An $O(1)$ RMRs Leader Election Algorithm

[Extended Abstract]

Wojciech Golab*
Department of Computer Science
University of Toronto
Toronto, Canada
wgolab@cs.toronto.edu

Danny Hendler†
Faculty of Industrial Engineering and Management
Technion
hendler@techunix.technion.ac.il

Philipp Woelfel‡
Department of Computer Science
University of Toronto
Toronto, Canada
pwoelfel@cs.toronto.edu

ABSTRACT

The *leader election* problem is a fundamental distributed coordination problem. We present leader election algorithms for the cache-coherent (CC) and distributed shared memory (DSM) models using reads and writes only, for which the number of remote memory references (RMRs) is constant in the worst case.

The algorithms use splitter-like objects [6, 8] in a novel way for the efficient partitioning of processes into disjoint sets that share work. As there is an $\Omega(\log n/ \log\log n)$ lower bound on the RMR complexity of mutual exclusion for n processes using reads and writes only [4], our result separates the mutual exclusion and leader election problems in terms of RMR complexity in both the CC and DSM models.

Our result also implies that any algorithm using reads, writes and one-time *test-and-set* objects can be simulated by an algorithm using reads and writes with only a constant blowup of the RMR complexity. Anderson, Herman and Kim raise the question of whether conditional primitives such as test-and-set and *compare-and-swap* are stronger than read and write for the implementation of local-spin mutual exclusion [3]. We provide a negative answer to this question, at least for one-time test-and-set.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

*Supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

†Supported in part by Sun Microsystems and by the Technion's Aly Kaufman Fellowship.

‡Supported by DFG grant WO1232/1-1.

General Terms

Algorithms, Performance, Theory

Keywords

Leader election, mutual exclusion, shared memory, remote memory references, test-and-set

1. INTRODUCTION

The leader election problem is a fundamental distributed coordination problem. In the leader election problem, exactly one process, the *leader*, should be distinguished from all other processes. Processes must output either a *win* or a *lose* value: the process elected as leader must output *win*, and all other processes must output *lose*.

We consider the time complexity of shared memory algorithms based on reads and writes under the *remote memory references* (RMR) complexity measure. The main contributions of this paper are leader election algorithms with $O(1)$ RMR complexity for the *cache-coherent* (CC) model and for the *distributed shared memory* (DSM) model. To the best of our knowledge, these are the first leader election algorithms using only reads and writes that have a sublogarithmic RMR complexity.

Our algorithms are based on a novel use of splitter-like objects for the efficient partitioning of processes into disjoint sets such that all processes in one set share work. Based on these algorithms, we are also able to prove that any algorithm for the CC or DSM model using read, write and one-time *test-and-set* can be simulated by an algorithm using read and write with only a constant blowup of the RMR complexity. Thus, one-time test-and-set is no stronger than read and write in terms of RMR complexity in the CC or DSM model.

The leader election problem is closely related to the mutual exclusion problem [11], and leader election may be regarded as "one-shot" mutual exclusion [12]. In particular, any algorithm that solves mutual exclusion also solves leader election.

Alur and Taubenfeld proved that for any mutual exclusion algorithm for two or more processes using reads and writes only, the first process to enter its critical section may have to perform an unbounded number of accesses to shared variables [1]. For leader election, this result implies that the process eventually elected as a leader may have to perform an unbounded number of shared variable accesses. As observed by Anderson, Herman and Kim [3], this result indicates that a time complexity measure that counts all shared memory accesses is meaningless for mutual exclusion; the same holds for leader election. Largely because of that, recent work on mutual exclusion uses the RMR complexity measure, which counts only remote memory references. These references cannot be resolved by a process locally and cause interconnect traffic. Recent mutual exclusion work also focuses on *local-spin* algorithms, for which all busy-waiting is done by means of read-only loops that repeatedly test locally accessible variables (see, e.g., [4, 5, 15, 16, 17]).

Anderson was the first to present a local-spin mutual exclusion algorithm using only reads and writes with bounded RMR complexity [2]. In his algorithm, a process incurs $O(n)$ RMRs to enter and exit its critical section, where n is the maximum number of processes participating in the algorithm. Yang and Anderson improved on that, and presented an $O(\log n)$ RMRs mutual exclusion algorithm based on reads and writes [7]. This is the most efficient known algorithm under the worst-case RMR complexity measure for both mutual exclusion and leader election using reads and writes only in both the CC and DSM models. A prior algorithm by Choy and Singh (with minor modifications to ensure termination) surpasses Yang and Anderson's algorithm in the context of leader election in the CC model by achieving an amortized complexity of $O(1)$ RMRs, while retaining $O(\log n)$ worst-case RMR complexity [9]. This algorithm is based on a cascade of splitter-like *filter* objects, and was originally proposed as a building block for adaptive mutual exclusion. Our algorithm improves on the above results by establishing a tight bound of $\Theta(1)$ RMRs in the worst case on leader election in both the CC and DSM models.

Anderson and Kim [4] proved a lower bound of $\Omega(\log n/\log\log n)$ on the RMR complexity of n-process mutual exclusion algorithms that use reads and writes only. This result improves on a previous lower bound of $\Omega(\log\log n/\log\log\log n)$ obtained by Cypher [10]. Both lower bounds hold also for algorithms that in addition use *conditional primitives*, such as test-and-set and *compare-and-swap*; lower RMR complexity can be attained with the help of non-conditional primitives such as *swap* [3]. This is somewhat surprising, as compare-and-swap is stronger than swap in Herlihy's wait-free hierarchy [13].

Anderson, Herman and Kim raise the question of whether conditional primitives are stronger than reads and writes in the context of mutual exclusion RMR complexity [3]. The known lower bounds provide no relevant information here as they are insensitive to the availability of conditional primitives. For one-time test-and-set, we provide a negative answer to this question by showing that, in both the CC and DSM models, it is no stronger than reads and writes in terms of RMR complexity for implementing *any* algorithm.

1.1 Model Definitions and Assumptions

In this paper we consider both the cache-coherent (CC) and distributed shared memory (DSM) multiprocessor architectures. Each processor in a CC machine maintains *local* copies of shared variables inside a cache, whose consistency is ensured by a coherence protocol. At any given time a variable is *remote* to a processor if the corresponding cache does not contain an up-to-date copy of the variable. In a DSM machine, each processor instead owns a segment of shared memory that can be locally accessed without traversing the processor-to-memory interconnect. Thus, every variable is *local* to a single processor and *remote* to all others.

In the presentation of our algorithm we assume that there is a unique process executing the algorithm on each processor. (Clearly, the RMR complexity of the algorithm can only improve if multiple processes execute on some or all of the processors.) An instruction of the algorithm causes a *remote memory reference* if it accesses a variable that is remote to the process that executes it. In DSM local-spin algorithms, each process has its own dedicated spin variables, stored in its local segment of shared memory. In contrast, in a CC machine it is possible for multiple processes to locally spin on the same shared variable. We assess the RMR complexity of a leader election algorithm by counting the worst-case total number of remote memory references required by a process to execute the algorithm.

In the model we consider processes are asynchronous but do not fail. (In fact, it follows from [1] that no fault-tolerant leader election algorithm that uses solely reads and writes exists.) Every process is *live*, meaning that once it begins executing an algorithm, it continues to take steps until its algorithm terminates.

The remainder of the paper is organized as follows. An overview of our leader election algorithms is provided in Section 2. In Section 3, we give a detailed description of the DSM algorithm. We then present a CC variant in Section 4 as an extension of the DSM algorithm. Section 5 discusses the RMR complexity of the one-time test-and-set primitive. Algorithm correctness proofs are provided in the full version of this paper, available through CiteSeer.

2. OVERVIEW OF THE ALGORITHMS

The algorithms proceed in asynchronous *merging phases* (described in more detail in Section 3.3). At the end of each merging phase, the set of processes is partitioned into *losers* and *contenders*. As the name suggests, a loser will not be elected leader, while a contender has a chance of being elected leader. Initially, all processes are contenders. Once a process has become a loser, it remains a loser thereafter.

The set of contenders is further partitioned into *teams*. Each team has a unique *head*; all its other members are called *idle*. Only the head of a team performs RMRs. The goal that each process performs only a constant number of RMRs is met by ensuring that each process can be a team head for only a constant number of phases, in each of which it may perform only a constant number of RMRs. After performing this predetermined number of RMRs, the team head selects an idle team member to be the new head, and loses.

An idle member merely waits (in a local-spin loop) until it is informed either that it has become a loser, or that it has become the head of a team. In fact, idle members are not even aware of the team to which they belong; only the head of the team knows its members.

Each team of contender processes is further classified either as *hopeful* or as a *playoff contender*. When a team is first formed (more about team formation shortly), it is hopeful; it will become a playoff contender in phase i if it does not "encounter" any other team in phase i. In each phase i, at most one team becomes a playoff contender; and if one does, it is called the *level-i* playoff contender.

The set of teams evolves from phase to phase as follows.

Initially, in phase 0, every process is the head of a hopeful team that has no other members. For any positive integer i, suppose that at the end of phase $i-1$, the contenders are partitioned into a set of teams. We now explain how the set of teams evolves during phase i. There are three possible outcomes for each hopeful team:

1. All members of the team become losers. The algorithm ensures that this does not happen to all hopeful teams.

2. The team becomes a level-i playoff contender. This happens for at most one hopeful team.

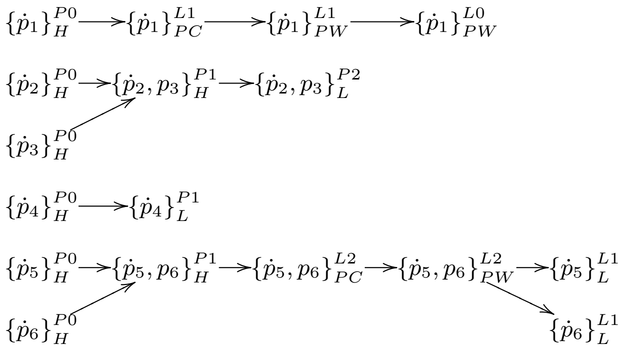
3. The team merges with other phase-i teams, forming a new, larger, phase-$(i+1)$ hopeful team. This new team proceeds to participate in phase $i+1$. The head of the original phase-i team may leave the new team and lose.

We prove that any hopeful team formed in phase i has at least $i+1$ members. Thus, the level-i playoff contender team (if one exists) has at least i members. The number of hopeful teams decreases in each phase, and eventually only playoff contender teams remain, say at the end of some phase $\ell \leq n$ (in fact it can be shown that $\ell \in O(\log n)$). Furthermore, for each $i \in \{1, \ldots, \ell\}$, there is at most one level-i playoff contender team. All such teams compete to select an *overall playoff winner* team, one of whose members is finally elected to be the overall leader.

The overall playoff winner team is selected as follows. Clearly there is a level-ℓ playoff contender team, where phase ℓ is the phase in which the last remaining hopeful team became a playoff contender. That team also becomes the level ℓ playoff winner. For each level i from $\ell-1$ down to 1, a level-i playoff winner team is determined as follows. The head of the level-$(i+1)$ playoff winner team and the level-i playoff contender team, if it exists, enter a two-process competition (leader election). The winner's team becomes the level-i playoff winner team. If there is no level-i playoff contender team, then the head of the level-$(i+1)$ playoff winner team will certainly win the competition, since there is no opponent.

In order to ensure that every process performs at most a constant number of RMRs during playoffs, the head of the level-$(i+1)$ playoff winner team for $i \geq 1$ selects a new team head to compete in level i, and then leaves the team. Since a level-j playoff contender team has at least j members, it follows that the resulting level-i playoff winner team is not empty. In particular, the level-1 playoff winner is not empty and becomes the level-0 playoff winner.

Finally, the algorithm elects a member of the level-0 playoff winner team (which, by the above argument, has at least one member). All other members of that team become losers. So, at the end of the algorithm, exactly one of the participating processes is elected as the leader, and all others become losers. An example execution of the algorithm is illustrated in Figure 1.



Notation: $\{p_1, \ldots, p_k\}_T^N$ represents a team consisting of $p_1, \ldots, p_k$ where T indicates the team type (H = hopeful, L = loser, PC = playoff contender, PW = playoff winner) and N denotes the corresponding team-building phase number (Pi for phase i) or playoff level (Li for level i). Dotted process IDs denote team heads.

Figure 1: Example of team evolution over time ($\rightarrow$ direction) leading to election of p_1.

3. DETAILED DESCRIPTION OF THE ALGORITHM FOR THE DSM MODEL

This section is organized as follows. In Section 3.1 we describe the notation that we use in the pseudo-code of the algorithm. In Section 3.2 we describe `LeaderElect`, the algorithm's main function. Section 3.3 gives a detailed description of the procedure for merging teams.

To simplify presentation as much as possible, the algorithm presented in Sections 3.2 and 3.3 uses a single variable for representing a set of idle team members, which requires $\Theta(n)$-bit words; in Section 3.4 we describe a variant of the algorithm that works with $\Theta(\log n)$-bit words.

3.1 Notational Conventions

In the algorithm pseudo-code provided in this section, we use the following notational conventions. Shared variables that exist over the entire duration of the algorithm are denoted by uppercase names; short-lived variables with function scope are denoted by lowercase names. Suppose that each process has its own local instance of variable V. We write V_p whenever we need to indicate that a pseudo-code line references the instance of V local to process p. We simply write V to indicate that the variable being referenced is the instance of V that is local to the process that executes the pseudo-code.

The algorithm proceeds in merging phases. Different merging phases use different "copies" of helper functions that operate on distinct sets of shared variables. One possible way of reflecting that in the code is to explicitly pass a *phase number* parameter to each function and then to index an array with this parameter whenever a "per-phase" variable is accessed. This, however, has the undesirable effect of cluttering the code and correctness proofs.

Instead, we use the following notational convention. In function calls made from `LeaderElect`, which is the main function of the algorithm, the name of the called function is indexed with the phase number. This signifies that the called function, and all the subfunctions it calls (either directly or indirectly) access the copy of the data structure corresponding to this phase. As an example, in line **5** of `LeaderElect` the following call is made: $\texttt{MergeTeam}_Z(T)$. When this call to `MergeTeam` executes, any reference to a shared variable done by it (or by the functions it calls) ac-

Algorithm 1: LeaderElect

```
   Output: A value in {win, lose}.
1  T ← ∅, Z ← 0, S ← success, work ← 0
2  while work < 3 ∧ S = success do
3  │  work ← work + 1
4  │  Z ← Z + 1
5  │  (S, T) ← MergeTeam_Z(T)
6  │  if T = ⊥ then
7  │  │  wait until T ≠ ⊥
8  │  end
9  end
10 if S = playoff ∧ Z ≥ 1 then
11 │  s ← 2PLeaderElect_Z()
12 │  if s = lose then
13 │  │  S ← lose
14 │  else
15 │  │  Z ← Z − 1
16 │  end
17 end
18 if S = playoff ∧ Z = 0 ∧ T = ∅ then
19 │  return win
20 end
21 if T ≠ ∅ then
22 │  q ← arbitrary process in T
23 │  write Z → Z_q
24 │  write S → S_q
25 │  write T − {q} → T_q
26 end
27 return lose
```

cesses the Z'th copy of that variable. An exception to this rule is the variable `PID` that stores a process identifier: every process has a single copy of `PID`.

3.2 The Function LeaderElect

The `LeaderElect` function is the main function of the algorithm. Let q be a process that executes it. `LeaderElect` uses the variables T, Z, S, and *work*, all local to q. Whenever q is a team head, the variable T stores the identifiers of all the idle members in q's team. Whenever q is an idle member, T is $\emptyset$. T is initialized to $\emptyset$, because when the algorithm starts, q is the head and single member of its team. The other variables, described in the following, are meaningful only when q is a team head. When q's team is hopeful, Z stores the number of the phase in which q's team is participating. If q's team becomes a playoff contender, then Z stores the playoff level in which q's team will compete next.

The status variable S has a value in {`lose`, `playoff`, `success`}. When q's team is hopeful, S equals `success`. When q's team is a playoff contender, S equals `playoff`. If $S =$ `lose`, then q's team has lost and all its team members are bound to lose, too. The variable *work* counts the number of merging phases that are performed by q as a team head.

Variable initialization is done in line **1**. In the while loop of lines lines **2**–**8**, q participates in at most three merging phases. As we prove, participating in three phases is enough to guarantee that the team size strictly increases from phase to phase. Before each phase, q increments *work* (line **3**) and Z (line **4**) to indicate its participation in another merging phase. Process q then calls the `MergeTeam` function. `MergeTeam`, described in detail in Section 3.3, is the heart of our algorithm. It implements the merging algorithm and returns a pair of values that are stored to q's S and T variables (line **5**). If the second value returned by `MergeTeam` (and stored to T) is $\perp$, then q is now an idle member of a new team. In this case, q spins on variable T until it becomes a team head again (line **7**). If q is the head of a team that competes in playoff level Z, for $Z \geq 1$ (line **10**), then q invokes the `2PLeaderElect` function, a constant-RMR two-process leader election algorithm whose details are presented in the full version of this paper. This step is skipped when q competes in playoff level 0 since there is no level 0 playoff contender team and q wins by default.

If q wins the level-i playoff competition, then it decrements Z (line **15**), as its team will next compete on level $i-1$. If the current level is 0 and q's team contains no idle team members (line **18**), then q is the single leader elected by the algorithm (line **19**). Otherwise, either q's team needs to participate in additional playoff competitions, or a process from q's team will eventually win. In either case, q arbitrarily selects an idle team member to be the new head (line **22**), copies its state to the local memory of the new head (lines **23**–**25**) and then loses (line **27**).

If q loses the level-i playoff competition, it sets S to `lose` (line **13**). Then, if its team is non-empty, q copies its state to a new head chosen from its team, making sure that all other team members eventually lose also (lines **21**–**25**). In either case, q loses (line **27**).

3.3 The Merging Algorithm

The merging algorithm is employed in every merging phase in order to coalesce phase-i teams into larger teams that proceed to participate in subsequent phases. The processes that participate in the merging algorithm of phase i are the heads of phase-i teams.

Each merging phase consists of several stages. As the algorithm is asynchronous, teams participating in different phases and stages may co-exist at any point of time. A merging phase consists of the following stages.

- **Finding other processes** — Every phase-i process that completes this stage, except for possibly one (subsequently called the *special process*), becomes aware of another phase-i process. In other words, it reads the `PID` of another phase-i process.

- **Handshaking** — Every non-special process q tries to establish a virtual *communication link* with the process it is aware of, p. If a link from p to q is successfully established, then p eventually becomes aware of q. This implies that p can write a message to q's local memory and spin on its local memory until q responds (and vice versa). Thus, after the establishment of the link, p and q can efficiently execute a reliable two-way communication protocol.

- **Symmetry breaking** — The output of the handshaking protocol is a directed graph over the set of participating processes, whose edges are the virtual communication links. This graph may contain cycles. In the symmetry-breaking phase, these cycles are broken by deleting some of these links and maintaining others. The output of this stage is a directed forest whose nodes are the heads of phase-i teams.

- **Team merging** — Each tree of size two or more in the resulting forest is now coalesced into a single, larger, phase-$(i+1)$ team. The head of the new team is a process from the phase-i team that was headed by the tree's root. The identifiers of all processes in the tree are collected and eventually reported to the new head.

The output of the merging phase is a set of new hopeful teams that proceed to participate in phase $i+1$ and, possibly, a single level-i playoff contender team. We now describe the algorithms that implement the above four stages in more detail.

3.3.1 Finding Other Processes

This stage is implemented by the `Find` function. It employs a splitter-like algorithm. In our implementation, the splitter consists of shared variables F and G. (Note that different instances of F and G are used by `Find` in different merging phases. See Section 3.1.) When process p executes `Find`, it first writes its identifier to F. It then reads G. If the value read is not $\perp$, then p has read from G the identifier of another process and `Find` returns that value. Otherwise, p writes its identifier to G and reads F. If the value read is the identifier of a process other than p, then it is returned by `Find`. Otherwise, `Find` returns $\perp$. Clearly a process incurs a constant number of RMRs as it executes the `Find` function. The proof of the following lemma is provided in the full version of the paper.

LEMMA 1. *The following claims hold.*

(a) *A call of* `Find` *by process p returns $\perp$ or the ID of some process $q \neq p$ that has called* `Find` *before p's call of* `Find` *terminated.*

(b) *At most one of the processes calling* `Find` *receives response $\perp$.*

3.3.2 Handshaking

Except for at most a single *special* process, which receives $\perp$ in response to a `Find` call, every process that calls `Find` becomes aware of one other process. Because of the asynchrony of the system, however, this information is not necessarily useful. E.g., it might be that p becomes aware of q but then p is delayed for a long period of time and q proceeds further in the computation or even terminates without being aware of p. Thus, if p waits for q, it might wait forever. The handshaking stage consists of a protocol between processes, through which they efficiently agree on whether or not they can communicate. The output of this stage for each process p is a list of outgoing links to processes that became aware of p (by calling `Find`) and, possibly, also a link to p from the single process it became aware of. If p and q share a link, then, eventually, both of them are aware of each other and of the existence of the virtual link between them.

The handshaking stage is implemented by the functions `LinkRequest` and `LinkReceive`. If q is aware of p, then q calls `LinkRequest(p)` to try to establish a link with p. Thus, a process calls `LinkRequest` at most once. We say that a *link from p to q is established*, if q's call to `LinkRequest(p)` returns 1.

A process p calls `LinkReceive` to discover its set of outgoing links. Technically, p and q perform a two-process leader election protocol to determine whether or not a link from p to q is established. This protocol is *asymmetric*, because it ensures that p (the recipient of link establishment requests) incurs no RMRs, whereas q (the requesting process) incurs only a constant number of RMRs.

Function `LinkRequest(p)`

Input: Process ID p
Output: a value in $\{0, 1\}$ indicating failure or success, respectively

```
1 write 1 → A_p[PID]
2 s ←read(B_p)
3 if  s = ⊥ then
4 |  link ← 1
5 else
6 |  link ← 0
7 end
8 write link → LINK_p[PID]
9 return link
```

Function `LinkReceive`

Output: set of processes to which link was established

```
1 B ← 1
2 forall process IDs q ≠ PID do
3 |  if A[q] = ⊥  then
4 |  |  LINK[q] ← 0
5 |  else
6 |  |  wait until LINK[q] ≠ ⊥
7 |  end
8 end
9 return {q | LINK[q] = 1}
```

The handshaking protocol to establish links with p uses the array $A_p[]$ and the variable B_p. (Note that different instances of these variables are used in different merging phases. See Section 3.1.) Processes p and q use B_p and entry q of A_p to agree on whether or not q succeeds in establishing a link with p. The output of this protocol is recorded in the LINK_p array: entry q of LINK_p is set if a link from p to q was established and reset otherwise.

To try and establish a link with p, `LinkRequest(p)` first sets the flag corresponding to q in the array A_p (line **1**). It then reads B_p to a local variable (line **2**). The link from p to q is established if and only if the value read in line **2** is $\perp$. If the link is established, q sets the bit corresponding to it in the array LINK_p, otherwise it resets this bit (lines **3**–**8**).

The execution of `LinkRequest` costs exactly three RMRs (on account of lines **1**, **2** and **8**). Each process calls function `LinkRequest` at most once because no process becomes aware of more than a single other process. On the other hand, it may be the case that many processes are aware of the same process p. Thus, multiple processes may request a link with p. Here we exploit the properties of the DSM model: when p executes `LinkReceive` it incurs no RMRs because it only accesses variables in its local memory segment, possibly waiting by spinning on some of them until a value is written.

When p executes `LinkReceive`, it first writes 1 to B_p (line **1**). Any process q that has not yet read B_p will fail in establishing a link with p. Process p proceeds to scan the array A_p. For each entry $q \neq p$, if q has not written yet to $A_p[q]$ then p resets $\text{LINK}_p[q]$ as the link from p to

q will not be established (lines **3**–**4**). Otherwise, p locally spins on $\text{LINK}_p[q]$ (line **6**) waiting for q to either set or reset this entry (indicating whether a link was established or not, respectively). Finally, the set of processes that succeeded in establishing a link with p is returned (line **9**). The key properties of the handshaking functions are captured by the following lemma.

LEMMA 2.

(a) Each call to `LinkReceive` *terminates.*

(b) Let L be the set returned by p's call to `LinkReceive`*. Then $q \in L$ if and only if a link from p to q is eventually established.*

(c) If q's call to `LinkRequest(`p`)` *terminates before p starts executing* `LinkReceive`*, then a link from p to q is established.*

PROOF.

Part (a): Consider a call to `LinkReceive` by process p. Assume by contradiction that the call does not terminate. This can only happen if a local spin in line **6** on the entry $\text{LINK}_p[q]$ for some process q does not terminate. It follows that $A_p[q] \neq \perp$ since otherwise line **6** is not reached for that q. It also follows that $\text{LINK}_p[q]$ remains $\perp$ forever. This is a contradiction because $A_p[q]$ can only be set to a non-$\perp$ value if q executes line **1** of `LinkRequest(`p`)` and in this case q eventually writes (in line **8** of `LinkRequest(`p`)`) a non-$\perp$ value to $\text{LINK}_p[q]$.

Part (b): We consider three cases.

Case 1: q executes line **2** of `LinkRequest(`p`)` *after* p executes line **1** of `LinkReceive`. In this case q reads a non-$\perp$ value in line **2** of `LinkRequest(`p`)` and eventually writes 0 to $\text{LINK}_p[q]$ in line **8**. Moreover, no process writes a different value to $\text{LINK}_p[q]$. Thus, the call to `LinkRequest(`p`)` made by q returns 0 (i.e. a link from p to q is not established) and $q \notin L$ holds.

Case 2: q executes line **2** of `LinkRequest(`p`)` *before* p executes line **1** of `LinkReceive`. In this case s becomes $\perp$ in line **2** of `LinkRequest(`p`)` and q eventually writes 1 to $\text{LINK}_p[q]$ in line **8** and returns 1. Since q writes to $A_p[q]$ before executing line **2**, it follows that $A_p[q] \neq \perp$ when p executes line **3** of `LinkReceive`. Consequently, no process other than q modifies the value of $\text{LINK}_p[q]$ and so $q \in L$ holds.

Case 3: q does not call `LinkRequest(`p`)` at all. Since $\text{LINK}_p[q]$ is set to 1 only when q executes `LinkRequest(`p`)`, it follows that $q \notin L$.

Part (c): If a call to `LinkRequest(`p`)` by process q terminates before p calls `LinkReceive` then we are under the conditions of Case 2 of the proof of Part (b). Hence $q \in L$ holds, and q's call to `LinkRequest(`p`)` returns 1. □

3.3.3 Symmetry Breaking

The functions `LinkRequest` and `LinkReceive` allow processes to establish communication links between them so that, eventually, both endpoints of each such link are aware of each other. However, the graph that is induced by these communication links may contain cycles. The `Forest` function calls the functions `Find`, `LinkRequest` and `LinkReceive` in order to establish communication links and then deletes some of these links in order to ensure that all cycles (if any) are broken. The deletion of these links may cause some processes to remain without any links. Each team head without links must lose (unless it is special) and, as a consequence, all the processes in its team also lose. It is guaranteed, however, that at least one team continues, either as a playoff contender or as a hopeful team that succeeded in establishing and maintaining a link.

Function `Forest`

Output: A success value in $\{0, 1\}$, a process ID (or $\perp$) and a set of processes

```
p ← Find
if p ≠ ⊥ then
    link ← LinkRequest(p)
else
    special ← 1
    link ← 0
end
ℒ ← LinkReceive()
if link = 1 then
    if ℒ ≠ ∅ ∧ PID > p then
        write 1 → CUT_p[PID]
        p ← ⊥
    else
        write 0 → CUT_p[PID]
    end
else
    p ← ⊥
end
forall q ∈ ℒ do
    wait until CUT[q] ≠ ⊥
    if CUT[q] = 1 then
        ℒ ← ℒ − {q}
    end
end
if p = ⊥ ∧ ℒ = ∅ ∧ special ≠ 1 then
    return (0, ⊥, ∅)
end
return (1, p, ℒ) ;
```

A call to `Forest` made by team head q returns a triplet of values: $(s_q, p_q, \mathcal{L}_q)$. The value of s_q indicates whether q is poised to fail ($s_q = 0$) or not ($s_q = 1$). If $s_q = 1$, then p_q and $\mathcal{L}_q$ specify q's neighbors in the graph of communication links: either p_q stores the identifier of q's parent if a link from p to q remains after cycles are broken, or $p_q = \perp$ if no incoming link to q remains; $\mathcal{L}_q$ is the (possibly empty) set of the identifiers of q's children, namely processes to which links from q remain.

We prove that the parent-child relation induced by these return values is consistent, and that the directed graph induced by that relation (with the edges directed from a node to its children) is indeed a forest: it is acyclic and the in-degree of every node is at most 1. We also prove that this forest contains at most one isolated node, and that at least one process r calling `Forest` does not fail. It follows that all trees in the forest (except for, possibly, one) contain two nodes or more. The teams whose heads are in the same such tree now constitute a new, larger, team that proceeds to the next phase.

A process q executing the `Forest` function first calls the

Find function and stores the returned value in the local variable p (line **1**). If Find returns $\perp$, then q sets the *special* local flag to indicate that it is the single process that is unaware of others after calling Find (line **5**). It also resets the *link* local variable to indicate that it has no parent link (line **6**). Otherwise, q requests a link from p and stores the outcome in *link* (line **3**). Regardless of whether q is special or not, it calls LinkReceive to obtain the set of links from it that are established (line **8**).

Lines **9**–**24** ensure that all cycles resulting from the calls to LinkRequest and LinkReceive (if any) are broken. Process q first tests if a link from its parent was established (line **9**) and if its set of outgoing links is non-empty (line **10**). If both tests succeed, then q may be on a cycle. In that case q deletes the link from its parent if and only if its identifier is larger than its parent's (line **10**). As we prove, this guarantees that all cycles (if any) are broken. To delete its link from p, process q writes 1 to the entry of the CUT_p array that corresponds to it (line **11**). Otherwise, q writes 0 to that entry so that p would know that this link is maintained (line **14**).

After dealing with the link from its parent, q waits (by spinning on the entries of its local CUT array) until all the processes that initially succeeded in establishing links from q indicate whether they wish to delete these links or to maintain them (lines **19**–**24**). If q is not special and was made isolated after the deletion of links, then the Forest function returns a code indicating that q should lose (line **26**). Otherwise, Forest returns $(1, p_q, \mathcal{L}_q)$ (line **28**), indicating that q should continue participating in the algorithm, as well as identifying q's parent and children in the resulting forest.

It is easily verified that a process executing Forest incurs a constant number of RMRs. Consider a set P of $m \geq 1$ processes, each calling Forest exactly once. Let $G = (V, E)$ be the directed graph where $V \subseteq P$ is the set of processes q with $s_q = 1$ and E is the set of edges (u, v) with $p_v = u$. The following lemma describes the correctness properties of Forest.

LEMMA 3.

(a) Every call to Forest *terminates.*

(b) If $p_v = u$ and $u \neq \perp$ then $u, v \in V$. Moreover $(u, v) \in E$ if and only if $v \in \mathcal{L}_u$.

(c) G is a forest.

(d) $|V| \geq 1$ and there is at most one vertex in V with (both in- and out-) degree 0.

PROOF.

Part (a): To obtain a contradiction assume there is a process r whose call to Forest does not terminate. Since r may only wait in line **20**, there must be a process $q \in \mathcal{L}$ such that $\mathrm{CUT}_r[q]$ is never set to a non-$\perp$ value. Since $q \in \mathcal{L}$, it follows from Lemma 2 (b) that q calls LinkRequest(r) and that a link from r to q is eventually established. Consequently, q eventually executes either line **11** or line **14** of the Forest function, a contradiction.

Part (b): Let $p_v = u$. It follows trivially from the algorithm that if $p_v \neq \perp$ then $s_v = 1$. Since $p_v = u$ it follows that when v executes line **1** of Forest, the variable p obtains the value u. Since p is not set to $\perp$ in line **17**, *link* must be set to 1 in line **3** and thus the link from u to v is established. Moreover, since v does not execute line **12**, it writes 0 into $\mathrm{CUT}_u[v]$ in line **14** and does not delete the link from u. Now consider u's execution of Forest. Since the link from u to v is established, from Lemma 2 (b), the set $\mathcal{L}$ returned by the call made by u to LinkReceive in line **8** contains v. Since $\mathrm{CUT}_u[v]$ is eventually set to 0, $s_u = 1$ and $v \in \mathcal{L}_u$ hold. Hence we have shown that $p_v = u$ implies $s_v = s_u = 1$ and $v \in \mathcal{L}_u$. Hence u and v are nodes in V and the edge relation $(u, v) \in E$ is well-defined for these nodes. Moreover, this already shows the direction $(u, v) \in E \Rightarrow v \in \mathcal{L}_u$.

For the other direction assume that $v \in \mathcal{L}_u$ holds (note that this implies also $s_u = 1$). Then $v \in \mathcal{L}$ holds after u executes line **8**, and a link from u to v is established. Hence, u executes line **20** for $q = v$ and eventually reads $\mathrm{CUT}_u[v] = 0$ (otherwise v would be removed from $\mathcal{L}$). It follows that v writes 0 to $\mathrm{CUT}_u[v]$. This implies, in turn, that v executes line **14** of Forest and that v's local variable p is set to u. Since p cannot be changed after that, v's call to Forest returns the triplet $(1, u, \mathcal{L}_u)$ and by definition $(u, v) \in E$ holds.

Part (c): By definition, $(u, v) \in E$ implies $p_v = u$. Hence the in-degree of every node in V is at most 1 and it suffices to prove that G is acyclic. To obtain a contradiction, assume G contains a directed cycle. Let $v_0, v_1, \ldots, v_{k-1}$ be the nodes on the cycle, i.e. $(v_i, v_{(i+1) \bmod k}) \in E$. Let $v_i = \max\{v_0, \ldots, v_{k-1}\}$ and assume w.l.o.g. that $i = 1$. From assumptions, $p_{v_1} = v_0$. Thus, v_1 executes line **10** of Forest when the value of its local variable p equals v_0. Moreover, v_1's call of LinkReceive in line **8** must return a set that contains v_2 (otherwise, from Lemma 2(b) and part (b) of this lemma, the link from v_1 to v_2 would not have been established). Hence, immediately after v_1 executes line **10**, $|\mathcal{L}| \geq 1$ holds. From the choice of v_1, we have $v_1 > p = v_0$. It follows that v_1 writes 1 to $\mathrm{CUT}_{v_0}[v_1]$ in line **11**. Now consider the execution of Forest by v_0. It is easily verified that v_0 removes v_1 from its set $\mathcal{L}$ by executing line **22** (for $q = v_0$). From part (b) of this lemma, $(v_0, v_1) \notin E$ holds. This is a contradiction.

Part (d): We say that a process $q \in P$ *loses* if $q \notin V$, i.e. $s_q = 0$ holds. From line **25**, a process with no children and no parent loses iff its local variable *special* does not equal 1. From Lemma 1 (b), the call to Find (in line **1**) returns $\perp$ for at most one process. Hence, there is at most one process for which the variable *special* is set to 1. It follows that at most a single node in G has no parent and no children.

It remains to show that V is not empty. If there is a process v^* for which the variable *special* is set to 1 in line **5**, then this process does not lose and so $v^* \in V$ holds. Assume otherwise. Then, for every process in P, the call to Find (in line **1**) returns a non-$\perp$ value. Let G' be the directed graph (P, E'), where $(u, v) \in E'$ iff v's call of Find in line **1** returns u. From assumptions, every node in P has an in-edge in G' and so G' contains a directed cycle. Let $(v_0, v_1, \ldots, v_{k-1}, v_0)$ be one such cycle, i.e. $(v_i, v_{(i+1) \bmod k}) \in E'$. The existence of this cycle implies that each process v_i, $0 \leq i < k$, calls LinkRequest($v_{(i-1) \bmod k}$) in line **3** of Forest. Let j, $0 \leq j < k$, be an index such that no process v_i, $0 \leq i < k$, $i \neq j$, finishes its execution of line **3** *before* process v_j does so. Hence, v_j finishes its call of LinkRequest($v_{(j-1) \bmod k}$) in line **3** before $v_{(j-1) \bmod k}$ calls LinkReceive and, according to Lemma 2 (c), a link from $v_{(j-1) \bmod k}$ to v_j is established.

Now let $E'' \subseteq E'$ be the set of established links, let $U \subseteq P$ be the set of processes that are an endpoint of at least one of these links, and let $G'' = (U, E'')$. We have already shown that $U \neq \emptyset$ and $E'' \neq \emptyset$ hold. Let $v := \max U$. We finish the proof by showing that $v \in V$, i.e. that v does not lose.

To obtain a contradiction, assume that v loses. It follows that when v executes line **25** of `Forest`, $p = \perp$ and $\mathcal{L} = \emptyset$ hold. Since $v \in U$, there must be another process u such that either $(u, v) \in E''$ or $(v, u) \in E''$ holds.

Case 1, $(v, u) \in E''$: Since a link from v to u was established, $u \in \mathcal{L}$ after v has finished line **8**. Process u can be removed from $\mathcal{L}$ only if v executes line **22**. As our assumptions imply that $\mathcal{L} = \emptyset$ holds when v executes line **25**, it must be that $\mathrm{CUT}_v[u]$ is set by u to 1 when it executes `Forest`. This can only happen if u executes line **11** with $p = v$. However, from our choice of v, $v > u$ holds and so the test of line **10** performed by u fails. Consequently u does not execute line **11**. This is a contradiction.

Case 2, $(u, v) \in E''$: In this case a link from u to v is established. It follows that when v executes line **1** of `Forest`, p gets value u. It also follows that v's call to `LinkRequest`(u) in line **3** returns 1. This implies, in turn, that p can be set to $\perp$ only in line **12**. However, because of the test in line **10**, this is only possible if $\mathcal{L} \neq \emptyset$ when v executes line **10**. Hence, there must be a process $u' \in P$ such that a link from v to u' has been established. Thus $(v, u') \in E''$ holds and we are under the conditions of Case 1 with $u = u'$. ☐

3.3.4 Putting it All Together: Team Merging

Merging phases are implemented by the `MergeTeam` function. It is called by a head of a phase-i hopeful team, q, and receives the set of q's (phase-i) idle team members as a parameter. Process q first calls `Forest` to try and merge its team with other phase-i teams (line **1**). As a response, it receives from `Forest` a triplet of values: $(s, p, \mathcal{L})$. Process q then tests whether it lost by checking whether $s = 0$ holds (line **2**), in which case it returns a `lose` response, along with its set of idle members, $\mathcal{T}$ (line **3**). In the main algorithm this will trigger a process in which all the idle members in q's team eventually lose also. If q did not lose, it checks whether it is the single isolated node of the graph induced by the return values of `Forest` (line **5**), in which case it returns the `playoff` status along with its unchanged set of idle members (line **6**). Process q's team is now the level-i playoff contender. Otherwise, q proceeds to perform the team-merging stage as follows. First, q adds its new children (whose identifiers are in $\mathcal{L}$) to the set $\mathcal{T}$ (line **8**). Next, it waits until each new child $r \in \mathcal{L}$ writes its set of idle members into entry $S_q[r]$. Then, q adds all these members to $\mathcal{T}$ (lines **9**–**12**). If q is the head of the new phase-$(i+1)$ team (line **13**), it returns a `success` status along with its new set of idle members (line **14**). Otherwise, q is an idle member of the new team, so it writes its set of idle members to the local memory of its new parent (line **16**), returning a `success` status and an empty set to indicate that it is now an idle team member (line **17**).

Let P be the set of team heads calling `MergeTeam`. Also, for $a \in P$, let $\mathcal{T}_a$ denote the set of p's idle team members. Thus a's team is the set $\{a\} \cup \mathcal{T}_a$. Now let all team heads $a \in P$ call `MergeTeam`$(\mathcal{T}_a)$ and let $(ret_a, \mathcal{T}'_a)$ be the corresponding return values (we prove that all these function calls terminate). A team head a can either *lose*, *succeed* or its team becomes a playoff contender, as indicated by the return value ret_a. Let $P' \subseteq P$ be the set of processes that succeed and remain team heads after their call to `MergeTeam` returns (i.e. the heads of the remaining hopeful teams). We denote by $\mathcal{T}^*$ the team that becomes playoff contender, i.e. the set consisting of that team's head and idle team members. If no team becomes a playoff contender during the call to `MergeTeam`, then $\mathcal{T}^* = \emptyset$.

Function `MergeTeam(`$\mathcal{T}$`)`

Input: A set $\mathcal{T}$ of process IDs
Output: A status in $\{$`lose`, `playoff`, `success`$\}$ and either a set of process IDs or $\perp$

1 $(s, p, \mathcal{L}) \leftarrow$ `Forest`()
2 **if** $s = 0$ **then**
3 | **return** (`lose`, $\mathcal{T}$)
4 **end**
5 **if** $p = \perp \ \wedge \ \mathcal{L} = \emptyset$ **then**
6 | **return** (`playoff`, $\mathcal{T}$)
7 **end**
8 $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{L}$
9 **for** $r \in \mathcal{L}$ **do**
10 | **wait until** $S[r] \neq \perp$
11 | $\mathcal{T} \leftarrow \mathcal{T} \cup S[r]$
12 **end**
13 **if** $p = \perp$ **then**
14 | **return** (`success`, $\mathcal{T}$)
15 **else**
16 | **write** $\mathcal{T} \rightarrow S_p$[`PID`]
17 | **return** (`success`, $\perp$)
18 **end**

The proof of the following lemma is provided in the full version of the paper. Part (d) implies that the size of hopeful teams increases from phase to phase. This is required, together with parts (b) and (c), in order to ensure the progress of the leader election algorithm. Part (e) ensures that we maintain the semantic correctness of our notion of a team, i.e. that each process is member of exactly one team and that every team has exactly one team head.

LEMMA 4. *The following claims hold.*

(a) Each call to the function `MergeTeam` *terminates.*

(b) At most one team becomes a level-i playoff contender.

(c) At least one team does not lose.

(d) For every process $a \in P'$ there is a different process $b \in P$ such that $\mathcal{T}_a \cup \mathcal{T}_b \cup \{b\} \subseteq \mathcal{T}'_a$.

(e) The following sets partition $\bigcup_{a \in P} \mathcal{T}_a \cup P$: $\mathcal{T}^*$, P', $\mathcal{T}'_b$ for $b \in P'$, and the set of processes in teams whose head $a \in P$ loses.

Based on the above lemma, the following theorem establishes the correctness of the algorithm. The proof is provided in the full version of the paper and relies on the observation that every team in phase k or level k, for $k \geq 1$, has at least k team members.

THEOREM 5. *Let P be a non-empty set of processes executing the algorithm* `LeaderElect`. *Then each process in P performs a constant number of RMRs, exactly one process returns* `win` *and all other processes return* `lose`.

It is easy to see that the space complexity of the algorithm is $O(n^2 \log n)$. It can also be shown that the response time (as defined in [9]) is $O(n \log n)$, despite the fact that the algorithm has constant RMR complexity.

3.4 Reducing Word Size Requirements

As mentioned earlier, the algorithm presented above requires a word size of $\Theta(n)$ for storing a team set in a single word. We now describe a simple modification that makes the algorithm work with realistic $O(\log n)$-bit variables. The key idea is that we represent a team set as a linked list of process identifiers.

The only functions that are modified are `LeaderElect` and `MergeTeam`, since all other functions operate on processes and do not manipulate teams at all. In the following description of the required changes, p is the process that executes the code.

3.4.1 MergeTeam

Let $p \to a_1 \to a_2 \ldots \to a_l$ be the linked list representing p's team set when p starts executing `MergeTeam`. In lines **8**–**12** of the pseudo-code of `MergeTeam`, presented in Section 3.3, p merges its team with the teams headed by all the processes in the set $\mathcal{L}$, the set of its children in the forest.

The new algorithm only merges p's team with some processes from the team of a *single* child $q \in \mathcal{L}$. In phase one and two, q's team has a size of one and two, respectively, and q's complete team is merged into p's team. Now assume that `MergeTeam` is called in phase three or higher and let $q \to b_1 \to b_2 \ldots \to b_m$ be the linked list representing q's team set. In this case p adds only b_1 and b_2 to its team set (it is easy to see that $m \geq 2$). Thus, the new team is now represented by the list $p \to b_1 \to b_2 \to a_1 \to a_2 \ldots \to a_l$. It can easily be verified that this can be done by p in a constant number of RMRs. This is enough to guarantee that team size strictly increases from phase to phase, as needed to establish Theorem 5. Thus, the correctness of the algorithm and the constant RMR complexity are maintained.

As we only add some of the processes from q's team to p's team, the teams headed by all the other processes in $\mathcal{L}$, as well as the remaining members of q's team, must lose. This is easily accomplished by starting a "lose process" along the linked lists of these processes, in which each of them notifies the next process that the team must fail, and then itself fails.

3.4.2 LeaderElect

Wherever in the original `LeaderElect` algorithm a process p checks whether p's team set equals $\perp$ (lines **6**, **7**, **18**, **21**), in the new `LeaderElect` algorithm p checks whether it is the last element of the linked list. Additionally, instead of selecting an arbitrary process to be the new head in line **22**, in the new algorithm p simply assigns the next process in the linked list to be the new head. Line **25** is no longer required since the next process in the list has a linked list of the remaining idle team members.

4. EXTENSION TO THE CC MODEL

The algorithm presented in Section 3 has an RMR complexity of $\Theta(n)$ in the CC model due to the loops on lines **2**–**8** of `LinkReceive`, lines **19**–**24** of `Forest`, and lines **9**–**12** of `MergeTeam`. Constant RMR complexity can be achieved by modifying the `LinkRequest` and `LinkReceive` functions so that the set of children returned by `LinkReceive` has size at most one. We denote by `LinkReceive-CC` the modified `LinkReceive` function. We also divide `LinkRequest` into two functions, `LinkRequestA-CC` and `LinkRequestB-CC`, which must be called in that order. In particular, every process that calls `LinkRequestA-CC`(p) must eventually also call `LinkRequestB-CC`(p). Extending the definition from Section 3.3.2, we say that a *link from p to q is established*, if q's call to `LinkRequestB-CC`(p) returns 1.

The DSM version of the leader election algorithm is modified by replacing the function `Forest` with `Forest-CC`, shown below, which incorporates the new calling sequence of the handshaking functions.

Function `Forest-CC`

```
p ← Find
if p ≠ ⊥ then LinkRequestA-CC(p)
ℒ ← LinkReceive-CC()
if p ≠ ⊥ then
    link ← LinkRequestB-CC(p)
else
    special ← 1
    link ← 0
end
/* resume from line 9 of Forest              */
```

The CC version of the handshaking functions where processes request links from p uses the following shared variables: A_p and B_p – integers, initially $\perp$. To perform a call to `LinkRequestA-CC`(p), a process q simply writes its `PID` to A_p. To perform `LinkReceive-CC`, p first saves A_p into a temporary variable, say a. If $a = \perp$, then p writes its own `PID` to B_p and returns $\emptyset$. Otherwise, a is the identifier of some $q \neq p$ that invoked `LinkRequestA-CC`(p), so p acknowledges having seen a by writing a to B_p, and returns $\{a\}$. Finally, to perform `LinkRequestB-CC`(p), q waits until $B_p \neq \perp$, and returns 1 if and only if $B_p = q$.

The modified handshaking functions satisfy the following properties, analogous to Lemma 2. The proof is provided in the full version of the paper.

LEMMA 6.

(a) Each call made to `LinkRequestB-CC`(p) *terminates, provided that p calls* `LinkReceive-CC`.

(b) Let L be the set returned by p's call to `LinkReceive-CC`. *Then $q \in L$ if and only if a link from p to q is eventually established.*

(c) If q's call to `LinkRequestA-CC`(p) *terminates before p starts executing* `LinkReceive-CC`, *then a link from p to* some *process (not necessarily q) is eventually established.*

Straight-forward extensions of the proofs of Lemma 3 and Theorem 5 yield analogous results for the CC variant of the leader election algorithm, where `Forest` is replaced by `Forest-CC`.

5. THE RMR COMPLEXITY OF ONE-TIME TEST-AND-SET

In this section we describe a *linearizable* [14] simulation of an n-process one-time test-and-set object by our leader election algorithm. A one-time test-and-set object assumes values from $\{0, 1\}$ and is initialized to 0. It supports a single operation, *test-and-set*. The test-and-set operation atomically writes 1 to the test-and-set object and returns the previous value.

Consider first the DSM model. Suppose that an algorithm A uses a one-time test-and-set object T that is local to some process p. Our goal is to be able to "plug" our simulation of all such objects T into A with only a constant blowup in the RMR complexity. Thus, as T resides in the local memory segment of process p, our simulation should allow p to apply operations to T without incurring RMRs at all. Any process $q \neq p$ should incur $O(1)$ RMRs when it applies an operation to T.

Our simulation uses three objects: an $(n-1)$-process constant-RMR leader election object LE_p (that can be implemented by using our leader election algorithm), a two-process constant-RMR leader election object $2LE_p$ (the implementation, discussed in the full version of the paper, must be asymmetric so that p incurs no RMRs), and a read-write register R_p initialized to $\perp$. As indicated by the subscript p, all these objects reside in p's local memory segment.

To apply the test-and-set operation on T, a process $q \neq p$ first reads R_p. If the result is not $\perp$, then q loses (i.e. returns 1). Otherwise, q writes its ID to R_p and then executes the $(n-1)$-process leader election algorithm of LE_p. If it is elected, q proceeds to compete against p on $2LE_p$. Only if it is also elected here does q win (i.e. return 0), otherwise it loses.

To apply the test-and-set operation on T, process p (to which T is local) first reads R_p. If it is not $\perp$, then q loses. Otherwise, p writes its ID to R_p and then competes on $2LE_p$ against the leader elected on LE_p (if any). Finally, q returns 0 if and only if it wins $2LE_p$.

It is easily verified that the RMR complexity of the simulation is as required and that the test-and-set operation of exactly one process returns response 0. As for linearizability, note that once an operation on T is completed, every subsequent operation returns 1 after reading a non-$\perp$ value from R_p. Thus, the single operation that returns 0 either completes before or executes concurrently with every other operation, and can always be placed first in the linearization order. The simulation works also in the CC model if `LeaderElect` is modified as per Section 4, though a slightly simpler simulation is possible using a single n-process leader election object. Thus, we get the following result.

THEOREM 7. *Any algorithm using one-time test-and-set objects, reads and writes can be simulated by an algorithm using only reads and writes with only a constant blowup in the RMR complexity in both the DSM and CC models.*

6. CONCLUSIONS AND FUTURE WORK

We have shown that one-time test-and-set can be implemented using atomic reads and writes in the CC and DSM models using $O(1)$ RMRs. It is interesting that our algorithm simultaneously achieves optimal RMR complexity and high response time. We do not currently know whether this is inherent in the leader election problem or merely a feature of our particular solution. In future work we plan to analyze our algorithms with respect to additional time complexity measures, and explore possible complexity trade-offs.

Acknowledgments

The authors are indebted to Vassos Hadzilacos who, in addition to providing useful comments on matters pertaining to this work, was kind enough to write down a high-level overview of our algorithm on which the contents of Section 2 are based. We would also like to thank Faith Ellen Fich for enlightening discussions about the leader election and mutual exclusion problems, as well as Hagit Attiya and the anonymous referees for their insightful comments.

7. REFERENCES

[1] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proc. RTSS 1992*, pp. 154–162, 1992.

[2] J. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Inf.*, 30(3):249–265, 1993.

[3] J. Anderson, T. Herman, and Y. Kim. Shared-memory mutual exclusion: Major research trends since 1986. *Dist. Comp.*, 16(2-3):75–110, 2003.

[4] J. Anderson and Y. Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proc. ACM PODC 2001*, pp. 90–99, Aug. 2001.

[5] J. Anderson and Y. Kim. Nonatomic mutual exclusion with local spinning. In *Proc. ACM PODC 2002*, pp. 3–12, July 2002.

[6] J. Anderson and M. Moir. Wait-free algorithms for fast, long-lived renaming. *Sci. Comp. Prog.*, 25(1):1–39, 1995.

[7] J. Anderson and J. Yang. Time/contention trade-offs for multiprocessor synchronization. *Inf. and Comp.*, 124(1):68–84, 1996.

[8] H. Attiya and A. Fouren. Adaptive and efficient wait-free algorithms for lattice agreement and renaming. *Theory of Comp. Sys.*, 31(2):642–664, 2001.

[9] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Dist. Comp.*, 8(1):1–17, 1994.

[10] R. Cypher. The communication requirements of mutual exclusion. In *ACM Proc. SPAA 1995*, pp. 147–156, July 1995.

[11] E. Dijkstra. Solution of a problem in concurrent programming control. *Comm. of the ACM*, 8(9):569, Sep. 1965.

[12] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[13] M. Herlihy. Wait-free synchronization. *ACM Trans. on Prog. Lang. and Sys.*, 13(1):123–149, Jan. 1991.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Prog. Lang. and Sys.*, 12(3):463–492, July 1990.

[15] Y. Kim and J. Anderson. Adaptive mutual exclusion with local spinning. In *Proc. DISC 2000*, pp. 29–43, Oct. 2000.

[16] Y. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proc. DISC 2001*, pp. 1–15, Oct. 2001.

[17] H. Lee. Transformations of mutual exclusion algorithms from the cache-coherent model to the distributed shared memory model. In *Proc. ICDCS 2005*, pp. 261–270, June 2005.