Local-Spin Group Mutual Exclusion Algorithms*
(Extended Abstract)

Robert Danek[1] and Vassos Hadzilacos[2]

[1] IBM Toronto Lab
[2] Department of Computer Science
University of Toronto

Abstract. Group mutual exclusion (GME) is a natural generalisation of the classical mutual exclusion problem. In GME, when a process leaves the non-critical section it requests a "session"; processes are allowed to be in the critical section simultaneously if they have requested the same session. We present GME algorithms (where the number of sessions is not known a priori) that use $O(N)$ remote memory references in distributed shared memory (DSM) multiprocessors, where N is the number of processes, and prove that this is asymptotically optimal even if there are only two sessions that processes can request. We also present an algorithm for two-session GME that requires $O(\log N)$ remote memory references in cache-coherent (CC) multiprocessors. This establishes a complexity separation between the CC and DSM models: there is a problem (two-session GME) that is provably more efficiently solvable in the former than in the latter.

1 Introduction

Group mutual exclusion (GME) [1] is a generalisation of the classical mutual exclusion problem [2, 3]. In GME there are N processes $1, 2, \ldots, N$, each having the structure shown in Figure 1. Each process alternates between a (possibly nonterminating) ***non-critical section*** (NCS) and a (terminating) ***critical section*** (CS). Each time a process leaves the NCS to enter the CS, it "requests" a positive integer (not necessarily the same each time) called a ***session***. Two processes are said to ***conflict*** if they are requesting different sessions. Processes coordinate their entry to the CS by executing the ***trying*** and ***exit protocols*** so that the following properties are satisfied:

Mutual exclusion: No two conflicting processes are in the CS simultaneously.
Lockout freedom: A process that leaves the NCS eventually enters the CS.
Bounded exit: A process completes its exit protocol in a bounded number of its own steps.
Concurrent entering: If process p is in the trying protocol while no process is requesting a conflicting session, then p completes the trying protocol in a bounded number of its own steps.

* Research supported in part by the Natural Sciences and Engineering Research Council of Canada.

repeat
  NCS
  Trying Protocol $\begin{cases} \text{Doorway (bounded)} \\ \text{Waiting room} \end{cases}$
  CS
  Exit Protocol
forever

Fig. 1. GME process structure

The ordinary mutual exclusion problem is the special case where each process p always requests session p, and thus any two processes conflict.

In some applications it may be important that processes enter the CS in a "fair" manner — i.e., roughly in the order in which they leave the NCS. To formalise this we assume that the trying protocol starts with a *bounded* section of code (i.e., one that contains no unbounded loops), called the ***doorway***; the rest of the trying protocol is called the ***waiting room*** (see Figure 1). The fairness requirements can now be stated as follows:

First-come-first served (FCFS): If process p completes the doorway before a conflicting process q starts the doorway, then q does not enter the CS before p [3, 4].

First-in-first-enabled (FIFE): If process p completes the doorway before a non-conflicting process q starts the doorway, and q enters the CS before p, then p enters the CS in a bounded number of its own steps [5, 6].

We can also strengthen the concurrent entering property to:

Strong concurrent entering: If process p completes the doorway before any conflicting process q starts the doorway, then p enters the CS in a bounded number of its own steps [6].

Model: We work in the context of the asynchronous, shared-memory model of computation. More precisely, we consider a system consisting of N processes, named $1, 2, \ldots, N$, and a set of shared variables. Each process also has its own private variables. Processes can communicate only by accessing the shared variables by means of read, write and COMPARE&SWAP operations. (COMPARE&SWAP(x, v, w) atomically reads the shared variable x, writes w into it iff its old value was v, and returns the old value.) One of our algorithms additionally uses FETCH&ADD(x, v), which atomically adds v to shared variable x and returns the old value of x. An execution is modeled as a sequence of process steps. In each step, a process either performs some local computation affecting only its private variables, or accesses a shared variable by applying one of the available operations. Processes take steps asynchronously: there is no bound on the number of other processes' steps that can be executed between two successive steps of a process. Every process, however, is *live*: if it has not terminated, it will eventually execute its next step.

Within this general framework, there are two models of shared memory: the ***distributed shared-memory*** (DSM) model and the ***cache-coherent*** (CC) model. These differ on where shared variables are physically stored and how processes can access them. These models are important for our results and we describe them next.

In the DSM model, each process has an associated memory module. Every shared variable is stored at the memory module associated with exactly one process. Accessing a variable stored at a different process's memory module causes the process to make a ***remote memory reference***.

In the CC model, all shared variables are stored in a global store that is not associated with any particular process. Every time a process reads a shared variable, it does so using a local (cached) copy of the variable. Whenever the cached variable is no longer valid — either because the process has never read the variable before, or because some process overwrote it in the global store — the process makes a remote memory reference and copies the variable into its local store (i.e., it caches the variable). Also, every time a process writes a variable, the process writes the variable to the global store (thereby invalidating all cached copies), which involves a remote memory reference. In our complexity analysis of algorithms under the CC model we assume that unsuccessful COMPARE&SWAP operations, i.e., ones that do not update the shared variable they access, do not invalidate cached copies of the variable.

Remote memory references are orders of magnitude slower than accesses to the local memory module (or cache); they also generate traffic on the processor-to-memory interconnect, which can be a bottleneck. For these reasons, the performance of many algorithms for shared memory multiprocessor systems depends critically on the number of remote memory references they generate [7]. In such systems, it is therefore important to design algorithms that minimise the number of remote memory references. An ordinary or group mutual exclusion algorithm is called ***local spin*** (under the DSM or CC model) if the maximum number of remote memory references made in any passage is bounded. (A ***passage*** is the sequence of steps executed by a process between the time it leaves the NCS and the time it next returns to it.) Many such algorithms have been devised for ordinary mutual exclusion; see [8] for a survey. In this paper we present local-spin algorithms for *group* mutual exclusion under the DSM model. To our knowledge, these are the first such algorithms (but see "Related Work" below for a caveat).

In the pseudocode description of algorithms we use the construct

$$\textbf{cobegin}\ S_1 \parallel \ldots \parallel S_k\ \textbf{coend}$$

where $S_1, \ldots, S_k$ are "threads", i.e., statements typically containing one or more unbounded loops. The meaning of this construct is that the threads are executed concurrently in an arbitrary fair interleaving. (A fair interleaving is one which, if infinite, contains an infinite number of instructions of *every* thread.) Furthermore, if one of the threads terminates, then the execution of the remaining threads is eventually suspended and the entire cobegin-coend statement terminates.

Outline of Results: In Section 2 we present three local-spin GME algorithms under the DSM model. Two of these algorithms satisfy strong fairness properties; the third sacrifices strong fairness to achieve greater concurrency. These three algorithms require $O(N)$ remote memory references per passage. We also prove that this is in some sense asymptotically optimal: In the DSM model, *any* GME algorithm (even for the special case of only two sessions and regardless of how powerful synchronisation primitives it may use), requires $\Omega(N)$ remote memory references for some passage. This is in sharp contrast to ordinary (as opposed to group) mutual exclusion, which can be solved in the DSM model using only $O(\log N)$ remote memory references, even if the shared variables can be accessed only by read and write operations [9].

In Section 3 we present an algorithm for two-session GME that requires only $O(\log N)$ remote memory references in the CC model. This algorithm uses COMPARE&SWAP and FETCH&ADD primitives to access shared memory. This result is interesting, as it provides a complexity separation between the DSM and CC models. We are not aware of other such separation results for these two models in asynchronous systems (but see "Related Work" below for similar results in synchronous systems).[3] By using a tournament-tree technique, this algorithm can be used to solve the M-session GME problem, for any fixed M, with $O(\log M \log N)$ remote memory references in the CC model.

We omit proofs from this extended abstract. They can be found in [10].

Related Work: Group mutual exclusion was first formulated and solved by Joung [1]. Many algorithms for this problem (or variants of it) have been proposed [1, 11, 4, 12, 6]. The only one of these that is local-spin in the DSM model is that by Keane and Moir [11]; it requires $O(\log N)$ remote memory references per passage. This algorithm, however, does not satisfy concurrent entering: there are executions where, although all processes request the same session, some processes are delayed arbitrarily long in the trying protocol. It satisfies a weaker liveness property called ***concurrent occupancy*** [4]. The difference between concurrent entering and concurrent occupancy turns out to be substantial: as our lower bound shows, there is no GME algorithm that satisfies concurrent entering and requires only $o(N)$ remote memory references per passage.

Kim and Anderson studied local-spin algorithms for ordinary mutual exclusion in *synchronous* systems, where there is a known maximum delay on the time to access a shared variable, and processes have access to reasonably accurate timers [13]. Their results provide a complexity separation between the DSM and CC models in such systems. They show that to solve ordinary mutual exclusion in synchronous systems, $\Theta(1)$ remote memory references are sufficient in the DSM model, while $\Theta(\log \log N)$ remote memory references are necessary (and sufficient) in the CC model. Intriguingly, this separation in the synchronous

[3] Truth in advertising: This separation result is not as strong as one might hope. In particular, we are able to prove it only in the context where COMPARE&SWAP and FETCH&ADD are both available, and under the assumption that unsuccessful COMPARE&SWAP operations do not invalidate cached copies. We view this result as a modest first step in exploring the relationship between the CC and DSM models.

model is in the reverse direction than the separation we show in this paper for the asynchronous model.

All the GME algorithms we present in this paper use as a "black box" — i.e., without any assumptions about how it works — an ***abortable*** algorithm for ordinary mutual exclusion that satisfies the FCFS property. (This is the property stated above keeping in mind that, in ordinary mutual exclusion, any two processes conflict.) An abortable mutual exclusion algorithm has, in addition to the trying and exit protocols, an ***abort protocol***. This can be invoked at any time while a process is waiting in its trying protocol and causes the process to re-enter the NCS in a bounded number of its own steps. Jayanti has devised an abortable FCFS mutual exclusion algorithm that requires $O(\log N)$ remote memory references per passage in the DSM and CC models [14]. We use this fact in our complexity analyses, and we refer to this algorithm as the ***underlying mutual exclusion algorithm***.

2 Local-Spin GME Algorithms for the DSM Model

In this section we present GME algorithms that require $O(N)$ remote memory references per passage in the DSM model, and show a matching lower bound.

2.1 "Fair" GME Algorithms

In this section we present two algorithms that emphasise fairness. The first of these satisfies mutual exclusion, lockout freedom, bounded exit, strong concurrent entering and FCFS. This algorithm is conceptually simple, but it does not satisfy FIFE. The second algorithm is an elaboration of the first, and satisfies FIFE in addition to all the other properties. The two algorithms are shown together in pseudocode form in Figure 2. The first consists of the non-shaded portions of the pseudocode; the second also includes the shaded portions. We now give an informal but hopefully informative presentation of the algorithms, cross-referenced to the pseudocode, explaining the actions of each process at a high level.

We start with the simple version that satisfies FCFS but not FIFE. (Refer to Figure 2, ignoring the shaded portions.) Upon leaving the NCS, a process p makes public its session by writing it into a shared variable $statusCS[p]$ (line 4).[4] It then executes the doorway portion of the underlying FCFS mutual exclusion algorithm denoted MutexDoorway (line 5), and notes the set of conflicting processes (lines 6–9). As p considers each process j to determine if they conflict, it sets to true a boolean variable $barricade[p, j]$ (line 8). This is a spin-lock used by p to wait for j, as we will see shortly. These actions comprise p's

[4] In all places where $statusCS[p]$ is written in the first version of the algorithm (initialisation, and lines 4 and 20), the second component *passage* is set to -1. So, in this version of the algorithm, we can think of this variable as containing only the first component *session*. The component *passage* plays a role in the second version of the algorithm.

```
shared variables:
statusCS: array[1..N] of record session: integer; passage: integer init (0, −1)
barricade: array[1..N][1..N] of boolean        ▷ barricade[p, j] is in p's memory, ∀j

capture: array[1..N] of integer init 0        ▷ capture[p] is in p's memory

private variables:
mysession: integer        ▷ session p wants to attend, set when p leaves the NCS
conflict_set: set of integer

passage: integer init 0
l_status: array[1..N] of record session: integer; passage: integer

repeat
    NCS
    passage := passage + 1
    for j ∈ {1..N} \ {p} do l_status[j] := statusCS[j]
    statusCS[p] := (mysession, −1)
    MutexDoorway
    conflict_set := ∅
    for j = 1 to N do
        barricade[p, j] := true
        if statusCS[j].session ∉ {0, mysession} then conflict_set := conflict_set ∪ {j}

    statusCS[p] := (mysession, passage)
    cobegin
        MutexWaitingRoom
        ||
        for each j ∈ conflict_set do await ¬barricade[p, j]
        ||
        await capture[p] = passage
    coend        ▷ when one co-routine terminates, go to the next line
    for j ∈ {1..N} \ {p} do
        if ∃v ≠ −1 : statusCS[j] = l_status[j] = (mysession, v) then
            capture[j] := l_status[j].passage
    CS
    statusCS[p] := (0, −1)
    for j = 1 to N do barricade[j, p] := false
    if mutex qualified then MutexExit else MutexAbort
forever
```

Fig. 2. "Fair" GME algorithms

doorway. The process then enters its waiting room, where it waits for one of two events: (a) the completion of the underlying mutual exclusion algorithm's waiting room, denoted MutexWaitingRoom (line 12), or (b) the clearing of spinlocks *barricade*$[p, j]$ by every conflicting process j (line 13). At that time, p enters the CS. If it does so because of event (a) (respectively, (b)) we say that it enters the CS ***mutex qualified*** (respectively, ***conflict-free qualified***).

When p leaves the CS it executes its exit protocol, which consists of the following actions: First p signals to other processes that it is no longer interested in a session by setting *statusCS*$[p]$ to 0 (line 20). For every process j, p then clears the spinlock *barricade*$[j, p]$ that j uses to wait for p (line 21). As a result, if j is waiting for p in line 13 of its waiting room (because j, in its doorway, found p to be conflicting) it stops doing so. Finally, p executes either the exit (MutexExit) or the abort (MutexAbort) protocol of the underlying mutual exclusion algorithm, depending on whether it entered the CS mutex qualified or conflict-free qualified (line 22).

This algorithm satisfies mutual exclusion, lockout freedom, bounded exit, FCFS, and strong concurrent entering. It does not, however, satisfy FIFE. We describe a scenario that shows how FIFE can be violated. The scenario involves processes p and q requesting the same session s and process r requesting a different session s'. Suppose that r enters the CS. Process p then leaves the NCS; while in the doorway, p notes the conflicting process r and completes the doorway. Process r leaves the CS and sets *session*$[r]$ to 0, but does not (yet) execute any more steps of its exit protocol. Process q leaves the NCS; in the doorway it notes no conflicting processes and so it enters the CS conflict free. However, because r has not yet cleared the spinlock *barricade*$[p, r]$, p is still waiting for r. Moreover, r has not yet executed MutexExit or MutexAbort and so p cannot become mutex qualified. Thus, we have a situation where p's doorway precedes q's and q enters the CS, but p cannot enter the CS in a bounded number of its own steps. This is a violation of FIFE.

To avoid this we embellish the algorithm with a "capturing mechanism". The overall idea is that a process such as q in the above scenario will, before entering the CS, "capture" a process such as p thereby enabling it to enter the CS without waiting. In more detail, the capturing mechanism, which is implemented by the shaded portions of pseudocode in Figure 2, works as follows. Each process p makes public in shared variable *statusCS*$[p]$ two pieces of information: the session number it is requesting and its current passage number. Process p starts its doorway by making a note of all other processes' *statusCS* variables (line 3). It then makes public its session number but writes -1 as its current passage; this indicates that p is active but still in the doorway. Process p then executes the doorway of the underlying mutual exclusion algorithm and notes all conflicting processes j, setting the spinlocks *barricade*$[p, j]$; these play exactly the same role as before. Finally, p indicates that it has completed the doorway by writing its passage number in the second component of *statusCS*$[p]$ (line 10). These actions comprise p's doorway. The process then enters the waiting room, where it now waits for one of *three* events to happen:

(a) The waiting room of the underlying mutual exclusion algorithm finishes (line 12). As before, we say that p enters the CS mutex qualified.
(b) The spinlocks *barricade*$[p, j]$ have been cleared for every conflicting process j (line 13). This indicates that all conflicting processes noted in p's doorway are done. We say that p enters the CS conflict-free qualified.
(c) Variable *capture*$[p]$ equals p's passage number (line 14). As we will see shortly, this indicates that p has been captured and can enter the CS without (further) waiting. In this case, we say that p enters the CS ***capture qualified***.

Just before entering the CS, p examines every other process j to determine whether to capture it (lines 16–18). Process p captures j iff: (i) p and j do not conflict; and (ii) *statusCS*$[j]$ has not changed since p made a note of it in its doorway (line 3) and, at that time, j had finished the doorway (and so the passage recorded in *statusCS*$[j]$ was not -1). If this is the case, p captures j by setting the spinlock *capture*$[j]$ to j's current passage number. (As we saw in case (c) above, a process completes the waiting room and becomes capture qualified when its *capture* variable is set to its current passage.) The capturing mechanism completes p's entry protocol. The exit protocol is identical to the previous algorithm's.

The opening of a new pathway to the CS via the capturing mechanism is cause for concern regarding mutual exclusion: We have to verify that a captured process is not enabled to enter the CS while another process requesting a different session is in the CS. Fortunately, this is the case but some interesting machinery is needed to prove that mutual exclusion is never violated.

Theorem 1. *The algorithm in Figure 2 satisfies mutual exclusion, lockout freedom, bounded exit, strong concurrent entering, FCFS and FIFE. Furthermore, in the DSM model, it requires $O(N)$ remote memory references per passage in addition to those used by the underlying mutual exclusion algorithm. Thus, combining this algorithm with Jayanti's abortable FCFS mutual exclusion algorithm [14], yields a GME algorithm with the above properties that requires $O(N)$ remote memory references per passage in the DSM model.*

2.2 "High-Concurrency" GME Algorithms

A drawback of both algorithms in Section 2.1 is illustrated by the following scenario: Let S be a set of processes, and p be a process that is not in S. Suppose that all processes in S leave the NCS at about the same time requesting the same session s, while p requests a different session s'. Further suppose that p publicises its session (line 4) before any process in S goes through the loop in the doorway that checks for conflicting processes (lines 7–9). Thus every process in S detects conflict and cannot be conflict-free qualified until after p leaves the CS. Moreover, assume that all processes in S took a snapshot of the *statusCS* variables (line 3) before any of them wrote its *statusCS* variable (line 4), so that there is no opportunity for any process in S to be capture qualified. Finally, suppose that the processes in $S \cup \{p\}$ execute through the doorway and waiting room in such a way that all processes in S become mutex qualified before p. This means that all the processes in S execute through the CS sequentially, even though they could all enter the CS concurrently.

We will now describe a "high-concurrency" GME algorithm that alleviates this problem by using a capturing mechanism. A process that enters the CS mutex qualified "captures" other processes that have requested the same session and enables them to enter the CS wait free. In doing so, our algorithm sacrifices FCFS and FIFE, the strong fairness properties of the previous algorithms.

```
shared variables:
statusCS: array[1..N] of {(0, OUT), (integer, IN), (integer, REQ)} init (0, OUT)
barricade: array[1..N][1..N] of boolean init true
                                      ▷ barricade[p, j] is in p's memory, ∀j

private variables:
mysession: integer      ▷ session p wants to attend, set when p leaves the NCS
conflict_set: set of integer

repeat
    NCS
    statusCS[p] := (mysession, REQ)
    MutexDoorway
    conflict_set := ∅
    for j = 1 to N do
        barricade[p, j] := true
        if statusCS[j] ∉ {(0, OUT), (mysession, REQ), (mysession, IN)} then
            conflict_set := conflict_set ∪ {j}
    cobegin
        MutexWaitingRoom
        ||
        for each j ∈ conflict_set do await ¬barricade[p, j]
        ||
        await statusCS[p] = (mysession, IN)
    coend               ▷ when one co-routine terminates, go to the next line
    for j = 1 to N do
        barricade[p, j] := true
        if ∃s ≠ mysession : statusCS[j] = (s, IN) await ¬barricade[p, j]
    if mutex qualified then
        for j ∈ {1..N} \ {p} do
            Compare&Swap(statusCS[j], (mysession, REQ), (mysession, IN))
    CS
    statusCS[p] := (0, OUT)
    for j = 1 to N do barricade[j, p] := false
    if mutex qualified then MutexExit else MutexAbort forever
```

Fig. 3. "High concurrency" GME algorithm using Compare&Swap

The algorithm is shown in Figure 3. Each process p has a shared variable *statusCS*[p] in which it publicises some information about itself in the form of a pair (s, v). The value of s is the session that p is requesting or 0; the value of v is one of OUT, REQ or IN. If $v = \text{IN}$ then p has been captured; if $v = \text{REQ}$ then p in in the trying protocol or the CS but has not been captured. If $v = \text{OUT}$ then p is in the NCS or in the exit protocol (and $s = 0$).

Upon leaving the NCS, p writes (s, REQ) into *statusCS*[p], where s is the session it is requesting (line 2), and executes the doorway of the underlying mutual exclusion algorithm (line 3). It then records the name of every conflicting process j and sets the spinlock *barricade*[p, j] on which it will wait for conflicting processes to finish, as in the previous algorithms (lines 4–8). These actions comprise the doorway.

Process p then enters the waiting room which consists of two phases. The first phase is as in the previous algorithm; p waits for one of three events (lines 9–13): (a) to be the winner in the underlying mutual exclusion algorithm (mutex qualified); (b) to find that all conflicting processes are done (conflict-free qualified); or (c) to be captured (capture qualified). In the second phase, p waits for captured conflicting processes to finish the CS (lines 14–16) and then, *if it completed the first phase of the waiting room mutex qualified*, p captures any processes that have requested the same session (lines 17–19). Waiting for captured conflicting processes is accomplished by (re)using the *barricade* spinlocks: p resets the spinlock *barricade*[p, j], and then waits for j to clear the spinlock in its exit protocol (line 22) if j is conflicting. Capturing processes that have requested the same session s as p is accomplished by writing (s, IN) into *statusCS*[j] for every process j whose *statusCS*[j] was previously (s, REQ) (line 19). Note that here we use the COMPARE&SWAP operation, so that the reading of *statusCS*[j] (to see if it is (s, REQ)) and its updating (to make it (s, IN)) are done in one atomic action. At this point, p has completed the trying protocol and enters the CS.

The exit protocol is very similar to the algorithms we have seen before: p indicates that it is out of the CS by writing $(0, \text{OUT})$ in *statusCS*[p] (line 21), clears the spinlocks of every process j that could be waiting for it (line 22), and then executes the exit or abort procedure of the underlying mutual exclusion algorithm, depending on whether it entered the CS mutex qualified or not (line 23).

Theorem 2. *The algorithm in Figure 3 satisfies mutual exclusion, lockout freedom, bounded exit and concurrent entering. Furthermore, in the DSM model, it requires $O(N)$ remote memory references per passage in addition to those used by the underlying mutual exclusion algorithm. Thus, combining this algorithm with Jayanti's abortable FCFS mutual exclusion algorithm [14], yields a GME algorithm with the above properties that requires $O(N)$ remote memory references per passage in the DSM model.*

2.3 Lower Bound on Remote Memory References

We now show that, in some sense, the preceding algorithms are asymptotically optimal in terms of the number of remote memory references they generate in the DSM model.

Theorem 3. *Any algorithm that satisfies mutual exclusion, lockout freedom, bounded exit and concurrent entering, and is local-spin under the DSM model, must perform $\Omega(N)$ remote memory references in some passage. This holds even for the two-session GME problem, i.e., when each process can request one of only two sessions.*

We now sketch a proof of this lower bound. Let **A** be an algorithm for two-session GME that is local-spin under the DSM model. Consider the following execution of **A**: Some process p requests session s, while no other process is active. Process p enters the CS (because **A** is lockout-free). At this point the remaining $N-1$ processes, each requesting the other session s', enter the trying protocol. Since p is in the CS and **A** satisfies mutual exclusion, none of the $N-1$ processes requesting s' can enter the CS. There is no bound on the amount of time that p can spend in the CS, so we assume that p stays in the CS until each of the $N-1$ processes in the trying protocol enters a busy-wait loop. Since **A** is local-spin under the DSM model, this implies that there are at least $N-1$ distinct variables on which the processes in the trying protocol are busy-waiting. For any such process to enter the CS, the local-spin variable(s) it is busy-waiting on must be updated. Thus, when p leaves the CS and executes the exit protocol, it must update at least $N-1$ remotely stored variables: If it updated any fewer then we could continue the execution in such a way that at least one process in the trying protocol would not enter the CS in a bounded number of its own steps after p is no longer active. This would violate the concurrent entering property that **A** is supposed to satisfy. Therefore, there is an execution of **A** in which a process executes $\Omega(N)$ remote memory references during a passage.

The difficulty in formalising this proof is that the idea of a process "entering a busy-wait loop", although intuitively clear, is not easy to express formally. A rigorous proof is given in [10] (see Chapter 3). Although the technical details of that proof are intricate, the basic intuition is contained in the simplified proof sketch we presented here.

3 Local-Spin Algorithms for GME in the CC model

We now turn our attention to the CC model. We will describe a two-session GME algorithm that uses (as a black box) an abortable FCFS ordinary mutual exclusion algorithm and requires $O(1)$ remote memory references in addition to those used by the underlying mutual exclusion algorithm.

The algorithm is shown in pseudocode in Figure 4. We assume that the two sessions that processes can request are 1 and 2. If $s \in \{1, 2\}$, $\overline{s}$ denotes the "other" session than s — i.e., $\overline{s} = 3 - s$. We provide a high-level overview of the algorithm, cross-referenced to the pseudocode. We start by describing the shared variables used in the algorithm, and then explain how they are used.

Associated with each session $s \in \{1, 2\}$ are two shared variables: a counter, *active*[s], and a spin-lock, *gate*[s]. Each process increments by one the counter of the session it is requesting when it enters the trying protocol (line 3) and atomically reads and decrements that counter by one when it leaves the CS (line 17). In addition, each process atomically reads and increments by $N+1$ the counter of the *other* session when it enters the waiting room (line 9), and decrements that counter by the same amount when it leaves the waiting room (line 15). Thus, if $active[s] = a(N+1) + b$, where $0 \le b \le N$, then there are exactly b processes requesting s in lines 4–17, and exactly a processes requesting $\overline{s}$ in lines 10–15. We will later see in more detail how this variable is used, but it is clear that by reading this variable a process can get some idea of how many active processes request each of the sessions.

```
    shared variables:
    gate: array[1..2] of record tag : {0..N}; state : {OPEN, CLOSED} init (0, CLOSED)
    active: array[1..2] of integer init 0

    private variables:
    mysession : {1, 2}          ▷ session p wants to attend, set when p leaves the NCS
    l_active: array[1..2] of integer                        ▷ private copy of active
    l_gate: record tag : {0..N}; state : {OPEN, CLOSED}      ▷ and of gate[$\overline{mysession}$]

    repeat
1       NCS
2       othersession := 3 − mysession  ▷ the opposite session than the one p requests
3       FETCH&ADD(active[mysession], 1)
4       l_gate := gate[othersession]
5       if l_gate ≠ (0, CLOSED) then
6           COMPARE&SWAP(gate[othersession], l_gate, (0, CLOSED))
7       COMPARE&SWAP(gate[othersession], (0, OPEN), (0, CLOSED))
8       MUTEXDOORWAY
9       l_active[othersession] := FETCH&ADD(active[othersession], N + 1)
10      cobegin
11          MUTEXWAITINGROOM
            ||
12          if l_active[othersession] mod (N + 1) > 0 then
13              await gate[mysession] = (0, OPEN)
14      coend                  ▷ when one coroutine terminates, go to the next line
15      FETCH&ADD(active[othersession], −(N + 1))
16      CS
17      l_active[mysession] := FETCH&ADD(active[mysession], −1)
18      if l_active[mysession] mod (N+1) = 1 and l_active[mysession] ≥ (N+1) then
19          gate[othersession] := (p, CLOSED)
20          if active[mysession] mod (N + 1) = 0 then
21              COMPARE&SWAP(gate[othersession], (p, CLOSED), (0, OPEN))
22      if mutex qualified then MUTEXEXIT else MUTEXABORT
    forever
```

Fig. 4. Two-session GME algorithm for the CC model

To enter the CS, a process executes the doorway of the underlying mutual exclusion algorithm (line 8) and then waits for one of two events: (a) the waiting room of the underlying mutual exclusion algorithm completes (line 11), in which case we say that the process enters the CS mutex qualified; or (b) it detects that no conflicting process is active (lines 12–13), in which case we say that the process enters the CS conflict-free qualified. We now discuss how a process detects that no conflicting process is active.

As alluded to earlier, a process requesting s can detect that there are no active processes currently requesting $\overline{s}$ by checking that $active[\overline{s}]$ **mod** $(N+1) = 0$. Unfortunately, a mechanism for detecting absence of conflicting requests that spins on the value of $active[\overline{s}]$ does not give us the desired complexity of only $O(1)$ remote memory references in the CC model.

For this reason we need a different mechanism for processes to detect the absence of conflicting requests. The shared variable $gate[s]$ provides this mechanism. At a high level, the idea behind this mechanism is that as soon as a process requesting $\overline{s}$ leaves the NCS, it "closes" $gate[s]$ by setting it to $(0, \text{CLOSED})$ (lines 4–7), thereby preventing processes requesting s from entering the CS conflict-free qualified (line 13). We refer to this as the "gate-closing" phase of the trying protocol. The subsequent opening of $gate[s]$ is accomplished as follows: As each process requesting $\overline{s}$ leaves the CS, it checks whether (a) it is the last such process to do so, and (b) there are conflicting processes in the waiting room. Process p, requesting session $\overline{s}$, does this by looking at the old value v of $active[\overline{s}]$ when it decremented it (line 18): If v **mod** $(N+1) = 1$, then p is the last one requesting $\overline{s}$ to leave the CS; and if $v \geq (N+1)$, then there are conflicting processes (requesting s) in the waiting room. If both of these conditions are satisfied, p attempts to "open" $gate[s]$ (lines 19–21). (The second condition, that a conflicting process be in the waiting room, is needed only to ensure that our algorithm makes $O(1)$ remote memory references in addition to those of the underlying mutual exclusion algorithm, not for any of the correctness properties.)

Process p should not "open" $gate[s]$ by using a simple assignment statement to set it to $(0, \text{OPEN})$. If p did this, there is the possibility that between the time when p reads $active[\overline{s}]$ (line 17) and the time when it writes $gate[s] = (0, \text{OPEN})$, some other process p' requesting session $\overline{s}$ executes its doorway (lines 2–9) and goes into its waiting room (lines 10–15) without closing the gate, since it already found it closed (lines 4–6). Process p could then set $gate[s] = (0, \text{OPEN})$ and return to the NCS, leaving us in a situation where $gate[s]$ is open, and a process requesting session $\overline{s}$ (i.e., p') has gotten past the gate-closing phase of the algorithm (lines 4–7) without ensuring that $gate[s]$ is closed. This is dangerous: after p' enters the CS, since $gate[s]$ is open, processes requesting session s can also enter the CS conflict-free (by completing line 13), thus violating mutual exclusion.

The preceding could be avoided if p, when it leaves the CS, could somehow do the following atomically: read and decrement the variable $active[\overline{s}]$, check if the value read meets the conditions for opening $gate[s]$ and, if it does, set $gate[s] = (0, \text{OPEN})$. This would ensure that other processes requesting session $\overline{s}$ (such as p' in the above scenario) could not race into the waiting room before p has the opportunity to set $gate[s] = (0, \text{OPEN})$. However, there is no realistic synchronization primitive that allows p to perform atomically all the required operations on $active[\overline{s}]$ and $gate[s]$. This leads us to devise a mechanism that achieves an equivalent effect.

Such a mechanism is located in lines 19–21. The first thing that p does in this fragment of the algorithm is to set $gate[s] = (p, \text{CLOSED})$. This essentially "tags" the $gate[s]$ variable with p's identifier. Using this tag, p can later determine if anyone else has written to $gate[s]$ since p last wrote to it. Process p then repeats the check of whether it is the last active process to leave the CS (line 20). If the recheck fails, then p knows that it should not open $gate[s]$ to processes requesting session s. If, however, the check succeeds, then p is still responsible for opening $gate[s]$. This is what p does in line 21. Again, however, p cannot use a simple assignment statement to set $gate[s] = (0, \text{OPEN})$ because of the danger of some other process requesting session $\overline{s}$ racing out of the NCS and into the waiting room. Instead, p makes use of the fact that it can safely assign $(0, \text{OPEN})$ to $gate[s]$ as long as no one wrote to $gate[s]$ since the time p performed the recheck. Specifically, by using COMPARE&SWAP, p atomically checks that its tag is still in $gate[s]$ (which it set before the recheck) and, if it is, it sets $gate[s] = (0, \text{OPEN})$ (line 21).

After a process carries out the "gate opening" procedure (lines 17–21), it completes the exit protocol by executing MUTEXEXIT or MUTEXABORT depending on whether it entered the CS mutex or conflict-free qualified (line 22).

Theorem 4. *The algorithm in Figure 4 solves the two-session FCFS GME problem. That is, it satisfies mutual exclusion, lockout freedom, bounded exit, concurrent entering and FCFS. Furthermore, in the CC model, it requires $O(1)$ remote memory references per passage in addition to those used by the underlying mutual exclusion algorithm. Thus, combining this algorithm with Jayanti's abortable FCFS mutual exclusion algorithm [14], yields a two-session GME algorithm that requires $O(\log N)$ remote memory references per passage in the CC model.*

Note that this "beats" the lower bound on the number of remote memory references per passage for the two-session GME problem in the DSM model (Theorem 3). Thus, two-session GME provides a complexity separation between the DSM and CC models. Consequently there is no general transformation that takes an algorithm that works in the CC model and turns it into one that solves the same problem and works as efficiently (within a constant factor) in the DSM model. This confirms the intuition that in general it is harder to design efficient local-spin algorithms for the DSM model than for the CC model.

Using the two-session GME algorithm as a building block, we can construct an algorithm for the M-session GME problem, for any fixed M, that requires $O(\log M \log N)$ remote memory references per passage in the CC model. The idea is to create a "tournament tree" with M leaves (one per session), and height $\lfloor \log_2 M \rfloor$. Each internal node of the tree corresponds to an instance of the two-session algorithm. A process p requesting session s starts at the leaf that corresponds to s and traces a path from that leaf to the root. At each internal node along that path, p executes the trying protocol of the corresponding two-session GME algorithm, using session 1 or 2 depending on whether p reached the node from its left or right child. When p completes the trying protocol of a node, it moves "up" to the node's parent; in the case of the root, p enters the CS. Upon leaving the CS, p retraces the same path in reverse order (from root to leaf) executing the exit protocols of the nodes it visits on the way "down" the tree. Yang and Anderson's algorithm for ordinary mutual exclusion [9] exhibits a similar recursive structure, though in that case the recursion is on the number of processes N, while here it is on the number of sessions M. A precise description of this algorithm, along with its correctness proof and remote memory reference complexity analysis in the CC model, can be found in [10].

Acknowledgments. We thank the anonymous referees for their comments.

References

1. Joung, Y.: Asynchornous group mutual exclusion. Distributed Computing **13** (2000) 189–206
2. Dijkstra, E.: Solution of a problem in concurrent programming control. Communications of the ACM **8** (1965) 569
3. Lamport, L.: The mutual exclusion problem: Parts I & II. Journal of the ACM **33** (1986) 313–348
4. Hadzilacos, V.: A note on group mutual exclusion. In: Proceedings of the 20th Annual Symposium on Principles of Distributed Computing. (2001) 100–106
5. Fischer, M., Lynch, N., Burns, J., Borodin, A.: Resource allocation with immunity to limited process failure. In: Proceedings of the 20th Annual Symposium on Foundations of Computer Science. (1979) 234–254
6. Jayanti, P., Petrovic, S., Tan, K.: Fair group mutual exclusion. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing. (2003)
7. Mellor-Crummey, J., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9** (1991) 21–65
8. Anderson, J., Kim, Y.J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distributed Computing **16** (2003) 75–110
9. Yang, J.H., Anderson, J.: A fast, scalable mutual exclusion algorithm. Distributed Computing **9** (1995) 51–60
10. Danek, R.: Local-spin group mutual exclusion algorithms. Master's thesis, University of Toronto (2004)
11. Keane, P., Moir, M.: A simple local-spin group mutual exclusion algorithm. IEEE Transactions on Parallel and Distributed Systems **12** (2001) 673–685
12. Vidyasankar, K.: A simple group mutual l-exclusion algorithm. Information Processing Letters **85** (2003) 79–85
13. Kim, Y., Anderson, J.: Timing-based mutual exclusion with local spinning. In: Proceedings of the 17th International Conference on Distributed Computing. (2003) 30–44
14. Jayanti, P.: Adaptive and efficient abortable mutual exclusion. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing. (2003)