Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Group mutual exclusion in linear time and space

Yuan He [a], K. Gopalakrishnan [b,*], Eli Gafni [a]

[a] *Department of Computer Science, University of California at Los Angeles, Los Angeles, CA-90095, United States*
[b] *Department of Computer Science, East Carolina University, Greenville, NC-27858, United States*

A R T I C L E   I N F O

Article history:
Received 1 May 2016
Received in revised form 12 April 2017
Accepted 30 May 2017
Available online 7 June 2017

Keywords:
Mutual exclusion
Group mutual exclusion
Remote memory reference complexity
Lamport's Bakery Algorithm
Black and White Bakery Algorithm

A B S T R A C T

We present two algorithms for the **Group Mutual Exclusion (GME) Problem** that satisfy the properties of *Mutual Exclusion, Starvation Freedom, Bounded Exit, Concurrent Entry and First Come First Served*. Both our algorithms use only simple read and write instructions, have $O(N)$ Shared Space complexity and $O(N)$ Remote Memory Reference (RMR) complexity in the Cache Coherency (CC) model. Our first algorithm is developed by generalizing the well-known Lamport's Bakery Algorithm for the classical mutual exclusion problem, while preserving its simplicity and elegance. However, it uses unbounded shared registers. Our second algorithm uses only bounded registers and is developed by generalizing Taubenfeld's Black and White Bakery Algorithm to solve the classical mutual exclusion problem using only bounded shared registers. We show that contrary to common perception our algorithms are the first to achieve these properties with this combination of complexities.

1. Introduction

Mutual Exclusion is a classical problem in distributed computing introduced by Dijkstra in 1965 [4]. The *Group Mutual Exclusion (GME)* problem, introduced by Joung in 2000 [10], is a natural generalization of the classical mutual exclusion problem.

Before formally stating the problem, we first motivate the generalization by the following illustrative example. In university environment we can think of the critical section as a room allocated by the University to students of all faiths to pray in the most neutral policy, FCFS (First-Come First-Serve). Naturally (to believers!), students of different faith cannot use the room to pray concurrently. However, students of the same faith are encouraged to pray concurrently. Since students pray individually, a student cannot overtake another waiting student to enter the prayer room just because a member of her faith is currently praying. This will prolong the waiting of the waiting student. Yet, students who arrive while only others of the same denomination are present can enter and pray without waiting. We assume that open minded Christian, or secular who might call to God at a dire strait, may pick a different denomination from time to time, though, they have to declare their pick upon arrival. If a member of different declared denomination is present, then service is FCFS. Students negotiate entrance to the prayer room by writing on a blackboard, intermittently taking a break from waiting to have coffee, and then come back to check the negotiated queue. Henceforth a student is a process, and each student depending on UID, has a space allocated on the blackboard to be written in exclusion.

* Corresponding author.
E-mail addresses: heyuan89@cs.ucla.edu (Y. He), gopal@ecu.edu (K. Gopalakrishnan), eli@ucla.edu (E. Gafni).

http://dx.doi.org/10.1016/j.tcs.2017.05.030

In Group Mutual Exclusion problem, processes repeatedly cycle through four sections of code viz., *Remainder Section, Entry Section, Critical Section (CS)* and *Exit Section*, in that order. An execution of the last three sections will be called an *invocation*. A process picks a session number when it leaves the remainder section and this session number can be different in each invocation. A process is said to be an *active* process, if it is in one of its invocations. Two active processes are in *conflict* if their session numbers are different. Unlike the classical mutual exclusion problem, multiple processes are allowed to be in the critical section at the same time, provided they are not conflicting. In fact, in the presence of active processes all of which are mutually non-conflicting, entry into the critical section takes bounded number of process steps. Formally, the problem consists of designing code for the entry section and the exit section such that the following four properties are satisfied.

P1 Mutual exclusion: No two conflicting processes can be in the critical section at the same time.
P2 Starvation freedom: If no process stays in the critical section forever, then any process that enters the entry section eventually enters the critical section.
P3 Bounded exit: After entering the exit section, a process is guaranteed to leave it within a bounded number of its own steps.
P4 Concurrent entry: In the absence of conflicting processes, a process in the entry section should be guaranteed to enter the critical section within a bounded number of its own steps.

The *Concurrent Entry* property is crucial to the GME problem. It was stated informally by Joung [10] and then later formalized by Hadzilacos [6]. The intent of this property is to ensure concurrency: active processes that request the same session, in the absence of conflicting processes, should be allowed to enter the CS without unnecessary synchronization among themselves.

In this paper, we will require that the algorithm satisfies the *First Come First Served (FCFS)* property in addition to the above four properties. The standard way to formalize this is to split the entry section into two sections viz., *Doorway Section* and *Waiting Room Section*. The doorway section of the code is free of "wait" statements, i.e., it can be completed by a process in a bounded number of its own steps. The waiting room section is where the actual synchronization with other conflicting processes occurs and may entail indefinite waiting. The notion of doorway was originally introduced by Lamport [12] in the context of classical mutual exclusion problem. We would say that process P_i *doorway precedes* process P_j, if P_i completes the doorway before P_j enters the doorway. Now, the *FCFS* property can be formally stated as given below.

P5 FCFS: If process P_i doorway precedes process P_j and the two processes request different sessions, then process P_j does not enter the critical section before process P_i.

If neither P_i doorway precedes P_j nor P_j doorway precedes P_i, we would say the two processes are *doorway concurrent*.

1.1. Model

We consider a system consisting of N processes, named $P_1, P_2 \ldots P_N$ and a set of shared variables. Each process also has its own private variables. Processes can communicate only by writing into and reading from shared variables. An execution is modeled as a sequence of process steps. In each step, a process performs some local computation or reads from a shared variable or writes into a shared variable. We assume that these steps of reading or writing are atomic. The processes take steps *asynchronously*. Specifically, this means that an unbounded number of steps of some other process could be performed in between two consecutive steps of a process. We assume that our processes are *live*; this means that if a process is active it will eventually execute its next step.

We allow only simple read and write operations on shared variables. We assume that these read and write operations are atomic. However, we do not assume that processes have access to more powerful synchronization operations such as atomic test-and-set and compare-and-swap.

There are two general architectural paradigms considered for shared memory in the literature, viz., Distributed Shared Memory (DSM) Model and Cache–Coherency (CC) Model. In the DSM model, each processor has its own memory module and each shared variable is assigned to a particular processor. When a processor is referencing a shared variable, it is locally accessible, if it is allocated to that processor itself; on the other hand, it is a remote access, if it is allocated to some other processor. In the CC model, all shared variables are stored in a global memory module that is not associated with any particular processor. Each processor also has a local cache and some hardware protocol ensures cache coherence, i.e., all copies of the same variable in different local caches are consistent. Every time a process reads a shared variable, it does so using a local (cached) copy of the variable. A local copy of the variable may not be valid, if the process has never read the variable before or if some process overwrote it in the global memory module. Whenever, upon reading a cached variable, the process is informed by the system that its value is invalid, it makes a remote memory reference and migrates the variable to its local cache. We assume that once a shared variable is brought into a process's local cache it remains there indefinitely. Also, every time a process writes a shared variable, the process writes the variable in the global memory module, which involves a remote memory reference. Note that, this action invalidates all cached copies of that variable. In this paper, we will be working exclusively under the cache–coherency model, a model of practical significance considering that virtually every modern multi-processor is cache–coherent.

By the term time complexity, we mean Remote Memory Reference (RMR) complexity in this paper. It is simply the worst case number of remote memory references performed by a process in one invocation. This is because remote memory references are the most time consuming operations as they involve interconnect traversal. Also, counting the number of remote memory references is probably the only reasonable way to evaluate the time complexity of mutual exclusion algorithms, since plain steps complexity of mutual exclusion algorithms is unbounded. An algorithm is called a *local-spin* algorithm if the maximum number of remote memory references made in any invocation is bounded.

By the term space complexity, we mean the total amount of shared space a solution entails. We do not count the private variables when measuring space complexity. There are two different situations of interest, one in which the shared variables are unbounded registers and the other in which the shared variables are bounded registers.

1.2. Our contribution and related work

Joung's original algorithm for the GME problem satisfies the four basic properties. It does not satisfy the FCFS property. Moreover, it has unbounded RMR complexity. Hadzilacos (see [6]) gave the first solution for the GME problem that has the FCFS property. His algorithm can be thought of as a modular composition of two independent algorithms, one, "the FCFS algorithm" provides FCFS property (but not necessarily guarantees mutual exclusion) and the other, "the ME algorithm" provides mutual exclusion property (but not necessarily FCFS). This algorithm has shared space complexity of $\Theta(N^2)$. It was (mistakenly in hindsight, see Section 4) claimed that the algorithm has RMR complexity of $O(N)$ in the CC model. The algorithm uses only bounded shared variables and simple read and write operations. It was left as an open problem to develop a solution (satisfying P1 through P5) for the GME problem that runs in linear time and space using only bounded shared variables.

Subsequently, in [8], Jayanti et al. presented an algorithm presumed to be of linear time and space, solving the challenge of Hadzilacos. Jayanti et al. retained the idea of modular composition and also the "ME algorithm" that Hadzilacos used. They came up with a clever modification to the "FCFS algorithm" of Hadzilacos to reduce the space complexity to $\Theta(N)$. Although they did not explicitly claim so, their algorithm was also deemed to have the linear RMR Complexity in view of the wrong claim by Hadzilacos.[1]

Both works use a slightly modified version of a classical mutual exclusion algorithm developed independently by Burns [2] and Lamport [13] as the "ME algorithm". This is an elegant algorithm that uses just one bit of shared space per process. In Section 4, we show that this algorithm actually has an intricate structure and has the worst case RMR complexity of $\Omega(N^2)$. It follows that algorithms of both Hadzilacos and Jayanti et al. are of RMR complexity $\Omega(N^2)$. Hence, the challenge posed by Hadzilacos has yet to be met. Our observation, and part of our contribution, that the challenge is still on, initiated this paper.

Our first algorithm, presented in Section 2, is a generalization of the classic Lamport's Bakery Algorithm to solve the GME problem while maintaining its simplicity and elegance. It uses unbounded registers to solve the GME problem (satisfying P1 through P5) and runs in linear time and space.

Takamura and Igarashi also made an attempt in [14] to generalize Lamport's Bakery Algorithm to solve the GME problem. They presented three different algorithms in that paper. However, none of their algorithms satisfies the concurrent entry property nor the FCFS property.

In 2004 (which is after the publication of [6] and [8]), Taubenfeld [15] came up with an elegant algorithm called *Black and White Bakery Algorithm* that solves the classical mutual exclusion problem with only bounded shared registers. His approach is a lot simpler than a prior method that bounds the registers of the Lamport's Bakery Algorithm developed by Jayanti et al. in [9]. Our second algorithm, presented in Section 6 is a generalization of the Black and White Bakery Algorithm to solve the GME problem. Our algorithm satisfies the properties P1 through P5 and runs in linear time and space using bounded shared registers. Thus, our algorithm is the first one to solve the open problem originally posed by Hadzilacos, a decade and a half ago.

We present a comparison of different GME algorithms in Table 1. To make the comparison fair, we include only those algorithms that solve the problem using only simple read/write instructions. The first row in the table describes the properties of Joung's original algorithm for the GME problem. The next four rows compare the algorithms for the GME problem that use unbounded shared registers. The last four rows compare the algorithms for the GME problem that use bounded shared registers.

2. Generalizing Lamport's Bakery Algorithm

In this section, we present a very simple algorithm for the GME problem by generalizing Lamport's Bakery Algorithm (see [12]) for the classical mutual exclusion problem. This algorithm is based on the method commonly used in bakeries, in

[1] For example, the recent paper by Bhatt and Huang [1] explicitly states that the RMR complexity of the algorithm by Jayanti et al. is $O(N)$.

Table 1
Comparison of GME algorithms that use only simple read/write instructions.

Algorithm	RMR	Space	P2?	P3?	P4?	P5?	Bounded?
Joung [10]	∞	O(N)	Yes	Yes	Yes	No	Yes
Takamura et al. [14] I	∞	O(N)	No	Yes	No	No	No
Takamura et al. [14] II	O(N)	O(N)	Yes	No	No	No	No
Takamura et al. [14] III	O(N)	O(N)	Yes	No	No	No	No
This work (**GLB Algorithm**)	O(N)	O(N)	Yes	Yes	Yes	Yes	No
Keane & Moir [11]	O(log N)	O(N)	Yes	No	No	No	Yes
Hadzilacos [6]	O(N^2)	O(N^2)	Yes	Yes	Yes	Yes	Yes
Jayanti et al. [8]	O(N^2)	O(N)	Yes	Yes	Yes	Yes	Yes
This work (**BWBGME Algorithm**)	O(N)	O(N)	Yes	Yes	Yes	Yes	Yes

which a customer receives a token number upon entering the store and the holder of the lowest token number is the next one served. We will refer to it as Generalized Lamport's Bakery (GLB) Algorithm.

The algorithm is presented in Fig. 1. It uses three shared variables. The first one *Session* is an integer array of size N and *Session*$[i]$ indicates the session number that process P_i requests in the current invocation to enter the CS. The second one is *Token*, an integer array of size N and *Token*$[i]$ represents the token number selected by process P_i. The third shared variable is *Choosing*, a boolean array of size N and *Choosing*$[i]$ is true would indicate that process P_i is currently attempting to make a new request to enter the critical section. The *Session* array and the *Token* array are initialized to zero and the *Choosing* array is initialized to false. The doorway of the algorithm consists of lines 3–6 and the waiting room is made up of lines 7–10.

When a process leaves the remainder section, it first sets the variable *Choosing*$[i]$ to true to signal other processes that it is about to make a new request to enter the critical section. Next, it places the desired session number in *Session*$[i]$. We assume that all session numbers are positive integers. It then selects its token number to be one more than the maximum of the token numbers of all other processes and places it in *Token*$[i]$. Finally, P_i sets *Choosing*$[i]$ to false to signal other processes that it has completed the doorway section for the new request.

Figure 1 Generalized Lamport's Bakery Algorithm.

```
Shared variables:
    Session: array[1..N] of integer, initially all 0
    Token: array[1..N] of integer, initially all 0
    Choosing: array[1..N] of boolean, initially all false

Private variables:
    mysession: integer, initially 0

1:  repeat
2:      REMAINDER SECTION

3:      $Choosing[i] := \mathbf{true}$
4:      $Session[i] := mysession$
5:      $Token[i] := 1 + max(\{Token[j] | (1 \le j \le n)\})$
6:      $Choosing[i] := \mathbf{false}$

7:      for $j := 1$ to $N$ do
8:          wait until $((Choosing[j] = \mathbf{false}) \vee (Session[j] \in \{0, mysession\}))$
9:          wait until $(((Token[i], i) < (Token[j], j)) \vee (Token[j] = 0) \vee$
                       $(Session[j] \in \{0, mysession\}))$
10:     end for

11:     CRITICAL SECTION

12:     $Token[i] := 0$
13:     $Session[i] := 0$
14: forever
```

In the waiting room, at line 8, for each other process P_j, process P_i checks to see whether *Session*$[j]$ is zero or same as *mysession*. In either case, there is no problem and so P_i can move on. On the other hand, if process P_j is requesting a conflicting session (i.e., $Session[j] \notin \{0, mysession\}$), P_i waits for P_j to finish the doorway and set *Choosing*$[j]$ to be false. Next, at line 9, process P_i waits on each conflicting process P_j (i.e., $Session[j] \notin \{0, mysession\}$) until either P_j has exited the critical section (i.e., $Token[j] = 0$) or P_j has a larger token number (i.e., $((Token[i], i) < (Token[j], j))$). It is possible that two conflicting processes pick the same token number. In that case, we use the process identifier to resolve the ties (line 9). The relation "less than" among ordered pairs of integers is defined as in the Lamport's Bakery Algorithm. That is $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$ (a, b, c and d are all integers). For simplicity, if $Token[i] = Token[j]$, we say the process with the smaller process identifier has the smaller token number. After the loop, process P_i enters the CS. In the exit section, P_i resets *Token*[i] to 0 and then *Session*[i] to 0.

3. Proof of correctness of the generalized Lamport's Bakery Algorithm

The algorithm satisfies the mutual exclusion property, starvation freedom property, bounded exit property, concurrent entry property and the FCFS property. Here we present a complete proof that the algorithm satisfies these properties.

Lemma 1. *The GLB algorithm has the FCFS property.*

Proof. Assume two conflicting processes i and j request the CS. Also assume that process i finishes the doorway before process j starts the doorway. So, when process j is computing its token number, it will read the token number of process i and select a larger token number. As process j has a larger token number, when process j checks with process i at line 9, it will wait for process i to exit the CS. □

Lemma 2. *The GLB algorithm has the Mutual Exclusion property.*

Proof. Suppose two conflicting processes i and j are in the CS at the same time with different sessions. By Theorem 1, process i and process j must be doorway-concurrent. Without loss of generality, we assume process i enters the CS first. Since processes i and j are doorway concurrent, process j has at least finished executing line 3 when process i checks on process j at line 8.

If process j has also finished updating *Session*[j] by that time, process i will wait for process j to finish the doorway, as they are requesting conflicting sessions. Since process i enters the CS first, it must have a smaller token number than process j. As a consequence, at line 9, process j will wait for process i because it finds that process i has a smaller token number. Therefore, process j cannot enter the CS until process i finishes the CS and resets its token number, which is contradicting with our assumption.

On the other hand, if process j has not finished updating *Session*[j] by that time, then process i will not wait on process j in line 8 as *Session*[j] $= 0$. However, if this is the case, process i has already selected its token number whereas process j has not yet begun selecting its token number. So, process j's token number is guaranteed to be larger than that of process i and process j will wait on process i at line 9. Therefore, process j cannot enter the CS until process i finishes the CS and resets its token number, which is contradicting with our assumption. □

Lemma 3. *The GLB algorithm has the Bounded Exit property.*

Proof. Since the exit section consists of two simple write instructions, a process that enters the exit section will trivially finish it within a bounded number of its own steps. □

We now define another property called **Deadlock Freedom**. Deadlock freedom simply means deadlocks cannot occur in the system. Informally, **Deadlock** occurs in the system when one or more processes are "trying to enter" their critical sections, but no process ever does so. Lamport (see page 329 in [13]) has shown that under the assumptions that no process stays in the critical section forever (which we are assuming), there are only finitely many processes (which we are assuming) and the algorithm satisfies the "Bounded Exit property" (which we have just now shown is the case for our algorithm), the deadlock freedom property can be formally stated as follows:

P6 Deadlock Freedom If one or more processes are forever trying to enter their critical section with no success for any of them, then there exists a process that enters the critical section infinitely often.

Lemma 4. *The GLB algorithm has the Deadlock Freedom property.*

Proof. Suppose the algorithm does not satisfy the deadlock freedom property. Then there is an execution of the algorithm in which a nonempty set S of processes enter the entry section but none of them enters the CS, and no process enters the CS infinitely often. We observe that no process can wait at line 8 forever since every process j that requests the CS will finish the doorway eventually and set *Choosing*[j] to false and process j enters the CS only finitely many times. Therefore, processes in the set S must wait on line 9 forever. There exists a process i that has the smallest token number among all processes in S and that process i will pass all other processes at line 9 and enter the CS, which is a contradiction with our assumption. □

It is easy to observe that the *Starvation Freedom* property immediately implies the *Deadlock Freedom* property. However, the converse is not necessarily true. Deadlock freedom means that the entire system of processes will always continue to make progress. However, it does not preclude the possibility of individual processes not making progress (i.e., waiting forever in the entry section).

Lemma 5. *The GLB algorithm has the Starvation Freedom property.*

Proof. Lamport has proved (see page 330 in [13]) that if an algorithm satisfies the deadlock freedom property and the FCFS property, then it necessarily satisfies the starvation freedom property. Hence, combining Lemma 1 and Lemma 4, we conclude that our algorithm has the starvation freedom property. □

Lemma 6. *The GLB algorithm has the Concurrent Entry property.*

Proof. Assume that process i requests a session, and no other process requests a different session. This means, for every other process j, it holds that $Session[j] \in \{0, mysession\}$. Since process i always checks whether $Session[j] \in \{0, mysession\}$ in both the busy–wait loops in line 8 and line 9, it cannot wait on either lines. So, it enters the CS within a bounded number of its own steps, thus satisfying the concurrent entry property. □

We now analyze both the time and space complexity of the algorithm. The algorithm uses three shared variables, *Session, Token* and *Choosing*, each of which is an array of size N. So, obviously the algorithm has $\Theta(N)$ shared space complexity. However, the token numbers used by this algorithm will grow in an unbounded manner, just like in the Lamport's Bakery Algorithm.

We now analyze the time complexity of the algorithm. As said before, by time complexity, we mean the RMR complexity of the algorithm because remote memory references are the most time consuming operations. Recall that in the CC model, all shared variables are stored in a global memory module and processes migrate them to their local cache to access them. In the waiting room, there are only two loops viz., the busy–wait loops in line 8 and line 9. In line 8, when process i is busy waiting for a process j, if $Choosing[j]$ changes to false, then process i will immediately terminate the wait. It is possible that (before process i observes the changed value of $Choosing[j]$) precess j sets the $Choosing[j]$ to true again and requests another conflicting session. In that case, since process i doorway-proceeds process j, process j will get a larger token number than process i. So, process j cannot enter the CS to finish that invocation and therefore cannot change the *Session* again until process i finishes the CS. Thus, line 8 can only involve a maximum of five RMR (three for $Choosing[j]$ and two for $Session[j]$). Similarly in line 9, when process i is busy-waiting on $Token[j]$, if $Token[j]$ changes, its new value will be either zero or a larger token number and in either case, process i will terminate the wait. Also, any change in $Session[j]$ will also entail a change in $Token[j]$ and thus terminate the wait. Hence, line 9 involves a maximum of five RMR (one for $Token[i]$, two for $Token[j]$, two for $Session[j]$). There are only a constant number of RMR in line 8 and line 9. As these two lines are enclosed within a for loop that can run a maximum of N times, it follows that the entire waiting room is of $O(N)$ RMR complexity. Note that the doorway made up of line 3 through line 6 involves only a constant number of RMR, except for the implicit loop in line 5. The implicit loop in line 5 has $O(N)$ RMR complexity as it involves inspecting the token numbers of all other processes. Finally, it is easy to see that the exit section consisting of lines 12–13 involves exactly two RMR. Hence, the overall RMR complexity of this algorithm in the CC model is $O(N)$.

We now summarize the results in the form of the following Theorem.

Theorem 1. *The Generalized Lamport's Bakery Algorithm presented in Fig. 1 solves the GME problem by satisfying all the five properties P1 through P5 in linear space (with unbounded registers) and time (RMR under the CC model) using only simple read and write operations.*

An earlier paper by Takamura and Igarashi [14] also made an attempt to generalize Lamport's Bakery Algorithm to solve the GME problem. They presented three different algorithms in that paper. Their first algorithm is fairly simple, but does not satisfy the starvation freedom property. Their second and third algorithms do satisfy the starvation freedom property. However, apart from being quite complicated, they do not satisfy the bounded exit property. None of the three algorithms satisfies the FCFS property nor concurrent entry property. However, all three of their algorithms satisfy a weaker property known as *concurrent occupancy* in the literature (see [11] and [6]). The GLB algorithm shown in Fig. 1 is the simplest and most natural generalization of Lamport's Bakery Algorithm for the GME problem.

4. A flaw in the literature

We now look at attempts to solve the GME problem using only bounded shared registers and simple read and write operations. Two prominent algorithms in this regard are that of Hadzilacos [6] and that of Jayanti et al. [8]. Both of the algorithms can be viewed as a modular composition of an "FCFS Algorithm" and an "ME Algorithm". Not only that both of the algorithms use modular composition technique, but also they both use the same "ME Algorithm". Hadzilacos's algorithm is of space complexity $\Theta(N^2)$ and he claimed that his algorithm is of $O(N)$ RMR complexity in the CC model. Hadzilacos posed it as an open problem to devise an algorithm to solve the GME problem in linear time and space using only bounded

Figure 2 Burns–Lamport ME algorithm.

```
1: L: $Competing[i] :=$ true
2: for $j := 1$ to $i - 1$ do
3:     if $Competing[j]$ then
4:         $Competing[i] :=$ false
5:         wait until not $Competing[j]$
6:         goto L
7:     end if
8: end for
9: for $j := i + 1$ to $N$ do
10:     wait until not $Competing[j]$
11: end for
```

shared variables. Jayanti et al. came up with a clever modification to the "FCFS algorithm" of Hadzilacos and reduced the space complexity to $\Theta(N)$. We now show that the "ME Algorithm" they used is of complexity $\Omega(N^2)$ in the CC model thus establishing that neither of these two algorithms has $O(N)$ RMR complexity.

The "ME algorithm" used by them is independently discovered by Burns [2] and Lamport [13]. This algorithm is depicted in Fig. 2. In this algorithm *Competing* is a shared array of size N. Each element of the array is a boolean variable, initialized to false. The code given in Fig. 2 is for process P_i. The variable j is a private variable.

We now briefly describe the algorithm. Every process P_i owns a single bit *Competing*$[i]$. Only P_i can write into *Competing*$[i]$; Other processes can read it. Before entering the CS, P_i sets its bit to true and checks all lower numbered processes (lines 1–3). If any of them, say process P_j, is found to have set its bit to true, then process P_i resets its bit to false, allowing the smaller-numbered process to make progress. It then waits for process P_j's bit to become false and then restarts the competition to enter the CS by going to line 1. Having checked all lower-numbered processes, process P_i then checks the higher-numbered ones and waits for each of them to set its bit to false. However, this time while P_i is waiting it does not set its bit to false. After that process P_i enters the CS. It turns out that this simple algorithm guarantees mutual exclusion. We refer the reader to [13] (see also [2]) for a proof of correctness.

To analyze the RMR complexity under the CC Model, consider the following sequence of events:

1. Process P_N sets its bit to true.
2. Process $P_{(N-1)}$ sets its bit to true.
3. Process P_N checks all lower-numbered processes and finds that $P_{(N-1)}$'s bit is set. So, P_N sets its bit to false and waits for *Competing*$[N-1]$ to become false.
4. Process $P_{(N-2)}$ sets its bit to true.
5. Process $P_{(N-1)}$ checks all lower-numbered processes and finds that process $P_{(N-2)}$'s bit is set. So, process $P_{(N-1)}$ sets its bit to false and waits for *Competing*$[N-2]$ to become false.
6. Process P_N now finds *Competing*$[N-1]$ to be false and so restarts the competition by setting its bit to true.
7. Process P_N checks all lower-numbered processes and finds that process $P_{(N-2)}$'s bit is set. So, P_N sets its bit to false and waits for *Competing*$[N-2]$ to become false.
8. Process $P_{(N-3)}$ sets its bit to true.
9. Process $P_{(N-2)}$ checks all lower-numbered processes and finds that process $P_{(N-3)}$'s bit is set. So, process $P_{(N-2)}$ sets its bit to false and waits for *Competing*$[N-3]$ to become false.
10. Process P_N now finds *Competing*$[N-2]$ to be false and so restarts the competition by setting its bit to true.
11. Process P_N checks all lower-numbered processes and finds that process $P_{(N-3)}$'s bit is set. So, P_N sets its bit to false and waits for *Competing*$[N-3]$ to become false.
12. .
13. .
14. .
15. Process P_N checks all lower-numbered processes and finds that process P_1's bit is set. So, P_N sets its bit to false and waits for *Competing*$[1]$ to become false.
16. Process P_1 checks all higher-numbered processes and finds that all the bits are false and enters CS.
17. Process P_1 exits the CS and sets its bit to false.

Note that during the above sequence of events, process P_N got blocked by each one of the lower numbered processes once. At the end of the above sequence, process P_1's request is satisfied. Also, at the end of the above sequence, the *Competing* bit of P_2 through P_N are all false. Now, we can create a similar sequence of events, but this time with only processes P_2 through P_N participating. We can recursively create a similar sequence of events again and again until finally we have only process P_N participating.

The net effect is that in this worst case scenario, process P_N got blocked by process $P_{(N-1)}$ a total of $(N-1)$ times, by process $P_{(N-2)}$ a total of $(N-2)$ times and so on. So, the total number of times that process P_N gets blocked by some other processes is

$$\sum_{k=1}^{N-1} k = N(N-1)/2 = \Omega(N^2)$$

In the CC model, at least one remote memory reference is involved each time a process gets blocked and hence the worst case RMR complexity of the algorithm in Fig. 2 is $\Omega(N^2)$ in the CC Model.

Therefore, the problem of developing a linear time (RMR) and linear (shared) space algorithm that uses only bounded shared variables for the GME problem, originally posed by Hadzilacos, is still open.

One might get the impression that we can immediately fix the problem, by plugging in some other mutual exclusion algorithm that has $O(N)$ RMR Complexity in place of Lamport–Burnst Mutual Exclusion Algorithm. Unfortunately, the situation is not that simple as we have to adapt the ME algorithm so that it provides concurrent entry for application in the development of GME algorithm using the modular composition technique. The Lamport–Burnst algorithm is easy to adapt by simply adding an extra condition to check whether the session number of the other process is the same as the session number of this process in all wait-until loops. On the other hand, the mutual exclusion algorithm of Taubenfeld [15] which has $O(N)$ RMR Complexity, as well as that of Yang and Anderson [16] which has $O(\log N)$ RMR Complexity, is not suitable for plugging in the modular design. Simply adding an extra condition to check the session number in all wait-until loops in these algorithms does not provide concurrent entry as they have more intricate structures. As far as we know, there is no mutual exclusion algorithm of $O(N)$ RMR complexity in the literature, that is easily adaptable to provide concurrent entry and can be plugged in the modular design. So, the problem of developing a linear time and linear space GME algorithm that uses only bounded registers is indeed a non-trivial problem. We develop such an algorithm in Section 6.

5. Black and White Bakery Algorithm

In 2004, Taubenfeld [15] came up with an elegant algorithm called **Black and White Bakery Algorithm** that solves the classical mutual exclusion problem with only bounded shared registers. In this section, we first review this algorithm and in the next section, we generalize the ideas developed in that paper and solve the GME problem with only bounded shared registers in linear time and space.

The key idea in this algorithm is to view the token as a colored token, i.e., the token has two components *color* and *number*. The color will be either black or white and the number will be a positive integer. As in Lamport's Bakery algorithm, we also use the *Choosing* shared variable. The algorithm also uses an additional shared bit variable called *GlobalColor* which can be either black or white. Unlike the other shared variables, *GlobalColor* is a multi-writer, multi-reader variable.

The algorithm is depicted in Fig. 3.

When process P_i wants to enter the CS, it first sets the *Choosing* variable to true to notify other processes that it is attempting to pick its token. Then it reads the value of *GlobalColor* and sets its own token color to the read value. It then picks a number which is greater than the token numbers of all processes which have the same token color as that of P_i. After having selected the colored token, the process resets its *Choosing* variable to false to indicate to other processes that it is done with picking a token.

Once P_i has got its colored token, it enters the waiting room and it waits until its colored token is the *lowest* and then enters the CS. The order between the colored tokens is defined as follows: If two tokens have the same color, the token with the smaller number is smaller. If two token have different colors, the token whose color is different from the *GlobalColor* is smaller. If two processes have the same token color and the same token number, the process identifiers are used to break the tie as in Lamport's Bakery algorithm.

When P_i is through with the CS, it sets the shared variable *GlobalColor* to the opposite of its own token color and then resets its own token number to zero. The setting of *GlobalColor* to the opposite color is to ensure that priority is given to waiting processes whose token color is the same as the one that P_i held.

If at a certain point of time t, the *GlobalColor* has a value of c, then the algorithm ensures that all processes with token color different from c that are in the entry section at time t enter the CS before any process with a token color of c.

After each time the *GlobalColor* changes, the token numbers again start from one and hence the token numbers used can only grow up to N, where N is the number of processes. We refer the reader to [15] for a complete exposition of the algorithm.

6. Solving GME with bounded registers

In this section, we generalize the ideas developed by Taubenfeld in [15] and solve the GME problem using bounded registers in linear time and space.

Our algorithm also uses a multi-writer, multi-reader shared bit variable called *GlobalColor* (see Fig. 4) which can only be black or white. All other shared variables used in the algorithm can only be written by one process even though they can be read by multiple processes. Each process uses a shared variable called *Token* that has three components viz., *session*, *color* and *number*. We assume that processes can read or write into this *Token* variable atomically even though it has three components. This is not an unreasonable assumption as this can be implemented without the aid of any higher level synchronization primitives by encoding three integers into a single integer using simple techniques (which we are not elaborating further here). Finally each process also has a boolean shared variable called *Choosing*. Unlike the *GlobalColor*

Figure 3 Black and White Bakery Algorithm.

```
1:  **repeat**
2:    **REMAINDER SECTION**
3:    $Choosing[i] :=$ **true**
4:    $Token[i].color := GlobalColor$
5:    $Token[i].number := 1 + max(\{Token[j].number \mid$
                    $Token[i].color = Token[j].color\})$
6:    $Choosing[i] :=$ **false**
7:    **for** $j := 1$ **to** $N$ **do**
8:      **wait until** $(Choosing[j] =$ **false**$)$
9:      **if** $Token[i].color = Token[j].color$ **then**
10:       **wait until**
          $(((Token[i].number, i) < (Token[j].number, j))\vee$
          $(Token[i].color \neq Token[j].color)\vee$
          $(Token[j].number = 0))$
11:     **else**
12:       **wait until**
          $((Token[i].color \neq GlobalColor)\vee$
          $(Token[i].color = Token[j].color)\vee$
          $(Token[j].number = 0))$
13:     **end if**
14:   **end for**
15:   **CRITICAL SECTION**
16:   **if** $Token[i].color =$ **black then**
17:     $GlobalColor :=$ **white**
18:   **else**
19:     $GlobalColor :=$ **black**
20:   **end if**
21:   $Token[i].number := 0$
22: **forever**
```

Figure 4 Header for Black and White Bakery GME Algorithm in Fig. 5.

Shared variables:

GlobalColor: a **bit** of type {**black**, **white**}, initialized arbitrarily
Token: **array**[1..*N*] of (*session*: **integer**, *color* ∈ {**black**, **white**, ⊥}, *number*: **integer**), initially all (0, ⊥, 0)
Choosing: **array**[1..*N*] of **boolean**, initially all false

Private variables:

mysession: **integer**, initially 0
mycolor: a **bit** of type {**black**, **white**}, initialized arbitrarily
mynumber: **integer**, initially 0
other: (*session*: **integer**, *color* ∈ {**black**, **white**, ⊥}, *number*: **integer**), initially (0, ⊥, 0)

variable, the token color of a process can be black or white or a special value denoted by ⊥ which indicates that the process has not yet set its token color.

The algorithm is depicted in Fig. 5 and we will refer to it as the Black and White Bakery GME (BWBGME) Algorithm. The doorway of the algorithm is made up of lines 3–15 and the waiting room section consists of lines 16–23. When a process leaves the remainder section, it picks a session number and then updates its *Token* variable to reflect it. It then sets its *Choosing* variable to be true to notify other processes that it has initiated the task of picking a token. The token color is set to be the same as the current value of the shared variable *GlobalColor*. The token number is set to be one more than the maximum of token numbers of conflicting processes with the same color and is set to be 1 in case there are no conflicting processes with the same color. The process then updates its *Token* variable to reflect the chosen color and number (line 14). It then sets its *Choosing* variable to be false to notify other processes that it is done with the task of picking a colored token (line 15).

In the waiting room, for each other process P_j, process P_i checks to see if it is an active conflicting process. If it is not, then there is no problem and P_i does not wait on P_j. If it is, then process P_i waits until P_j has completed selecting its token color and number, if it has initiated the task (line 17). Process P_i then checks whether it has priority over P_j (lines 18–22). If so, it does not wait on process P_j and otherwise it waits on P_j. The priority order between conflicting processes is the same as that in the Black and White Bakery Algorithm.

Figure 5 Black and White Bakery GME Algorithm.

```
 1: repeat
 2:   REMAINDER SECTION
 3:   $Token[i] := (mysession, \perp, 0)$
 4:   $Choosing[i] :=$ true
 5:   $mycolor := GlobalColor$
 6:   $mynumber := 0$
 7:   for $j := 1$ to $N$ do
 8:     $other := Token[j]$
 9:     if $((other.color = mycolor) \wedge (other.session \notin \{0, mysession\}))$ then
10:       $mynumber := \max(other.number, mynumber)$
11:     end if
12:   end for
13:   $mynumber := mynumber + 1$
14:   $Token[i] := (mysession, mycolor, mynumber)$
15:   $Choosing[i] :=$ false
16:   for $j := 1$ to $N$ do
17:     wait until
          $((Choosing[j] = \textbf{false}) \vee (Token[j].session \in \{0, mysession\}))$
18:     if $Token[j].color = mycolor$ then
19:       wait until $(((mynumber, i) < (Token[j].number, j)) \vee$
          $(Token[j].color \neq mycolor) \vee (Token[j].session \in \{0, mysession\}))$
20:     else
21:       wait until $((GlobalColor \neq mycolor) \vee$
          $(Token[j].color = mycolor) \vee (Token[j].session \in \{0, mysession\}))$
22:     end if
23:   end for
24:   CRITICAL SECTION
25:   if $mynumber \neq 1$ then
26:     if (not OPPOSITECOLOR($mycolor$)) then
27:       if $mycolor =$ black then
28:         $GlobalColor :=$ white
29:       else
30:         $GlobalColor :=$ black
31:       end if
32:     end if
33:   end if
34:   $Token[i] := (0, \perp, 0)$
35: forever
```

Figure 6 Method OPPOSITECOLOR(*color*) for Black and White Bakery GME Algorithm in Fig. 5.

```
1: for $j := 1$ to $N$ do
2:   $other := Token[j]$
3:   if $((other.session \neq 0) \wedge (other.color = \overline{color}))$ then
4:     return true
5:   end if
6: end for
7: return false
```

At the time of exiting, process P_i checks whether its token number is 1 (line 25). If so, it just resets its *Token* variable to the initial value and exits. If not, it checks whether there is an active process (not necessarily a conflicting process) with the *opposite token color* (see Fig. 6). The *opposite token color*, denoted by $\overline{color}$, is defined to be *black* if $color$ = white, and vice versa. If so, it just resets its *Token* variable to the initial value and exits. If not, then it updates the *GlobalColor* to the opposite of its token color and then resets its *Token* variable to the initial value. In particular, note that token color is reset to $\perp$. This is important to ensure that other processes do not erroneously use a process's old token color while determining the priority.

The generalization of the Black and White Bakery Algorithm to solve the GME problem is quite tricky. While the formal proof of correctness can be found in Section 7, we provide some main insights into our generalization here. In the original Black and White Bakery Algorithm, when a process is attempting to enter the CS, it selects its token color as the current global color and its token number to be one more than the maximum of token numbers of processes with the same color. When it exits the CS, it simply updates the *GlobalColor* to be the opposite of its own token color. A naive generalization of Black and White Bakery Algorithm would simply add an additional check to see whether the other process is a conflicting process in all busy–wait loops. This naive generalization, nevertheless, cannot ensure the correctness in the case of group mutual exclusion. In the GME, as processes with the same session and different token colors can be in the CS at the same time, if a process leaving the CS simply updates the *GlobalColor* as before, it may erroneously allow conflicting processes to stay in the CS simultaneously.

Consider the following scenario. Initially, the *GlobalColor* is white. Processes P_i and P_j request the session S and get the token color of white. Then, P_i and P_j enter the CS concurrently because there is no conflict. When P_i is exiting, it sets the *GlobalColor* to black. Next, a process P_k requests the same session S and gets its token color of black. It is easy to see that P_k enters the CS by the concurrent entry property. After that, a process P_l starts to request a conflicting session S' and gets the token color of black. In the waiting room, P_l waits for P_j since they have different token color and the *GlobalColor* is black. However, if we let P_k exit the CS and then simply set the *GlobalColor* to white, P_l will stop waiting for P_j as it sees the *GlobalColor* is different from its token color. Hence, two conflicting processes P_j and P_l will be in the CS simultaneously, thus violating the mutual exclusion property.

To get a correct generalization, we observed a key invariant of the original Black and White Bakery Algorithm.

Invariant 1. *After a process P_i gets its token color from the GlobalColor, the GlobalColor cannot be flipped twice, before the process P_i finishes the critical section and gets out of the exit section.*

In the original Black and White Bakery Algorithm, suppose a process P_a gets a token color of black and after some time the *GlobalColor* gets changed to white by some process P_b. Now, in order for some other process P_c to change it again to black, P_c must be a white process. However, P_c can change it only while it exits. In order to exit, P_c must first enter the CS. As *GlobalColor* is currently white and P_a has a color of black, P_a will have the higher priority over P_c to enter the CS. So, the *GlobalColor* cannot be changed again until P_a gets out completely.

The fundamental idea in generalizing the Black and White Bakery Algorithm is to ensure that this invariant is maintained. It is not difficult to see that doing so solves the mutual exclusion violation illustrated in the previous scenario. Although processes P_j and P_k with different token color stay in the CS at the same time, the *GlobalColor* will not be updated when P_k executes the exit section (as otherwise the *GlobalColor* is flipped twice since P_j got its token color and before it exits) and therefore, P_l will still wait for P_j until it exits.

In order to maintain this invariant in the generalization, when a process finishes the CS, we let it check whether there is another active process with the opposite token color. If there exists such a process, then the exiting process does not update the *GlobalColor*. Otherwise, the process updates the *GlobalColor* to be the opposite color of its own token color. Also, while a process is checking this condition, it may unintentionally access the old token color of another process even though that process has already finished the previous invocation. To prevent this from happening, we let processes reset their token color to empty ($\perp$) at the end of the exit section.

However, the new *GlobalColor* updating mechanism is not enough to keep this invariant. A process in the doorway may read the (opposite) *GlobalColor* and then stop before updating its token color. Hence, another process in the exit section would have no clue as to which color this process will get (it will see a color of $\perp$ for this process). It could erroneously think that there are no active processes with the opposite token color and flip the *GlobalColor* (while in fact there is an active process with the opposite color). We can devise an intricate sequence of execution to show that the key invariant does not hold if there is such a process that stops just before writing down its token color in the doorway.

In order to handle this, we use a different scheme (from that in the original Black–White algorithm) for token number picking (lines 7–12) and add another condition to check before updating the *GlobalColor* (line 25). When a process is picking a token number, it determines the maximum of token numbers of conflicting processes with the same token color and then increments it by 1. If there are no such processes, it selects its token number to be 1. At the time of exiting, a process does not even attempt to update the *GlobalColor* if its token number is 1. On the other hand, if its token number is 2 or more, then it attempts to update the *GlobalColor* (it actually does if there is no active process with the opposite token color).

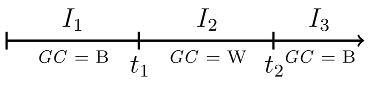
These changes are necessary to solve the issue of overlooking a process with the opposite token color having a "$\perp$" value. Suppose that a hanging process P_h has read a value of black for *GlobalColor* and has not yet written into its token color variable. Suppose that some other process changes the *GobalColor* from black to white after some time. Later, if a process P_u starts to execute the algorithm and attempts to change the *GlobalColor* again, then P_u must be a white-colored process with a token number greater or equal to 2. In order for process P_u to get a token number of 2 or more, there must exist an active conflicting white-colored process P_v with a smaller token number. Clearly, one of P_u and P_v must be in conflict with the hanging process P_h because either P_u or P_v has a different session with P_h. Therefore, at least one of P_u or P_v would have waited (in line 17) for P_h to finish the token selection before it enters the CS because P_h is hanging at the doorway when P_u and P_v enter the waiting room. Whichever is the case, as P_u can enter the CS only after P_v has left the CS, we can conclude that the "hanging process" P_h has really written down the read *GlobalColor* to its token variable by the time P_u is checking P_h's token color in the exit section. This shows that the scenario that we mentioned before cannot possibly occur anymore. A formal proof that our generalization maintains this key invariant is available in the next section (see Lemma 9).

7. Proof of correctness of the Black and White Bakery GME Algorithm

In this section, we present a complete and formal proof of the correctness and complexity analysis of Black and White Bakery GME Algorithm. In particular, we show that our algorithm satisfies the properties P1 through P5, uses only bounded registers and has linear time and space complexity.

We use the notation $P_i@x$ to denote that process P_i is executing line x of the algorithm. The notation $P_i@x \rightarrow P_j@y$ means process P_i executes line x before process P_j executes line y. We use $\overline{color}$ to denote the *opposite* color (i.e, $\overline{color} =$ black if $color =$ white, and vice versa). $\overline{color}$ is not defined if $color = \perp$. It is easy to see that *GlobalColor* always has a value of either black or white. Consider the execution of the algorithm on the global time line. We use I_i to denote the time interval that the *GlobalColor* remains as some color after it has been flipped $(i-1)$ times from the beginning. Once the *GlobalColor* is changed to the opposite color at the end of I_i, the *GlobalColor* remains as that color during the interval I_{i+1}. We use t_i to represent the time point at which the *GlobalColor* is flipped to the opposite color at the end of I_i. Note that, t_i will be the starting point of the time interval I_{i+1}. Process P_{t_i} denotes the process that flips the *GlobalColor* at t_i.

Here is an example to show a possible execution of the algorithm.



At the beginning, the *GlobalColor* is initialized to black, and it remains black in the time interval I_1. At the time point t_1, the *GlobalColor* is set to white by a process P_{t_1}. Then, the *GlobalColor* remains as white in the time interval I_2. Note that we are not claiming that the *GlobalColor* will not be updated during I_2. However, the *GlobalColor* will not be updated to black during I_2 by the very definition of the interval I_2. The *GlobalColor* remains as white in the time interval I_2 until a process P_{t_2} sets it to black at time point t_2.

Lemma 7. *If a process P_i gets a Token.number of 2 or more, then when P_i is calculating its Token.number in the doorway (lines 7–13), there must exist a conflicting process P_j with the same Token.color as P_i and a smaller Token.number. Moreover, P_i cannot enter the CS before P_j finishes the CS.*

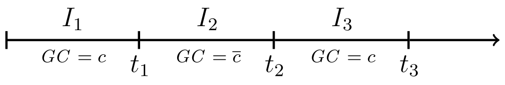
Proof. According to line 9, when a process is calculating its *Token.number*, it will ignore any process with the same session or different *Token.color*. Assume there doesn't exist such a conflicting process P_j with the same *Token.color* as P_i and smaller *Token.number*. It is easy to see that P_i will get a *Token.number* of 1, which is a contradiction with P_i getting a *Token.number* of 2 or more. Therefore, such P_j must exist when P_i is calculating its *Token.number*.

Moreover, as P_j has the same *Token.color* as P_i and a smaller *Token.number*, P_i will wait for P_j at line 19 until P_j exited the CS and reset its *Token*. Thus, our claim is true. □

Lemma 8. *If a process P_{t_i} flips the GlobalColor at the time point t_i, then there must exist a conflicting process $P_{\widetilde{t_i}}$ with the same Token.color as P_{t_i} and a smaller Token.number when P_{t_i} was calculating its Token.number (lines 7–13). Moreover, both P_{t_i} and $P_{\widetilde{t_i}}$ execute line 5 in the time interval I_i.*

Proof. Since process P_{t_i} flips the *GlobalColor* at time point t_i, according to line 25, P_{t_i} must have the *Token.number* of 2 or more. By Lemma 7, there must exist a conflicting process $P_{\widetilde{t_i}}$ with the same *Token.color* and a smaller *Token.number* when P_{t_i} was calculating its *Token.number* in the doorway. Next, we show that both P_{t_i} and $P_{\widetilde{t_i}}$ execute line 5 in time interval I_i.

We prove our claim using mathematical induction. Without loss of generality, assume the *GlobalColor* is initialized to c.



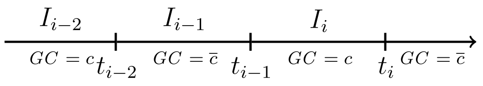
K = 1: Process P_{t_1} flips the *GlobalColor* from c to $\bar{c}$ at t_1. Obviously, P_{t_1} and $P_{\widetilde{t_1}}$ must have the *Token.color* of c. Since I_1 is the first and the only time interval that the *GlobalColor* is c before time point t_1 and so, P_{t_1} and $P_{\widetilde{t_1}}$ must execute line 5 in I_1.

K = 2: Process P_{t_2} flips the *GlobalColor* from $\bar{c}$ to c at t_2. P_{t_2} and $P_{\widetilde{t_2}}$ must have the *Token.color* of $\bar{c}$. Since I_2 is the only time interval that the *GlobalColor* is $\bar{c}$ before t_2, P_{t_2} and $P_{\widetilde{t_2}}$ execute line 5 in I_2.

Assume the claim holds for $K = (i-1)$.

***K* = *i*:** Process P_{t_i} may flip the *GlobalColor* either from c to $\bar{c}$ or $\bar{c}$ to c at time point t_i. We consider only the case where *GlobalColor* flips from c to $\bar{c}$ at time point t_i as the argument is similar in the other case.

If P_{t_i} flips the *GlobalColor* from c to $\bar{c}$ at t_i, it is easy to see both P_{t_i} and $P_{\widetilde{t_i}}$ have the *Token.color* of c.



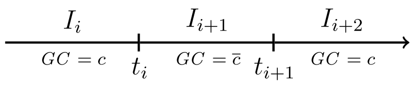
We prove our claim by showing both P_{t_i} and $P_{\widetilde{t_i}}$ execute line 5 in I_i

1. Assume P_{t_i} does not execute line 5 in I_i, then it must execute line 5 before t_{i-2} since the *GlobalColor* is $\bar{c}$ during I_{i-1}. By induction hypothesis, in time interval I_{i-1}, $P_{t_{i-1}}$ and $P_{\widetilde{t_{i-1}}}$ execute line 5 and then write their *Token.color* of $\bar{c}$. Clearly, P_{t_i} must conflict with either $P_{t_{i-1}}$ or $P_{\widetilde{t_{i-1}}}$ since $P_{t_{i-1}}$ and $P_{\widetilde{t_{i-1}}}$ have different sessions. Therefore, according to line 17, at least one of $P_{t_{i-1}}$ or $P_{\widetilde{t_{i-1}}}$ will wait for P_{t_i} to finish the doorway to set *Choosing*$[t_i]$ to false. By Lemma 7, $P_{t_{i-1}}$ cannot enter the CS before $P_{\widetilde{t_{i-1}}}$ finishes the CS. So, when $P_{t_{i-1}}$ enters the CS, P_{t_i} has already wrote down its *Token.color* of c and finished the doorway. After $P_{t_{i-1}}$ finishes the CS and checks the condition at line 26, it will find out that P_{t_i} has the opposite *Token.color*, and so, $P_{t_{i-1}}$ will not update the *GlobalColor*, which is a contradiction with the assumption that $P_{t_{i-1}}$ flips the *GlobalColor* at t_{i-1}. Therefore, process P_{t_i} executes line 5 in time interval I_i.
2. Since P_{t_i} executes line 5 in I_i, it must read the *Token.number* of $P_{\widetilde{t_i}}$ in I_i, which implies $P_{\widetilde{t_i}}$ does not reset its *Token* until P_{t_i} reads it in I_i. Assuming $P_{\widetilde{t_i}}$ does not execute line 5 in I_i, it must execute line 5 before t_{i-2}. Obviously, $P_{\widetilde{t_i}}$ conflicts with either $P_{t_{i-1}}$ or $P_{\widetilde{t_{i-1}}}$. Therefore, before $P_{t_{i-1}}$ enters the CS, $P_{\widetilde{t_i}}$ has already written down its *Token.color* of c and finished its doorway. $P_{\widetilde{t_i}}$ keeps its *Token* at least until P_{t_i} reads it in I_i and this implies $P_{\widetilde{t_i}}$ keeps its *Token* when $P_{t_{i-1}}$ is exiting. Hence, in the exit section, $P_{t_{i-1}}$ will find that $P_{\widetilde{t_i}}$ has the opposite *Token.color* and so, $P_{t_{i-1}}$ will not flip the *GlobalColor*, which is a contradiction. So, process $P_{\widetilde{t_i}}$ executes line 5 in I_i.

We have shown that if P_{t_i} flips the *GlobalColor* from c to $\bar{c}$ at t_i, then both P_{t_i} and $P_{\widetilde{t_i}}$ execute line 5 in I_i.
Hence, we have proved our claim for the case $K = i$ and therefore the result follows by mathematical induction. □

Lemma 9. *After a process P_i executes line 5, the GlobalColor cannot be flipped more than once before P_i finishes the exit section (line 34).*

Proof. Without loss of generality, we assume P_i reads the *GlobalColor* of c at line 5 in the time interval I_i. Assume that the *GlobalColor* is flipped twice at time t_i and t_{i+1} before P_i finishes the exit section. $P_{t_{i+1}}$ represents the process that flips the *GlobalColor* at t_{i+1}.



By Lemma 7 and Lemma 8, in I_{i+1}, two conflicting processes $P_{t_{i+1}}$ and $P_{\widetilde{t_{i+1}}}$ execute line 5 and then finish the CS. So, P_i must conflict with either $P_{t_{i+1}}$ or $P_{\widetilde{t_{i+1}}}$. Before $P_{t_{i+1}}$ enters the CS, P_i must finish the doorway and write down its *Token.color* of c. Thus, after $P_{t_{i+1}}$ has finished the CS, it will find out that P_i has the opposite *Token.color* and fail to update the *GlobalColor*, which is a contradiction with our assumption that $P_{t_{i+1}}$ flips the *GlobalColor* at t_{i+1}. Hence, we have proved the lemma. □

Lemma 10. *If process P_i and process P_j request conflicting sessions and process P_i finishes the doorway before process P_j executes line 4, then process P_j does not enter the CS before process P_i finishes the CS.*

Proof. For the sake of concreteness, assume process P_i gets the *Token.color* of c. Process P_j may have a *Token.color* of either c or $\bar{c}$. We consider both possibilities.

1. If process P_j has the *Token.color* of c, it will get a larger *Token.number* than P_i as P_j has not begun to calculate its *Token.number* when P_i finishes the doorway. Thus P_j will wait for P_i at line 19 until P_i finishes the CS and resets its *Token* at line 34.
2. If process P_j gets the *Token.color* of $\bar{c}$, that means the *GlobalColor* was flipped to $\bar{c}$ after P_i read the *GlobalColor* of c at line 5. By Lemma 9, the *GlobalColor* cannot be flipped again to c until P_i finishes the exit section. Hence, P_j will find out it has the different *Token.color* from P_i and waits for it at line 21. None of the conditions in line 21 can become true until P_i finishes the CS and resets its *Token*$[i]$.

We have shown that in all cases, process P_j cannot enter the CS before process P_i finishes the CS. □

Theorem 2. *The BWBGME algorithm satisfies the FCFS property.*

Proof. If processes P_i and P_j request conflicting sessions and P_i finishes the doorway before P_j starts the doorway, by Lemma 10, P_j cannot enter the CS before process P_i enters the CS. □

Theorem 3. *The BWBGME algorithm satisfies the Mutual Exclusion property.*

Proof. Suppose the algorithm does not satisfy the mutual exclusion property. At some point of time, there exist two processes P_i and P_j requesting different sessions that are in the CS simultaneously. By Lemma 10, neither P_i nor P_j can finish their doorway before the other executes line 4. Thus, processes P_i and P_j must have an overlap when they execute lines 4–15. Since P_i and P_j have different sessions, they wait for each other at line 17 to finish the doorway to set *Choosing* to false before determining which one of them has the priority.

For the sake of concreteness, we assume that process P_i enters the CS with the *Token.color* of c. When P_i checks line 18 for P_j, it may have the *Token*[j]*.color* of c or $\bar{c}$. We analyze both cases.

1. Process P_i finds that P_j has the same *Token.color* of c. In order for P_i to enter the CS, it must have a smaller *Token.number* in comparison to P_j. So, process P_j cannot enter the CS before P_i finishes the CS and resets its *Token*[i] as it has the larger *Token.number*, contradicting the assumption.
2. Process P_i finds that process P_j has the opposite *Token.color* of $\bar{c}$. There are two possibilities: (I) after process P_j reads the *GlobalColor* of $\bar{c}$ at line 5, the *GlobalColor* is flipped to c and then process P_i executes line 5 get the *Token*[i]*.color* of c or (II) after process P_i reads the *GlobalColor* at line 5 of c, it is flipped to $\bar{c}$ and then process P_j reads it at line 5 and gets the *Token*[j]*.color* of $\bar{c}$. In order for process P_i enter the CS, it must find out the *GlobalColor* is $\bar{c}$ when it checks line 21. Therefore, by Lemma 9, only case II can be true. On the other hand, when process P_j executes line 21, it will find that none of the conditions are satisfied because the *GlobalColor* cannot be flipped to c again by Lemma 9. Therefore, P_j will wait for P_i until it finishes the exit section, contradicting the assumption that P_j and P_i stay in the CS at the same time.

We have proved that in all cases we get a contradiction with our assumption, and hence, the algorithm satisfies the Mutual Exclusion property. □

Theorem 4. *The BWBGME algorithm satisfies the Bounded Exit property.*

Proof. Since the exit section does not contain any busy–wait loop, a process that enters the exit section will finish it within a bounded number of its own steps. □

Theorem 5. *The BWBGME algorithm satisfies the Concurrent Entry property.*

Proof. If a process P_i requests a session, and no other process requests a different session, for every other process P_j, it holds that *Token*[j]*.session* $\in \{0, mysession\}$ from the view of P_i. Since process P_i always checks *Token*[j]*.session* $\in \{0, mysession\}$ in all busy–wait lines (lines 17,19 and 21), P_i will not wait at any line when there is no conflict and so, it enters the CS within a bounded number of its own steps. □

Theorem 6. *The BWBGME algorithm satisfies the Deadlock Freedom property.*

Proof. Suppose the algorithm does not satisfy the deadlock freedom property. There is an execution of the algorithm in which a nonempty set S of processes enter the entry section but none of them enters the CS, and no process enters the CS infinitely often. As every process in the entry section will finish the doorway eventually and set the *Choosing* to be false, processes in set S cannot wait on line 17 forever. Let S_1 be the subset of S that consists of processes having the *Token.color* of c, and S_2 be the subset of S that consists of processes having the *Token.color* of $\bar{c}$. Since no process enters the CS infinitely often, the *GlobalColor* cannot be changed infinitely often. Assume the *GlobalColor* remains as c after some time.

We observe that processes in S_2 cannot wait on line 21 forever, because any process in S_2 will find that their *Token.color* is different from the *GlobalColor* and stop waiting. So, processes in S_2 can only wait on line 19. Also, we observe that processes in S_2 cannot wait for any process in S_1 on line 19, because processes in S_2 will find that they have different *Token.color* with processes in S_1 at line 19, which will make processes in S_2 immediately terminate waiting. Thus, a process in S_2 must wait for another process in S_2 at line 19. In the set S_2, there must be a process that has the smallest *Token.number* and that process will pass all other processes in S_2 at line 19 and enter the CS. This situation contradicts with our assumption that processes in S cannot enter the CS. So $S_2 = \emptyset$.

On the other hand, a process in S_1 cannot wait for any other processes in S_1 at line 21 because it will find they have the same *Token.color* and stop waiting. Therefore, every process in S_1 must wait for another process in S_1 at line 19. In the set S_1, there must exist a process that has the smallest *Token.number* among all processes in S_1 and that process will pass all other processes in S_1 and enter the CS, which is a contradiction. So $S_1 = \emptyset$.

Hence, we have $S = S_1 \cup S_2 = \emptyset$, which contradicts with our assumption and the theorem is proved. □

Theorem 7. *The BWBGME algorithm satisfies the Starvation Freedom property.*

Proof. Lamport has proved that [13] if an algorithm satisfies the deadlock freedom property and the FCFS property, then it necessarily satisfies the starvation freedom property. Hence, using Theorem 2 and Theorem 6, we infer that the algorithm has the starvation freedom property. □

Theorem 8. *The shared variables used in the BWBGME algorithm are bounded.*

Proof. Obviously, every shared variable used in the algorithm is bounded except possibly *Token.number* and *Token.session*. However, if at all the *Token.session* variable is unbounded, it is due to the nature of the application and not due to the design of the algorithm. In other words, if the underlying application uses only a bounded number of sessions, so will our algorithm. Hence, we need to concern ourselves with only *Token.number* variable. Here we show that the value of *Token.number* cannot be larger than $N+1$.

If not, there must exist an execution sequence

$$\begin{aligned} &P_{i_1}@5 \rightarrow P_{i_1}@13 \rightarrow P_{i_2}@13 \rightarrow P_{i_3}@13 \rightarrow \dots \\ &\qquad\qquad\qquad \rightarrow P_{i_j}@13 \rightarrow \dots \rightarrow P_{i_{N+2}}@13 \end{aligned} \tag{1}$$

such that all processes in the sequence have the same *Token.color* of c, and for all j except 1, P_{i_j} gets its *Token*$[i_j]$*.number* at line 13 by incrementing the previous largest number *Token*$[i_{j-1}]$*.number*. P_{i_1} gets its *Token.number* of 1 by virtue of it not finding any conflicting process with the same *Token.color* when it executes lines 7–12. It is easy to see that for any j, P_{i_j} and $P_{i_{j+1}}$ have different session numbers as $P_{i_{j+1}}$ increments P_{i_j}'s *Token.number*. Here we prove the theorem by showing that such a sequence doesn't exist.

We first show that every process in the sequence (1) executes line 5 in the same time interval I.

If not, there exist two processes in the sequence, say P_{i_k} and $P_{i_{k+1}}$ that execute line 5 in different intervals. As they have the same *Token.color*, after one of them executes line 5, the *GlobalColor* is flipped twice before the other process executes line 5. By Lemma 9, the previous process already resets its *Token* before the *GlobalColor* is flipped twice, and therefore, $P_{i_{k+1}}$ will fail to read P_{i_k}'s *Token.number* and increment the number, which contradicts our assumption that every process in the sequence (1) increments the previous largest *Token.number*. So, every process in the sequence (1) executes line 5 in the same interval I.

Since we only have N processes, by the pigeonhole principle, at least two processes, say P_{i_x} and P_{i_y} (not necessarily distinct processes, but with $x < y$), from the above sequence $(P_{i_1}, P_{i_2}, .., P_{i_{N+2}})$ must exit their CS before $P_{i_{N+2}}$ executes line number 3. So, we have

$$P_{i_x}@34 \rightarrow P_{i_y}@34 \rightarrow P_{i_{N+2}}@3 \tag{2}$$

It is easy to see that every two adjacent processes in sequence (1) have different sessions, hence, for any j, $P_{i_{j+1}}$ cannot enter the CS before P_{i_j} completely exits and resets its *Token*$[i_j]$. Consequently, we can conclude that P_{i_1} and P_{i_2} must have exited by the time P_{i_y} exits (as P_{i_1} and P_{i_2} are the first two processes in the sequence 1, P_{i_x} and P_{i_y} are some processes in the sequence (1) and the processes in the sequence (1) exit one after the other). Thus, before $P_{i_{N+2}}$ begins to execute the doorway, P_{i_1} and P_{i_2} must already have finished their exit section, which is represented by the following sequence.

$$P_{i_1}@34 \rightarrow P_{i_2}@34 \rightarrow P_{i_{N+2}}@3 \tag{3}$$

By our previous claim, P_{i_1}, $P_{i_2} \dots P_{i_{N+2}}$ execute line 5 in the same interval and so they all get the same color, say c. In order for P_{i_1} to enter the CS, all processes conflicting with P_{i_1} in the system, having *Token.color* of $\bar{c}$ must have already finished the exit section and reset their *Token*. In order for P_{i_2} to enter the CS, all processes conflicting with P_{i_2} in the system, having *Token.color* of $\bar{c}$ must have already finished the exit section and reset their *Token*. Every process that has the *Token.color* of $\bar{c}$ must be in conflict with either P_{i_1} or P_{i_2}, as these two processes have different session numbers. Hence, after P_{i_2} finishes the CS and checks the condition at line 26, it will find that there is no process with the opposite *Token.color* of $\bar{c}$ in the system. So, P_{i_2} will flip the *GlobalColor* to $\bar{c}$, which immediately starts a new time interval I'. This contradicts our previous claim that every process in the sequence (1) executes line 5 in the same interval I (as $P_{i_{N+2}}$ in particular will execute line 5 in a different interval).

So, we have shown that such a sequence of processes cannot exist and so proved that the value of the shared variable *Token.number* is bounded by $N+1$. □

Theorem 9. *The BWBGME Algorithm has $O(N)$ shared space complexity and $O(N)$ RMR Complexity under the CC model.*

Proof. It is trivial to observe that algorithm is of $O(N)$ shared space complexity. Therefore, we focus on proving the RMR complexity here. From line 17 through line 22, there are three busy–wait loops, one each at lines 17, 19 and 21. Since line 19 and line 21 are located in different branches, it is not difficult to see that a process that goes through lines 17–22 executes only two busy–wait loops viz., line 17 and line 19 or line 17 and line 21.

In line 17, when a process P_i is busy waiting for a process P_j, if *Choosing*$[j]$ changes to false, then process P_i will immediately terminate the wait. It is possible that process P_j executes a new invocation and resets the *Choosing*$[j]$ to true again with another conflicting session. However, in that case, since process P_i doorway proceeds P_j, according to the FCFS property, P_j cannot reenter the CS before P_i. So, this can occur at most once. Thus, line 17 can only involve at most five RMR (three for *Choosing*$[j]$ and two for *Token*$[j]$).

Suppose process P_i in the entry section executes line 19 and waits for a process P_j. If process P_j changes its *Token*$[j]$, it will be either resetting its *Token*, or getting a new *Token* in a new invocation. If process P_j resets its *Token*$[j]$, P_i will immediately terminate the wait because it will detect *Token*$[j]$.*session* is 0. However, it is possible that P_j wants to reenter the CS with a new *Token* before P_i detects that P_j is in the remainder section. In view of the FCFS property, process P_j cannot reenter the CS before process P_i enters the CS. So, sooner or later process P_i will be able to compare its *Token*$[i]$ with that of process P_j. Therefore, the maximum number of RMR at line 12 is two (for *Token*$[j]$).

Suppose process P_i in the entry section executes line 21 and waits for another process P_j. Without loss of generality, assume P_i gets the *Token.color* of c. If the *GlobalColor* is set to $\bar{c}$, then process P_i will find out *GlobalColor* $\neq$ *mycolor* and stop the wait. Note that *GlobalColor* cannot be changed twice to c again when process P_i is waiting at line 21 in view of Lemma 9. Hence, process P_i will be able to detect the fact that the *GlobalColor* has changed sooner or later.

It is important to realize that if the variable *GlobalColor* is overwritten, its cached copies will immediately be invalidated, even if the actual value of the *GlobalColor* does not really change. This could potentially cause multiple RMR when a process P_i is waiting in line 21. Suppose that the *GlobalColor* is set again to c by another process P_k with the *Token*$[k]$.*color* of $\bar{c}$ executing the exit section. Now, this situation will cause P_i to migrate the *GlobalColor* to cache again, but P_i will continue to be blocked as the value of *GlobalColor* has not been really changed. In that case, by Lemma 9, P_k cannot start a new invocation and set the *GlobalColor* to c again before P_i enters the CS. This is because, if P_k gets its *Token.color* of $\bar{c}$ in the new invocation, the *GlobalColor* is actually flipped to $\bar{c}$ after P_i executes line 5. When P_k finishes the CS in the new invocation, it will find out P_i has the opposite *Token*$[i]$.*color* of c and fail to set the *GlobalColor*. So, P_i will see that the *GlobalColor* is updated, but still get blocked at line 21 for at most $N-1$ times in the whole loop lines 16–23. Hence, the amortized RMR complexity of *GlobalColor* for each process P_j is constant.

For other conditions at line 21, if process P_j resets the *Token*$[j]$, P_i will be able to pass process P_j. It is possible that P_j may want to reenter the CS before P_i detects that P_j is in the remainder section. In the new invocation, process P_j may execute line 3 and write *Token*$[j]$ of $(mysession, \perp, 0)$ and block P_i again. However, by the FCFS property, process P_j cannot reenter the CS before process P_i and so P_i will be able to detect the new *Token*$[j]$ or that the *GlobalColor* is flipped when process P_j finishes the doorway, which would make process P_i stop waiting. Hence, the overall RMR made at line 21 is constant.

Since the RMR complexity of lines 17, 19 and 21 is constant and lines 17–22 are enclosed within a *for loop* that can run a maximum of N times, it follows that the waiting room (lines 16–23) is of $O(N)$ RMR complexity.

It is trivial to see that lines 8–11 costs constant RMR and so, lines 7–12 is of $O(N)$ RMR complexity. It is not difficult to see that the function used in the algorithm is of $O(N)$ RMR complexity. It is also seen that other lines involve constant RMR. Hence, the overall RMR complexity of this algorithm is $O(N)$ in the CC model. □

We summarize the results by stating the properties of our algorithm in the following theorem.

Theorem 10. *The BWBGME Algorithm presented in Fig. 5 solves the GME problem by satisfying all the five properties P1 through P5 in linear space (with bounded registers) and time (RMR under the CC model) using only simple read and write operations.*

8. Conclusions and open problems

We presented two algorithms for solving the GME Problem. Both our algorithms satisfy all the five properties P1 through P5, use only simple read and write operations, run in linear time and linear space. Our first algorithm made use of unbounded shared registers whereas the second algorithm used only bounded shared registers. There is no algorithm in the literature that achieves the combination of both linear time and space complexity while satisfying all five properties in the model in which we are working. As a side contribution, we clarified a flaw in the literature.

Even though the token numbers in our BWBGME algorithm are bounded, the bound is $N+1$ which is not a constant. We leave the development of a linear time and linear space GME algorithm satisfying all five properties, which uses only simple read and write instructions and whose shared registers are bounded by a constant, as an open problem.

Although the five properties mentioned in the problem statement given in the section 1 are the most important properties, there is no end to the wish list of desirable properties. We now discuss some other esoteric properties considered in the literature.

The FCFS property captures fairness across processes requesting different sessions, but not across processes requesting same sessions. This fairness property is captured by the *first-in-first-enabled* (FIFE) property, described below. The FIFE property was first introduced by Fischer et al. [5] in the context of l-exclusion problem.

P7 FIFE: If process P_i doorway precedes process P_j and the two processes request the same sessions, and P_j enters the CS before process P_i, then P_i enters the CS within a bounded number of its own steps.

Another interesting property is the so-called *Strong Concurrent Entry* (SCE) property. It was first stated in [8] and is a strengthening of the basic concurrent entry property. This property is desirable as late processes (even if they are conflicting) cannot prevent a process from entering the CS within a bounded number of its own steps.

P8 SCE: If a process P_i has completed its doorway, and P_i doorway precedes every active process that requests a different session from that of P_i, then P_i enters the CS within a bounded number of its own steps.

Our algorithms do not satisfy the FIFE property or the strong concurrent entry property. It would be nice to develop algorithms that satisfy the FIFE property and SCE property in addition to having all the desirable properties of our BWBGME algorithm stated in Theorem 10. We leave that as an open problem. In [8], a couple of algorithms that additionally satisfy the FIFE property and SCE property are developed by using the notion of *abortable mutual exclusion*. In [3] an algorithm satisfying FIFE property in addition to other properties is developed. However, none of them runs in bounded linear space. Moreover, these algorithms use the "co-begin co-end construct" with the additional assumption that if one co-routine terminates naturally, all the other coroutines are aborted, thus deviating from the simple model of a process that we are working with in this paper.

Acknowledgements

The authors would like to thank an anonymous referee who found a subtle error in an earlier version of our Black and White Bakery GME algorithm. The authors would also like to thank Alex Aravind for pointing out a typo in the Black and White Bakery GME Algorithm that was published in the conference version [7].

References

[1] V. Bhatt, C-C. Huang, Group mutual exclusion in $O(\log n)$ RMR, in: Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC'10, 2010, pp. 45–54.
[2] James E. Burns, Complexity of Communication Among Asynchronous Parallel Processes, Ph.D. thesis, Georgia Institute of Technology, 1981.
[3] R. Danek, V. Hadzilacos, Local-spin group mutual exclusion algorithms, in: Proceedings of the 18th International Symposium on Distributed Computing, DISC'04, in: Lecture Notes in Comput. Sci., vol. 3274, Springer-Verlag, 2004, pp. 71–85.
[4] E. Dijksra, Solution of a problem in concurrent programming control, Commun. ACM 8 (9) (1965) 569.
[5] M. Fischer, N. Lynch, J. Burns, A. Borodin, Resource allocation with immunity to limited process failure, in: Proceedings of the 20th IEEE Annual Symposium on Foundations of Computer Science, FOCS'79, 1979, pp. 234–254.
[6] V. Hadzilacos, A note on group mutual exclusion, in: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, PODC'01, 2001, pp. 100–106.
[7] Yuan He, K. Gopalakrishnan, E. Gafni, Group mutual exclusion in linear time and space, in: Proceedings of the 17th International Conference on Distributed Computing and Networking, ICDCN'16, ACM, New York, NY, USA, ISBN 978-1-4503-4032-8, 2016, Article No. 22.
[8] P. Jayanti, S. Petrovic, K. Tan, Fair group mutual exclusion, in: Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing, PODC'03, 2003, pp. 278–284.
[9] P. Jayanti, K. Tan, G. Friedland, A. Katz, Bounding Lamport's Bakery Algorithm, in: Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'01, in: Lecture Notes in Comput. Sci., vol. 2234, Springer-Verlag, 2001, pp. 261–270.
[10] Y. Joung, Asynchronous group mutual exclusion, Distrib. Comput. 13 (4) (2000) 189–206.
[11] Patrick Keane, Mark Moir, A simple local-spin group mutual exclusion algorithm, in: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, PODC'99, 1999, pp. 23–32.
[12] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Commun. ACM 17 (8) (1974) 453–455.
[13] L. Lamport, The mutual exclusion problem: parts I and II, J. ACM 33 (2) (1986) 313–348.
[14] M. Takamura, Y. Igarashi, Group mutual exclusion algorithms based on ticket orders, in: Proceedings of the 9th Annual International Computing and Combinatorics Conference, COCOON'03, in: Lecture Notes in Comput. Sci., vol. 2697, Springer-Verlag, 2003, pp. 232–241.
[15] G. Taubenfeld, The Black–White Bakery Algorithm, in: Proceedings of the 18th International Symposium on Distributed Computing, DISC'04, in: Lecture Notes in Comput. Sci., vol. 3274, Springer-Verlag, 2004, pp. 56–70.
[16] J.H. Yang, J. Anderson, A fast, scalable mutual exclusion algorithm, Distrib. Comput. 9 (1) (1995) 51–60.