Constant RMR Group Mutual Exclusion for Arbitrarily Many Processes and Sessions

Liat Maor
The Interdisciplinary Center, Herzliya, Israel

Gadi Taubenfeld
The Interdisciplinary Center, Herzliya, Israel

Abstract

Group mutual exclusion (GME), introduced by Joung in 1998, is a natural synchronization problem that generalizes the classical mutual exclusion and readers and writers problems. In GME a process requests a session before entering its critical section; processes are allowed to be in their critical sections simultaneously provided they have requested the same session.

We present a GME algorithm that (1) is the first to achieve a constant Remote Memory Reference (RMR) complexity for both cache coherent and distributed shared memory machines; and (2) is the first that can be accessed by arbitrarily many dynamically allocated processes and with arbitrarily many session names. Neither of the existing GME algorithms satisfies either of these two important properties. In addition, our algorithm has constant space complexity per process and satisfies the two strong fairness properties, first-come-first-served and first-in-first-enabled. Our algorithm uses an atomic instruction set supported by most modern processor architectures, namely: read, write, fetch-and-store and compare-and-swap.

2012 ACM Subject Classification Theory of computation; Theory of computation → Distributed computing models; Theory of computation → Shared memory algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Group mutual exclusion, RMR complexity, unbounded number of processes, fetch&store (FAS), compare&swap (CAS)

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.30

Acknowledgements We thank the anonymous referees for their constructive suggestions.

1 Introduction

1.1 Motivation and results

In the *group mutual exclusion* (GME) problem n processes repeatedly attend m sessions. Processes that have requested to attend the same session may do it concurrently. However, processes that have requested to attend different sessions may not attend their sessions simultaneously. The GME problem is a natural generalization of the classical mutual exclusion (ME) and readers/writers problems [9, 12]. To see this, observe that given a GME algorithm, ME can be solved by having each process uses its unique identifier as a session number. Readers/writers can be solved by having each writer requests a different session, and having all readers request the same special session. This allows readers to attend the session concurrently while ensuring that each writer attends in isolation. The GME problem has been studied extensively since it was introduced by Yuh-Jzer Joung in 1998 [21, 22].

A simple example has to do with the design of a concurrent queue or stack [6]. Using a GME algorithm, we can guarantee that no two users will ever simultaneously be in the *enqueue.session* or *dequeue.session*, so the enqueue and dequeue operations will never be interleaved. However, it will allow any number of users to be in either the enqueue or

dequeue session simultaneously. Doing so simplifies the design of a concurrent queue as our only concern now is to implement concurrent enqueue operations and concurrent dequeue operations.

In this paper, we present a GME algorithm that is the first to satisfy several desired properties (the first two properties are satisfied only by our algorithm).

1. *Suitability for dynamic systems*: All the existing GME algorithms are designed with the assumption that either the number of processes or the number of sessions is a priori known. Our algorithm is the first that does not make such an assumption:
   - it can be accessed by an arbitrary number of processes; that is, processes may appear or disappear intermittently, and
   - the number and names of the sessions are not limited in any way.
2. $O(1)$ *RMR complexity*: An operation that a process performs on a memory location is considered a remote memory reference (RMR) if the process cannot perform the operation locally on its cache or memory and must transact over the multiprocessor's interconnection network in order to complete the operation. RMRs are undesirable because they take long to execute and increase the interconnection traffic. Our algorithm
   - achieves the ideal RMR complexity of $O(1)$ for Cache Coherent (CC) machines; and
   - is the first to achieve the ideal RMR complexity of $O(1)$ for Distributed Shared Memory (DSM) machines. (In Subsection 1.3, we explain why this result does not contradict the lower bound from [11].)

   This means that a process incurs only a constant number of RMRs to satisfy a request (i.e, to enter and exit the critical section once), regardless of how many other processes execute the algorithm concurrently.
3. $O(1)$ *space per process*: A small constant number of memory locations are allocated for each process. On DSM machines, these memory locations reside in the process local memory; on CC machines, these locations reside in the shared memory.
4. *Strong fairness*: Requests are satisfied in the order of their arrival. That is, our algorithm satisfies the *first-come-first-served* and *first-in-first-enabled* properties, defined later.
5. *Hardware support*: Atomic instruction set that is supported by most modern processor architectures is used, namely: read, write, fetch-and-store and compare-and-swap.

We point out that when using a GME as a ME algorithm, the number of processes is the same as the number of sessions (each process uses its identifier as its session number). Thus, in GME algorithms, in which the number of sessions is a priori known also the number of processes must be known, at least when these GME algorithms are used as ME algorithms or readers and writers locks.

Our GME algorithm is inspired by J. M. Mellor-Crummey and M. L. Scott MCS queue-based ME algorithm [28]. The idea of our GME algorithm is to employ a queue, where processes insert their requests for attending a session. The condition when a process p may attend its session depends on whether p's session is the same as that of all its predecessors. Otherwise, p waits until p is notified (by one of its predecessors) that all its predecessors which have requested different sessions completed attending their sessions.

A drawback of the MCS ME algorithm is that releasing a lock requires spinning – a process p releasing the lock may need to wait for a process that is trying to acquire the lock (and hence is behind p in the queue) to take a step before p can proceed. The ME algorithms in [13] overcome this drawback while preserving the simplicity, elegance, and properties of the MCS algorithm. We use a key idea inspired by [13] in our GME algorithm to ensure that a process releasing the GME lock will never have to wait for a process that has not attended its session yet.

Another key idea of our algorithm is to count down completed requests for attending a session by moving a pointer by one node (in the queue) for each such request and to ensure the integrity of this scheme by gating the processes that have completed attending a session (and are now trying to move the pointer) through a mutual exclusion lock.

1.2 The GME problem

More formally, the GME problem is defined as follows: it is assumed that each process executes a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section (CS)*, and *exit*.

A process starts by executing its remainder section. At some point, it might need to attend some session, say s. To attend session s, a process has to go through an entry code that guarantees that while it is attending this session, no other process is allowed to attend another session. In addition, once a process completes attending a session, the process executes its exit section in which it notifies other processes that it is no longer attending the session. After executing its exit section, the process returns to its remainder.

The group mutual exclusion problem is to write the code for the *entry section* and the *exit section* so that the following requirements are satisfied.

- *Mutual exclusion*: Two processes can be in their CS at the same time, only if they request the same session.
- *Starvation-freedom*: If a process is trying to enter its CS, then this process must eventually enter its CS.
- *Group concurrent entering* (*GCE*): If a process p requests a session s while no process is requesting a conflicting session, then (1) some process with session s can complete its entry section within a bounded number of its own steps, and (2) p eventually completes its entry section, even if other processes do not leave their CS.
- *Group bounded exit* (*GBE*): If a process p is in its exit section, then (1) some process can complete its exit section within a bounded number of its own steps, and (2) p eventually completes its exit section.

GCE precludes using a given mutual exclusion algorithm as a solution for the GME problem since GCE enables processes to attend the same session concurrently.

Our algorithm also satisfies the following strong fairness requirements. To formalize this, we assume that the entry code starts with a bounded section of code (i.e., one that contains no unbounded loops), called the *doorway*; the rest of the entry code is called the waiting room. The fairness requirements, satisfied by our algorithm, can now be stated as follows:

- *First-come-first-served* (*FCFS*): If a process p completes its doorway before a process q enters its doorway and the two processes request different sessions, then q does not enter its CS before p enters its CS [16, 26].
- *First-in-first-enabled* (*FIFE*): If a process p completes its doorway before a process q enters its doorway, the two processes request the same session, and q enters its CS before p, then p enters its CS in a bounded number of its own steps [20].

We notice that FCFS and FIFE do not imply starvation-freedom or group concurrent entering.

1.3 Further explanations

To illustrate the various GME requirements, imagine the critical section as a lecture hall that different professors can share for their lectures. Furthermore, assume that the lecture hall has one entrance door and one exit door. When solving the GME problem, the property of mutual exclusion guarantees that two different lectures cannot be arranged in the lecture hall simultaneously, while starvation-freedom guarantees that the lecture hall will eventually be reserved for every scheduled lecture.

Assuming that only one lecture is scheduled, group concurrent entering ensures that all the students who want to attend this lecture can enter the lecture hall through the entrance door, possibly one after the other, and attend the lecture. Furthermore, at any given time, when there are students who want to attend the lecture, at least one of them can always enter the lecture hall without any delay. Similarly, group bounded exit ensures that all the students who want to leave a lecture can do so through the exit door, possibly one after the other. Furthermore, at any given time, at least one of them can exit the lecture hall without delay.

Group concurrent entering and group bounded exit are first introduced and formally defined in this paper. They are slightly weakened versions of two known requirements (formally defined below) called concurrent entering and bounded exit. Using the lecture hall metaphor, assuming that only one lecture is scheduled, concurrent entering ensures that all the students who want to attend this lecture can enter the lecture hall *together*. Similarly, bounded exit ensures that all the students who want to leave a lecture can do so *together*. So, why have we not used these two stronger requirements?

Danek and Hadzilacos lower bound. Let n denotes the total number of processes. In [11], it is proven that $\Omega(n)$ RMRs are required for any GME algorithm that satisfies mutual exclusion, starvation-freedom, concurrent entering, and bounded exit, in the DSM model, using basic primitives of any strength. This result holds even when the number of sessions is only two. (Concurrent entering and bounded exit are as defined below [16].) Since we are aiming at finding a solution that has $O(1)$ RMR complexity, we had to weaken either concurrent entering, bounded exit, or both. (GME would not be interesting if the mutual exclusion or starvation-freedom properties are weakened.)

Group concurrent entering. To avoid an inefficient solution to the GME problem using a traditional ME algorithm and forcing processes to be in their CS one-at-a-time, even if all processes are requesting the same session, Joung required that a GME algorithm satisfies the following property (which he called concurrent entering):

- If some processes request a session and no process requests a different session, then the processes can concurrently enter the CS [21].

The phrase "can concurrently enter," although suggestive, is not precise. In [24, 25], Keane and Moir were the first to give a precise definition that captures their interpretation of Joung's requirement (which they also called concurrent entering):

- *Concurrent occupancy*: If a process p requests a session and no process requests a different session, then p eventually enters its CS, even if other processes do not leave their CS. (The name "concurrent occupancy" is from [16].)

In [16], Hadzilacos gave the following interpretation, which is stronger than that of Keane and Moir.

- *Concurrent entering*: If some process, say p, is trying to attend a session s while no process is requesting a conflicting session, then p completes its entry section in a bounded number of its own steps.

To circumvent the Danek and Hadzilacos $\Omega(n)$ lower bound, we looked for a slightly weaker version of concurrent entering that would still capture the property that Joung intended to specify. We believe that group concurrent entering, which is strictly stronger than concurrent occupancy, is such a property. We point out that our algorithm actually satisfies the following stronger version of group concurrent entering,

- *Strong group concurrent entering*: If a process p requests a session s, and p completes its doorway before any conflicting process starts its doorway, then (1) some process with session s can complete its entry section within a bounded number of its own steps, and (2) p eventually completes its entry section, even if other processes do not leave their CS.

Strong group concurrent entering (SGCE) is a slightly weakened version of a known property called strong concurrent entering [20].

Group bounded exit. Our group bounded exit property is replaced by the following two (weaker and stronger) properties in previously published papers.

- *Terminating exit*: If a process p enters its exit section, then p eventually completes it [24].
- *Bounded exit*: If a process p enters its exit section, then p eventually completes it within a bounded number of its own steps [16].

Again, to circumvent the Danek and Hadzilacos $\Omega(n)$ lower bound, we have defined group bounded exit, which is slightly weaker than bounded exit and is strictly stronger than terminating exit.

Open question. We have modified both concurrent entering and bounded exit. Is this necessary? With minor modifications to the Danek and Hadzilacos lower bound proof, it is possible to prove that their lower bound still holds when replacing only bounded exit with group bounded exit. Thus, to circumvent the lower bound, the weakening of concurrent entering is necessary. However, the question of whether it is possible to circumvent the lower bound by replacing only concurrent entering with group concurrent entering, and leaving bounded exit as is, is open.

1.4 Related work

Table 1 summarizes some of the (more relevant) GME algorithms mentioned below and their properties. The group mutual exclusion problem was first stated and solved by Yuh-Jzer Joung in [21, 22], using atomic read/write registers. The problem is a generalization of the mutual exclusion problem [12] and the readers and writers problem [9] and can be seen as a special case of the drinking philosophers problem [8].

Group mutual exclusion is similar to the room synchronization problem [6]. The room synchronization problem involves supporting a set of m mutually exclusive "rooms" where any number of users can execute code simultaneously in any one of the rooms, but no two users can simultaneously execute code in separate rooms. In [6], room synchronization is defined using a set of properties that is different than that in [21], a solution is presented, and it is shown how it can be used to efficiently implement concurrent queues and stacks.

In [24, 25], a technique of converting any solution for the mutual exclusion problem to solve the group mutual exclusion problem was introduced. The algorithms from [24, 25] do not satisfy group concurrent entering and group bounded exit and have $O(n)$ RMR complexity, where n is the number of processes. (By a mistake, in some of the tables in [24, 25], smaller RMR complexity measures are mentioned.) In [16], a simple formulation of concurrent entering is proposed which is stronger than the one from [24], and an algorithm is presented that satisfies this property.

In [20], the first FCFS GME algorithm is presented that uses only $O(n)$ bounded shared registers, while satisfying concurrent entering and bounded exit. Also, it is demonstrated that the FCFS property does not fully capture the intuitive notion of fairness, and additional fairness property, called first-in-first-enabled (FIFE) was presented. Finally, the authors presented a reduction that transforms any *abortable* FCFS mutual exclusion algorithm, into a GME algorithm, and used it to obtained GME algorithm satisfying both FCFS and FIFE.

A GME algorithm is presented in [11] with $O(n)$ RMR complexity in the DSM model, and it is proved that this is asymptotically optimal. Another algorithm in [11] requires only $O(\log n)$ RMR complexity in the CC model, but can be used just for two sessions.

Our algorithm satisfies FCFS fairness. That is, if the requests in the queue are for sessions 1, 2, 1, 2, 1, 2 and so on, those requests would be granted in that order. Yet, for practical considerations, one may want to batch all requests for session 1 (and, separately, for session 2) and run them concurrently. Our algorithm does not support "batching" of pending requests for the same session, as FCFS fairness and "batching" of pending requests for the same session are contradicting (incompatible) requirements. This idea was explored in [5], where a GME algorithm is presented that satisfies two "batching" requirements call pulling and relaxed-FCFS, and requiring only $O(\log n)$ RMR complexity in the CC model. Reader-Writer Locks were studied in [7], which trade fairness between readers and writers for higher concurrency among readers and better back-to-back batching of writers.

An algorithm is presented in [14] in which a process can enter its critical section within a constant number of its own steps in the absence of any other requests (which is typically referred to as contention-free step complexity). In the presence of contention, the RMR complexity of the algorithm is $O(min(k, n))$, where k denotes the interval contention. The algorithm requires $O(n^2)$ space and does not satisfy fairness property like FCFS or FIFE.

In [2], a GME algorithm with a constant RMR complexity in the CC model is presented. This algorithm does not satisfy group concurrent entering (or even concurrent occupancy) and FCFS. However, it satisfies two other interesting properties (defined by the authors) called simultaneous acceptance and forum-FCFS.

In [17], the first GME algorithm with both linear RMR complexity (in the CC model) and linear space was presented, which satisfies concurrent entering and bounded exit, and uses only read/write registers. A combined problem of ℓ-exclusion and group mutual exclusion, called the group ℓ-exclusion problem, is considered in [29, 32].

Besides the algorithms mentioned above, for the shared-memory model, there are algorithms that solve the GME problem under the message-passing model. Several types of the network's structure were considered, for example, tree networks [4], ring networks [33], and fully connected networks [3]. In [3, 23, 31], quorum-based message-passing algorithms are suggested in which a process that is interested in entering its CS has to ask permission from a pre-defined quorum.

2 Preliminaries

2.1 Computational model

Our model of computation consists of an asynchronous collection of n deterministic processes that communicate via shared registers (i.e., shared memory locations). Asynchrony means that there is no assumption on the relative speeds of the processes. Access to a register is done by applying operations to the register. Each operation is defined as a function that gets as arguments one or more values and registers names (shared and local), updates the value of the registers, and may return a value. Only one of the arguments may be a name of a *shared* register. The execution of the function is assumed to be atomic. Call by reference is used when passing registers as arguments. The operations used by our algorithm are:

- *Read:* takes a shared register r and simply returns its value.
- *Write:* takes a shared register r and a value *val*. The value *val* is assigned to r.
- *Fetch-and-store* (FAS): takes a shared register r and a local register ℓ, and atomically assigns the value of ℓ to r and returns the previous value of r. (The fetch-and-store operation is also called *swap* in the literature.)

Table 1 Comparing the properties of our algorithm with those of several GME algorithms.

GME Algorithms	Group bounded exit BE/GBE	Group concurrent entering CE/GCE	Fairness FCFS/ FIFE	Unknown number of processes & sessions	Shared space for all processes	RMR in CC	RMR in DSM	Hardware used
Joung 1988	BE	CE	✗	✗	$O(n)$	∞	∞	read/write
Keane & Moir 1999	✗	✗	✗	✗	$O(n)$	$O(n)$	$O(n)$	read/write
Hadzilacos 2001	BE	CE	FCFS	✗	$O(n^2)$	$O(n^2)$	∞	read/write
Jayanti et.al. 2003	BE	CE	FCFS FIFE	✗	$O(n)$	$O(n^2)$	∞	read/write
Danek&Hadzilacos 2004	BE	CE	FCFS FIFE	✗	$O(n^2)$	$O(n)$	$O(n)$	CAS fetch&add
Bhatt & Huang 2010	BE	CE	✗	✗	$O(mn)$	$O(min(k, \log n))$	∞	LL/SC
He et. al. 2018	BE	CE	FCFS	✗	$O(n)$	$O(n)$	∞	read/write
Aravid&Hesselink 2019	BE	✗	FIFE	✗	$O(L)$	$O(1)$	∞	fetch&inc
Gokhale & Mittal 2019	BE	CE	✗	✗	$O(n^2)$	$O(min(c, n))$	$O(n)$	CAS fetch&add
Our algorithm	GBE	GCE	FCFS FIFE	✓	$O(n)$	$O(1)$	$O(1)$	CAS fetch&store

✓ - satisfies the property
✗ - does not satisfy the property
n - number of processes
m - number of sessions
k - point contention
c - interval contention
L - a constant number s.t. $L > min(n, m)$
BE - bounded exit
GBE - group bounded exit
CE - concurrent entring
GCE - group concurrent entring

- *Compare-and-swap* (CAS): takes a shared register r, and two values: *new* and *old*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise, r is left unchanged and the value *false* is returned.

Most modern processor architectures support the above operations.

2.2 The CC and DSM machine architectures

We consider two machine architecture models: (1) Cache coherent (CC) systems, where each process (or processor) has its own private cache. When a process accesses a shared memory location, a copy of it migrates to a local cache line and becomes locally accessible until some other process updates this shared memory location and the local copy is invalidated; (2) Distributed shared memory (DSM) systems, where instead of having the "shared memory" in one central location, each process "owns" part of the shared memory and keeps it in its own local memory. These different shared memory models are illustrated in Figure 1.

A shared memory location is locally accessible to some process if it is in the part of the shared memory that physically resides on that process' local memory. Spinning on a remote memory location while its value does not change, is counted only as *one* remote operation that causes communication in the CC model, while it is counted as *many* operations that cause communication in the DSM model. An algorithm satisfies *local spinning* (in the CC or DSM models) if the only type of spinning required is local spinning.

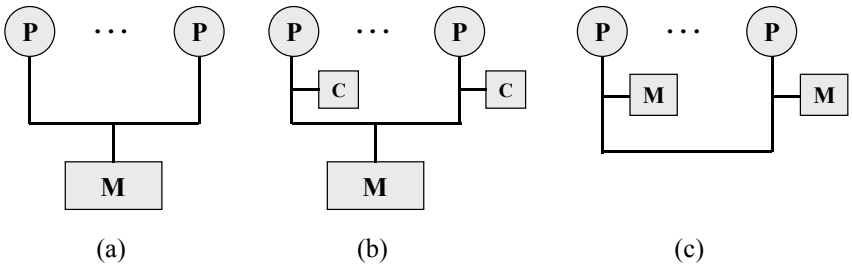

Figure 1 Shared memory models. (a) Central shared memory. (b) Cache Coherent (CC). (c) Distributed Shared Memory (DSM). P denotes processor, C denotes cache, M denotes shared memory.

2.3 RMR complexity: counting remote memory references

We define a *remote reference* by process p as an attempt to reference (access) a memory location that does not physically reside in p's local memory or cache. The remote memory location can either reside in a central shared memory or in some other process' memory.

Next, we define when remote reference causes *communication*. (1) In the DSM model, any remote reference causes communication; (2) in the CC model, a remote reference to register r causes communication if (the value of) r is not (the same as the value) in the cache. That is, communication is caused only by a remote write access that overwrites a different value or by the first remote read access by a process that detects a value written by a different process.

Finally, we define time complexity when counting only remote memory references. This complexity measure, called RMR complexity, is defined with respect to either the DSM model or the CC model, and whenever it is used, we will say explicitly which model is assumed.

- *The RMR complexity* in the CC model (resp. DSM model) is the maximum number of remote memory references which cause communication in the CC model (resp. DSM model) that a process, say p, may need to perform in its entry and exit sections in order to enter and exit its critical section since the last time p started executing the code of its entry section.

3 The GME Algorithm

Our algorithm has the following properties: (1) it has constant RMR complexity in both the CC and the DSM models, (2) it does not require to assume that the number of participating processes or the number of sessions is a priori known, (3) it uses constant space per process, (4) it satisfies FCFS and FIFE fairness, (5) it satisfies the properties: mutual exclusion, starvation-freedom, SGCE, and GBE, (6) it uses an atomic instruction set supported by most modern processor architectures (i.e., read, write, FAS and CAS).

3.1 An informal description

The algorithm maintains a queue of nodes which is implemented as a linked list with two shared objects, *Head* and *Tail*, that point to the first and the last nodes, respectively. Each node represents a request of a process to attend a specific session. A node is an object with a pointer field called *next*, a boolean field called *go*, an integer field called *session*, and two

status fields called *status* and *active*. Each process p has its own *two* nodes, called $Nodes_p[0]$ and $Nodes_p[1]$, which can be assumed to be stored in the process p's local memory in a DSM machine, and in the shared memory in a CC machine. Each time p wants to enter its CS section, p uses alternately one of its two nodes. We say that a process p is *enabled* if p can enter its CS in a bounded number of its own steps.

In its doorway, process p initializes the fields of its node as follows:

- *session* is set to the session p wants to attend, letting other processes know the session p is requesting (line 2).
- *next* is a pointer to the successor's node and is initially set to null. This field is being updated later by p's successor (line 11).
- *go* is set to *false*. Later, if p is not enabled, p would spin on its *go* bit until the value is changed to *true*. The *go* bit is the only memory location a process may spin on.
- *status* is set to *WAIT*. This field is being used to determine if a process is enabled. When a process becomes enabled, it sets this field to *ENABLED* (line 26). When process p sees that its predecessor is not enabled (line 13), p spins on its *go* bit (line 14). Otherwise, p informs its predecessor that p has seen that the predecessor is enabled (and hence p does not need help), by setting its predecessor's *status* field to NO_HELP. When a process p sees that its *status* is *ENABLED* (line 30), p tries to help its successor to become enabled and notifies the successor by setting p's own *status* to TRY_HELP.
- *active* is set to *YES*. This field is being used to determine whether p's node is active or not. A node is active if there is a process p that is currently using the node in an attempt to enter p's critical section.

At the end of its doorway, process p threads its node to the end of the queue (line 7). Afterward, p checks what its state is. The state can be one of the following:

1. its node is the first in the queue,
2. its predecessor requests the same session, or
3. its predecessor requests a different session.

In the first case, p can safely become enabled and enters its CS. In the second case, p becomes enabled only if its predecessor is enabled. In the third case, p eventually becomes enabled, once all the processes it follows completed their CSs. We observe that in the exit section, each process causes *Head* to be advanced by exactly one step. So, if p's predecessor's node is inactive, it implies that all the processes that p follows completed their CSs, and thus, p can become enabled and enters its CS.

In the last two cases, once p is enabled, p checks whether it should help its predecessor advance *Head*, by checking if p's predecessor's node is inactive. If the predecessor's node is inactive, then *Head* should point to the node after this inactive node, which is p's node. Therefore, in such a case, p advances *Head* to point to its node.

Once p is enabled to enter its CS, p notifies its successor by setting p's *status* to *ENABLED*. Next, p checks if it has a successor that requests the same session and needs help also to become enabled. If so, p tries to help its successor to become enabled. Only then p enters its CS. The processes that may enter their CS simultaneously are: the process, say p, that *Head* points to its node, and every process q that (1) requests the same session as p, and (2) no conflicting process entered its node between p's node and q's node.

Most of the exit code is wrapped by a mutual exclusion lock. This ensures that each process can cause *Head* to be advanced by a single step every time a process completes its CS. A process that completes its CS and succeeds in acquiring the ME lock tries to advance

Head. If the process succeeds in advancing *Head*, then *Head* value is either *null* or points to the next node in the queue. If *Head* is not *null*, the process changes the *go* bit to *true* in the node that *Head* points to. By doing so, the process lets the next process becoming enabled.

If p fails to advance *Head*, this means that some other process either,

1. enters the queue after p sets *Tail* to *null* (line 38),
2. enters the queue but has not notified its predecessor yet (line 11), or
3. has not entered the queue yet (line 7).

In the first case, the process, say q, in its entry section overrides *Head* to point to q's node (line 9) because q's predecessor is *null*, and so q "advances" *Head* for p. In the latter cases, q in its entry section overrides $Head$ to point to q's own node because it sees q's predecessor's node is inactive, and so q "advances" *Head* for p. Afterward, p releases the ME lock, changes the index of its current node (for the next attempt to enter p's critical section), and completes its exit section.

To guarantee that our GME algorithm satisfies group bounded exit, the mutual exclusion used in the exit section (lines 36 and 49) must satisfy three properties, (1) starvation-freedom, (2) bounded exit, and (3) a property that we call *bounded entry*. Bounded entry is defined as follows: If a process p is in its entry section, while no other process is in its critical section or exit section, some process can complete its entry section within a bounded number of its own steps.[1] While the important and highly influential MCS lock [28] does not satisfy bounded exit, there are variants of it, like the mutual exclusion algorithms from [10, 13, 19], that satisfy all the above three properties.

We will use one of the mutual exclusion algorithms from [13, 19], since (in addition to satisfying the above three properties) each one of these algorithms satisfies the following properties which match those of our GME algorithm: (1) it has constant RMR complexity in both the CC and the DSM models, (2) it does not require to assume that the number of participating processes is a priori known, (3) it uses constant space per process, (4) it satisfies FCFS, (5) it uses the same atomic instruction set as our algorithm, (6) it makes no assumptions on what and how memory is allocated (in [10] it is assumed that all allocated pointers must point to even addresses).

3.2 The algorithm

Two memory records (nodes) are allocated for each process. On DSM machines, these two records reside in the process local memory; on CC machines, these two records reside in the shared memory. In the algorithm, the following symbols are used:

& – this symbol is used to obtain an object's memory location address (and not the value in this address). For example, $\&var$ is the memory location address of variable var.

→ – this symbol is used to indicate a pointer to data of a field in a specific memory location. For example, assume var is a variable that is a struct with a field called $number$. We now define another variable $loc := \&var$ s.t. loc points to var. Using $loc \rightarrow number$ we would get the value of $var.number$.

Q – the queue in the algorithm is denoted by Q. Q is only used for explanations and does not appear in the algorithm's code.

[1] It is interesting to notice that the bounded entry property cannot be satisfied by a ME algorithm that uses only read/write atomic registers [1], [30] (page 119).

Algorithm 1 The GME algorithm: Code for process p.

Type: $QNode$: { session: int, go: bool, next: QNode*,
active: $\in$ {*YES*, *NO*, *HELP*}
status: $\in$ {*ENABLED*, *WAIT*, *TRY_HELP*, *NO_HELP*} }

Shared: $Head$: type QNode*, initially null ▷ pointer to the first node in Q
$Tail$: type QNode*, initially null ▷ pointer to the last node in Q
$Lock$: type ME lock ▷ mutual exclusion lock
$Nodes_p[0, 1]$: each of type QNode, initial value immaterial ▷ nodes local to p in DSM

Local: s: int ▷ the session of p
$node_p$: type QNode*, initial value immaterial ▷ pointer to p's currently used node
$pred_p$: type QNode*, initial value immaterial ▷ pointer to p's predecessor node
$next_p$: type QNode*, initial value immaterial ▷ pointer to p's successor node
$temp\_head_p$: type QNode*, initial value immaterial ▷ temporarily save the head
$current_p$: $\in$ {0, 1}, initial value immaterial ▷ the index for p's current node

procedure THREAD(s: int) ▷ s is the session p wants to attend
▷ *Begin Doorway*

1: $node_p$:= &$Nodes_p[current_p]$ ▷ pointer to current node for this attempt to enter p's CS
2: $node_p \rightarrow session := s$ ▷ p's current session
3: $node_p \rightarrow go$:= false ▷ may spin locally on it later
4: $node_p \rightarrow next$:= null ▷ pointer to successor
5: $node_p \rightarrow status := WAIT$ ▷ p isn't enabled
6: $node_p \rightarrow active := YES$ ▷ p's node is active

7: $pred_p$:= FAS(Tail, $node_p$) ▷ p enters its current node to Q
▷ *End Doorway*
8: **if** $pred_p$ = null **then** ▷ was Q empty before p entered?
9: Head := $node_p$ ▷ $node_p$ is the first in Q
10: **else** ▷ p has pred
11: $pred_p \rightarrow next := node_p$ ▷ notify pred
12: **if** $pred_p \rightarrow session = s$ **then** ▷ do we have the same session?
13: **if not** CAS($pred_p \rightarrow status$, *ENABLED*, *NO_HELP*) **then**
▷ should wait for help from pred?
14: **await** $node_p \rightarrow go$ = true ▷ wait until released by pred with the same session
15: **else if not** CAS($pred_p \rightarrow active$, *YES*, *HELP*) **then** ▷ should help advance Head?
16: Head := $node_p$ ▷ help advance Head
17: **end if**
18: **else** ▷ we have different sessions
19: **if** CAS($pred_p \rightarrow active$, *YES*, *HELP*) **then** ▷ pred's node is still active?
20: **await** $node_p \rightarrow go$ = true
▷ wait until release by a process with a different session
21: **else** ▷ pred's node is inactive in Q thus p is enabled
22: Head := $node_p$
23: **end if**
24: **end if**
25: **end if**

26: $node_p \rightarrow status := ENABLED$ ▷ can enter the CS
▷ *Try helping the successor*
27: $next_p := node_p \rightarrow next$ ▷ save next pointer locally
28: **if** $next_p \neq$ null **then** ▷ has successor?
29: **if** $next_p \rightarrow session = s$ **then** ▷ we have the same session
30: **if** CAS($node_p \rightarrow status$, *ENABLED*, *TRY_HELP*) **then**

```
31:              next_p → go := true                          ▷ make your successor enabled
32:          end if
33:      end if
34:  end if

35:  critical section

36:  Acquire(Lock)                                            ▷ Mutual exclusion entry section

37:  temp_head_p := Head                                      ▷ save current head locally
38:  if CAS(Tail, temp_head_p, null) then     ▷ remove node from tail if it is the only node in Q
39:      CAS(Head, temp_head_p, null)                         ▷ try removing it from the head
40:  else if temp_head_p → next ≠ null then                   ▷ head has successor
41:      temp_head_p := temp_head_p → next                    ▷ advance the temp head
42:      Head := temp_head_p                                  ▷ advance the head
43:      temp_head_p → go := true                             ▷ enable the new head
44:  else if not CAS(temp_head_p → active, YES, NO) then
                                ▷ someone in Tail but hasn't notified to its predecessor in time
45:      temp_head_p := temp_head_p → next                    ▷ advance the temp head
46:      Head := temp_head_p                                  ▷ advance the head
47:      temp_head_p → go := true                             ▷ enable the new head
48:  end if

49:  Release(Lock)                                            ▷ Mutual exclusion exit section

50:  current_p := 1 − current_p                               ▷ toggle for further use
end procedure
```

3.3 Further explanations

To better understand the algorithm, we explain below several delicate design issues which are crucial for the correctness of the algorithm.

1. *Why does each process p need two nodes $Nodes_p[0]$ and $Nodes_p[1]$?* This is done to avoid a *deadlock*. Assume each process has a single node instead of two, and consider the following execution. Suppose p is in its CS, and q completed its doorway. p resumes and executes its exit section. p completes its exit section while q is in the queue but has not notified p that q is p's successor (line 11). p leaves its *status* field as *ENABLED* and changes its *active* field to *NO* (line 44), so q should be able to enter its CS, no matter what session q requests. p starts another attempt to enter its CS, before q resumes and executes either line 13 or line 19 (depends on which session p requests). p uses its single node and sets *status* to *WAIT* and *active* to *YES* in its doorway (lines 5 and 6, respectively). Now, q continues and (by executing either line 13 or line 19) sees that p is not enabled and p's node is active, so q spins on its *go* bit. Also, p (by executing either line 13 or line 19) sees that q is not enabled and its node is active, so p also spins on its *go* bit. No process will release q, and a deadlock occurs. This problem is resolved by having each process owns two nodes.
2. *Why do we need the CAS operations at lines 13 and 30?* The CAS operations at these lines prevent a potential race condition that may violate the *mutual exclusion* property. Assume we replace the CAS operations at lines 13 and 30, as follows:

- At line 13, p checks if $pred_p \rightarrow status \neq ENABLED$. If so, p waits at line 14. Otherwise, at line **14.5**, p executes $pred_p \rightarrow status = NO\_HELP$.
- At line 30, p checks if $node_p \rightarrow status = ENABLED$. If so, at line **30.5**, p executes $node_p \rightarrow status = TRY\_HELP$ and then continues to line 31 and helps p's successor.

Suppose p is the predecessor of q, and they both request the same session s. p executes line 30, sees that p's *status* is *ENABLED*, and continues to line 30.5 but does not execute this statement yet. Then, q executes line 13, sees that p's *status* is *ENABLED*, executes line 14.5, changes p's *status* to NO_HELP and continues to q's CS. q completes its CS, executes q's exit section, and starts the algorithm again using q's second node. q requests the same session as before, s, and continues to q's CS since q's predecessor is enabled. q completes its exit code and enters the entry code again using q's first node, but now q requests a different session $s' \neq s$. Notice, q's first node is the same node that p has seen as its successor. q continues to line 20 (because it does not request the same session as its predecessor). And so, q waits until its *go* bit is set to *true*. Now, p executes line 30.5 that changes p's *status* to TRY_HELP, continues to line 31 that sets q's first node's *go* bit to *true* and enters its CS. q sees that its *go* bit is *true* and also enters its CS. Therefore, both p and q, which request different sessions, are in their CSs at the same time.

3. *Why do we need the CAS operations at lines 15, 19, and 44?* The CAS operations at these lines are used to prevent a potential race condition that may cause a *deadlock*. Assume we replace the CAS operations are at lines 15, 19, and 44, as follow:
   - At line 15, p checks if $pred_p \rightarrow active \neq YES$. If so, p sets *Head* to its node at line 16. Otherwise, at line **16.5**, p executes $pred_p \rightarrow active = HELP$.
   - At line 19, p checks if $pred_p \rightarrow active = YES$. If so, at line **19.5**, p executes $pred_p \rightarrow active = HELP$.
   - At line 44, p checks if $temp\_head_p \rightarrow active = YES$. If so, p executes lines 45-47 and advances *Head*. Otherwise, p continues to line **47.5** and executes $temp\_head_p \rightarrow active = NO$.

   Suppose p is at line 44 while its successor q is at line 15. q executes line 15 and sees its predecessor's node's *active* equals to *YES*. So q continues to line 16.5 but does not execute it yet. Now, p continues and sees that the *active* field of the first node in the queue is *YES*, so p continues to line 47.5. Then, p sets this node's *active* field to *NO*, while q sets it to *HELP*. Next, p completes its exit section and q enters its CS. Since no process advanced *Head*, *Head* still points to the same node. Assume another process, r, wants to enter its CS and requests a different session than q. r starts the algorithm and gets q's node as its predecessor's node (at line 7). r continues to line 19, as r requests a different session than its predecessor q, and sees that its predecessor's node's *active* field is set to *YES*. Then, r continues to line 19.5, notifying that it did not help to advance *Head*, and waits at line 20 for the *go* bit to be set to *true*. q completes its CS, advances *Head* at line 42, sets the new first node's *go* bit to *true* (line 43), and completes its exit code. But the new first node is q's node, since no process advanced *Head* when p completed its CS. All the new processes will wait until r becomes enabled, but no process can help r becoming enabled and a deadlock occurs.
4. *Why don't we use a dummy node?* The head is being set for the first time at line 9 by the first process that executes the algorithm. The head can be set in line 9 only by one process, the first process, because of the use of the FAS operation at line 7. Only the first process returns null from this operation. The other times that a process may set the head at line 9 is when another process, say q, sets the tail to null in q's exit section, and then q should set the head to null and clear the queue. That means the algorithm is returned to its initial state.

5. *Why have we added line 39, although the algorithm is correct without line 39?* We can remove line 39, and the algorithm would still be correct, as we would override Head at line 9 with the next process that executes the algorithm. We have added this line for semantics reasons, as we do not want to get into a situation where Head points to a node that is no longer active while there are no processes that want to execute the algorithm. That is, when no processes are executing the algorithm, Head and Tail should be null.
6. *Is it essential to include lines 43 and 47 within the ME critical section?* We can move lines 43 and 47 outside the ME critical section (CS), and the algorithm would still be correct. At these lines, we use a local variable *temp_head*, which no other process can change. We placed these lines inside the ME CS for better readability. If we move these lines outside the ME CS, we would need to check if we executed line 38, line 40, or line 44, and only if we executed lines 40 or 44, we then should set *go*.
7. *Who can set process p's go bit to true when p waits at line 20?* By inspecting the code, we can see that p's *go* bit can be changed to *true* either in the entry section (line 31) or in the exit section (lines 43 and 47). Assume p spins on its *go* bit at line 20. p would stop spinning when its *go* bit changed to *true* by another process. Since p is at line 20, p has already tested the condition at line 12 and got *false*. This means that p has requested a different session than its predecessor. Thus, p's predecessor will not reach line 31 because the predecessor will see (line 29) that its successor requests a different session. Each process that acquires the ME lock causes *Head* to be advanced by exactly one step. Therefore, the process that will change p's *go* bit to *true* is the last process that acquires the ME lock and requests the same session as p's predecessor.
8. *The algorithm might become simpler if one can obviate the use of Head. Is the use of Head necessary?* We have tried to simplify the algorithm by not using Head, as done for mutual exclusion in the implementation of the MCS lock [28]. Solving the GME problem is more complex than solving ME. There are more possible race conditions that should be avoided, and using Head helped us in the design of the algorithm. In particular, in the exit code, in lines 43 & 47 the new process at the head of the queue is enabled, by a process that is exiting. We do not see how to implement this in constant time without using Head.

4 Correctness Proof

In this section, we prove that the algorithm satisfies the following properties.

▶ **Theorem 1.** *The GME algorithm has the following properties:*

1. *it satisfies mutual exclusion, starvation-freedom, strong group concurrent entering, and group bounded exit;*
2. *it satisfies FCFS and FIFE fairness;*
3. *it has constant RMR complexity in both the CC and the DSM models;*
4. *it does not require to assume that the number of participating processes or the number of sessions is a priori known;*
5. *it uses constant space per process;*
6. *it uses an atomic instruction set supported by most modern processor architectures, namely, read, write, fetch&store (FAS) and compare&swap (CAS).*

For the lack of space, the proof is omitted. A very detailed proof of Theorem 1 appears in [27].

5 Discussion

With the wide availability of multi-core systems, synchronization algorithms like GME are becoming more important for programming such systems. In concurrent programming, processes (or threads) are often sharing data structures and databases. The GME problem deals with coordinating access to such shared data structures and shared databases.

We have presented a new GME algorithm that is the first to satisfy several desired properties. Based on our algorithm, it would be interesting to design other GME algorithms, such as abortable GME [18] and recoverable GME [15], which will preserve the properties of our algorithm.

References

1 R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, 1992.

2 A. Aravind and W.H. Hesselink. Group mutual exclusion by fetch-and-increment. *ACM Trans. Parallel Comput.*, 5(4), 2019.

3 R. Atreya, N. Mittal, and S. Peri. A quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set. *IEEE Transactions on Parallel and Distributed Systems*, 18(10), 2007.

4 J. Beauquier, S. Cantarell, A. K. Datta, and F. Petit. Group mutual exclusion in tree networks. In *Proc. of the 9th International Conference on Parallel and Distributed Systems*, pages 111–116, 2002.

5 V. Bhatt and C.C. Huang. Group mutual exclusion in $O(log\ n)$ RMR. In *Proc. 29th ACM Symp. on Principles of Distributed Computing*, pages 45–54, 2010.

6 G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronization. In *Proc. of the 13th Annual Symposium on Parallel Algorithms and Architectures*, pages 122–133, 2001.

7 I. Calciu, D. Dice, Y. Lev, V. Luchangco, V.J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM symposium on Principles and practice of parallel programming*, PPoPP '13, page 157–166, February 2013.

8 K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646, 1984.

9 P.L. Courtois, F. Heyman, and D.L Parnas. Concurrent control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, 1971.

10 T.S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR-93-02-02, Dept. of Computer Science, Univ. of Washington, 1993.

11 R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *18th international symposium on distributed computing*, 2004. *LNCS 3274* Springer Verlag 2004, 71–85.

12 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

13 R. Dvir and G. Taubenfeld. Mutual exclusion algorithms with constant rmr complexity and wait-free exit code. In *Proc. of the 21st international conference on principles of distributed systems (OPODIS 2017)*, October 2017.

14 S. Gokhale and N. Mittal. Fast and scalable group mutual exclusion, 2019. `arXiv:1805.04819`.

15 W. Golab and A. Ramaraju. Recoverable mutual exclusion. In *Proc. 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.

16 V. Hadzilacos. A note on group mutual exclusion. In *Proc. 20th symp. on Principles of distributed computing*, pages 100–106, 2001.

17 Y. He, K. Gopalakrishnan, and E. Gafni. Group mutual exclusion in linear time and space. *Theoretical Computer Science*, 709:31–47, 2018.

18 P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, pages 295–304, 2003.

19 P. Jayanti, S. Jayanti, and S. Jayanti. Towards an ideal queue lock. In *Proc. 21st International Conference on Distributed Computing and Networking*, ICDCN 2020, pages 1–10, 2020.

20 P. Jayanti, S. Petrovic, and K. Tan. Fair group mutual exclusion. In *Proc. 22th ACM Symp. on Principles of Distributed Computing*, pages 275–284, July 2003.

21 Yuh-Jzer Joung. Asynchronous group mutual exclusion. In *Proc. 17th ACM Symp. on Principles of Distributed Computing*, pages 51–60, 1998.

22 Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

23 H. Kakugawa, S. Kamei, and T. Masuzawa. A token-based distributed group mutual exclusion algorithm with quorums. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1153–1166, 2008.

24 P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proc. 18th ACM Symp. on Principles of Distributed Computing*, pages 23–32, 1999.

25 P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7), 2001.

26 L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

27 L. Maor. Constant RMR group mutual exclusion for arbitrarily many processes and sessions. Master's thesis, The Interdisciplinary Center, Herzliya, Israel, August 2021.

28 J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

29 M. Takamura, T. Altman, and Y. Igarashi. Speedup of Vidyasankar's algorithm for the group k-exclusion problem. *Inf. Process. Lett.*, 91(2):85–91, 2004.

30 G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

31 M. Toyomura, S. Kamei, and H. Kakugawa. A quorum-based distributed algorithm for group mutual exclusion. In *Proc. of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 742–746, 2003.

32 K. Vidyasankar. A simple group mutual ℓ-exclusion algorithm. *Inf. Process. Lett.*, 85(2):79–85, 2003.

33 Kuen-Pin Wu and Yuh-Jzer Joung. Asynchronous group mutual exclusion in ring networks. In *Proc. 13th Inter. Parallel Processing Symposium and 10th Symp. on Parallel and Distributed Processing*, pages 539–543, 1999.