An $O(1)$ RMRs Leader Election Algorithm*

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

Danny Hendler‡
Department of Computer Science
Ben-Gurion University
Israel
hendlerd@cs.bgu.ac.il

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

January 6, 2010

Abstract

The *leader election* problem is a fundamental coordination problem. We present leader election algorithms for multiprocessor systems where processes communicate by reading and writing shared memory asynchronously, and do not fail. In particular, we consider the cache-coherent (CC) and distributed shared memory (DSM) models of such systems. We present leader election algorithms that perform a constant number of remote memory references (RMRs) in the worst case.

Our algorithms use splitter-like objects [6, 9] in a novel way, by organizing active processes into teams that share work. As there is an $\Omega(\log n)$ lower bound on the RMR complexity of mutual exclusion for n processes using reads and writes only [10], 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; proving this is easy in the CC model, but presents subtle challenges in

*A preliminary version of this paper appeared in [16].

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

‡Part of this work was done while the second author was a postdoc fellow in the University of Toronto and the Technion. Supported in part by the Technion's Aly Kaufman Fellowship.

§Supported by DFG grant WO1232/1-1.

the DSM model, as we explain later. Anderson, Herman and Kim raise the question whether conditional primitives such as test-and-set and *compare-and-swap* can be used, along with reads and writes, to solve mutual exclusion with better worst-case RMR complexity than is possible using reads and writes only [3]. We provide a negative answer to this question in the case of implementing one-time test-and-set.

1 Introduction

The leader election problem is a fundamental distributed coordination problem. In this 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 leader election in multiprocessor systems where processes communicate by reading and writing shared memory. In particular, we consider the cache-coherent (CC) and distributed shared memory (DSM) models of such systems. Processes are asynchronous and do not fail. (In fact, it follows from [1] that no fault-tolerant leader election algorithm that uses only reads and writes exists.) Not every process is required to participate in an execution. Every process is *live*, meaning that once it has begun executing its leader election algorithm, it continues to take steps until the algorithm terminates. The leader election algorithm must eventually terminate provided that every active process is live. Solving the leader election problem subject to these requirements is non-trivial and necessitates inter-process communication; the naive algorithm where some designated process always returns *win* and any other process always returns *false* is incorrect in executions where at least one process is active but the designated process is not active, because no leader is elected in such executions.

We measure time complexity using the *remote memory references* (RMR) complexity measure,

and present algorithms whose worst-case RMR complexity is constant. To the best of our knowledge, our algorithms are the first leader election algorithms that have sub-logarithmic worst-case RMR complexity while relying only on reads and writes.

Our algorithms use splitter-like objects in a novel way to partition processes efficiently into disjoint work-sharing *teams*. Based on these algorithms, we are also able to prove that *any* algorithm for the CC or DSM model using atomic 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. (Our simulation is intended for blocking algorithms and does not preserve progress properties such as lock-freedom.) Thus, one-time test-and-set is no stronger than read and write in terms of RMR complexity in both the CC model and the DSM model.

The leader election problem is closely related to the mutual exclusion problem [13], and leader election may be regarded as one-time mutual exclusion [14]. Thus, any algorithm that solves mutual exclusion also solves leader election. Alur and Taubenfeld proved that for any mutual exclusion or leader election 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 traffic on the processor-to-memory interconnect.

Recent mutual exclusion work also focuses on *local-spin* algorithms, in which all busy-waiting is done by means of read-only loops that repeatedly test locally accessible variables (see, e.g., [4, 5, 19, 20, 22]). 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 in the course of each *passage* (an entry to the critical section, followed by the corresponding exit), where n is the maximum number of processes participating in the algorithm. Yang and Anderson improved on that, and presented a mutual exclusion algorithm based on reads and writes where processes incur $O(\log n)$ RMRs per passage [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 per passage, while retaining $O(\log n)$ worst-case RMR complexity [11]. 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 proved a lower bound of $\Omega(\log n / \log\log n)$ on the worst-case (per passage) RMR complexity of n-process mutual exclusion algorithms that use reads and writes only [4]. This result improved on a previous lower bound of $\Omega(\log\log n / \log\log\log n)$ obtained by Cypher [12]. Fan and Lynch [15] proved an $\Omega(n \log n)$ lower bound on the *state change* cost of read/write mutual exclusion. Recent work by Attiya, Hendler and Woelfel [10] proved a tight lower bound of $\Omega(n \log n)$ on the RMR complexity of read/write mutual exclusion. They prove that every read/write mutual

exclusion algorithm has an execution in which each process performs at most a single passage and the total number of RMRs incurred by the participating processes is $\Omega(n \log n)$.

These lower bounds hold also for algorithms that, in addition to reads and writes, 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 [17].

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]. That is, they ask whether conditional primitives such as test-and-set and compare-and-swap can be used, along with reads and writes, to solve mutual exclusion with better worst-case RMR complexity than is possible using reads and writes only. 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 worst-case RMR complexity for implementing *any* algorithm.

The *system response time* of a mutual exclusion algorithm is the time interval between subsequent entries to the critical section, where a time unit is the minimal interval in which every active process performs at least one step [8, 11]. Similarly, the system response time of a leader election algorithm is the time interval between the time when the execution starts and the time when the leader is determined, where a time unit is defined in the same manner. Table 1 presents worst-case lower bounds on the system response time and space complexity for variants of our leader election algorithm.

Although our algorithms have optimal $O(1)$ RMR complexity, they all have non-constant system

Architecture	Register size (bits)	RMR complexity	space complexity	response time
DSM	$O(n)$	$O(1)$	$\Theta(n^2 \log n)$	$\Theta(n \log n)$
CC	$O(n)$	$O(1)$	$\Theta(n \log n)$	$\Theta(n \log n)$
DSM	$O(\log n)$	$O(1)$	$\Theta(n^2 \log n)$	$\Theta(n \log n)$
CC	$O(\log n)$	$O(1)$	$\Theta(n \log n)$	$\Theta(\log n)$

Table 1: Worst-case complexity comparison for variants of our leader election algorithm.

response time. Specifically, our DSM algorithms have high worst-case response time of $\Theta(n \log n)$. Our results thus establish that low RMR complexity does not imply low system response time.

1.1 Model Definitions and Assumptions

In this paper, we consider multiprocessor systems where processes communicate using atomic reads and writes. Processes take steps asynchronously and do not fail. In each step, a process reads or writes one shared variable and performs a finite amount of local computation. A process is *active* in an execution if it takes at least one step in that execution. The subset of processes that are active in an execution is not fixed in advance. Every active process is *live*, meaning that if its algorithm has not terminated then it eventually completes its next step.

We consider both the cache-coherent (CC) and distributed shared memory (DSM) multiprocessor architectures. Each processor in a CC architecture 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 architecture, each processor instead owns a segment of shared memory that can be locally accessed without traversing the processor-to-memory interconnect. Thus, each 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 that may execute the algorithm on each processor. We denote the set of these processes by P and let n denote its size. (Clearly, the RMR complexity of the algorithm, expressed in terms of n, can only improve if multiple processes execute on some or all of the processors.) A step 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 architecture it is possible for multiple processes to locally spin on the same shared variable. We define the RMR complexity of a leader election algorithm as the maximum number of remote memory references required by a process to execute the algorithm, taken over all processes (winners and losers) and all possible executions.

2 Overview of the Algorithms

In this section we provide a high-level description of our leader election algorithm. To facilitate its understanding, we then present a detailed example computation of the algorithm in Section 2.1.

Our leader election 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 *players*. As the name suggests, a loser cannot become the elected leader, while a player has a chance of becoming the leader. Initially, all processes are players. Once a process becomes a loser, it remains a loser thereafter.

The set of processes is further partitioned into *teams*. Each team has a unique member called *head*; all of 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 achieved by ensuring that each

process can be a team head for only a constant number of merging phases, in each of which it may perform only a constant number of RMRs. After completing this predetermined number of phases, the team head selects an idle team member from its team to be the new head, leaves its team, and loses. Thus, the former team head becomes the head of a new *losing* team that includes only itself.

An idle team 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 a team knows its members.

Each team of players is further classified either as a *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. A hopeful team that exists at the beginning of phase i is called a *phase-i* team. During phase i, such a team may become a *losing* team, as decided by the team head, in which case all players on the team become losers.

The set of teams evolves from phase to phase as follows. Initially, every process is a player and is the head of a hopeful team that includes only itself. For any positive integer i, at the beginning of phase i, the players are partitioned into a set of teams. We now explain how these teams evolve during phase i. There are three possible outcomes for each phase-i hopeful team:

1. The team becomes a losing team, and all of its members become losers. The algorithm ensures that this does not happen to all hopeful teams.

2. The team becomes a level-i playoff contender. This may happen 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.

Our algorithms ensure that any phase-$(i+1)$ hopeful team 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 we prove in Section **??** that $\ell \in O(\log n)$ holds, where n is the maximum number of processes). 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 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 wins the level-i competition by default.

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-j playoff winner team is not empty.

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

2.1 An Example Computation

Figure 1 depicts an example computation performed by processes $p_1, \ldots, p_8$. Each process p_i participates (initially as a team head and later possibly as an idle team member) in one or more merging phases (with phase numbers increasing in the direction of the leftmost arrow of Figure 1). Then, either p_i and its team lose or they contend in one or more playoff levels (with level numbers decreasing in the direction of the rightmost arrow of Figure 1).

At the beginning of merging phase 1, each process is the head of a hopeful team with an empty set of idle members. In the course of phase 1, the phase-1 teams headed by p_2, p_3 and p_4 are combined and proceed as a single phase-2 team headed by p_2. Likewise, the phase-1 teams headed by p_5, p_6 and p_7 are combined and proceed as a single phase-2 team headed by p_6. The team headed by process p_8 becomes the single level-1 playoff contender and will next compete at level 1 of the playoffs. Finally, the team headed by p_1 becomes a losing team after phase 1. It therefore does not proceed to merging phase 2 nor does it become a level-1 playoff contender.

In phase 2, the two phase-2 teams headed by p_2 and p_6 are combined and proceed as a single phase-3 team headed by p_6. No team loses in phase-2 nor is there a level-2 playoff contender. The single phase-3 team, headed by p_6 becomes the single level-3 playoff contender. It then proceeds to compete at level 3 of the playoffs, headed by p_3: p_6 leaves its team and loses after being a team head for three phases.

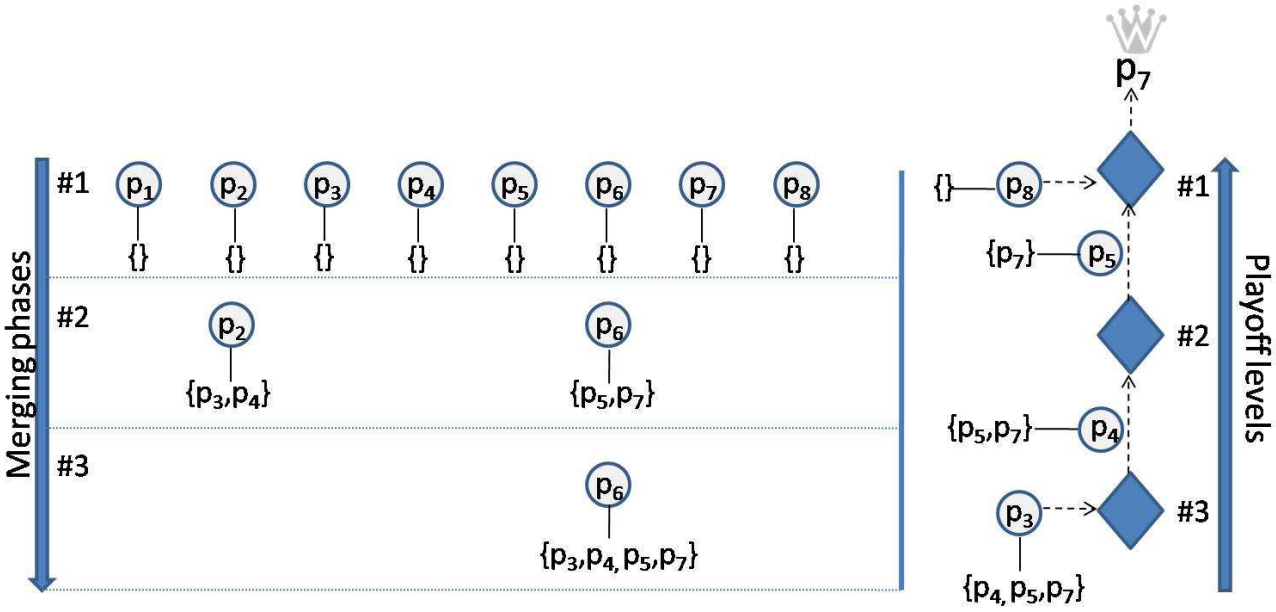

Figure 1: An example computation of our leader election algorithm. Team head processes are encircled and connected by a line segment to the set of idle team members. The diamond shapes on the right represent objects on which processes that participate in the playoffs compete.

The team headed by p_3 is the only competitor at level 3 of the playoffs and thus proceeds to compete at level 2. The team is now headed by p_4: p_3 leaves its team and loses after heading it in level 3 of the playoffs. Since there is no level-2 playoff contender, this team proceeds to compete at level 1. It is now headed by p_5: p_4 leaves the team and loses after being the team head in level 2 of the playoffs.

The teams headed by p_5 and p_8 finally compete at level 1 of the playoffs. Assuming that p_8 was relatively slow, it loses at level 1. The team that was headed by p_5 wins level 1 of the playoffs and is the single overall playoff winner. It now consists only of p_7: p_5 leaves the team and loses after heading it in level 1 of the playoffs. Finally, since p_7 is the single remaining member of this team,

it is elected as a leader.

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. In Section 3.4, we prove the correctness of our algorithm and its constant RMR complexity. 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 3.5 we describe a variant of the algorithm that works with $\Theta(\log n)$-bit words.

3.1 Notational Conventions

We use the following notational conventions in the algorithm pseudo-code. Shared variables are denoted by uppercase names; local variables 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. We denote by $s := A$ the assignment of value A to a private variable s, and by **write** $B := C$ the assignment of value C to shared variable B. Similarly, we denote by s (e.g., $A := s$) the value read from private variable s, whereas **read**(A) (e.g., $B :=$ **read**(A)) denotes the value read from shared variable A. We omit the **read** operator in the special case of local spin loops (e.g., we write **wait until** $Flag = 1$ rather than **wait until read**$(Flag) = 1$).

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

Algorithm 1: LeaderElect

Output: A value in $\{\texttt{win}, \texttt{lose}\}$.

Var.: T – set of process IDs, records the idle team members if process is a team head

Var.: Z – integer, records number of current team merging phase or playoff level if process is a team head

Var.: S – element of $\{\texttt{hopeful}, \texttt{lose}, \texttt{playoff}\}$, records status of process

Var.: $work$ – integer, records number of team merging phases or playoff levels completed by a process as team head

```
/* init per-process variables */
```
1 $T := \emptyset$, $Z := 0$, $S := \texttt{hopeful}$, $work := 0$

```
   /* execute at most three merging
      phases                          */
```
2 **while** $work < 3 \ \wedge \ S = \texttt{hopeful}$ **do**

3 $\quad work := work + 1$

4 $\quad Z := Z + 1$

5 $\quad (S, T) := \texttt{MergeTeam}_Z(T)$

```
   /* see if I became idle          */
```
6 $\quad$ **if** $T = \perp$ **then**

```
   /* wait until either my
      team loses or I become
      a team head again             */
```
7 $\quad\quad$ **wait until** $T \neq \perp$

8 $\quad$ **end**

9 **end**

```
   /* execute one playoff level     */
```
10 **if** $S = \texttt{playoff} \wedge Z \geq 1$ **then**

11 $\quad s := \texttt{TwoProcessLE}_Z()$

12 $\quad$ **if** $s = \texttt{lose}$ **then**

```
   /* team lost                     */
```
13 $\quad\quad S := \texttt{lose}$

14 $\quad$ **else**

```
   /* team won                      */
```
15 $\quad\quad Z := Z - 1$

16 $\quad$ **end**

17 **end**

```
   /* see if I'm the leader         */
```
18 **if** $S = \texttt{playoff} \ \wedge \ Z = 0 \ \wedge \ T = \emptyset$ **then**

19 $\quad$ **return** win

20 **end**

```
   /* I'm not the leader, so choose
      a new team head and lose      */
```
21 **if** $T \neq \emptyset$ **then**

22 $\quad t :=$ arbitrary process in T

```
   /* pass control to t             */
```
23 $\quad$ **write** $Z_t := Z$

24 $\quad$ **write** $S_t := S$

25 $\quad$ **write** $T_t := T - \{t\}$

26 **end**

27 **return** lose

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

Variable initialization is done on line **1**. In the while loop of lines **2**–**8**, p participates as a head in at most three merging phases. As we later prove, participating in three phases is enough to guarantee that the team size strictly increases from phase to phase. (Participating in only two phases is not sufficient.) Before each phase, p increments *work* (line **3**) and Z (line **4**) to indicate its participation in another merging phase. Process p 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 in p's S and T variables (line **5**). The specification of `MergeTeam` is formally stated in Lemma 5, and is informally summarized below:

Specification 1. *For any execution of* `LeaderElect` *and any merging phase number, i, consider the calls to* $\texttt{MergeTeam}_i$ *made by the heads of phase-i hopeful teams. For each such team head p, let T_p denote the argument of p's call to* $\texttt{MergeTeam}_i$ *and let (S'_p, T'_p) denote the response. Then:*

(a) If p's phase-i team becomes the level-i playoff contender then $S'_p =$ `playoff` *and $T'_p = T_p$. (This may happen for at most one p.)*

(b) If p's phase-i team becomes a losing team then $S'_p =$ `lose` *and $T'_p = T_p$. (This may not happen to all phase-i teams.)*

(c) If p's phase-i team becomes part of a phase-$(i+1)$ hopeful team then $S'_p =$ `hopeful`*. Moreover, if p becomes an idle member of this phase-$(i+1)$ team, then $T'_p = \perp$, otherwise p becomes a team head and T'_p is the set of all its team members.*

Note that if $S'_p =$ `hopeful` and $T'_p = \perp$ (i.e., p becomes an idle team member), then p enters a local-spin loop on line **7**, where it waits until it becomes a team head again (i.e., $T \neq \perp$). Once this occurs, p may be part of a losing team, a hopeful team for some merging phase, the playoff contender or winner team for some playoff level, or even the overall playoff winner team.

If p is the head of a team that competes in playoff level Z, for $Z \geq 1$ (line **10**), then p invokes the `TwoProcessLE` function, a constant-RMR two-process leader election algorithm whose details are presented in Section 8. Note that p must check on line **10** whether $Z \geq 1$ in the event that Z_p was overwritten with 0 by another process on line **23**, in which case p is part of the overall playoff winner team (i.e., the level-1 playoff winner team).

If p wins the level-i playoff competition, then it decrements Z (line **15**), as its team will next compete on level $i-1$. If p's team has won level 1 and contains no idle team members (line **18**), then p is the single leader elected by the algorithm (line **19**). Otherwise, either p's team needs to participate in additional playoff competitions, or a process from p's team will eventually win. In either case, p 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 p loses the level-i playoff competition, it sets S to `lose` (line **13**). Then, if its team is non-empty, p 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, p 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 phase $i+1$. The processes that participate in the merging algorithm of phase i are the heads of phase-i hopeful 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 team head that completes this stage, except for possibly one (subsequently called the *special process*), becomes aware of another phase-i team head. That is, it reads from some shared variable the `PID` of another phase-i team head.

- **Handshaking** — Every non-special process p tries to establish a virtual *communication link* with the process it is aware of, say q. If a link from q to p is successfully established, then q eventually becomes aware of p. This implies that q can write a message to p's local memory and spin on its local memory until p 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 forest defines a phase-$(i+1)$ team consisting of the union of phase-i teams whose heads are in the tree. 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 team head.

The output of the merging phase is a set of new hopeful teams for phase i that proceed to participate in phase $i+1$ and, as we will show, at most 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 function `Find`, which 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 (line **1**). It then reads G (line **2**). If the value read is not $\perp$, then p has read from G the identifier of another process and `Find` returns that value (line **10**). Otherwise p writes its identifier to G (line **4**) and reads F (line **5**). If the value read is the identifier of a process other than p, then it is returned by `Find` (line **10**). Otherwise `Find` returns $\perp$ on line **10**. It is easy to see from the code that a process incurs a constant number of RMRs as it executes `Find`. The key properties of `Find` are summarized by Lemma 2.

Function `Find`

Output: Either $\perp$ or the ID of another process that executes `Find`

Var.: F, G – process ID or $\perp$, initially $\perp$

```
1  write F := PID
2  s := read(G)
3  if s = ⊥ then
4  |  write G := PID
5  |  s := read(F)
6  |  if s = PID then
7  |  |  s := ⊥
8  |  end
9  end
10 return s
```

Lemma 2. *For any execution where each process calls* `Find` *at most once:*

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

(b) At most one of the calls to `Find` *returns $\perp$.*

Proof: *Part (a):* The only values that can be written to F or G are process IDs. Hence, the value returned by `Find` is either the ID of a process having called `Find` before p's call terminated or $\perp$. The conditional statement on line **6** ensures that no process returns its own ID.

Part (b): Assume the contrary. Then there are two calls to `Find`, by distinct processes p and q, that return $\perp$. From the code, `Find` returns $\perp$ only if the process that executes it reads its own ID from F on line **5** (the value read here cannot be $\perp$ because the process that executes `Find` has written its process ID to F before executing line **5**). Since a process calls `Find` at most once, it can get response $\perp$ only if it writes `PID` to F on line **1** and reads it back in line **5**. It follows that both p and q execute line **4**. Assume without loss of generality that p writes to F on line **1** before q does. If p writes to G on line **4** before q writes to F, then q must read a non-$\perp$ value on line **2**. This contradicts the fact that q executes line **4**. It follows that by the time p writes to G on line **4**, q has already overwritten p's value in F. This implies, in turn, that p does not read its own ID in line **5**, receiving response $\perp$, a contradiction. □

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 usable. 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.

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

```
Function LinkRequest(p)
    Input: Process ID p
    Output: a value in {0, 1} indicating
            link establishment failure or
            success, respectively
    Var.: A_p[1..N] – array of values in
          {⊥, 1}, one instance per
          process, initially ⊥
    Var.: B_p – value in {⊥, 1}, one
          instance per process, initially ⊥
    Var.: LINK_p[1..N] – array of values in
          {⊥, 0, 1}, initially ⊥
1   write A_p[PID] := 1
2   s := read(B_p)
3   if s = ⊥ then
4   |   link := 1
5   else
6   |   link := 0
7   end
8   write LINK_p[PID] := link
9   return link
```

```
Function LinkReceive
    Output: set of processes to which
            link was established
    Var.: A[1..N], B, LINK[1..N] –
          shared with LinkRequest
1   B := 1
2   forall process IDs q ≠ PID do
3   |   if A[q] = ⊥ then
4   |   |   LINK[q] := 0
5   |   else
    |   |   /* wait for requesting
    |   |      process              */
6   |   |   wait until LINK[q] ≠ ⊥
7   |   end
8   end
9   return {q | LINK[q] = 1}
```

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 no longer be able to establish 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 3. *For any process p, and for any execution where p calls* `LinkReceive` *at most once and every process $q \neq p$ calls* `LinkRequest(`p`)` *at most once:*

(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 the local spin loop on 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 (on 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 on line **2** of `LinkRequest(`p`)` and eventually writes 0 to $\text{LINK}_p[q]$ on 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 $\perp$ on line **2** of `LinkRequest(`p`)` and q eventually writes 1 to $\text{LINK}_p[q]$ on 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, p waits on line **6** of `LinkReceive` until q writes 1 to $\text{LINK}_p[q]$ and $q \in L$ 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 incident 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.

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 this 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 calling `Forest` does not fail. It follows that all trees in the forest (except for, possibly, one) contain two nodes or more.

Function `Forest`

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

Var.: p – ID of process encountered while executing `Find`

Var.: *link* – Boolean, indicates whether comm. link from process p was established

Var.: *special* – Boolean, indicates whether this process encountered no other while executing `Find`.

Var.: $\mathcal{L}$ – set of processes to which comm. link was established.

Var.: CUT[1..N] – array of $\{0, 1, \perp\}$, i'th entry indicates whether the comm. link to process i will be cut ($\perp$ = undecided)

```
   /* look for another process                    */
1  p := Find()
2  if p ≠ ⊥ then
   |  /* request link from p                      */
3  |  link := LinkRequest(p)
4  else
5  |  p = ⊥ /* so I'm special                     */
6  |  special := 1
7  |  link := 0
8  end
   /* see who requested links                    */
9  L := LinkReceive()
   /* break cycles                               */
10 if link = 1 then
   |  /* there is a link from p, so
   |     cut it if needed                         */
11 |  if L ≠ ∅ ∧ PID > p then
   |  |  /* link will be cut                      */
12 |  |  write CUT_p[PID] := 1
13 |  |  p := ⊥
14 |  else
   |  |  /* link will be kept                    */
15 |  |  write CUT_p[PID] := 0
16 |  end
17 else
   |  /* no link, forget p                        */
18 |  p := ⊥
19 end
   /* check for cut links                        */
20 forall l ∈ L do
21 |  wait until CUT[l] ≠ ⊥
22 |  if CUT[l] = 1 then
23 |  |  L := L − {l}
24 |  end
25 end
   /* see if team must lose                      */
26 if p = ⊥ ∧ L = ∅ ∧ special ≠ 1
   then
27 |  return (0, ⊥, ∅)
28 end
   /* team lives on as hopeful or
      playoff contender                          */
29 return (1, p, L)
```

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 **6**). It also resets the *link* local variable to indicate that it has no parent link (line **7**). 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 **9**).

Lines **10**–**25** 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 **10**) and if its set of outgoing links is non-empty (line **11**). 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 **11**). 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 **12**). Otherwise, q writes 0 to that entry so that p would know that this link is maintained (line **15**).

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 **20**–**25**). 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 **27**). Otherwise, `Forest` returns $(1, p_q, \mathcal{L}_q)$ (line **29**), 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 $p_v = u$. (Note that the edges in E correspond to a subset of the links established by calling `LinkRequest`/ `LinkReceive`, namely those that were not cut by the execution of line **12** of `Forest`.) For notational simplicity, in the proofs that follow we do not distinguish between a process and its PID. Thus, e.g., we let v denote both process v and its identifier. The following lemma describes the correctness properties of `Forest`.

Lemma 4. *For any execution where each process calls* `Forest` *at most once:*

(a) Every call to `Forest` *terminates.*

Moreover, letting $(s_p, p_p, \mathcal{L}_p)$ denote the value returned by process p's call to `Forest`*, and letting $G = (V, E)$ denote the directed graph where V is the set of processes that call* `Forest` *and $E = \{(u, v) : p_v = u\}$:*

(b) If $p_v = u$ and $u \neq \perp$ then $u, v \in V$. Furthermore, $(u, v) \in E$ 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 q whose call to `Forest` does not terminate. From Lemmas 2 and 3, all the calls made from `Forest` terminate. Since q may wait inside `Forest` only on line **21**, there is a process $l \in \mathcal{L}$ such that $\text{CUT}_q[l]$ is never set to a non-$\perp$ value. Since $l \in \mathcal{L}$, it follows from Lemma 3 (b) that l calls `LinkRequest(`q`)` and that a link from q to l is eventually established. Consequently, l eventually executes either line **12** or line **15** of the

`Forest` function and sets $\mathrm{CUT}_q[l]$ to a non-$\perp$ value, a contradiction.

Part (b): Let $p_v = u$. It follows by lines **26**–**29** of 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 $\perp$ on line **18**, *link* must be set to 1 on line **3** and thus the link from u to v is established. Moreover, since v does not execute line **13**, it writes 0 into $\mathrm{CUT}_u[v]$ on line **15** 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 3 (b), the set $\mathcal{L}$ returned by the call made by u to `LinkReceive` on line **9** 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 $v \in \mathcal{L}_u$. It follows that u and v are nodes in V and the edge relation $(u, v) \in E$ is well-defined. Moreover, this also establishes 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 **9**, and so a link from u to v is established. Hence, u executes line **21** with $q = v$ 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 **15** 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 $(v_0, v_1, \ldots, v_{k-1}, v_0)$ be such a 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 **11** of `Forest` when the value of its local variable p equals v_0. Moreover, v_1's call of `LinkReceive` on line **9** must return a set that contains v_2 (otherwise, from Lemma 3(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 **11**, $|\mathcal{L}| \geq 1$ holds. From the choice of v_1 and by Lemma 2(a) we have $v_1 > p = v_0$. It follows that v_1 writes 1 to $\mathrm{CUT}_{v_0}[v_1]$ on line **12**. 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 **23** (with $l = v_0$). 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 **26**, a process with no children and no parent loses if and only if its local variable *special* does not equal 1. From Lemma 2 (b), the call to `Find` (on 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 **6**, then this process does not lose and so $v^* \in V$ holds. Assume otherwise. Then, for every process in P, the call to `Find` (on line **1**) returns a non-$\perp$ value. Let G' be the directed graph (P, E'), where $(u, v) \in E'$ if and only if v's call of `Find` on 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(`$v_{(i-1) \bmod k}$`)` on 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(`$v_{(j-1) \bmod k}$`)` on line **3** before $v_{(j-1) \bmod k}$ calls

`LinkReceive` and, according to Lemma 3 (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 **26** 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 **9**. Process u is removed from $\mathcal{L}$ only if v executes line **23**. As our assumptions imply that $\mathcal{L} = \emptyset$ holds when v executes line **26**, 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 **12** with $p = v$. However, from our choice of v and from Lemma 2(a), $v > u$ holds and so the test of line **11** performed by u fails. Consequently u does not execute line **12**. 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 is assigned value u. It also follows that v's call to `LinkRequest(`u`)` on line **3** returns 1. This implies, in turn, that p can be set to $\perp$ only on line **13**. However, because of the test on line **11** and by Lemma 2(a), this is only possible if $\mathcal{L} \neq \emptyset$ when v executes line **11**. 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 $u = u'$. □

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

Input: A set $\mathcal{T}$ of process IDs

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

Var.: $S[1..N]$ – array of either $\perp$ or set of process IDs, one instance per process, initially $\perp$

1 $(s, p, \mathcal{L}) :=$ `Forest()`
2 **if** $s = 0$ **then**
  `/* my team has lost */`
3 **return** (`lose`, $\mathcal{T}$)
4 **end**
  `/* if I have no parent or children */`
5 **if** $p = \perp \ \wedge \ \mathcal{L} = \emptyset$ **then**
  `/* my team is the single playoff contender of this phase */`
6 **return** (`playoff`, $\mathcal{T}$)
7 **end**
  `/* my new children join my team */`
8 $\mathcal{T} := \mathcal{T} \cup \mathcal{L}$
9 **for** $r \in \mathcal{L}$ **do**
10 **wait until** $S[r] \neq \perp$
  `/* my children's offspring join my team */`
11 $\mathcal{T} := \mathcal{T} \cup S[r]$
12 **end**
13 **if** $p = \perp$ **then**
  `/* I am the head of the new team */`
14 **return** (`hopeful`, $\mathcal{T}$)
15 **else**
16 **write** S_p[`PID`] $:= \mathcal{T}$
  `/* I am an idle member of the new team */`
17 **return** (`hopeful`, $\perp$)
18 **end**

3.3.4 Putting it All Together: Team Merging

Merging phases are implemented by the `MergeTeam` function. It is called by the head (say q) of a phase-i hopeful team, 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 chain reaction 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 processes from $\mathcal{L}$ to its set $\mathcal{T}$ (line **8**). (These are q's children in the graph induced by the return value of `Forest`.) Next, q waits until each new child $r \in \mathcal{L}$ writes its set of idle members into $S_q[r]$. 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 `hopeful` 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 `hopeful` status and $\perp$ 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 a'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 $(status_a, \mathcal{T}'_a)$ denote the corresponding return values (we prove that all these function calls terminate).

A team head a can either lose, remain hopeful, or its team becomes a playoff contender, as indicated by the return value $status_a$. Let $P' \subseteq P$ denote the set of processes whose call to `MergeTeam` returns the `hopeful` status. We denote by $\mathcal{T}^*$ the team that becomes a 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 $\mathcal{T}^* = \emptyset$.

The following lemma describes the vital properties of `MergeTeam`. 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 a member of exactly one team and that every team has exactly one team head.

Lemma 5. *For any execution where each process calls* `MergeTeam` *at most once, let P denote the set of processes that call* `MergeTeam`*, for each $p \in P$, let $\mathcal{T}_p$ denote the parameter of p's call, suppose for each $p \in P$ that $\mathcal{T}_p \cap P = \emptyset$, and suppose for any $p, q \in P$ that $p \neq q \Rightarrow \mathcal{T}_p \cap \mathcal{T}_q = \emptyset$. Then, in the execution under consideration:*

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

(b) At most one call returns $(\texttt{playoff}, \bullet)$.

(Informally, at most one team becomes a playoff contender in this merging phase.)

(c) At least one call does not return $(\texttt{lose}, \bullet)$.

(Informally, at least one team does not lose in this merging phase.)

Moreover, letting $(status_p, \mathcal{T}'_p)$ denote the response of p's call, $P' = \{p \in P : status_p = \texttt{hopeful}\}$, and $\mathcal{T}^* = \{p\} \cup \mathcal{T}_p$ where p is the process for which $status_p = \texttt{playoff}$ (or $\mathcal{T}^* = \emptyset$ if there is no

such p):

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

For $a \in P$, let $(s_a, p_a, \mathcal{L}_a)$ be the return value of a's call to `Forest` on line **1**. Let $G = (P, E)$ be a graph over the nodes of P with E defined as follows: $(a, b) \in E$ if and only if $p_b = a$. We first prove parts (a)-(c) and then prove a claim from which parts (d)-(e) follow directly.

Proof of Lemma 5 (a)-(c): *Part (a):* By Lemma 4 (a), the call on line **1** terminates. By definition, $s_a = 0$ for all processes $a \in P - P'$, except for the single process that is the head of $\mathcal{T}^*$ (if any). Thus all these processes exit `MergeTeam` on line **3** or line **6**. It remains to verify that calls to `MergeTeam` made by processes in P' terminate.

We call a vertex $a \in P'$ *bad* if the call to `MergeTeam` by a does not terminate. From Lemma 4 (c), G is a forest. Assume there is a bad vertex in P' and let a be one such bad vertex, for which there is no other bad vertex on the path from a to a leaf. Then a waits indefinitely on line **10** for some $b \in P$. It follows that the set $\mathcal{L}_a$, returned by a's call to `Forest` on line **1**, contains b. Hence b is a child of a in a tree of G and so $p_b = a$ and $s_b = 1$ after b's call to `Forest` on line **1** returns. Moreover, as we assume b is not bad, its call to `MergeTeam` terminates. It therefore eventually executes line **16** (since $s_b = 1$) and writes a non-$\perp$ value to $S_a[b]$. This contradicts our assumption that a waits for b indefinitely.

Part (b): A team becomes special only if its head, a, executes line **6**. This implies that the condi-

tions of line **5** hold for a and so it is a root without children in G. From Lemma 4 (d), there is at most one such node in G.

Part (c): Follows directly from Lemma 4 (d). □

Proof of Lemma 5 (d)-(e): Recall that if a team head a loses during a call to function `MergeTeam`, or if its team becomes a playoff contender (i.e. a is the head of $\mathcal{T}^*$), then $\mathcal{T}'_a = \mathcal{T}_a$. Parts (d) and (e) of Lemma 5 follow immediately from that fact and the following claim.

1. A process is in P' if and only if it is a root of a tree of G and has at least one child.
2. If $a \in P'$ holds then a's new team is the union of the sets $\{b\} \cup \mathcal{T}_b$ for all nodes b in the tree rooted at a.

From the definition of P', process a is in P' if and only if it returns from `MergeTeam` on line **14**. This happens if and only if $s_a = 1 \wedge p_a = \perp \wedge \mathcal{L}_a \neq \emptyset$ holds. It follows that a is a root with at least a single child.

We now prove the second part of the claim. Let u be a node in a tree T with root a. Since $s_u = 1$ and u's call of `MergeTeam` terminates (from part (a) of Lemma 5), u executes line **13**. We use a simple structural induction to show that at the time such a process u executes this line, the set $\mathcal{T} \cup \{u\}$ contains exactly the nodes in $\{b\} \cup \mathcal{T}_b$ for all nodes b in the subtree rooted at u. For $u = a$ this proves the claim because a's execution returns this set $\mathcal{T}$ on line **14**.

If u is a leaf then $\mathcal{L}_u = \emptyset$ and $p_u \neq \perp$ (since $u \in P'$ and so u's team does not become a playoff contender). It is easy to see that in this case the variable $\mathcal{T}$ does not change during the execution of `MergeTeam` and thus retains its original value $\mathcal{T}_u$. If u is an inner node, then all of its children

are added to $\mathcal{T}$ on line **8**. Moreover, for each of its children v line **11** is executed with $r = v$ after $S_u[v]$ becomes non-$\perp$. Since v is not a root, it writes on line **16** its local variable $\mathcal{T}$ to $S_u[v]$. From the induction hypothesis, this is the union of $\{b\} \cup \mathcal{T}_b$ for all nodes b on the path from v to a leaf, not including $\{v\}$. It follows easily that u's local variable $\mathcal{T}$ has the claimed value when u executes line **13**. □

3.4 Correctness of the Leader Election Algorithm

Based on Lemma 5, the following theorem establishes the correctness of the leader election algorithm. The proof, presented in detail below, relies on the observation that every team in phase or level k, for $k \geq 1$, has at least k team members (see Lemma 10 below).

Theorem 6. *Consider an arbitrary execution where some subset P of processes executes* `LeaderElect`*. Then each process in P performs a constant number of RMRs, exactly one process in P returns* `win` *and all other processes in P return* `lose`*.*

In order to prove Theorem 6, we need to prove a few lemmas. Let P be a set of processes, each of them calling `LeaderElect` exactly once. As before, we partition the processes into *teams*, each of which consists of a team head and idle team members. Let a be a process in P. The variable T_a, set by the call to `MergeTeam` on line **5** of `LeaderElect` made by a, specifies whether a is a team head or not. Process a is a team head if $T_a \neq \perp$ and is an idle team member otherwise. A team is a set of processes $\{a\} \cup T_a$ for some team head a. There is one exception to this rule, however, which is needed to ensure that teams are disjoint at all times: as soon as process a finishes its write in line **25**, its team consists of itself only (although T_a might not be empty). A process that finishes its call of `LeaderElect` is the head of a team of size 1, containing itself only.

Each team is in one of three states, as determined by the status S_a of its team head a. It is in the *losing state* if $S_a = \texttt{lose}$, it is in the *playoffs state* if $S_a = \texttt{playoff}$, and otherwise it is in the *hopeful state*. These states correspond to the three team types – losing, playoff contender, and hopeful – informally defined in Section 2. We say that the team is *in phase* k if it is in the hopeful state and the Z variable of its team head equals k. A team member is in phase k, if its team is in phase k. We say that the team is *in level* k, if it is in the playoffs state and the Z variable of its team head equals k. A team member is in level k, if its team is in level k.

We first prove some claims about the semantic correctness of the team building process.

Lemma 7. *In any execution where some subset P of processes executes* `LeaderElect`*:*

(a) The phase number of a team member never decreases.

(b) The teams partition P.

Part (a) of the lemma ensures that we can apply Lemma 5 for all `MergeTeam` calls: if process a calls `MergeTeam` then a is a team head in phase Z_a. Since a increments Z_a before it calls $\texttt{MergeTeam}_{Z_a}$, part (a) of Lemma 7 ensures that this is the first (and last) time a calls $\texttt{MergeTeam}_{Z_a}$. Part (b) shows that every process a is either a team head or an idle team member of exactly one team, i.e. there is exactly one team head b such that $a \in T_b$. This yields the following corollary.

Corollary 8. *In any execution where some subset P of processes executes* `LeaderElect`*:*

1. *Every process calls an instance of* `MergeTeam` *at most once.*

2. *If a is an idle team member, then there is a team head b such that $a \in T_b$.*

Proof of Lemma 7: We prove the lemma by induction on the total number of operations performed by processes. Clearly, every process calls `MergeTeam` at least once on line **5** (with $Z = 1$) and the claim is true before any process has called `MergeTeam`. To obtain a contradiction, consider the first operation, performed by some process, that falsifies the claim. We call this the *bad operation*.

Assume first that the bad operation decreases the phase number of some team member. Let a be the head of that team. It follows from the induction hypothesis (part (b) of the lemma) that a is not an idle team member of any team. Hence, Z_a can only be decreased by decrementing it on line **15**. However, this is possible only if a's team is in the playoffs state.

Otherwise, assume that the bad operation yields a violation of part (b) of the lemma. Teams can change only during a `MergeTeam` call or on line **25**. Lemma 5 (e) ensures that a call of `MergeTeam` only leads to a new partition of P into teams. Assume, then, that the bad operation is the execution of line **25** by process a with $t = b$. Let $\mathcal{T}_a$ and $\tilde{\mathcal{T}}_a$ respectively denote the set of idle members in a's team just before and after line **25** executes. Then after the execution of line **25**, b is a new team head and $\mathcal{T}_b = \mathcal{T}_a - \{b\}$. Also, by our definition of teams, $\tilde{\mathcal{T}}_a = \emptyset$. It follows that the team $\{a\} \cup \mathcal{T}_a$ is partitioned into two teams – $\{a\}$ and $\{b\} \cup \mathcal{T}_b$ – and all teams still partition P. □

Fact 9. *If a is an idle team member, then it is executing line* **5***, line* **6** *or line* **7** *of* `LeaderElect`*.*

Proof: Variable T_a can be set to $\perp$ only by a call of a to `MergeTeam` on line **5**. As long as a remains an idle team member, it waits on line **7**. □

We need some notation for starting the following lemma. We denote by $work_c$ the value of process c's local variable $work$. For a set of processes S, we let $\Phi(S) = \sum_{c \in S}(3 - work_c)$.

Lemma 10. *In any execution where some subset P of processes executes* `LeaderElect`*:*

1. *Any team in phase k or level k, for $k \geq 1$, has at least k team members, unless it is a team formed when a process executes line* **25** *of* `LeaderElect`.

2. *Let a be a team head in phase k. If a's execution is before line* **3** *of* `LeaderElect`*, or after line* **5** *and before line* **25**, *then the following inequality holds.*

$$\Phi(\{a\} \cup T_a) \;\geq\; 2^k + 2 \tag{1}$$

Proof: We first prove the second part of the lemma. The proof proceeds by induction on k. If $k = 0$, then no process has called `MergeTeam` yet and all teams are singletons. Since in this case $work_a = 0$ for every team head a, $\Phi(\{a\} \cup T_a) = 3$ holds, as claimed.

Now let $k \geq 1$. We prove that inequality (1) holds immediately after team head a finishes executing line **5**, where it calls $\texttt{MergeTeam}_k$. Let $\mathcal{T}_a$ denote a's set of idle team members just before it calls $\texttt{MergeTeam}_k$. Assume a finishes executing line **5**, for $Z_a = k > 0$, as a team head, and let T_a be its set of idle team members immediately after the call of $\texttt{MergeTeam}_k$ returns. Also, let b be another process calling $\texttt{MergeTeam}_k$, such that $\{b\} \cup \mathcal{T}_b \cup \mathcal{T}_a \subseteq T_a$ (such a process b exists according to Lemma 5 (d)). From the induction hypothesis, $\Phi(\{a\} \cup \mathcal{T}_a), \Phi(\{b\} \cup \mathcal{T}_b) \geq 2^{k-1} + 2$ hold just before a and b call $\texttt{MergeTeam}_k$. Since all processes in $\mathcal{T}_a \cup \mathcal{T}_b$ are idle, their *work* variables do not increase (see Fact 9). Variables $work_a$ and $work_b$ are incremented by 1 when a and b respectively execute line **3**. It follows that $\Phi(\{a\} \cup T_a) \geq \Phi(\{a\} \cup \mathcal{T}_a) + \Phi(\{b\} \cup \mathcal{T}_b) - 2 \geq 2^k + 2$ holds.

If, after calling $\texttt{MergeTeam}_k$, process a proceeds to line **25** and leaves its team, then $\Phi(T_a) \geq 2^k + 2$ no longer holds since $\Phi(T_a) = (3 - work_a) = 0$. However, process a never returns to a line of `LeaderElect` before line **25**, and so no longer satisfies the hypothesis of part 2 of the lemma.

This completes the proof of the lemma for all teams in the hopeful state.

We next prove the first part of the lemma. For teams whose heads are in the hopeful state, the lemma follows from inequality (1), since each member of the team headed by a contributes at most 2 to $\Phi(\{a\} \cup T_a)$ (every such member clearly called `MergeTeam` at least once). Specifically, if a is in phase $Z_a \geq 0$ then it has at least $(2^{Z_a} + 2)/2 = 2^{Z_a - 1} + 1 \geq (Z_a - 1) + 1 = Z_a$ members.

Next, we prove the first part of the lemma for teams in the playoffs state. We prove the claim by induction on the level numbers (considered in decreasing order). A team *enters the playoffs state* if its head, say a, calls `MergeTeam` on line **5** and the return value is $S_a =$ `playoff`. The induction basis is the level in which a enters the playoffs state. If a team led by a enters the playoffs state, then a leaves the while loop and executes line **11**. Since a was hopeful before it entered the playoffs state, we get from the previous part of the proof that a's team has at least Z_a members when it enters that state.

Assume the claim holds for level k. If a's call of `TwoProcessLE` on line **11** in level k returns `lose`, then S_a is set to `lose` on line **13** and the team is not in the playoffs state anymore. Otherwise, Z_a is decremented and, if $Z_a \geq 1$ holds after that, a elects a new team head, b, from its team members (on line **25**) and leaves the team. Process b exists, since by the first part of the proof a's team has size at least 2 before it enters the playoffs. From induction hypothesis, the size of a's team before performing line **25** is at least k. Since a decrements Z_a and then sets $Z_b := Z_a$ on line **23**, the new team led by b has at least Z_b team members also after a leaves the team. □

Corollary 11. *If a team head a is in the hopeful state in phase $k \geq 1$ while it executes a line beyond line* **5***, then one of a's team members will be in the hopeful state in phase $k + 1$.*

Proof: If a itself will not be in the hopeful state in phase $k + 1$, then it cannot execute line **4**

again. Hence it eventually leaves its team and elects a new team head $b \in T_a$ by executing line **25** (note that, from Lemma 10, T_a contains at least one idle team member). According to Fact 9, b executes a line between line **5** and line **7** at the time it is elected as a new team head. Clearly, b is also in the hopeful state at this time. After electing b as the new head, its level is set to k in line **23**. From the second part of Lemma 10, at that time $\Phi(\{b\} \cup T_b) \geq 2^k + 1 > 0$ holds. It follows that either b or a process in T_b will be in a hopeful state in phase $k+1$. □

We say that a team *plays a playoffs game in level* i, if its team head executes line **11** with $Z = i$. A team *loses* that playoffs game if the function call on that line returns `lose`, otherwise it *wins* the playoffs game.

Lemma 12. *For any execution where a subset P of processes executes* `LeaderElect`*, there is an integer m such that, for all $1 \leq k \leq m$, there are at least one and at most two teams playing a playoffs game in level k.*

The following remark is useful for the proof.

Remark 13. *If a team T plays a playoffs game in level i, then one of the following holds.*

- *T's head has called* $\texttt{MergeTeam}_i$ *on line* **5** *and received response* `playoff`*. From Lemma 5 (b), this happens to at most a single team head in phase i, or*
- *T's head won a playoff game in level $i+1$.*

Proof of Lemma 12: Since each hopeful team in phase i has at least i members (Lemma 10, part 1), no hopeful team ever reaches phase $i > n$. Let $1 \leq m \leq n$ be the maximum integer such that a team head a calls $\texttt{MergeTeam}_m$. Lemma 5 (c) guarantees that, among all team heads calling

`MergeTeam`$_m$, there is at least one team head a which remains a team head after the call and does not enter the losing state. From Corollary 11, a's call of `MergeTeam`$_m$ returns `playoff` because otherwise at least one member of a's team would enter phase $m+1$ in the hopeful state and thus call `MergeTeam`$_{m+1}$. By Remark 13, at most a single process, say a, plays a playoffs game in level m after a call of `MergeTeam`$_m$ returns response `playoff`. It follows that a is the only process playing a playoffs game in level m. This proves the lemma for $k = m$, which is the base case of the induction that follows.

Assume that at least one and at most two teams play a playoffs game in level $k+1$, $1 \leq k \leq m-1$. From the semantics of two-process leader election, exactly one of these teams wins the playoffs game. The winning team has at least one idle team member (Lemma 10) and thus its head elects a new head on line **25**, just before it leaves the team and loses. Clearly from the algorithm, the new team head then enters the playoffs game in level k. Hence, at least one team enters the playoffs game in level k. From Remark 13, the only other possibility for playing a playoffs game in level k (besides winning the playoffs game in level $k+1$) is by joining the playoffs as a result of a `MergeTeam`$_k$ call on line **5**. Since this can happen for at most one team in each level, at most two teams play the playoffs game in level k. □

We can now prove the high-level correctness properties of `LeaderElect`, stated earlier as Theorem 6.

Proof of Theorem 6: The while loop starting on line **2** is executed at most three times. Since all subfunction calls require only a constant number of RMRs, and as `LeaderElect` itself consists only of a constant number of steps, the total number of RMRs is constant.

We now prove that every process in P finishes its execution of the `LeaderElect`. Assume there

is a set of processes, $P' \subseteq P$, for each of which the call of `LeaderElect` does not terminate. It follows from Lemma 5 and the termination property of `TwoProcessLE` that all the processes of P' wait on line **7** even after all the processes of $P - P'$ have already finished executing the algorithm. At this time, the processes of $P - P'$ are heads of singleton teams. On the other hand, each process $a \in P'$ is an idle team member because $T_a = \perp$. Since a team is by definition a set $\{b\} \cup T_b$, where b is a team head, no team contains the processes of P'. This is a contradiction to Lemma 7 (b).

It remains to prove that exactly one process in P returns `win`. We call a team $W = T_a \cup \{a\}$ headed by a an *overall winning team of size s*, if a executes line **18** and if $Z_a = 0 \wedge S_a = \texttt{playoff} \wedge |W| = s$ holds when it does. If a is the head of an overall winning team W_s of size $s \geq 2$, then a will elect on line **25** one of its idle team members $a' \in T_a$ as the new head of the team $T_a = W_s - \{a\}$ and will then lose. Clearly, $Z_{a'} = 0$ and $S_{a'} = \texttt{playoff}$ when a' becomes the team head of T_a and, following Fact 9, it will proceed to line **18** afterwards. Hence, $W_{s-1} := T'_a$ is an overall winning team of size $s - 1$.

It is easily seen that there are only two ways in which a team W_s can become an overall winning team. Either there is an overall winning team $W_s \cup \{b\}$, headed by b, and process b leaves it as described above; or the team head c of team W_s executes line **15**, thus decrementing Z_c from 1 to 0. Since the latter requires that c wins the playoffs game in level 1, this is the case for exactly one team. Hence, there is a unique largest overall winning team W_s. Let s be the size of W_s. It follows that for each $1 \leq i \leq s$, there is a unique overall winning team W_i, such that $W_1 \subseteq W_2 \subseteq \cdots \subseteq W_s$. By definition, if a^* is the head of the unique overall winning team W_1 of size 1, then $Z_{a^*} = 0 \wedge S_{a^*} = \texttt{playoff} \wedge T_{a^*} = \emptyset$ holds when a^* executes line **18**. Since these conditions are necessary and sufficient for a process a^* to return `win`, a^* is the unique winner of the leader

election algorithm. □

3.5 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.

3.5.1 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 entire 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 (from the first part of Lemma 10, $m \geq 2$). 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 the correctness of

Theorem 6 is maintained. Thus, the correctness of the algorithm and its constant RMR complexity are maintained.

As we only add some of the processes of 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 chain reaction along the linked lists of these processes, in which each of them notifies the next process that the team must fail, and then fails itself.

3.5.2 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 Cache-Coherent Model

The algorithm presented in Section 3 has RMR complexity of $\Theta(n)$ in the CC model due to the loops of lines **2**–**8** of `LinkReceive`, lines **20**–**25** of `Forest`, and lines **9**–**12** of `MergeTeam`. Constant RMR complexity in the cache-coherent (CC) model 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 the modified `LinkReceive` function by `LinkReceive-CC`. We also divide `LinkRequest` into two functions – `LinkRequestA-CC` and `LinkRequestB-CC` – which must

be called in that order. Specifically, 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`

```
1 p := Find()
2 if p ≠ ⊥ then LinkRequestA-CC(p)
3 L := LinkReceive-CC()
4 if p ≠ ⊥ then
5 |   link := LinkRequestB-CC(p)
6 else
7 |   special := 1
8 |   link := 0
9 end
  /* resume from line 10 of Forest                                    */
```

The CC handshaking protocol for establishing links uses shared arrays $A[]$ and $B[]$, indexed by process ID, all of whose elements initialized to $\perp$. The algorithms of the handshaking functions are shown below.

To perform a call to `LinkRequestA-CC(p)`, a process q simply writes its `PID` to $A[p]$ on line **1**. To perform `LinkReceive-CC`, p first saves $A[p]$ into the temporary variable a on line **1**. If $a = \perp$, then p writes its `PID` to $B[p]$ on line **3** and returns $\emptyset$. Otherwise, a is the identifier of some $q \neq p$ that invoked `LinkRequestA-CC(p)`, and so p acknowledges having seen a by writing a to $B[p]$ on line **6**, and returns $\{a\}$. Finally, to perform `LinkRequestB-CC(p)`, q waits on line **1** until $B[p] \neq \perp$,

Function `LinkRequestA-CC(`p`)`

Var.: $A[1..N]$ – array of $\perp$ or process ID, initially $\perp$

1 **write** $A[p]$:= `PID`

Function `LinkRequestB-CC(`p`)`

Output: a value in $\{0, 1\}$ indicating link establishment failure or success, respectively

Var.: $B[1..N]$ – array of $\perp$ or process ID, initially $\perp$

1 **wait until** $B[p] \neq \perp$
2 **if read**$(B[p])$ = `PID` **then**
3 | **return** 1
4 **else**
5 | **return** 0
6 **end**

Function `LinkReceive-CC`

Output: set of processes to which link was established

Var.: $A[1..N], B[1..N]$ – shared with `LinkRequestA-CC` and `LinkRequestB-CC`

1 a := **read**$(A[$`PID`$])$
2 **if** $a = \perp$ **then**
3 | **write** $B[$`PID`$]$:= `PID`
4 | **return** $\emptyset$
5 **else**
6 | **write** $B[$`PID`$]$:= a
7 | **return** $\{a\}$
8 **end**

and returns 1 if and only if it reads its `PID` from $B[p]$. The properties of the new CC handshaking functions are analogous to those stated in Lemma 3 in the DSM model, and are stated in the following lemma.

Lemma 14. *The following claims hold.*

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

(b) Let L *be the set returned by* p*'s call to* `LinkReceive-CC`*. Suppose that every process that calls* `LinkRequestA-CC(`p`)` *subsequently also calls* `LinkRequestB-CC(`p`)`*, but* p *never calls* `LinkRequestA-CC(`p`)`*. Then* $L = \{q\}$ *if and only if a link from* p *to* q *is eventually established.*

(c) *Suppose that q's call to* `LinkRequestA-CC(`p`)` *terminates before p invokes* `LinkReceive-CC`*, and that every process that calls* `LinkRequestA-CC(`p`)` *subsequently also calls* `LinkRequestB-CC(`p`)`*. Then a link from p to* some *process (not necessarily q) is eventually established.*

Proof:

Part (a): Termination of `LinkRequestA-CC(`p`)` is trivial, so consider `LinkRequestB-CC(`p`)`. Suppose that q executes `LinkRequestB-CC(`p`)` and p executes `LinkReceive-CC`. Then p eventually executes line **3** or line **6** of `LinkReceive-CC`, in either case causing $B[p] \neq \perp$ to hold. As $B[p]$ is not written again, q eventually completes line **1** of `LinkRequestB-CC`.

Part (b): Suppose that p's call to `LinkReceive-CC` returns the set $\{q\}$. Then p assigns $B[p] := q$ on line **6** during this call and $B[p]$ is never written again. It follows that q called `LinkRequestA-CC(`p`)`, and so eventually also calls `LinkRequestB-CC(`p`)` by assumption. Since $B[p]$ is initially $\perp$ and is only written by p, once, it follows that q reads q from $B[p]$ on line **2** of `LinkRequestB-CC(`p`)` and returns 1. Similarly, if q's call to `LinkRequestB-CC(`p`)` returns 1, then it must be that p assigned $B[p] := q$ in `LinkReceive-CC`. Since p never calls `LinkRequestA-CC(`p`)`, it follows that $q \neq p$, and so `LinkReceive-CC` returns $\{q\}$ after writing $B[p]$ on line 6.

Part (c): Suppose that q's call to `LinkRequestA-CC(`p`)` terminates before p starts executing `LinkReceive-CC`. Since q completed line **1** of `LinkRequestA-CC` and since no process assigns $\perp$ to $A[p]$, it follows that $A[p] \neq \perp$ when p reads $A[p]$ on line **1** of `LinkReceive-CC`. Thus, p returns a nonempty set, say $\{z\}$ for some $z \neq \perp$. By part (b), it follows that a link from p to z is eventually established. □

Based on Lemma 14, it is easily shown that the properties of the `Forest` function, established by Lemma 4, hold also for `Forest-CC`. The proof is essentially the same. The only required modifications are that (1) references to the invocation and return value of `LinkRequest` are replaced with references to the invocation of `LinkRequestA-CC` and return value of `LinkRequestB-CC`, and (2) the proof uses Lemma 14 instead of Lemma 3. Thus, the following result follows.

Theorem 15. *Let P be a non-empty set of processes executing the algorithm* `LeaderElect` *in the CC model, where* `Forest` *is replaced by* `Forest-CC`*. Then each process in P performs a constant number of RMRs, exactly one process returns* `win` *and all other processes return* `lose`*.*

5 Space Complexity

In this section we analyze the space complexity of the leader election algorithms we presented. The space complexity depends crucially on the maximum number of merging phases, which we denote by M. Clearly from the algorithm, the maximum number of playoff levels is also M. The following lemma provides an upper bound on M.

Lemma 16. *Suppose that k processes invoke* `LeaderElect`*. Then $M \leq \lfloor \log_2 k \rfloor$ holds.*

Proof: We prove the lemma by induction. Let t_i be the number of hopeful teams after the i'th merging phase. From initialization, $t_0 = k$ holds. From Lemma 5 part (d), we get that $t_i \leq \lfloor t_{i-1}/2 \rfloor$, hence $t_i \leq \lfloor k/2^i \rfloor$ holds. Since $t_i \geq 1$ if and only if merging phase i exists, we get that $M \leq \lfloor \log_2 k \rfloor$, as required. □

The following lemma shows that the bound of Lemma 16 is tight.

Lemma 17. *There is an execution of* `LeaderElect` *with* $\Omega(\log n)$ *phases.*

Proof: For simplicity and without loss of generality, we assume in this proof that n is an integral power of 2. We construct an execution of the algorithm in which all n processes participate, and in which there are exactly $\log_2 n$ merging phases. The execution is constructed iteratively. Iteration i, for $1 \leq i \leq \log_2 n - 1$, corresponds to phase i.

Let P_i denote the subset of processes that are the heads of hopeful teams at the beginning of the i'th phase. Our construction meets the invariant that $|P_i| = n/2^{i-1}$, for $i \in \{1, \cdots, \log_2 n\}$. This invariant clearly holds at the beginning of phase 1.

For $1 \leq j < \log_2 n$, assume we have constructed the first $j - 1$ iterations; we now describe the construction of the j'th iteration. From assumptions, $|P_j| = 2k$, for some integer k. Let $q_1, \cdots, q_{2k}$ denote the processes of P_j when ordered in a *decreasing* order of their PID. Also, let $Q_1 = \{q_1, \cdots, q_k\}$ and $Q_2 = \{q_{k+1}, \cdots, q_{2k}\}$. In iteration j, we schedule the processes of P_j so that all of them receive a *hopeful* response from `MergeTeam`. However, the processes of Q_1 remain team-heads right after the call returns (i.e., $\mathcal{L} \neq \emptyset$ holds for them), whereas the processes of Q_2 become idle team members after the call returns. The key idea of the construction is to schedule the processes in the j'th phase so that, for $l = 1, \cdots, k$, process q_l 'sees' only process q_{k+l} and

vice-versa. Thus, the graph corresponding to phase j (which is the input to the `Forest` calls of phase j) will consist of k directed cycles of size 2. The symmetry of each of these cycles is broken by function `Forest` in favor of the process with the bigger PID. We now describe the construction in detail.

Initially, we let each of the processes of Q_1 perform the function `MergeTeam` until it is about to perform line **4** of `Find`, and then we halt the process. We now construct the first cycle. We let q_1 execute line **4** of `Find` and write its PID to G. We then let q_{k+1} execute line **1** of `Find` and write its PID to F. We then let q_{k+1} execute line **2** of `Find` and read q_1's PID from G. Finally, we let q_1 execute line **5** of `Find` and read q_{k+1}'s PID from F. This completes the construction of the first cycle. We continue in this manner with the other $k-1$ pairs of processes in P_j, until we construct all of the k cycles.

It is easily seen that, for each $i \in \{1, \cdots, k\}$, process q_i breaks the cycle with q_{k+i} by deleting the link from q_{k+i} to q_i in line **12** of the `Forest` function. It follows from the condition of line **26** of `Forest` that the teams of q_i and q_{k+i} are merged, and that when q_i returns from `Forest` it is the head of this new team. This completes the construction of phase j.

As the number of hopeful teams decreases by exactly half in each phase, we can continue in this manner for exactly $\log_2 n$ phases. □

Theorem 18. *The space complexity of the* `LeaderElect` *algorithm is* $\Theta(n^2 \log n)$.

Proof: The dominant factors in the space complexity of `LeaderElect` are the per-process arrays A[], LINK[], CUT[], and S[] used by the functions `LinkRequest`, `LinkReceive`, `Forest` and `MergeTeam`. These arrays are of size n and a copy of these arrays is required for every phase. The result now follows from Lemmas 16 and 17. □

The space complexity can be reduced to $O(nM)$ in the CC version of the algorithm by taking advantage of the fact that `LinkReceive-CC` returns a set of size at most one. More precisely, the arrays CUT (used by the `Forest-CC` function) and S (used by the `MergeTeam` function) can be replaced with single-word variables, which reduces the space requirement to $O(1)$ per process per phase or level.

6 System Response Time

The *system response time* of a mutual exclusion algorithm is the time interval between subsequent entries to the critical section, where a time unit is the minimal interval in which every active process performs at least one step [8, 11]. Leader election may be regarded as one-time mutual exclusion [14]. The system response time of a leader election algorithm is the time interval between the time when the execution starts and the time when the leader is determined, where a time unit is defined in the same manner. In the following, we analyze the system response times of the leader election algorithms we have presented.

Theorem 19. *The system response time of the* `LeaderElect` *algorithm is* $O(n \log n)$.

Proof: We first analyze the maximum time during which a process is hopeful. Consider some merging phase m. A process completes `Find`, `LinkRequest`, `LinkRequestA-CC` and `LinkReceive-CC` in $O(1)$ time units, since these functions do not involve waiting. `LinkRequestB-CC` also requires $O(1)$ times units, since if q executes line **1**, waiting for p to write B_p, then p has invoked `Find` (and hence `Forest-CC`) in phase m and will complete `LinkReceive-CC` in $O(1)$ time units. `LinkReceive` requires $O(n)$ time units, since if p executes line **6** of `LinkReceive`, waiting for some process q, then

q invoked `LinkRequest(`p`)` in phase m and p's busy-wait loop terminates after $O(1)$ time units. In function `Forest`, if p reaches line **21**, waiting for some process q, then q invoked `LinkRequest(`p`)` in line **3** of `Forest` and reaches line **12** or line **15** within $O(n)$ time units (after completing `LinkReceive`), and so p spends a total of $O(n)$ time units busy-waiting in the loop of lines **20**–**25**. Thus, a call to `Forest` completes in $O(n)$ time units. Similarly, `Forest-CC` completes in $O(1)$ time units since p waits for at most one process to write some entry of $\text{CUT}_p[]$ and that process does so within $O(1)$ time units.

A slightly more complex argument shows that a call to `MergeTeam` completes in $O(n)$ time units. Consider first the DSM model. If p executes line **10** of `MergeTeam`, waiting for some process q, then q invoked `Forest` on line **1** of `MergeTeam` in phase m, and so q reaches the loop of line **9** in $O(n)$ time units. Let T denote the subtree of the forest corresponding to phase m, to which processes p and q belong. From Lemma 4 part (c), there exists a process, say r, participating in phase m and belonging to T, for which $\mathcal{L} = \emptyset$ after it executes line **1** of `MergeTeam`. Then r completes `MergeTeam` in $O(n)$ time units, since it skips the loop of lines **9**-**12**. Let h denote the height of p in T. A straightforward induction on the height of nodes in T shows that p completes `MergeTeam` $O(n + h)$ time units after it reaches line **9**, hence it completes `MergeTeam` in $O(n)$ time units. From Lemma 16, we get that a process is hopeful for $O(n \log n)$ time units. In the CC model, an analogous argument shows that p completes `MergeTeam` in $O(1 + h)$ time units and is hopeful for $O(n \log n)$ time units.

It is easy to see that `TwoProcessLE` – the two process leader election algorithm – requires $O(1)$ time units. Thus, again from Lemma 16, the total time spent by a process performing this algorithm in all levels is $O(\log n)$. It follows that the overall playoff winner team is determined

after $O(n \log n)$ time units. Since the maximum size of any team is n, after at most n additional time units every process terminates and the identity of the leader is determined. □

The following theorem shows that the bound of Theorem 19 is tight, at least for the DSM variant of the algorithm. This is noteworthy, since it establishes that low RMR complexity does not necessarily imply low system response time.

Theorem 20. *The system response time of the DSM variant of the* `LeaderElect` *algorithm is* $\Omega(n \log n)$.

Proof: We construct an execution, very similar to that constructed in Lemma 17, that has the desired system response time. As in Lemma 17, we assume that n is an integral power of 2. In order not to duplicate the description of the execution provided in the proof of Lemma 17, we only describe the difference between the two executions.

The execution we construct here is composed of $\log_2 n + 1$ iterations. As in Lemma 17, each of the first $\log_2 n$ iterations corresponds to a merging phase. In the last iteration, all the active processes run until they complete their operations. Let $1 \leq i < \log_2 n$ be a phase number, let P_i be as defined in Lemma 17, and let $R_i = P - P_i$. In the description of the construction that appears in Lemma 17, we disregarded the processes of R_i. To obtain a lower bound on system response time, however, we need to also schedule the steps of these processes. This is because the system response time increases by 1 only after every active process takes at least a single step. Consequently, in the following description, we assume that every process of R_i takes at least a single step between any two consecutive steps of a process from P_i. The construction of Lemma 17, as well as the construction we provide here, guarantees that no playoff teams are formed as

long as there are hopeful teams. Consequently, the processes of R_i are either idle team members or members of teams that have already lost. It follows that steps taken by these processes do not result in the election of a leader.

Let k, Q_1 and Q_2 be as in Lemma 17. In the beginning of phase i, we construct k directed cycles, each containing one process from Q_1 and one from Q_2, in exactly the same manner this was done in Lemma 17. Next, we let all the processes of P_i run until they are about to begin the loop on lines **2**–**8** of function `LinkReceive`. We then let these processes execute this loop in lock-step until they all finish it. Clearly, this increases the system response time of the execution by $\Theta(n)$. Finally, we let all the processes of P_i run until they all return from `MergeTeam`. This completes the construction of phase i.

Continuing in this manner, we construct the first $\log_2 n$ iterations of the execution and obtain a response time of $\Omega(n \log_2 n)$. Since a playoff team is formed only at the end of phase $\log_2 n$, a leader is not elected during these phases. Finally, in the $(\log_2 n + 1)$'th and last iteration, we let all processes run until they terminate.

□

If the modifications described in Section 3.5 are applied, as well as the optimization that reduces space complexity to $\Theta(n \log n)$, then the response time of the CC version of the algorithm decreases to $\Theta(\log n)$. As argued in the proof of Theorem 19, `Forest-CC` takes only $O(1)$ time units. Consequently the space-optimized `MergeTeam` function also executes in $O(1)$ time units, since the calling process no longer needs to wait on line **10** for its children to report their team sets on line **16**; instead, the caller directly reads at most two processes from its only child's linked list of team members. The chain reaction that causes the remaining team members (of the child)

to lose can be completed at a one-time cost of $O(N)$ time units where N is the maximum team size. Thus, a process waits $O(M + N)$ time units on line **7** of `LeaderElect` while its teammates progress through merging phases and playoff levels, where M is the maximum phase number, and additional $O(N)$ time units subsequently before it is notified of its team's outcome. The total cost is $O(M + N)$ time units, which is $O(\log n)$ since $M, N \in \Theta(\log n)$ in the CC version of the algorithm with $O(\log n)$-bit words. Since some process may be active until phase M, it follows that the response time is $\Theta(\log n)$ in the worst case.

7 Simulation of One-Time Test-And-Set Objects

Linearizability [18] is widely accepted as a correctness condition for concurrent objects. Informally, it states that each operation must appear to take effect instantaneously at some point in the interval of time bounded by its invocation and response.

In this section we describe linearizable simulations of an n-process one-time *test-and-set* object for both the DSM and the CC models. Our simulations use leader election algorithms as black box components. 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`, which atomically writes 1 to the object and returns the previous value. A leader election algorithm is a function that returns a value in $\{\texttt{win}, \texttt{lose}\}$, indicating whether the calling process is a leader (it has won) or not. (The correctness properties of a leader election algorithm were defined in Section 1.)

7.1 Simulation for the DSM model

Suppose that an algorithm $\mathcal{A}$ 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 $\mathcal{A}$ 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 building blocks: an $(n-1)$-process $O(1)$-RMR leader election algorithm `LE`, which can be implemented using `LeaderElect`; a two-process $O(1)$-RMR leader election algorithm $\texttt{2LE}_p$, for process p and one other process (whose ID is not known a priori), which can be implemented as explained in Section 8; and a read-write register R_p, initially $\perp$. The key property of $\texttt{2LE}_p$ is that it is *asymmetric*: it guarantees that p incurs no RMRs, and that the process competing with p incurs only a constant number of RMRs. Similarly, R_p resides in p's local memory segment and can be accessed by p without incurring RMRs. The simulation algorithm is presented below as function `test-and-set-DSM`.

To apply the test-and-set operation on T, a process q first reads R_p on line **1**. If the result is not $\perp$, then q returns one.

Otherwise, q writes its PID to R_p on line **4**. Subsequent steps depend on q's identity. Suppose that $q \neq p$. Then q executes the $(n-1)$-process leader election algorithm `LE` on line **8**. If it is elected, q proceeds to compete against p using $\texttt{2LE}_p$ on line **9**. Only if it is also elected here does q return 0, otherwise it returns 1. If, on the other hand, $q = p$, then it proceeds directly to compete using $\texttt{2LE}_p$ on line **7** against the leader elected using `LE` on line **8** (if any). Finally, q returns 0 if it wins $\texttt{2LE}_p$, and returns 1 otherwise. The following lemma establishes the correctness

Function `test-and-set-DSM`

Output: value in $\{0, 1\}$

Var.: R_p – process ID or $\perp$

Var.: `LE` – $O(1)$-RMR leader election algorithm

Var.: $\texttt{2LE}_p$ – $O(1)$-RMR 2-process leader election algorithm (for process p and one other, 0-RMR for process p)

```
1  if read(R_p) ≠ ⊥ then
   |  /* another process has set object before me                    */
2  |  return 1
3  else
4  |  write R_p := PID
5  end
   /* if object is local to me                                       */
6  if PID = p then
   |  /* I need only win against the winner amongst all other processes */
7  |  if 2LE_p() = win then return 0 else return 1
   |  /* else if I won against all processes other than p           */
8  else if LE() = win then
   |  /* competition between p and the winner amongst all others     */
9  |  if 2LE_p() = win then return 0 else return 1
10 else
11 |  return 1
12 end
```

of `test-and-set-DSM`.

Lemma 21. *The function* `test-and-set-DSM` *simulates a linearizable one-time test-and-set object. In the DSM model, the cost of a* `test-and-set` *operation is* $O(1)$ *RMRs. Specifically, if the* `test-and-set` *operation is applied by the process to which the test-and-set object is local, then that process incurs no RMRs.*

Proof: First consider linearizability. Suppose that at least one process executes `test-and-set-DSM`. Note that by lines **1**–**5**, no process invokes `LE` or $\texttt{2LE}_p$ more than once in the same execution. Thus, only the single process $q \neq p$ that wins `LE` on line **8** (if it exists) competes using $\texttt{2LE}_p$. Also, a process returns 0 if and only if it wins $\texttt{2LE}_p$ on line **7** or line **9**. From the semantics of $\texttt{2LE}_p$, exactly one process wins $\texttt{2LE}_p$ and returns 0, whereas all other processes return 1. Furthermore, from lines **1**–**5** it follows that once a `test-and-set` operation completes, every subsequently invoked operation returns 1. Thus, the single operation that returns 0 can always be placed first in the linearization order.

Finally, termination and the claimed RMR complexity of the simulation follow from Theorem 6 and Lemma 25 (see Section 8). □

7.2 Simulation for the CC model

To simulate a `test-and-set` operation in the CC model, we use the function `test-and-set-CC` shown below. The simulation uses a shared register R, analogous to R_p in `test-and-set-DSM`, as well as an n-process $O(1)$-RMR leader election algorithm `LE`, which can be implemented using the CC variant of `LeaderElect`. (In the CC model, the register R cannot be assigned statically to a

specific process.) The algorithm is somewhat simpler than that of `test-and-set-DSM`, since all processes can spin on the same register and incur only a single RMR.

Function `test-and-set-CC`

```
   Output: value in {0, 1}
   Var.: R – process ID or ⊥
   Var.: LE – O(1)-RMR leader election algorithm
 1 if read(R) ≠ ⊥ then
 2 |   return 1
 3 else
 4 |   write R := PID
 5 end
 6 if LE() = win then
 7 |   return 0
 8 else
 9 |   return 1
10 end
```

Lemma 22. *The function* `test-and-set-CC` *simulates a linearizable one-time test-and-set object in the CC model. The cost of a* `test-and-set` *operation is* $O(1)$ *RMRs.*

Proof: The proof of linearizability is similar to that used in the proof of Lemma 21. Termination and constant RMR complexity follow from Theorem 15 and Lemma 25 (see Section 8).

□

We can now prove the main result of this section.

Theorem 23. *Any algorithm, for either the DSM or the CC model, 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 of the RMR complexity, in a manner that preserves safety properties but not liveness properties (other than termination).

Proof: Follows immediately from Lemmas 21 and 22. □

8 Auxiliary Algorithms

In order to make the paper self-contained, we now describe simple two-process constant-RMR leader election algorithms that can be used to implement line **11** of algorithm `LeaderElect` in Section 3, as well as the two-process asymmetric leader election algorithm $\texttt{2LE}_p$ used in function `test-and-set-DSM` of Section 7.

Consider first the function `TwoProcessLE` invoked in line **11** of algorithm `LeaderElect`, which determines the playoff winner at each level. The following implementation uses function `Forest`.

Function `TwoProcessLE`

Output: A value in $\{\texttt{win}, \texttt{lose}\}$

```
1 (s, p, L) := Forest()
2 if s = 1 ∧ p = ⊥ then
3 |   return win
4 else
5 |   return lose
6 end
```

Assume that at most two processes call the `TwoProcessLE` function. Each process calls `Forest` and obtains a triplet $(s, q, \mathcal{L})$. Then, it checks whether it is a tree root in the graph G induced by the return values of `Forest`, i.e. whether $s = 1$ and $p = \perp$ (refer to Section 3.3.4). If this is the case then the process wins, otherwise it loses. Parts (c)-(d) of Lemma 4 guarantee that G is a tree

with either one or two nodes (due to part (d) of the lemma G may not contain two isolated roots). Hence, exactly one process calling `Forest` becomes a root. This yields the following corollary to Lemma 4.

Corollary 24. *If at least one and at most two processes call* `TwoProcessLE`*, then exactly one of them wins.*

Next, consider the leader election algorithm $\texttt{2LE}_p$ used in the function `test-and-set-DSM` of Section 7. Although $\texttt{2LE}_p$ is only a two-process algorithm, it cannot be implemented using `TwoProcessLE` as it must be local to p. That is, p must not incur any RMRs while executing $\texttt{2LE}_p\texttt{()}$. To that end, we take advantage of the fact that the ID of one of the processes executing $\texttt{2LE}_p$, namely p, is known a priori. Let q denote an arbitrary process other than p. To elect a leader, p and q respectively invoke the functions `TwoProcessLE-p` and `TwoProcessLE-q` shown below. The static shared variables are as follows: A – Boolean, initially 0; and B – integer 0..3, initially 0.

Lemma 25. *Suppose that at least one of the following events occurs:*

1. *Process p calls* `TwoProcessLE-p`*.*
2. *Process q calls* `TwoProcessLE-q`*.*

Then every function call terminates, exactly one function call returns `win`*, and, if both function calls are made, the other returns* `lose`*. Moreover,* `TwoProcessLE-p` *incurs zero RMRs and* `TwoProcessLE-q` *incurs $O(1)$ RMRs in the DSM model.*

Proof: First, consider termination. The only busy-wait loop occurs on line **5** of `TwoProcessLE-p`. Suppose that p reaches this line. Then p read $B \neq 0$ on line **2**, so q must have completed line **1**

Function `TwoProcessLE-p`

Output: A value in $\{\texttt{win}, \texttt{lose}\}$
Var.: A – value in $\{0, 1\}$, initially 0
Var.: B – value in [0..3], initially 0

```
1  A := 1
2  if B = 0 then
3  |  return win
4  else
5  |  wait until B ≠ 1
6  |  if B = 2 then
7  |  |  return win
8  |  else
9  |  |  return lose
10 |  end
11 end
```

Function `TwoProcessLE-q`

Output: A value in $\{\texttt{win}, \texttt{lose}\}$
Var.: A, B – shared with `TwoProcessLE-p`

```
1  write B := 1
2  if read(A) = 1 then
3  |  write B := 2
4  |  return lose
5  else
6  |  write B := 3
7  |  return win
8  end
```

of `TwoProcessLE-q`. In this case, q eventually executes line **3** or line **6**, allowing p to complete `TwoProcessLE-p`. The claimed RMR complexity follows from the structure of the functions and the fact that all shared variables are local to p in the DSM model. □

9 Discussion

We presented leader election algorithms using reads and writes only, for both the CC and the DSM models, for which the worst-case number of RMRs is constant. These are the first leader election algorithms using only reads and writes that have a sub-logarithmic worst-case RMR complexity. This result separates the mutual exclusion and leader election problems in terms of worst-case RMR complexity.

We also showed that one-time test-and-set can be simulated from reads and writes with only a

constant number of RMRs, in both the CC and the DSM models. Moreover, we showed that any algorithm, using read, write, and one-time test and set, can be simulated by an algorithm using read and write only with only a constant blowup in RMR complexity. A follow-up paper by Golab et al. [21] proves a corresponding result for conditional primitives such as compare-and swap and load-linked/store-conditional.

It is noteworthy that our algorithm for the DSM model simultaneously achieves optimal RMR complexity and high system response time. This establishes that low RMR complexity does not necessarily imply low system response time. The question is open whether this represents a tradeoff between RMR complexity and system response time that is inherent to the leader election problem, or is merely a feature of our particular algorithm.

Styer and Peterson consider the space-complexity of symmetric mutual exclusion and related problems [23]. Among other results, they prove a lower bound of $\lceil \log n \rceil + 1$ on the space-complexity of leader election in the model we consider. As summarized by Table 1, all the algorithms we present have super-linear space complexity. Styer and Peterson also present a leader election algorithm with $\Theta(\log n)$ space-complexity, thus establishing that their bound is tight. Since a linear-space lower bound is known for mutual exclusion, their results separate the mutual exclusion and leader election problems in terms of space complexity.

The RMR complexity of the logarithmic-space leader election algorithm of [23] is unbounded for the DSM model and at least linear for the CC mode. Finding the tight space-complexity bound on constant-RMR leader election is another interesting open question.

Acknowledgements

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 for enlightening discussions about the leader election and mutual exclusion problems, as well as Hagit Attiya and the anonymous referees for their insightful comments. Finally, we are grateful to Robert Danek for his feedback on an earlier draft of the paper.

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 V. Bortnikov. Adaptive and Efficient Mutual Exclusion. *Distributed Computing.*, 15(3):177–189, 2002.

[9] 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.

[10] H. Attiya, and D. Hendler, and W. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proc. STOC 2008*, pp. 217–226, May 2008.

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

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

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

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

[15] R. Fan, and N. Lynch. An Omega $(n \log n)$ lower bound on the cost of mutual exclusion. In *Proc. PODC 2006*, pp. 275–284, July 2006.

[16] W. Golab, D. Hendler, and P. Woelfel. An O(1) RMRs leader election algorithm. In *Proc. ACM PODC 2006*, July 2006.

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

[18] 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.

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

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

[21] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. PODC 2007, pp. 3–12.

[22] 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.

[23] E. Styer and G.L. Peterson Tight Bounds for Shared Memory Symmetric Mutual Exclusion Problems. In *Proc. PODC 1989*, pp. 177–191, Aug 1989.