\}$, initially arbitrary.

$F_p \in 2^{\{1,\dots,n\}}$, initially arbitrary.

$q \in \{1, \dots, n\}$, initially arbitrary.

$TestAndSet_p()$:

1 $index_p := \mathsf{INDEX}$
  $[round_p := ROUND]$;
2 $\mathsf{CHOOSE}_p := index_p$;
3 **if** $(index_p \neq \mathsf{INDEX})$
  then $res_p := lost$;
4 **else** $res_p := \mathsf{OneShot}[index_p].TestAndSet_p()$
  $[OWNER := p$ if $res_p = won]$
5 **return** res_p ;

$Reset_p()$:

6 (∗ Find index of free test-and-set object ∗)
6.1 $F_p := I \setminus \left\{index_p\right\}$
  $[Q_p := \{\}]$;
6.2 **forall** $q \neq p$
6.3 **do** $F_p := F_p \setminus \left\{\mathsf{CHOOSE}_q\right\}$
  $[Q_p := Q_p \cup \{q\}\,;\, choose_p[q] := \mathsf{CHOOSE}_q]$;
6.4 $free_p :=$ select $x \in F_p$
7 (∗ Wash free object. This is not an atomic operation. ∗)
7.1 call $\mathsf{OneShot}[free_p].Wash_p()$
7.2 wait until $\mathsf{OneShot}[free_p].Wash_p()$ returns
  $[owner(free_p) = \perp]$
  (∗ Make free object available ∗)
8 $\mathsf{INDEX} := free_p$
  $[ROUND := ROUND + 1; OWNER := \perp]$;

Protocol 3.1: A bounded space long-lived n processor test-and-set protocol.

into several lines $x.i$, we also write $p@x$ if $p@x.i$ for some i. We write $p.x$ for the action of executing line x by process (or sometimes action) p.

We will use $\rightarrow$ to denote both the partial order in the run $\langle \mathcal{A}, \rightarrow \rangle$ over the high-level actions (i.e., those on the long-lived test-and-set object), and the total order over all low level actions (i.e., both on the registers and the one-shot test-and-set objects). Note that $x \to R(y)$ is equivalent to $x.5 \to y.6$ and that $R(x) \to y$ is equivalent to $x.8 \to y.1$.

We prove correctness of the protocol using invariants. The proof consists of two stages. First we will show that invariance of the following two predicates

$$p@5\text{–}8 \wedge \mathit{res}_p = \mathit{won} \Rightarrow \mathit{OWNER} = p \tag{P1}$$

$$p@5 \wedge \mathit{res}_p = \mathit{lost} \Rightarrow (\mathit{OWNER} \neq p \ \wedge\ \mathit{OWNER} \neq \bot) \ \vee\ (\mathit{round}_p \neq \mathit{ROUND}) \tag{P2}$$

implies that (T1) and (T2) hold for every run. This shows, using Theorem 2.10, that all runs are linearisable w.r.t. S_{TS}. In the second stage of the proof, we show that (P1) and (P2) are indeed invariant and thus that Protocol 3.1 implements a test-and-set object.

Lemma 3.1. *Invariance of (P1) and (P2) in Protocol 3.1 implies that (T1) and (T2) hold for all runs of Protocol 3.1.*

Proof. If (T1) is violated, then for some x_p and y_q in $\mathcal{W}$ with $x_p \neq y_q$ we have $x_p.5 \to y_q.6$ and $y_q.5 \to x_p.6$. Hence x and y overlap and we conclude that $p \neq q$. Because the underlying execution is atomic, either $x_p.5 \to y_q.5$ or $y_q.5 \to x_p.5$. In the first case, just before $y_q.5$ we have $p@6$ and $\mathit{res}_p = \mathit{won}$ as well as $q@5$ and $\mathit{res}_p = \mathit{won}$, so by (P1) $\mathit{OWNER} = p$ and $\mathit{OWNER} = q$ and then $p = q$, a contradiction. The other case is similar.

Suppose $y_q \in \mathcal{L}$. To show (T2) holds, we have to show there exists a $z_r \in \mathcal{W}$ with $z_r \not\to y_q$ and $y_q \not\to R(z_r)$ and that for all $x_p \in \mathcal{W}$, if $R(x_p) \to y_q$ we have $x_p \to R(z_r)$, and if $y_q \to x_p$ we have $z_r \to R(x_p)$

By property (P2) we have the following two cases.

1. At $y_q.5$, $\mathit{OWNER} = r$ and $r \neq q$ and $r \neq \bot$. Then for some $z_r \in \mathcal{W}$, $z_r.4 \to y_q.5 \to z_r.8$. Hence $z_r \not\to y_q$ and $y_q \not\to R(z_r)$. Moreover, if $R(x_p) \to y_q$, using the above also $x_p.5 \to z_r.8$

   Now if $z_r.5 \to x_p.5$ at $x_p.5$ we have $r@5\text{–}8 \ \wedge\ p@5$. By (P1), then $p = r$. Because z_r and x_p are concurrent, this cannot happen. Because the underlying run is atomic, we conclude $x_p.5 \to z_r.5$, and hence $x_p \to R(z_r)$.

   If $y_q \to x_p$, then $z_r.4 \to y_q.5 \to x_p.1 \to x_p.5$. If $x_p.5 \to z_r.5$ then at $p.5$ we have again $r@5 \ \wedge\ p@5$, which we have already shown is impossible. We conclude $z_r.5 \to x_p.5$, and hence $z_r \to R(x_p)$.
2. At $y_q.5$, $\mathit{round}_q \neq \mathit{ROUND}$. Then for some z_r, $y_q.1 \to z_r.8 \to y_q.5$. Again $z_r \not\to y_q$ and $y_q \not\to R(z_r)$. Moreover, if $R(x_p) \to y_q$, using the above $x_p.5 \to z_r.8$ and by similar reasoning of the previous item $x_p \to R(z_r)$.

   If $y_q \to x_p$, then $y_q.1 \to z_r.8 \to y_q.5 \to x_p.1$. This implies $z_r.5 \to x_p.6$, and hence $z_r \to R(x_p)$.

This completes the proof. □

We introduce the following 9 smaller invariants.

$$p@5\text{–}8 \;\wedge\; res_p = won \quad \text{implies} \quad index_p = \mathsf{INDEX} \;\wedge\; owner(index_p) = p \tag{I1}$$

$$res_p = lost \quad \text{implies} \quad OWNER \neq p \tag{I2}$$

$$p@4 \;\wedge\; owner(index_p) = \bot \quad \text{implies} \quad index_p = \mathsf{INDEX} \tag{I3}$$

$$owner(i) = q \;\wedge\; q \neq \bot \quad \text{implies} \quad q = OWNER \;\vee\; \mathsf{INDEX} \neq i \tag{I4}$$

$$p@5 \;\wedge\; res_p = lost \;\wedge\; OWNER = \bot \quad \text{implies} \quad round_p \neq ROUND \tag{I5}$$

$$p@6.2\text{–}8 \;\wedge\; q \in Q_p \;\wedge\; q@4 \quad \text{implies} \quad index_q = choose_p[q] \;\vee\; index_q = \mathsf{INDEX} \tag{I6}$$

$$p@6\text{–}8 \quad \text{implies} \quad res_p = won \tag{I7}$$

$$p@7\text{–}8 \;\wedge\; q@4 \quad \text{implies} \quad (index_q \neq free_p \wedge \mathsf{INDEX} = index_p \neq free_p) \tag{I8}$$

$$\mathsf{INDEX} \neq index_p \quad \text{implies} \quad ROUND \neq round_p \tag{I9}$$

Lemma 3.2. *If (I1), (I7) and (I8) are invariant for Protocol 3.1, then for all operations on* $\mathsf{OneShot}[i]$*, with* $i \in I$*, executed by Protocol 3.1, (TS1), (TS2) and (TS3) hold.*

Proof. If (I8) is invariant, then if p washes $\mathsf{OneShot}[free_p]$ at 7, for all test-and-set operations executed by q at 4 on $\mathsf{OneShot}[index_q]$, $free_p \neq index_q$.

If (I1) and (I7) are invariant, then if both p and q wash an object, then actually $p = q$. Hence for all actions A on $\mathsf{OneShot}[i]$ (including washing operations), $\neg(A \| Wash(\mathsf{OneShot}[i]))$.

By assumption, the implementation $\mathcal{I}$ of $\mathsf{OneShot}[free_p]$ and the implementation of $\mathsf{OneShot}[free_p].Wash_p()$ together guarantee that $\mathsf{OneShot}[free_p].Wash_p()$ returns with $init(\mathcal{I})$.

Now for every run of the protocol, the conditions of Theorem 2.4 are met. One shot test and set objects impose no further restrictions on wellformed runs. Hence each run in the partition (in Theorem 2.4) is wellformed, and so for all actions on one shot test and set objects in the run, S_{TS} holds. □

Lemma 3.3. *In Protocol 3.1, (I1) through (I9) hold initially.*

Proof. Because initially, for all p, $p@1$, (I1), (I3), (I5), (I6), (I7) and (I8) trivially hold. (I2) holds initially, because initially $OWNER = \bot$. (I4) holds because initially $owner(i) = \bot$ for all $i \in I$. (I9) holds because initially $index_p = \mathsf{INDEX} = 0$ □

Lemma 3.4. *In Protocol 3.1, (I1) through (I9) are invariant.*

Proof. We prove invariance of each invariant separately.

(I1) : Steps 1, 2, and 8 of p set $\neg p@5\text{–}8$. Steps 5, 6, and 7 of p do not change $index_p$, INDEX or $owner(index_p)$ (step 7 of p does not alter $owner(index_p)$ by the construction of F_p). Step 3 only sets $p@5$ if $p_{index} \neq \mathsf{INDEX}$ and hence $res_p = lost$.

Now consider step 4 by p. If it sets $res_p = won$, then by (TS1) – which holds by Lemma 3.2 – according to which at $p.4$, $owner(index_p) = \bot$, which by (I3)

implies $index_p$ = INDEX (not altered by $p.4$). Also, by (TS1) after step 4 we have $owner(index_p) = p$.
Only steps 4, 7 and 8 of $q \neq p$ can affect (I1) by possibly altering INDEX or $owner(index_p)$ while p@5–8 and $res_p = won$.
But if q@7 or q@8, then $res_q = won$ by (I7). Using the assumption that p@5–8 and $res_p = won$ (else (I1) trivially holds) and that (I1) holds initially, we conclude $p = q$. This case is handled in the previous paragraph.
This leaves step 4 for $q \neq p$ while p@5–8 and $res_p = won$. This only affects $owner(index_p)$ if $index_q = index_p$. Then by (I1), $owner(index_q) \neq \perp$. But then by Lemma 3.2 (TS2) holds, which implies that step 4 does not affect $owner(index_p)$ at all.

(I2) : No other processor but p can set $OWNER = p$. But p only sets $OWNER = p$ when $res_p = won$.

(I3) : Only step 3 of p sets p@4, but only if $index_p$ = INDEX.
The only step of q that could set $owner(index_p) = \perp$ is 7. However, similar as before, step 7 of $q \neq p$ does not change $owner(index_p)$ by (I8) as long as p@4. And if $q \neq p$ changes INDEX from $index_p$ to a value unequal to $index_p$ (by step 8), then q@8 implies $res_q = won$ by (I7) and hence we can apply (I1) to show INDEX = $index_q$ and hence $index_p = index_q$, and thus $owner(index_p) = q \neq \perp$.

(I4) : The only step that changes $owner(i)$ to a value unequal to $\perp$ is 4. But then it also sets $OWNER = p$. The only steps that change INDEX and $OWNER$ are 4 (handled already) and 8. But if 8 of p sets INDEX = i, then $free_p = i$ at 7 and hence after 7, $owner(i) = \perp$.

(I5) : Step 1–2 and 5–8 of p set $\neg p$@5.
Step 3 only sets p@5 if $p_{index} \neq$ INDEX and hence by (I9) $ROUND \neq round_p$. This leaves step 4. But if this step returns $res_p = lost$, then by (TS2) – which holds by Lemma 3.2 – we have $owner(index_p) \neq p \wedge owner(index_p) \neq \perp$. By (I4) then $OWNER \neq \perp$ or $INDEX \neq index_p$. Because we assumed $OWNER = \perp$ we conclude $INDEX \neq index_p$ and hence again by (I9), $ROUND \neq round_p$.

(I6) : Step 1–6 and 8 of p set $\neg p$@6.2–8. Steps 6.2, 6.4, and 7 do not alter Q_p or $choose_p[]$. Step 6.1 sets $Q_p = \{\}$ satisfying (I6).
For each q, step 6.3 of p sets $Q_p := Q_p \cup \{q\}$, but also $choose_p[q] :=$ CHOOSE_q. If q@4–5, then $\text{CHOOSE}_q = index_q$ and the condition holds. If $\neg q$@4–5 the condition holds trivially
Only step 8 of $r \neq p$ changes INDEX. But if r@8, then by (I7) and (I1) we conclude that $p = r$. This case is handled above.
Only step 3 of a $q \in Q_p$ can invalidate (I6) if it reaches q@4, but then there are two cases:

1. $q.2 \to p.6.3$: $\text{CHOOSE}_q = index_q$ when $p.6.3$ is executed for this q, hence $choose_p[q] = index_q$.
2. $p.6.3 \to q.2$: Then $p.1 \to q.3 \to p.8$ ($q.3 \to p.8$ by assumption that p@7–8). By (I1), INDEX = $index_p$. Now because q reaches q@4, $index_q$ = INDEX = $index_p$.

(I7) : By the protocol, $OWNER = p$ implies $res_p = won$. Then (I7) follows from wellformedness (maintaining (TS3)) of the run.

(I8) : If $p@7$, then (I8) holds by virtue of (I6) and the fact that after step $p.6$ $Q_p = I \setminus \{p\}$ and $free_p$ is selected from $F_p = I \setminus (\bigcup_{q \in Q_p} choose_p[q] \cup \left\{index_p\right\})$. Also note that INDEX = $index_p$ by (I1).

(I9) : The only step that changes INDEX is 8, but this step also increments *ROUND*. Hence whatever value p read for $round_p$ at step 1 when reading $index_p$ must have been smaller.

□

The correctness of our protocol immediately follows from these invariants, and the results of Section 2.

Theorem 3.5. *In Protocol 3.1, (P1) and (P2) are invariant. Therefore Protocol 3.1 implements a wait-free long-lived test-and-set object.*

Proof. (P1) follows from (I1) and (I4). Invariants (I2) and (I5) imply (P2). (I1) through (I9) are invariant by lemma 3.4. Therefore (P1) and (P2) are invariant too.

By Lemma 3.1, then (T1) and (T2) hold for every run of Protocol 3.1. By Theorem 2.10 then all runs of Protocol 3.1 are linearisable w.r.t. S_{TS}. □

We conclude by stating the time (measured in number of shared register accesses) and space requirements of our protocol.

Theorem 3.6. *Protocol 3.1 uses* $n + 1$ *one-shot test-and-set objects, and* n *single writer registers and* 1 *(multi-writer) register containing* $\lceil \log n + 1 \rceil$ *bits each.*

Let t *be the worst case expected running time of the one-shot test-and-set operation, then the worst case expected running time of the test-and-set operation of Protocol 3.1 is* $3 + t$.

Let w *be the worst case running time needed to wash the one-shot test-and-set object. Then the worst case running time of the reset operation is* $n + w$.

Proof. Immediately follows from the code of Protocol 3.1. □

Using the results of the appendix, we obtain the following corollary.

Corollary 3.7. *Long-lived test-and-set can be implemented using* $4n^2 + 4$ *(multi-writer) registers of* 2 *bits each, plus* n *single writer registers and* 1 *(multi-writer) register containing* $\lceil \log n + 1 \rceil$ *bits each.*

The test-and-set operation takes a worst-case expected time of $O(\log n)$*; the reset operation takes* $5n$ *time units in the worst case.*

References

[AGTV92] AFEK, Y., GAFNI, E., TROMP, J., AND VITÁNYI, P. M. B. Wait-free test-and-set. In *6th WDAG* (Haifa, Israel, 1992), A. Segall and S. Zaks (Eds.), LNCS 647, Springer-Verlag, pp. 85–94.

[AKKV88] AWERBUCH, B., KIROUSIS, L. M., KRANAKIS, E., AND VITÁNYI, P. M. B. A proof technique for register atomicity. In *8th FST&TCS* (Pune, India, 1988), K. V. Nori and S. Kumar (Eds.), LNCS 338, Springer Verlag, pp. 286–303.

[BGHM95] BUHRMAN, H., GARAY, J. A., HOEPMAN, J.-H., AND MOIR, M. Long-lived renaming made fast. In *14th PODC* (Ottawa, Ont., Canada, 1995), ACM Press, pp. 194–203.

[Her91] HERLIHY, M. P. Wait-free synchronization. *ACM Trans. Prog. Lang. & Syst.* **13**, 1 (1991), 124–149.

[HW90] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. & Syst.* **12**, 3 (1990), 463–492.

[IL87] ISRAELI, A., AND LI, M. Bounded time stamps. In *28th FOCS* (Los Angeles, CA, USA, 1987), IEEE Comp. Soc. Press, pp. 371–382.

[IS92] ISRAELI, A., AND SHAHAM, A. Optimal multi-writer multi-reader atomic register. In *11th PODC* (Vancouver, B.C., Canada, 1992), ACM Press, pp. 71–82.

[KST91] KIROUSIS, L. M., SPIRAKIS, P., AND TSIGAS, P. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. In *5th WDAG* (Delphi, Greece, 1991), S. Toueg, P. G. Spirakis, and L. Kirousis (Eds.), LNCS 579, Springer-Verlag, pp. 229–241.

[Lam86] LAMPORT, L. On interprocess communication. Part I: Basic formalism, part II: Algorithms. *Distr. Comput.* **1**, 2 (1986), 77–101.

[LAA87] LOUI, M. C., AND ABU-AMARA, H. H. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing Research*, F. P. Preparata (Ed.), vol. 4. JAI Press, 1987, pp. 163–183.

[MA94] MOIR, M., AND ANDERSON, J. H. Fast, long-lived renaming. In *8th WDAG* (Terschelling, The Netherlands, 1994), G. Tel and P. M. B. Vitányi (Eds.), LNCS 857, Springer-Verlag, pp. 141–155. To appear in Science of Computer Programming.

[TV91a] TROMP, J., AND VITÁNYI, P. M. B. A randomized algorithm for two-process wait-free test-and-set. Tech. Rep. CT-91-10, Institute for Language, Logic and Information, University of Amsterdam, 1991.

[TV91b] TROMP, J., AND VITÁNYI, P. M. B. Randomized wait-free test-and-set. Tech. Rep. CS-R9113, CWI, Amsterdam, 1991.

A From 2 to *n* Process One-Shot Test-And-Set

We still have to show that our construction of a long-lived test-and-set object from a one-shot object is reasonable, in the sense that we can implement the one-shot test-and-set together with a washing operation. In this appendix we show that this is indeed the case, by constructing an n process one-shot test-and-set object from a 2-process one which is easily washed by a single processor. The construction is only a slight modification of existing protocols. Our exposition on this topic is therefore brief, and will point to relevant other publications where appropriate.

Recall that for long-lived test-and-set objects, each winner of the test-and-set eventually resets it. In other words, for all $x \in \mathcal{W}$, $R(x)$ exists. For one-shot test-and-set objects this is not the case, because the reset operation is not defined. In this case, $R(x)$ for a $x \in \mathcal{W}$ was defined in Section 2.4 as a virtual action ordered after all regular actions in the run.

For one-shot test-and-set, property (T1) now implies that $\mathcal{W}$ contains at most one action.

Registers
  DOOR $\in \{open, closed\}$, a multi-writer register, initially *open*.
  OneShot2Proc[], a tree of 2 processor one-shot test-and-set objects,
    indexed by binary strings of length $\lceil \log n \rceil - 1$ or smaller,
    initially $owner(i) = \perp$.
Local variables of processor p
  $res_p \in \{won, lost\}$, initially arbitrary.
  $\ell \in \{0, \ldots, \lceil \log n \rceil\}$, initially arbitrary.

TestAndSet$_p$() :

1 **if** (DOOR = *open*)
2 **then** DOOR := *closed* ;
  $\ell := \lceil \log n \rceil$;
3   **repeat** $res_p :=$ OneShot2Proc$[p[\ell - 1{:}1]]]$.*TestAndSet*$_{p[\ell]}$() ;
    $\ell := \ell - 1$;
  until $res_p = lost \vee \ell = 0$;
  else $res_p := lost$;
4 **return** res_p ;

Wash():
  forall binary strings s of length $\lceil \log n \rceil - 1$ or smaller
  do OneShot2Proc$[s]$.*Wash*() ;

Protocol A.1: From 2 to n processor one-shot test-and-set.

(T1)' $|\mathcal{W}| \leq 1$.

Given this constraint, (T2) simplifies to

(T2)' If $|\mathcal{L}| > 0$, then $|\mathcal{W}| = 1$, and for $x \in \mathcal{W}$, for each $y \in \mathcal{L}$, $x \not\to y$.

In Protocol A.1 we present the construction of a n-process single shot test-and-set using a 2-process single-shot test-and-set as a primitive. The idea is to play a tournament [AGTV92] in a tree of 2-processor test-and-set objects, using multi-writer registers inside these 2-processor test-and-set objects to ease the search for an opponent [BGHM95] and to enable a single processor to wash the entire data structure.

Any 2-processor test-and-set object using single writer registers in its implementation (e.g., the construction in [TV91a, TV91b]) can be used if each single writer register is replaced by a multi-writer one (which can be written by any of the n processors playing the tournament). The construction of the tree guarantees that at most two processors ever play any test-and-set.

Each 2-processor test-and-set has 2 'inputs', labelled 0 and 1. A processor entering such test-and-set over the input with label i plays the 2-processor test-and-set protocol for processor i.

The tournament tree of 2-processor test-and-sets is constructed as follows. The root of the tree (at level 1) is a single 2-processor test-and-set labelled ε

(the empty string). The tree has $\lceil \log n \rceil$ levels. If test-and-set x is connected over input i to test-and-set y with label s one level higher in the tree, then x has label $(s\,i)$.

Interpret the name p of any of the n processors $\{0, \dots, n-1\}$ as an $\lceil \log n \rceil$ binary string, padded in the front with 0 bits when appropriate. The index of the rightmost, least significant, bit is 1. Let $p[i]$ denote the i-th bit of p's name, and let $p[i{:}1]$ denote the substring of p's name obtained by taking the rightmost i, i.e., least significant, bits.

Now the path for p through the tree to win is easily determined using p's name. At level ℓ, p plays the test-and-set with label $p[\ell - 1{:}1]$ over input $p[\ell]$. p enters the tournament at level $\ell = \lceil \log n \rceil$, and if p wins, then he advances to a higher level $\ell - 1$. If p wins test-and-set ε at level 1 (and advances to level 0), then p wins the tournament and hence the n-processor test-and-set.

The tournament construction guarantees that $(T1)'$ holds. Processors entering the tournament first check to see whether a 'global' door is open, and only enter and close the door if they find this door open. This construction guarantees that $(T2)'$ also holds.

We will now formally prove correctness of Protocol A.1.

Definition A.1. *Let $\langle \mathcal{A}, \to \rangle$ be a complete run (see Sect. 2.1) over Protocol A.1. For every action $x \in \mathcal{A}$ define*

$$\textit{level}(x) = \textit{the minimal level visited by } x \textit{ in the tournament tree},$$

where level$(x) = 0$ for the winner, and level$(x) = \lceil \log n \rceil + 1$ if x never enters the tournament tree but looses because of a closed door.

For all processors $p \in \{0, \dots, n-1\}$ and all $\ell \in \{0, \dots, \lceil \log n \rceil\}$, define the peers of p at level $\ell + 1$,

$$T_\ell(p) = \left\{ y_q \in \mathcal{A} \mid (\textit{level}(y_q) < \lceil \log n \rceil + 1) \ \wedge\ (q[\ell{:}1] = p[\ell{:}1]) \right\}.$$

The peers $T_\ell(p)$ contain all actions that actually compete in the test and set and are possible contenders for p at level $\ell + 1$. We have the following lemma, which states if there is a contender for p at level $\ell + 1$ then there is exactly one such contender at level ℓ or lower.

Lemma A.2. *Let $\langle \mathcal{A}, \to \rangle$ be a complete run over Protocol A.1. For all $\ell \in \{0, \dots, \lceil \log n \rceil\}$, and all $p \in \{0, \dots, n-1\}$ with $|T_\ell(x_p)| > 0$, there exists exactly one $y_q \in T_\ell(x_p)$ with level$(y_q) \leq \ell$.*

Proof. By induction on ℓ.

For $\ell = \lceil \log n \rceil$, there is at most one x_p with $x_p \in T_\ell(p)$. If $|T_\ell(p)| > 0$, there is exactly one.

Suppose the result is proven for all $\ell > t$. Now consider $\ell = t$, and pick an arbitrary $p \in \{0, \dots, n-1\}$ with $|T_t(p)| > 0$. Let p^0 be p with $p[t+1] = 0$ and let p^1 be p with $p[t+1] = 1$. For all $y_q \in T_t(p)$ either $y_q \in T_{t+1}(p^0)$ or $y_q \in T_{t+1}(p^1)$. Those y_q with $y_q \in T_{t+1}(p^0)$ play the 2-processor test-and-set with label $p[t:1]$ at level $t+1$ over input 0; by the induction hypothesis there is at most one such y_q.

Those y_q with $y_q \in T_{t+1}(p^1)$ play the 2-processor test-and-set with label $p[t:1]$ at level $t+1$ over input 1; by the induction hypothesis there is at most one such y_q.

Because $|T_t(p)| > 0$, there is at least one such y_q. By the properties of the test-and-set object and the fact that the run is complete there is exactly one winner that enters level t. □

Theorem A.3. *(T1)' and (T2') hold for all runs of Protocol A.1. Hence Protocol A.1 implements an n-process one-shot test-and-set object.*

Proof. (T1)', i.e., $|\mathcal{W}| \leq 1$, immediately follows from Lemma A.2, taking $\ell = 0$.

(T2)', i.e., if $|\mathcal{L}| > 0$, then $|\mathcal{W}| = 1$, and for $x \in \mathcal{W}$, for each $y \in \mathcal{L}$, $x \not\to y$, is proven as follows.

Note that the only value written to DOOR is *closed*. Then all reads after the first write to DOOR return *closed*. Now suppose to the contrary that $p \in \mathcal{W}$, $q \in \mathcal{L}$, $q \to p$. But then $q.2 \to p.1$ and hence p would have read DOOR = *closed*, meaning $p \in \mathcal{L}$ contrary to assumption.

If $|\mathcal{L}| > 0$, then at least one processor plays the test-and-set. The first processor p to enter the test-and-set reads DOOR = *open* at $p.1$ and enters the tournament. Hence $|T_0(p)| > 0$ and therefore by Lemma A.2 for some $y_q \in T_0(p)$ *level*$(y_q) = 0$ i.e., $y_q \in \mathcal{W}$.

The second part from the theorem follows from Theorem 2.10. □

Theorem A.4. *Let Protocol A.1 use the 2-process test-and-set of Tromp and Vitányi [TV91a, TV91b]. Then its worst case expected running time of the test-and-set operation is $O(\log n)$, and it uses at most $4n$ registers of 2 bits each. The worst case running time of the washing operation is $4n$.*

Proof. A processor can advance at most $\lceil \log n \rceil$ levels up the tree, playing a one-shot test-and-set at each level. The 2-process test-and-set object of Tromp and Vitányi [TV91a, TV91b] has a worst case expected running time of $O(1)$. Hence our protocol has a worst case expected running time of $O(\log n)$.

Protocol A.1 uses $2^{\lceil \log n \rceil - 1} < 2n$ single-shot test-and-set objects. The 2-process test-and-set object of Tromp and Vitányi [TV91a, TV91b] uses two 4-valued registers. Hence our protocol uses at most $4n$ registers of 2 bits each, plus 1 binary register for the 'door'.

Washing each register takes 1 time unit. This proves the upper bound on the time needed to wash the object. □