THE REMOTE MEMORY REFERENCE COMPLEXITY OF THE
DISK PAXOS ALGORITHM

Netanel Raviv, 066411794. Raziel Hess-Green, 303705727.

July 4, 2011

1 Introduction

The consensus problem is a fundamental problem in the design of distributed algorithms. The problem is as follows - Given a collection of processes, each process can propose a value. If the algorithm terminates, there will be a unique value $v$ such that:

- $v$ had been proposed by at least one of the processes.
- Each process is able to learn that $v$ is the chosen value.

As proven by [5], this problem is insolvable in an asynchronous system, even in the presence of a single failure. In order to achieve some basic consensus in such a system several solutions were proposed. One of the most useful algorithms is the Paxos algorithm, first presented by [1]. In this algorithm, consistency is promised, but the system is not guaranteed to stop. Yet, under some stability assumptions a consensus is guaranteed. In the latter case, in order to guarantee success the Paxos algorithm may use a leader election algorithm.

In modern networks the use of commodity disks as independent units has become more and more popular. Yet, some thought needs to be done in order to adapt existing algorithms to this model. This so called Disk Model was introduced by [3], where an adaptation of the classic Paxos algorithm to this model was also made.

In the Disk Model, it seems rather natural that one of the complexity measures will be counting how many remote references were made by processes to the independent disks. Such a measure is known as RMR complexity (RMRC). In this paper, we will analyze the RMRC of the disk paxos, as presented in [3]. Moreover, the overall RMRC of [3] is strongly dependent on the way in which we choose to elect the leader, if in fact we choose to do so. An efficient algorithm (regarding RMRC) for that purpose was introduces by [4] in a different model. Our challenge will be to adapt this algorithm to the Disk Model in order to use it in the Disk Paxos algorithm.

Our results considering the RMRC analysis of Disk Paxos relays on the stability of the system, which allows also to use the leader election algorithm. In the natural Disk model our results were mediocre. For that reason we tried to extend our perception of the algorithm through the use of other known models. A more successful result was achieved in the CC model. In addition, The adaptation of the algorithm in [4] to the disk model required additional assumptions on the system besides stability, such as restricted communication between processes. This may be useful for systems in which process-to-disk communication is much cheaper than process-to-process communication.

This article is organized as follows - Section 2 consists of article summaries. Section 3 will explain how do derive the Disk Paxos algorithm from the classic paxos algorithm. In section 4 we will analyze the RMRC of the Disk Paxos algorithm in various models. Finally, Section 5 will explain how the algorithm from [4] can be implemented in the Disk model.

2 Article Summary

Paxos Made Simple

This paper is an abbreviation and simplification of previously presented results from [1] by the same author (Leslie Lamport).

In general, The purpose of the Paxos algorithm is to implement a fault tolerant solution for the asynchronous consensus problem in distributed systems. Clearly, since the consensus problem is insolvable in asynchronous
systems (Or more precisely, there is no fault tolerant solution for this problem), the algorithm does not guarantee success, i.e the algorithm does not necessarily halts at all possible executions.

The Paxos algorithm, as presented here, divides the participant processes into 3 different sets (A process can be a member of more than one set) -

- **proposers** - processes that propose a value to be agreed on.
- **acceptors** - processes that do not propose a value themselves but cooperate with the proposers to come to an agreement.
- **learners** - passive processes that can only learn what is the value that was agreed on.

The algorithm consists of two phases - defined for proposers. Each proposer can be at any given time in any of the phases regardless of other proposers.

**Phase 1**: The proposer sends a prepare request, which consists of a ballot number and a label (namely, a title “prepare”) to a majority of the acceptors. Any acceptor receiving such a request can either respond with a commit not to accept any requests with number less than the current number together with a value that he already had committed on (if such a value exists), or ignore it. A proposer that receives answers from most of the acceptors can move to the second phase.

**Phase 2**: The proposer chooses a value \( v \) that is either his value to propose (in case that the acceptors at the previous phase did not already commit on any value) or a value that was received from an acceptor at phase 1. Afterwards, that proposer will send an accept request to a majority of the acceptors. An accept request consists of a label, a ballot number (The same number as in phase 1) and the value \( v \). An acceptor receiving an accept request numbered \( n \), and did not commit not to accept any requests numbered less than \( m > n \), can accept it.

There are several possibilities for the role of the learner. In general, a learner that wants to find out if a value had been chosen must find out that most of the acceptors accepted it. One possible option is to have each acceptor inform each learner about a chosen value. Another option, which will be more of interest for us on this paper, is to have a distinguished learner, or a set of distinguished learners, that will be informed about a chosen value from the acceptors and will inform the other learners. For that reason, The algorithm for leader election (presented in [4]) will be of special interest for us here. Such an algorithm will allow us not only to choose a distinguished learner/s, but also to choose a distinguished proposer to prevent a situation of competition between proposers.

**Disk Paxos**

The Disk Paxos algorithm presented in [3] is a variant of the classic Paxos for asynchronous fault tolerant consensus (as in the classic version, termination is promised only in consistent mode, i.e. when it is possible to choose a leader), accommodated to fit a network of processes and disks.

The model (inspired by the growing use of commodity disks in modern networks), is of \( n \) Processes and \( D \) Disks where the disks are “passive” in the sense that they only store data that the processes can read and write “actively”.

The algorithm consists of 2 phases, the first allows choosing a value to agree upon (after the phase is complete), and the second makes it a consensus. The desired situation for the algorithm is that a leader is elected, and he alone performs the algorithm, in this case termination is promised. If a unique leader was not elected, consistency is still promised, but not termination.

The main Idea is similar to the classic version, and uses ballots to propose and decide the output. Each process \( p \) has a Private block of memory, denoted \( d\text{block}[p] \), containing the components:

- \( mbal \) - the current number of the ballot.
- \( bal \) - the highest ballot number encountered in phase 1.
- \( val \) - the value being committed on.

The disks are referred to as blocks denoted \( d\text{block}[d][p] \), for any disk each process \( p \) writes only to block \( p \), but reads any part of the disk. Both phases are done in similar fashion, while the value to be proposed is chosen only after a successfully finished phase 1.

A phase is done as follows: a process chooses a value for \( d\text{block}[p].mbal \) and saves some special non - input in \( d\text{block}[p].bal \) and \( d\text{block}[p].val \) in phase 1, or chooses a value for \( d\text{block}[p].val \) and \( d\text{block}[p].bal \) as we will describe later in phase 2. Then \( p \) writes it’s \( d\text{block}[p] \) to \( d\text{block}[d][p] \) for a majority of the disks. After writing, \( p \) tries to read from a majority of the disks \( d\text{block}[d][q] \) for all \( q \neq p \), aborting if he finds any \( mbal \) with a value higher than it’s own. If this reading is successful, the phase is complete. If not - the process returns to the beginning of phase 1 (if it so pleases).

After completing phase 1, \( p \) must choose a value, for that he chooses the value of the highest numbered ballot that had a proposed value (i.e. \( d\text{block}[p].val \neq \text{non - input} \)), if there is no such a ballot, \( p \) can choose any value he
wants. For bal he simply uses mbal’s value.
After phase2 p has decided on his value successfully.

Another issue is recovering from faults, which is done by reading disk[d][p] for most of the disks, and choosing a higher mbal than any encountered, and starting phase1 with it.

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

According to the perspective of this paper, the main contribution of this article is a constant RMRC algorithm for Leader Election, both for the CC and the DSM models. The processes in this article are assumed to be asynchronous, but they do not fail. The definition of a remote reference is the conventional one - At the CC model any reference to a cache-variable which is not up-to-date with its corresponding counterpart is considered remote. At the DSM model any reference to a variable which is not on the local memory of the process that performs the reference will be a remote one.

The algorithm is based on team merges. Every team has a distinguished leader, and all the operations of the algorithm are done by it. The rest of the processes in each team are called idle. At the outset every process is the leader of the singleton team that consists of himself. From now on the algorithm performs merging phases. In each merging phase a team can either lose (and all its members lose also), merge with another team or become a playoff contender. Once a team becomes a playoff contender it goes out of the merging process and starts to participate in playoffs. Each merging phase results in at most one playoff contender team, and in phase i the teams consist of at least i members. Therefore there are at most n (# of processes) merging phases. After the merging phases, the playoff phases commence. The playoffs are performed in a decreasing order, starting from the index of the last merge, say Z. The merging phase numbered Z has only a single team, and it is automatically declared as playoff winner. Henceforth the playoffs are as follows - at playoff i the head of the winning team of the i+1 playoff and the head of the playoff contender team of phase i (If it exists. If not he wins automatically) are racing in a 2-process leader election. The winner among them continues with his team to the next playoff. The chosen leader will be the head of the 0th playoff phase winner team. Note that the constant RMRC of this algorithm is achieved by the fact that every process can serve as team leader only 3 times.

![Figure 2.1: Functions call graph for the leader election algorithm](image)

As can be seen in Figure 2.1, The main algorithm uses several sub-routines. Later on this paper we will modify these routines, Thus a brief description of them is requisite. Note that from MergeTeam and 2P-LeaderElec downwards, every merging phase uses different instances of the function (This observation will be of much use to us later on this paper). In all of the following descriptions, q is the executing process.

**Find** - Due to the simplicity of this function the reader is referred to section 3.3.1 of the full version of the article, page 17.

**LinkRequest and LinkReceive** - (Section 3.3.2 of the extended abstract) Both functions are used to create a directed graph (that will later become a forest) from the connections made by **Find**. **LinkRequest(p)** is intended
to let the process \( q \) become one of \( p \)'s sons (returns 0 or 1 to indicate if \( p \) confirmed the connection or not). \texttt{LinkReceive} is intended to let \( p \) know about his set of sons. The process \( q \) in fact renounces all the processes that did not read \( \perp \) from the variable \( B_q \), or did not begin to perform \texttt{LinkRequest}(\( q \)) (Line 1 in \texttt{LinkReceive} and lines 2-7 in \texttt{LinkRequest}). The array \( A_q \) is used to make \( q \) wait for the processes that commenced \texttt{LinkRequest}(\( q \)) but did not yet finish it, or to renounce the processes that did not do so (Line 1 in \texttt{LinkRequest} and lines 3-7 in \texttt{LinkReceive}). This way the array \( LINK_q \) will contain 1 in the places of which a mutual connection between \( q \) to that process was made, and 0 otherwise.

**Forest** - (Section 3.3.3 of the extended abstract) This function treats the process as tree nodes (hence the name), has no input and outputs a success value (indicating \textit{win} or \textit{lose}), a process ID indicating \( q \)'s parent and the set of \( q \)'s sons. The function uses \texttt{Find} to couple \( q \) with another process and then uses \texttt{LinkRequest} and \texttt{LinkReceive} to let \( q \)'s parent know about him, and let \( q \) know about his sons. If \( q \) fails to couple with another process, \( \texttt{LinkRequest} \) returns 0, \( q \) becomes either an \textit{idle} member or loses. If \( \texttt{Find} \) fails (i.e. returns \( \perp \)) \( q \) becomes the \textit{special} process (Line 5), that later may become the head of this phase’s playoff contender team. If \( q \) manages to couple with another process, the set \( L \) (returned by \texttt{LinkReceive}) is traversed in order to break any cycles in the graph, when the symmetry breaker is the ID of the process (Lines 10-15). The loop in lines 19-24 is intended to disconnect between \( q \) and the sons that cut him off. If all of \( q \)'s sons cut him off (which will result in \( L = \emptyset \)) and he is not the special process and \texttt{LinkRequest} had failed (which means \( p = \perp \)) he is bound to lose (Line 25, 26).

Else, \( (1, p, L) \) is returned:
- if \( q \) is the special process (which means \( p = \perp \)) and \( L = \emptyset \) then \( q \) is the leader of the team that will become the playoff contender in \texttt{MergeTeam}.
- if \( p = \perp \) and \( L \neq \emptyset \), than \( q \) is the leader of a non-playoff team (called \textit{hopeful}).
- if \( p \neq \perp \) than \( q \) is an idle member of his team, and his parent in the forest is \( p \).

**MergeTeam** - (Section 3.3.4 of the extended abstract) The main function of the merging phases. It receives the set of current team members as input, and outputs the status of the team after the current merging phase (\textit{lose}, \textit{playoff}, \textit{success}), the new set of members if \( q \) is the leader, or \( \perp \) if he is an idle member. After summoning \texttt{Forest} in line 1, \textit{lose} is returned if \( q \) had lost in \texttt{Forest}. If \( q \) was the \textit{special} node in \texttt{Forest}, he becomes a leader and his team becomes a playoff contender (Lines 5-7). Otherwise, \( q \) will perform a union operation between all the sons of his own sons (The set \( L \) of \( p \)'s sons was computed in \texttt{Forest} also) in lines 8-12. Afterwards, if \( q \) is the root of the tree computed in \texttt{Forest}, \( q \) is the chosen leader (Line 14), and if not \( q \) will notify the root about his set of sons and will become idle (Lines 15-18). Note that a \textit{success} return value indicates that the team is \textit{hopeful}.

**2PLeaderElect** - For the same reason as in \texttt{Find}, the reader is referred to section 8 of the full version of the article, page 60.

**LeaderElect** - (Section 3.1 of the extended abstract) The main function of the algorithm. Has no input and returns a value in \{\textit{win}, \textit{lose}\}. The function uses the variables -
- \textit{T} - The set of the team members.
- \textit{Z} - The phase number (merge or playoff).
- \textit{S} - The outcome of \texttt{MergeTeam} or \texttt{2PLeaderElect}.
- \textit{work} - used to bound (to 3) the number of times a process can serve as a team head.

Lines 2-8 contain the loop that deals with merging phases. Note that processes which are not leaders (called \textit{idle processes}) are waiting in line 7. Lines 10-17 contain the (single) playoff that \( q \) may participate in. If that playoff is the 0th and results in a victory, \( q \) wins (Line 19), else he passes the reign to another arbitrary process (Lines 21-26).

### 3 Deriving Disk Paxos From Classic Paxos

Initially, one may not fully understand the precise connection between the model described in [3] and the Classic Paxos Algorithm described in [2]. Although [3] contains a section with the same title as above, The description of the algorithm does not distinguish between the roles of proposers, acceptors and learners. As a consequence, many redundant RMRs might be taken into account. A more precise analysis of the behavior of each one of the possible roles could be informative.

Firstly, let us identify the operation of Message passing in the Disk model. As mentioned in [3], In each one of the phases a process \( p \), for every disk \( d \), writes \texttt{dblock}[\( p \)] to \texttt{disk}[\( d \)][\( p \)] and reads \texttt{disk}[\( d \)][\( q \)] for every \( p \neq q \). The operation of writing \texttt{dblock} to a disk could be regarded as sending the message that it contains to whom that will read it. Reading \texttt{disk}[\( d \)][\( q \)] for all \( p \neq q \) could similarly be thought of as receiving the previously sent messages. We will distinguish between four possible types of a \texttt{dblock} -
• A dblock having a valid mbal only (the rest of the fields are NULL or non-input) will be considered a prepare request, as defined in [2].

• A dblock having a valid bal,inp fields only will be similarly considered an accept request.

• A dblock having an inp value only will be called a terminal block.

• An empty dblock, or (Optional) one having a bal value only, will be called a learning block.

These definitions will allow a reading process to know which type of message he is reading.

Secondly, as will be concluded from the following analysis, the role of the acceptor is redundant in the Disk model (or alternatively, the proposer’s role is combined with that of the acceptor). The processes in this model can be divided into 2 groups only - proposers and learners. The precise analysis is as follows -

**proposer**

According to [2], a proposer p will begin Phase I by sending a prepare request, together with a ballot number n, to a majority of the acceptors. In the Disk model, the equivalent will be writing the new ballot number n in his own dblock[p].mbal, and writing dblock to disk[d][p] to some portion (or all) of the disks. A process that acts as a proposer can reset the fields bal and inp to null, since they have no use at this stage. As a result, a dblock having a valid mbal field only can be recognized by other processes as a prepare request. In [2], phase 1 comes to an end when a majority of the acceptors have responded to p’s prepare request with the appropriate commitment. In the Disk model this stage is redundant, since while p is reading other processes’ q block from a disk, he can distinguish either that q has already issued a proposer number higher than his own (in case that q is a proposer) or that it is empty, and thus q is a learner. If q happens to be a proposer with a higher ballot number, p can abort the phase and either become a learner or retry phase 1 with a higher ballot number. Thus p can end phase 1 in the Disk model after reading a majority of other processes’ block without encountering a ballot number larger than his own. Moreover, the acceptors’ commitment in [2] is in fact done in the Disk model by the proposer at the moment that he reads a prepare block.

In order to begin phase 2, p must choose a proper inp value. If non of the blocks p has read in phase 1 contained a valid inp field, p can choose any input value v. Otherwise, v will be that inp value. In [2], p begins phase 2 by sending an accept request to most of the acceptors. In the Disk model this is also redundant, since p can write his own dblock (containing inp=v and bal=n) to a majority (or all) of the disks - and every other proposer that will read his block can deduce that p is in phase 2 with the value v. After finishing the write phase, p will continue to read all the other processes’ blocks. If he will encounter a ballot number higher than his own, he again can abort the ballot. If not, p can finish the algorithm with the value v.

**learner**

In [2] there are several possibilities for the role of the learner. Most relevant for us will be to choose, in a way that will be discussed further in this paper, a set of distinguished learners. Each such a learner p needs to continuously read blocks from the disks and remember each accept request he encounters. If the same value v appears in most of the disks (i.e appears in disk[d][q] for over half of the disks, and some process q) and no higher ballot number is found at the rest of the disks (i.e in disk[d][w] for over half of the disks and every other process w), he can accept v. An issue left open at the article is how the distinguished learner/s inform the rest of the processes that a value had been decided. We propose the following way - once the learner learns that a value v had been decided, he writes a terminal block with the value v into his dblock and write it to disk[d][p] for every d. Any process that encounters a terminal block, regardless of his role, should end the algorithm with the value v. After finishing the writes at this stage, p can abort the algorithm. Note that the same action can be done also by a proposer that finishes the algorithm.

An optional addition to the role of the learner could be to cooperate with the proposers in the following sense - If p finishes reading all of the disks without encountering enough accept requests, he can write the max mbal value to his own dblock.bal, and write it to disk[d][p] for every d (Namely, write a learning block). This will help proposers to discover if a ballot number larger than their own exists.

4 RMR Analysis of Disk Paxos

The model presented in [3] is not a regular model for RMR complexity memory. Nevertheless we will try, as a first step, to evaluate it “as is” regarding the RMR complexity. In the model, there are n processes, each process p has a small block of memory dblock[p], and there are D remote disks to which each process writes to a designated block disk[d][p]. Since the non-remote block is very limited, and on the contrary, the remote blocks essentially contain
everything, it is hard to hope for much. Although termination is not even promised, in a stable state, where a leader can be elected, termination is guaranteed. So we will try to analyze RMR complexity with respect to a stable state.

Even under this restriction, we cannot present interesting results, since to complete the algorithm, a process must reach a higher ballot number than any proposed before him. Since these numbers are unbounded, it may take any (finite) number of rounds (assuming, as in the original text, that in each execution of phase 1, the ballot number increases by n, the number of processes). Consequently, we will consider a small modification - choosing a ballot number higher than any seen yet (in case of failure). Thus we will need no more than two phase 1, and one phase 2 operations to reach consensus, one phase 1 to encounter a higher ballot number (if exists), and one to succeed in writing the highest numbered ballot.

We have 3 phases:

1. Choosing:
   In this phase a process $p$:
   writes $dblock[p]$ to $disk[d][p]$ for a majority of the disks $- \frac{D}{2} + 1$ RMRs.
   then reads for a majority of the disks $disk[d][q]$ for all processes $q \neq p$ (as long as no greater mball than his own is encountered), each read is remote $- (n - 1) \cdot (\frac{D}{2} + 1)$ RMRs.
   We have all together $n \cdot (\frac{D}{2} + 1)$ RMR C.

2. Committing:
   First decide the value of $dblock[p].inp$ based solely on input gathered on phase 1, no RMR C.
   Next the same as in phase 1 is repeated (regarding the amount of read/write operations), so the complexity is the same: $n \cdot (\frac{D}{2} + 1)$ RMR C.

3. Recovering
   Look up $disk[d][p]$ for a majority of the disks $- \frac{D}{2} + 1$ RMR C.

So adding this together, we get the upper bound: $3n \cdot (\frac{D}{2} + 1)$ for the disk model after the system is stable with the modification mentioned above.

Without a leader (i.e. if the system is not stable), there may be any number of phases, since there may be more than one process executing the algorithm. In this case all we can say, is that if there are $i$ phases, the RMR complexity will be $i \cdot n \cdot (\frac{D}{2} + 1)$.

Disk-Paxos RMR complexity in the Cache Coherence model

In the Cache Coherence model, we assume each process has a cache large enough to cache all the data it needs. When a cached value is invalid (i.e. changed by another process) a remote access is needed in order to read the value. This refinement of observation, in the face of unbounded rounds, is not much help. So again, we will start analyzing after stability is reached, meaning there is a leader who is preforming the algorithm alone. In this case it is enough for the leader process to read all the disks once, and eventually his ballot number will pass any other ballot existing (all he needs is to write). So the RMR complexity is bounded to $(n + i - 1) \cdot (\frac{D}{2} + 1)$, where $i$ is the number of rounds needed for the leaders ballot number to become the maximum ballot number. Similarly, with respect to the modification mentioned above, it suffices to write thrice, (since two operations of phase 1 and one of phase 2 are enough) so the bound is $(n + 2) \cdot (\frac{D}{2} + 1)$.

In the general case (without stability) in this model, each phase will cost only the writing, and reading changed values, but formalizing this does not give much (due to the estimation needed for a process $p$, of how many changes happened between different operations of the phases).

Disk-Paxos RMR complexity in the DSM model

In the Distributed Shared Memory model, each process has control of a block of the shared memory, and the rest of the memory is considered remote. This model is not appropriate for the algorithm, since a process must read most of the disks, so it seems we won’t get any note worthy difference from the original disk model of [1], by assigning the blocks in any specific way.

---

1 the -1 comes from the fact that in each disk $d$ we read, we can skip $disk[d][p]$
Analysis of Classic Paxos derived from Disk Paxos

After setting the stage, we will try to use the above, to refine the RMR Analysis of the algorithm, taking use of a distinguished group of learners. Again we conclude that without achieving stability, there is no use for the analysis, as there are always unbounded RMR situations. This is due to the fact that different processes may compete indefinitely on the highest ballot number, and any leader will have to keep up with this. This means unbounded RMR.

So after the system is stable, or in any case a proposer succeeds in committing to a value, a learner would have to read the last updates of that proposer in most of the disks. We take account of what happens in the different models:

- [3] model:
  Here the leader will have to search most of the disks, in order to determine that most of them are “committed” to a certain value. This is no better than in the case without separating the learners and the proposers, since the only difference is the $\frac{D}{2} + 1$ writes that the learner is free of doing.

- Cache Coherent model:
  In the CC model, a learner would only have to look for new changes, so after a proposer leader is chosen, the learner needs only to see the $\frac{D}{2} + 1$ accept blocks. To evaluate the more general case, where we include phases without a leader chosen, for each phase done the leader will have to add $\frac{D}{2} + 1$ remote reads, so for $i$ phases, we have $i \cdot (\frac{D}{2} + 1)$ RMR.

- Distributed Shared Memory model:
  As stated earlier in this paper, to obtain better bounds with the DSM model, we need to distribute the memory in some smart way. We don’t see such a way in the general scheme, and distinguishing the learners from the proposers seems not much of help. A small achievement can be attained by letting each learner have a block of a proposer, so he needs only to search remotely for the rest of the proposers blocks. The problem is this forces the proposers to make remote references. Anyhow, the benefit is negligible.

To complete the discussion, we will show a way for the leader of the learners to inform the rest of the learners about the output of the algorithm. Since the interesting results in this section come from the CC model, we will use this model. All we need is to set a special block in some disk (for instance in the block that belongs to the learner leader), and the rest of the learners will spin on it - which is a single remote reference. when the learner-leader decides a result is found, it has only to write that value to that special place. More on choosing a leader in the disk model in the following section.

5 Leader Election Algorithm in the Disk Model

As already mentioned, leader election is crucial for the progress of the Paxos algorithm in the disk model. In this paper we will mainly be concerned about efficient RMR complexity for the leader election algorithm. For that reason, in this section we will try to implement the RMR-efficient leader election presented in [4]. There are two main difficulties in our way - one is that the algorithm presented in [4] is not at all fault tolerant, and the other is that [4] uses busy-wait on local cache variable on several occasions. The latter is problematic since no cache is considered in the disk model, or more precisely - no cache that is accessible to processes other than the process that owns it. Moreover, busy-waiting on a remote variable in the disk model will be RMRC-devastating, and it will not allow us to analyze the complexity properly. Therefore, We will be obliged to make two changes in the disk model in order to produce the desired algorithm -

1. Processes do not fail. (Disks can still fail).
2. Some restricted communication between processes is allowed.

The first change is obviously crucial for the success of the algorithm in [4]. We will also assume that a disk that fails does not comes back to life during the entire execution of the algorithm, and that at most half of the disks fail. Weaker assumptions could have been taken, such as the existence of a failure-detector, but we will not explore this option here. Another assumption we will make is that the disks are indexed by some totally ordered set. This assumption is not obligatory, and is only intended to lower the RMRC occasionally. The allowing of communication is merely intended to optimize the RMRC, and the communication that is allowed is as follows -
Only predetermined messages can be sent. Each message can contain a name of a variable and a process ID. The purpose of this mechanism is to simulate busy-waiting on a variable \( V \) - Whenever a process \( p \) needs to busy-wait on a variable, it stops making steps. Another process \( q \) that changes \( V \) sends the message \( "(V,q)" \) to \( p \), meaning that the copy of \( V \) on the disks has changed by \( q \) and \( p \) can check its value again. Note that \( V \) is found only on the disks, as in the ordinary disk model.

As can be seen in figure (2.1) the algorithm LeaderElect is based upon several simpler functions. We will now show how to implement each and every of the above functions in the disk model. These implementations will bare only minor changes from the ones in [4], all of which will concern the correctness of the functions in our model. In order to do so let us define the operations at hand -

**Notations**: high-level operations will be noted by upper-case letters, and low-level by lower-case. In the following algorithms lower-case letters will denote local variables, and upper-case letters will denote variables on the disks. The term “PID” will denote the ID of the executing process.

- \( x := \text{READ}(V) \) - The executing process will read \( V \) from the disks according to their order (If indeed such an order exists). Once some value is read successfully, it is returned and assigned into \( x \). This operation will be useful in cases that a variable changes its value only once, or alternatively in cases that we know that the last write to the variable has ended (e.g after waking up).

- \( \text{WRITE}(V) := x \) - The process will write the value \( x \) into the variable \( V \) in all of the (active) disks, again, according to their order.

- \( x := \text{READ}(w,V) \) - The process will try to read the variable \( V \) from all the (active) disks. If it encounters some value \( \neq w \), it stops reading and assigns it into \( x \). Otherwise, it returns \( w \) iff most (over half) of the disks are active. The result of this operation in case that over half of the disks has failed, as the result of the entire algorithm, is undefined.

- \( \text{send}(p,\{x,PID\}) \) - The process, whose ID is \( PID \), will send the message \( x \) (which is a variable name) to process \( p \).

- \( \text{sleep}(x,q) \) - The process will sleep until the process \( q \) sends the message \( x \). If \( q = \star \) the process will wait for a message labeled \( x \) from any of the processes.

**Proposition 1.** The above definitions allow a form of new-old inversion in the following scenario -

Assume there is a variable \( V \) in two disks (denoted DISK1, DISK2), and \( p_1, p_2, p_3 \) are performing the following operations -

- \( p_1 \) - WRITE(\( V \)):=1
- \( p_2 \) - WRITE(\( V \)):=2

Let us assume that context switches occur in such a way that DISK1.\( v \) = 2 and DISK2.\( v \) = 1 (It is possible if \( p_1 \) writes to DISK1, \( p_2 \) writes to DISK1&2, and then \( p_1 \) writes to DISK2). Then,

- \( p_3 \) - READ(\( V \))

Let us also assume that prior to \( p_3 \)'s READ operation, DISK1 fails. \( p_3 \) will then read \( v \) from DISK2 and will return 1, which is the previously written value. □

In order to overcome this difficulty we will use the \( \text{READ}(w,V) \) operation when necessary. The next observation will be of more use to us, and will identify the cases in which the above scenario cannot happen, and thus the algorithms we will present will remain as correct as their version in [4].

**Proposition 2.** If only one process is performing only one write into a variable \( V \), no new-old inversion between two previously written values will occur.

Note that in this case it is possible that prior to the ending of a WRITE operation, a corresponding READ operation will return the initial value instead of the written value. This case is equivalent to an ordinary read operation that is being performed before the write - and therefore each correctness proof of an algorithm that involves only read/writes of this kind will still hold.

We will now present the algorithms from [4] in their Disk-model version -
Algorithm 1 Find

Output: Either ⊥ or the ID of another process that executes Find.

Variables: \( F, G \) - process IDs, initially ⊥.

1. \( \text{WRITE}(F) := \text{PID} \)
2. \( s := \text{READ}(\bot, G) \)
3. \( \text{if } s = \bot \text{ then} \)
   4. \( \text{WRITE}(G) := \text{PID} \)
   5. \( s := \text{READ}(\text{PID}, F) \)
   6. \( \text{if } s = \text{PID} \text{ then} \)
      7. \( s := \bot \)
   8. \( \text{end} \)
9. \( \text{end} \)
10. \( \text{return } s \)

Find

(presented as algorithm 1) To overcome possible disk failures, each disk will contain all instances of the variables \( F, G \) for each one of the merging phases (There is a bounded number of phases - See Section 2). It remains to prove that the correctness proof of this function (Lemma 2 from the full version of [4], Section 3.3.1, page 18) still holds. Note that besides the high level operation, the pseudo-code presented in algorithm (1) differs from the one at [4] only in line 5. This change is necessary because of the following scenario (As implied in Proposition 1) - \( p \) and \( q \) write their PID to the \( F \)’s simultaneously such that \( \text{disk1}.F=p, \text{disk2}.F=q \), and then simultaneously read \( \bot \) from \( G \). They write to \( G \) in some order and then \( p \) reads \( \text{disk1}.F \), sets his \( s \) to \( \bot \) in line 7 and returns \( \bot \). Before \( q \) does the next step, \( \text{disk1} \) fails, and then \( q \) reads \( \text{disk2}.F \). Since \( \text{disk2}.F=q \), \( q \) also returns \( \bot \) for the same reason. This will contradict part (b) of the lemma.

Proof. part a of the lemma obviously still holds for the same reason. As for part b, let us assume that two processes \( p \) and \( q \) both return \( \bot \). It follows that both of them executed line 7, since this is the only way of returning \( \bot \) (This is because otherwise, the variable \( s \) received its value from \( F \) in line 5, but \( F \) cannot contain \( \bot \) since line 1 was executed). We conclude that the condition in line 6 was true for both \( p \) and \( q \), thus the READ operation in line 5 returned \( p,q \) respectively(*). Let us note that if both \( p \) and \( q \) entered the if at line 3 then they must have read \( \bot \) from \( G \). Due to the nature of the operation \( \text{READ}(\bot, G) \), it follows that any of the processes’ WRITE to \( G \) in line 4 happened after the other process READ \( G \) in line 2. As a result, we can deduce that any of the READ operations in line 5 occurred after the other process had executed line 1. Therefore, since either of the processes’ PID was eventually written to most of the disks, it is not possible that both \( p \) and \( q \) received their own PID, since if (w.l.o.g) \( p \) got his PID, then he did not encounter \( q \)’s PID, and his own PID is written in most of the disks. In that case, \( q \) must return \( p \)’s PID in line 5, since it is written in most of the disks. Note that this proof also holds for more than 2 processes, since if the READ in line 5 does not encounter a value different than the PID of some process \( p \), it must be that \( p \)’s PID is written in most of the disks.

LinkRequest and LinkReceive

Algorithm 2 LinkRequest(p)

Input: process ID \( p \).
Output: a value in \( \{0, 1\} \) indicating a failure or success, respectively.
1. \( \text{WRITE}(A_p[PID]) := 1 \)
2. \( s := \text{READ}(B_p) \)
3. \( \text{if } s = \bot \text{ then} \)
   4. \( \text{link} = 1 \)
5. \( \text{else} \)
   6. \( \text{link} = 0 \)
7. \( \text{end} \)
8. \( \text{WRITE}(\text{LINK}_p[PID]) := \text{link} \)
9. \( \text{send}(p,\{\text{LINK},PID\}) \)
10. \( \text{return } \text{LINK} \)
Algorithm 3 LinkReceive

Output: set of processes to which a link was established.

1 WRITE(B) := 1
2 forall processes IDs q ≠ PID do
   3 if READ(⊥, A_PID[q]) = ⊥ then
      link[q] := 0
   4 else
      5 if READ(⊥, LINK_PID[q]) = ⊥ then sleep(LINK, q)
      6 link[q] := READ(LINK_PID[q])
   7 end
9 end
10 return \{ q | link[q] = 1 \}

(Algorithms 3, 2) We will analyze LinkReceive and LinkRequest together since they use some mutual variables, and subtle issues regarding the correctness of Proposition 2 must be taken into account.

The implementation of LinkRequest in our model is identical to that in [4] beside the different definitions we gave to READ and WRITE, and the wake-up message in line 9.

First, let us note that the variables A_p[PID] and LINK_p[PID] are being written into only once (†), since a different instance of them is being used in every merging phase, and LinkRequest is summoned only once per process and per merging phase. Moreover, The variable B_p is also written into only by the process p itself (line 1 of LinkReceive) and only once per merging phase. For these reasons, Proposition 2 becomes handy and the correctness proof LinkRequest follows immediately from the one at [4].

Second, in order to validate Proposition 2 for LinkReceive, let us note several differences between the above implementation of LinkReceive, and the one in [4] -

- read and write operations from [4] were replaced by their high-level counterparts.
- An operation of the type READ(w,V) was added instead of the regular read at line 3, for RMRC optimization and prevention of new-old inversion with the initial value.
- A local array named “link” was added. This was in order to validate (†), since an ordinary WRITE at this stage might have cause a new-old inversion. Moreover, in [4] the executing process p reads and writes to LINK_p only to compute the return value at line 10. Therefore we would rather to perform this computation locally, as it will simultaneously validate (†) and save the some redundant RMR.
- In order to validate the local computation of “link”, after waking up in line 7, p (The executing process) needs to acquire the new value of LINK_PID[p], which he now knows that is not ⊥, and assign it into link[q].

Forest

In this algorithm (presented here as algorithm 4), the variable CUT_p[PID] is being written into only once, and thus Proposition 2 holds. Since the owner of CUT_p[PID] needs to busy-wait on this variable, a wake-up message is being sent in line 16. The READ operation in line 21 is intended to prevent new-old inversion.

2PLeaderElect

This is a very simple function, which involves only one call to the function Forest and local variables only. Its execution will be identical to the one in [4] (the code is presented only in the full version of the paper, Section 8, page 60).

MergeTeam

Again, the variable S_p[PID] is being written into only once, and a wake-up message is send in line 17. (See algorithm 5)

LeaderElect

In this algorithm (presented as algorithm 6), some modification of variables is required in order to validate the correctness proof. In addition, subtle analysis of the use of the common variables (denoted Z,S,T in the article) is
Algorithm 4 Forest

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

1 $p \leftarrow \text{Find}$
2 if $p \neq \perp$ then
3   $\text{link} \leftarrow \text{LinkRequest}(p)$
4 else
5   special $\leftarrow 1$
6   link $\leftarrow 0$
7 end
8 $\text{set} \leftarrow \text{LinkReceive}()$
9 if $\text{link} = 1$ then
10   if $\text{set} \neq \emptyset$ \land \text{PID} > p then
11     \text{WRITE}(\text{CUT}_p[\text{PID}]) := 1$
12     $p \leftarrow \perp$
13   else
14     \text{WRITE}(\text{CUT}_p[\text{PID}]) := 0$
15   end
16   $\text{send} (p, \{\text{CUT}, \text{PID}\})$
17 else
18   $p \leftarrow \perp$
19 end
20 forall $q \in \text{set}$ do
21   if $\text{READ}(\perp, \text{S}[q]) \neq \perp$ then sleep($\text{CUT}, q$)
22   if $\text{READ}(\text{CUT}[q]) = 1$ then
23     $\text{set} \leftarrow \text{set} \setminus \{q\}$
24 end
25 end
26 if $p = \perp$ \land \text{set} = \emptyset \land \text{special} \neq 1$ then
27   return $(0, \perp, \emptyset)$
28 end
29 return $(1, p, \text{set})$

Algorithm 5 MergeTeam($\tau$)

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

1 $(s, p, \text{set}) \leftarrow \text{Forest}$
2 if $s = 0$ then
3   return $(\text{lose}, \tau)$
4 end
5 if $p = \perp$ \land \text{set} = \emptyset$ then
6   return $(\text{playoff}, \tau)$
7 end
8 $\text{set} \leftarrow \tau \cup \text{set}$
9 forall $r \in \text{set}$ do
10   if $\text{READ}(\perp, \text{S}[r]) \neq \perp$ then sleep($\text{S}, r$)
11   $\tau \leftarrow \tau \cup \text{READ}(\text{S}[r])$
12 end
13 if $p = \perp$ then
14   return $(\text{success}, \tau)$
15 else
16   \text{WRITE}(\text{S}_p[\text{PID}]) := \tau$
17   $\text{send}(p, \{\text{S}, \text{PID}\})$
18   return $(\text{success}, \perp)$
19 end
Algorithm 6 LeaderElect

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

1. \( t \leftarrow \emptyset, z \leftarrow 0, s \leftarrow \text{success}, work \leftarrow 0 \)
2. while \( work < 3 \land s = \text{success} \) do
3. \( \quad work \leftarrow work + 1 \)
4. \( \quad z \leftarrow z + 1 \)
5. \( \quad (s, t) \leftarrow \text{MergeTeam}_2(t) \)
6. \( \quad \text{if } t = \perp \text{ then} \)
7. \( \quad \quad \text{if } \text{READ}(\perp, M.T) = \perp \text{ sleep}(M, +) \)
8. \( \quad \quad \quad t = \text{READ}(M.T), s = \text{READ}(M.S), z = \text{READ}(M.Z) \)
9. \( \quad \text{end} \)
10. \( \text{end} \)
11. if \( s = \text{playoff} \land z \geq 1 \) then
12. \( \quad \text{temp} \leftarrow 2P\text{LeaderElect}_2() \)
13. \( \quad \text{if } \text{temp} = \text{lose} \text{ then} \)
14. \( \quad \quad s \leftarrow \text{lose} \)
15. \( \quad \text{else} \)
16. \( \quad \quad z \leftarrow z - 1 \)
17. \( \quad \text{end} \)
18. \( \text{end} \)
19. if \( s = \text{playoff} \land z = 0 \land t = \emptyset \) then
20. \( \quad \text{return win} \)
21. \( \text{end} \)
22. if \( t \neq \emptyset \) then
23. \( \quad q \leftarrow \text{arbitrary process in } t \)
24. \( \quad m \leftarrow (t \setminus \{q\}, s, z) \)
25. \( \quad \text{WRITE}(M_q) := m \)
26. \( \quad \text{send}(q, \{M, \text{PID}\}) \)
27. \( \text{end} \)
28. \( \text{return lose} \)
needed. Although in the original algorithm the variables $T,Z,S$ are local to the process, they need to be updated by another process (lines 23-25 in the original algorithm). In our algorithm the variable $M$ contains as sub-fields the variables $S,T,Z$ and its content is being read from/written into only when a communication through it needs to be done - i.e in line 24, where the current leader needs to throne the next one (and notify a loss if needed), and in line 8 where the process needs to know if he is the leader of this phase. The rest of the operations on $S,T,Z,M$ are done on their local counterparts, $s,t,z,m$. Proposition 2 is not useful here, since multiple writes into $M_q$ are possible. The correctness will follow from the following lemma.

**Lemma.** Two different writes to the variable $M_q$ cannot overlap.

**Proof.** As follows from the definition of the original algorithm, these lines (23-25 in the original algorithm, 24 in algorithm 6) are executed only by a team leader, and their purpose is to throne a different leader to the same group. A fact that follows from the correctness proof of MergeTeam is that every process belongs to exactly one team in every merging phase. Assume that there exists a process $q$ such that two writes to $M_q$, by $p_1,p_2 \neq q$, overlap each other. It follows that $p_1,p_2$ were leaders of teams that contained $q$, and of course that in different phases. W.l.o.g assume that $p_1$ is the team leader in an earlier phase, say level $i$. Hence, $q$ is the leader of level $i+1$, and have to throne the subsequent leader himself. A contradiction will be obtained by the fact that $p_2$’s write to $M_q$ can only begin after $q$’s will finish executing lines 8-23 of algorithm 6. (Since before being throned, $q$ was at most at the “busy-wait” in line 7, and $p_2$ was throned by a leader which was $q$, or one of his subsequent leaders). Thus two writes to $M_q$ cannot overlap. □

**Corollary.** The operations on $M_q$ are linearizable through atomicity, and the correctness of algorithm 6 follows.

**RMR complexity**

The high level operations we applied are performing $d$ (# of disks) low-level operations at the worst case. In each one of the algorithms we added no more than a constant number of high level operations. Therefore, when regarding to read/write operations only, the $O(1)$ RMRC of [4] falls down to $O(d)$ in our case. Into this analysis we must add the message passing, which also is $O(1)$ per algorithm.

**References**


