An Efficient Weak Mutual Exclusion Algorithm

Paolo Romano
INESC-ID

Luis Rodrigues
INESC-ID/IST

Abstract

The Weak Mutual Exclusion (WME) is a recently proposed abstraction which, analogously to classical Distributed Mutual Exclusion (DME), permits to serialize concurrent accesses to a shared resource. Unlike DME, however, the WME abstraction regulates the access to a replicated *shared resource and is solvable in the presence of less restrictive synchrony assumptions, i.e. in an asynchronous system augmented with an eventually perfect failure detector.*

This paper presents an efficient WME algorithm which outperforms previous solutions in terms of both communication latency and message complexity, while relying on minimal synchrony assumptions.

1. Introduction

Context. The distributed mutual exclusion problem (DME) [3], [5], [17] requires to define a distributed coordination algorithm aimed at resolving conflicts resulting from concurrent, distributed processes accessing a *single, indivisible* resource, that can only support one user at a time. An user accessing the resource is said to be in its critical section (CS), and the (safety) property guaranteeing that *at any time* at most one process is in its CS is known as *mutual exclusion*.

Very recently, the work in [16] has formalized a variant of DME, namely the Weak Mutual Exclusion problem (WME), which regulates the access to a *replicated* shared resource whose copies are locally maintained by every participating process. More in detail, the WME problem is derived by extending the classical DME specification in a twofold direction. On one hand, by explicitly modelling the interactions with the replicated shared resource to be accessed in mutual exclusion, which is viewed as a deterministic state machine. On the other hand, by relaxing the classical mutual exclusion property in order to detach it from the notion of real-time. Unlike the classical DME, in fact, in the WME problem a CS can be granted not only in case there is currently (i.e. at the same time) no other process in the CS, but rather as long as the whole sequence of established CSs can be reordered to yield a sequential history in which: 1) no two CSs overlap over time, 2) the order of establishment of the CSs and of the operations executed within the CSs on the (replicated) shared resource does not contradict the (partial) order in the original history, and 3) the state trajectories of the set of replicas of the shared resource are equivalent to a serial execution over a single copy of the resource. Further, the specification of the WME problem allows aborting already established CS instances in case of failure suspicions. In this case, the process is ejected from its CS and any pending operation must fail (i.e. not be executed by any process) delivering an explicit notification to the application level.

Related Research. Over the years, the mutual exclusion problem has been investigated both in the failure-free model [15], [18] and under the assumption that the processes accessing the shared resource can fail according to the crash-failure model [1], [13]. The large number of DME algorithms proposed in literature are typically classified into two main categories [14], [17]: token-based or permission-based. In token-based algorithms, e.g. [15], [18], only the site holding the token can execute the CS and make the final decision on the next site to enter the CS. In permission-based algorithm, e.g. [3], [12], a requesting site can execute the CS only after it has received permission from each member of a subset of sites in the system and every site receiving a CS request message participates in making the final decision.

Independently of the chosen algorithmic approach, the DME problem is solvable, in presence of failures, only if processes are able to accurately distinguish unresponding processes from crashed ones [5] (i.e. without the possibility of *any* mistake). Unfortunately, phenomena such as network congestions, partitions or nodes' overloads, can make it very hard, or even impossible, to guarantee perfectly accurate failure detections [4] in real-life distributed systems. On the other hand, the WME problem was shown to be solvable in presence of much more relaxed synchrony assumptions, namely the availability of an eventually perfect failure detector, or $\diamond$P [4]. The $\diamond$P failure detector is allowed to generate an unbounded number of false failure suspicions, and is required to ensure accurate failure indications only eventually, i.e. after an unknown, yet finite, time. In other words, with respect to the classical DME problem, the WME is solvable in much more general (and realistic) system models such as the partially synchronous (also named eventually synchronous) one [6], in which the bounds on communication latency and relative process speed are only guaranteed to hold after some unknown time.

Contribution. In this paper we introduce a novel WME algorithm which sensibly outperforms the one presented in [16], while still tolerating a minority of faulty processes and relying on the weakest synchrony assumptions required for solving WME, i.e. the availability of an eventually perfect failure detector [4].

Unlike the WME algorithm in [16], which triggers a (costly) consensus instance [7] for entering and exiting from the critical section, as well as to issue operations, the algorithm presented in this paper resorts to using consensus exclusively in the case of failure suspicions of the process currently in the CS. In absence of failures, the proposed algorithm relies on an efficient token-asking, broadcast-based coordination scheme [14], which resembles those employed by classical DME's algorithms [18], [15]. This allows the presented WME algorithm to sensibly outperform the WME algorithm previously presented in [16], in terms of both message complexity and communication latency. On the other hand, unlike classical DME token-based algorithms, which require to enforce the uniqueness of the token *at any time*, the presented WME algorithm is designed to tolerate the simultaneous coexistence of multiple tokens, possibly generated by the occurrence of false failure suspicions.

Structure of the paper. Section 2 formalizes the considered system model. The the WME's specification is recalled in Section 3. Section 4 presents the WME algorithm, and discusses its correctness. An analysis of its performance is provided in Section 5.

2. System Model

We consider a classical crash-prone asynchronous message passing system model consisting of a set of n processes $\Pi = \{1, \ldots, n\}$ $(n > 1)$. We do not consider Byzantine failures: a process either correctly executes the algorithm assigned to it, or crashes and stops forever. We denote the crash of a process with the event $crash_i$. A process that does not crash is said to be $correct$. Communication takes place over reliable channels guaranteeing that messages are eventually delivered by the intended receiver, unless either the sender or the receiver crashes. We assume the existence of a discrete global clock whose ticks are the set of natural numbers $\mathbb{N}$. The time instant in which an event e is generated is denoted as $\mathcal{T}(e)$. Note that the global clock is a fictional device to which processes have no direct access.

The system is augmented with a distributed failure detector oracle [4], in the sense that every process has access to a local failure detector module that provides hints on the set of crashed processes. The algorithm described in this paper relies on an Eventually Perfect failure detector ($\Diamond$P), namely the weakest failure detector for solving WME [16], which is specified by the following two properties [4]: every crashed process is eventually suspected by every correct process (*Strong completeness*) and, there is a time after which every correct process is not suspected by every correct process (*Eventual strong accuracy*).

Users, stubs, and shared resource replicas. Each process i hosts a user u_i, a stub s_i, and a replica of the shared resource r_i. A user u_i only interacts with its local stub s_i to request exclusive access to the shared resource and issue operations on it. The stub s_i acts as a wrapper on the local replica of the shared resource r_i and coordinates with the other processes to ensure that the operations executed on r_i are equivalent to an execution over a single copy of the shared resource (which is required to be consistent with the order of establishment of the mutual exclusion). Users, stubs and replicas of the shared resources are modeled as deterministic automata that communicate by exchanging input and output events [5]. A stub s_i interacts with its local replica r_i of the replicated resource through the following classes of events from the domain $SRevents$:

i) $invoke_i[op]$ is an input event of r_i (resp. output event for s_i), which triggers the execution of the operation $op \in Operations$, where $Operations$ is the set of admissible operations for the replicated shared resource automaton. We assume each op to be univocally identifiable (this is accomplishable by associating an unique id with each operation, which we omit to simplify presentation).

ii) $response_i[op, res]$ is an output event of r_i (resp. input event for s_i) which notifies the stub about the result $res \in Results$ produced by the execution of a previously issued operation op on r_i, where $Results$ is the set of possible results that the shared resource automaton can produce in output.

The interaction with a replica r_i is assumed non-blocking, i.e. if r_i receives a $invoke_i[op]$ event it eventually generates the corresponding $response_i[op, res]$ unless process i crashes. A user u_i and its local stub s_i interact using the following six classes of events from the domain $USevents$:

i) try_i is an input event of s_i (resp. output event of u_i) which indicates the wish of u_i to enter its CS. In this case we say that i *volunteers*.

ii) $crit_i[CS\_id]$, where $CS\_id \in \mathbb{N}$, is an input event of u_i (resp. output event of s_i) which is used by s_i to grant u_i access to the critical section instance CS_id.

iii) $issue_i[CS\_id, op]$, where $CS\_id \in \mathbb{N}$ and $op \in Operations$, is an input event of s_i (resp. output event of u_i), which is used by u_i to issue an operation op on the shared resource.

iv) $outcome_i[CS\_id, op, res]$, where $CS\_id \in \mathbb{N}$, $op \in Operations$ and $res \in Results$, is an input event of u_i (resp. output event of s_i) which notifies the result res of the execution of operation op by r_i.

v) $exit_i[CS\_id]$ is an input event of s_i (resp. output event of u_i) which indicates the wish of u_i to leave the critical section instance CS_id. In this case we say that i *resigns*.

vi) $rem_i[CS\_id]$ is an input event of u_i (resp. output event of s_i) which notifies u_i that it can continue its work out of its critical section instance.

vii) $ejected_i[CS\_id]$ is an input event of u_i (resp. output event of process s_i) which notifies u_i that s_i was forced to exit from the critical section CS_id.

An operation that was issued by a user u_i through a $issue_i[CS\_id, op]$ event, and which is followed neither by the corresponding $outcome_i[CS\_id, op, res]$ event, nor by an $ejected_i[CS\_id]$ event is called a *pending* operation. If s_i generates an $outcome_i[CS\_id, op, res]$ event for a pending operation, we say that the operation was successfully executed, or simply succeeded. If s_i generates an $ejected_i[CS\_id]$ event for a pending operation, we say that the operation failed to execute, or simply failed.

An event e is said to be *associated with a CS instance* CS_id iff i) e is an event exchanged between a user and a process (i.e., $e \in USevents$), and ii) e is either the try event that determined the establishment of the CS instance CS_id or e has CS_id as the value of its CS instance identifier parameter. The events $issue_i[CS\_id, op]$ and $invoke_i[op']$ (resp. $outcome[CS\_id, op, res]$ and $response_i[op', res]$) are said to be *correlated* iff $op = op'$, i.e. they are associated with the same operation op.

Histories and Sub-Histories. A history H is the (possibly infinite) sequence of events produced by the automatons of the system (i.e., users, processes and replicas of the shared resource), including any process crashes event. A history H induces a time-based irreflexive partial ordering relation $<_H^{\mathcal{T}}$ on its events:

$$e_0 <_H^{\mathcal{T}} e_1 \Leftrightarrow \mathcal{T}(e_0) < \mathcal{T}(e_1)$$

A *process subhistory*, $H|i$ (H at i), of a history H is the subsequence of all events in H generated by process i. We define the *user-stub subhistory* H^{US} as the subsequence of the history H restricted to the events exchanged between stubs and users, i.e. H^{US}=$H \cap USevents$. Analogously, the *stub-resource subhistory* H^{SR} is defined as the subsequence of the history H restricted to the events exchanged between stubs and replicas of the shared resource, i.e. H^{SR}=$H \cap SRevents$. The subsequence of the user-stub subhistory H^{US} restricted to the $issue_i[CS\_id, op]$ and $outcome[CS\_id, op, res]$ events is called the *user-stub successful operations subhistory* and denoted as $H^{US|op}$. We define a *CS instance subhistory*, H_{id}^{CS}, as the subsequence of the user-process subhistory H^{US} restricted to the events associated with CS instance id. We define the $init$ event of a CS instance subhistory H_{id}^{CS}, as the try event in H that determined the establishment of CS instance id, and denote it as $\mathcal{I}(H_{id}^{CS})$. The $final$ event of a CS instance subhistory H_{id}^{CS}, denoted with $\mathcal{F}(H_{id}^{CS})$, is defined as the first event in the set $\{rem_i[CS\_id], eject_i[CS\_id], crash_i\}$ in H.

Well-formed CS instance subhistories. A *CS instance subhistory*, H_{id}^{CS} is said to be *well-formed* if and only if it is a prefix of the cyclically ordered sequences $\mathcal{S}^1$ or $\mathcal{S}^2$, where $\mathcal{S}^1$ is defined as $\mathcal{S}^1 := (try_i\ crit_i[id]\ \texttt{OPS}\ exit_i[id]\ rem_i[id])$, $\texttt{OPS}$ being any sequence of $issue_i[CS\_id, op]$ and $outcome_i[CS\_id, op, res]$ events generated by the following context free grammar:

$$\texttt{OPS} := (issue_i[id, op]\ outcome_i[id, op, res]\ \texttt{OPS} \mid \varepsilon)$$

and $\mathcal{S}^2$ is defined as $\mathcal{S}^2 := (\ try_i\ crit_i[id]\ \texttt{INT\_OPS}\ )$, $\texttt{INT\_OPS}$ being any sequence of $issue_i[id, op]$, $outcome_i[id, op, res]$ and $ejected_i[id]$ events generated according to the following context free grammar:

$$\texttt{INT\_OPS} := (\ issue_i[id, op]\ outcome_i[id, op, res]\ \texttt{INT\_OPS} \mid issue_i[id, op]\ ejected_i[id] \mid ejected_i[id])$$

Informally, any well-formed CS instance subhistory H_{id}^{CS} starts with the establishment of a new critical section instance, through the try_i-$crit_i[CS\_id]$ events. Once entered the critical section instance, u_i can (sequentially) issue an arbitrary number (possibly null) of operations, through the $issue_i[CS\_id, op]$ events. In case u_i is not ejected by its CS, it can explicitly resign through the $exit_i[CS\_id]$, $rem_i[CS\_id]$ events.

Complete CS instance subhistories. A well-formed CS instance subhistory H_{id}^{CS} is *complete* iff: i) it has no pending operations, and ii) the CS instance is concluded via either a voluntarily resignation or an ejection or a crash, formally $\mathcal{F}(H_{id}^{CS}) \neq \emptyset$.

A *legal completion* of a well-formed history H is a well-formed history obtained by completing or deleting any not complete CS instance subhistory H_{id}^{CS} by adding or removing events from H according to the following rules:

1) if $H_{id}^{CS} = \{try_i\}$ then either append a $crit_i[id]$ event or remove $crit_i$, deleting the whole CS instance subhistory,

2) for any pending operation op issued by user u_i within CS instance CS_id, append zero or more $invoke_i[op]$, $response_i[op, res]$ and $outcome_i[CS\_id, op, res]$ correlated events, preserving its well-formedness,

3) if, after applying rules 1 and 2, H_{id}^{CS} is not empty, append either an $eject_i[id]$ event, or the pair of events $exit_i[id]$-$rem_i[id]$, or the $rem_i[id]$ event so to complete it while preserving its well-formedness.

Equivalent and Isomorphic Histories. Two histories H and H' are said *equivalent* if for every process $i \in \Pi$ $H|i$=$H'|i$. A stub-resource subhistory H^{SR} is *isomorphic* to a user-stub successful operations subhistory $H^{US|op}$ iff:

A) a bijection $\mathcal{M}$ exists from H^{SR} to $H^{US|op}$ such that $\forall e \in H^{SR}$ and $\forall e' \in H^{US|op}, \mathcal{M}(e) = e' \Leftrightarrow e$ and e' are correlated events, and

B) $\mathcal{M}$ is an *order isomorphism* with respect to $<_H^{\mathcal{T}}$, i.e. $\forall \{e_0, e_1\} \in H^{SR}, e_0 <_H^{\mathcal{T}} e_1 \Leftrightarrow \mathcal{M}(e_0) <_H^{\mathcal{T}} \mathcal{M}(e_1)$.

Informally, a stub-resource subhistory and a user-stub successful operations subhistory are isomorphic if each event in H^{SR} has a corresponding event in $H^{US|op}$ (and vice versa) (condition A above), and if the order of the $issue$ and $outcome$ events exchanged between the users and the stubs matches the order of the correlated $invoke$ and $response$ events exchanged between the stubs and the replicas of the shared resource (condition B above).

CS-sequential Histories. We define the irreflexive partial order $<_{\mathcal{H}}^{CS}$ on well-formed, complete CS instance subhistories of the history H as follows:

$$H_{id}^{CS} <_H^{CS} H_{id'}^{CS} \Leftrightarrow \mathcal{F}(H_{id}^{CS}) <_H^{\mathcal{T}} \mathcal{I}(H_{id'}^{CS})$$

A well-formed history H is *CS-sequential* iff $<_{\mathcal{H}}^{CS}$ is a total order relation for its user-stub subhistory H^{US}. Note that, by this definition, if a user-stub subhistory is *CS-sequential* then two CS instances never overlap over time. This is equivalent, in a sense, to the classical mutual exclusion property [5] (which requires that "No two processes are in the CS at the same time") except from that, unlike in the original DME problem, the "owner of the CS" can be, in our case, pre-empted by the delivery of an $ejected$ event.

3. The Weak Mutual Exclusion Problem

An algorithm solves the WME problem if, under the assumption that every user is well-formed, any run of the algorithm satisfies the following six properties [16]:

Safety Properties

Weak Mutual Exclusion: For every history H there exists a legal completion H_*, such that:

WME1: H_*^{US} is equivalent to a CS-sequential user-stub subhistory S.

WME2: $<_{H_*}^{CS} \subseteq <_S^{CS}$

WME3: the stub-resource subhistory H_*^{SR} is isomorphic to the user stub subhistory of S, S^{US}.

1CS: The stub-resource subhistory H_*^{SR} is equivalent to a serial execution on a single replica of the shared resource.

Well-formedness: For any $i \in \Pi$, the history describing the interaction between u_i and s_i is well-formed.

Liveness Properties

Starvation-Freedom A correct process i that volunteers eventually enters the critical section, if no other process stays forever in its critical section.

CS-Release Progress: If a correct process resigns, it enters its remainder section.

Operation Progress: If a correct process issues an operation, eventually this either fails or succeeds, and eventually all issued operations succeed.

The weak mutual exclusion property requires that the CS instance subhistories can be reordered (without violating the ordering of events in the original process subhistories) to yield a history in which no two CS instances overlap over time (*WME1*), while preserving the real time ordering of acquisitions of the critical sections (*WME2*). *WME3* constrains the order of execution of the operations on the replicas of the shared resource to be consistent with the execution order perceived by the user while interacting with its stub. Note that *WME* does not force processes to exchange mutual information on the state of the local copies of the shared resource, r_i, whose state trajectories could be therefore allowed to arbitrarily diverge. Such runs are ruled out by the *1CS* property which guarantees that the replicated shared resource's history is 1-copy serializable [2].

The liveness properties provide non-blocking guarantees on the establishment and release of the CS, as well as on the execution of operations on the replicated shared resource. Furthermore, to rule out trivial solutions which could constantly fail to execute operations, the *Operation Progress* property requires that after some unknown, but finite, time any issued operated is successfully executed.

4. The Algorithm

In this section we present a token-based algorithm which solves the WME problem using a $\diamond$P failure detector and tolerates a minority of process crashes.

Overview. A run of the algorithm evolves as a sequence of epochs, each one being univocally associated with a sequentially increasing identifier. An epoch starts with a *normal phase*, in which the stub executes the pseudo-code defined in Figure 1, and is (possibly) concluded by a *termination phase* in which the stub executes the termination protocol reported in Figure 2. The code executed in runs without any failure suspicion (i.e. or simply *nice* runs) resembles the classical token-passing broadcast-based DME algorithm in [15], extended in order to handle the execution of operations on the replicated resource. The termination protocol, on the other hand, relies on a consensus service[1] [7] to ensure that processes concluding the current epoch agree on a consistent global state prior to entering the normal phase of a fresh new epoch.

The switching between the normal and the termination phases is controlled by the $block$ variable, which is set to false as soon as the termination protocol is activated, disabling all the input events defining the stub's behavior during nice runs (see Figure 1), and is re-set to the true value only when the termination protocol is concluded (see Figure 2). Therefore, at each process, the normal and the termination phases never overlap. Further, a stub tags all its output messages with the current epoch identifier, and only accepts messages tagged with the current epoch identifier.

Local variables. Each stub s_i maintains the following local variables: $curOwner$, storing the token owner's identity, which is initialized by all processes with a common, predetermined, value, namely s_1; $reqId$, namely a sequentially increasing integer identifier used to tag the CS establishment requests; $granted$, namely a n-entries array, whose j-th entry keeps track of the latest CS instance already granted to s_j; $curSN$, a global sequence number which is used to impose a total order on the sequence of both i) invoked operations and ii) CS-instance subhistories; $reqs$, a FIFO-order queue storing the received CS establishment requests; $opHist$, which stores the ordered sequence of operations issued within an epoch; $lastIssuedOp$, storing the last operation to have been locally issued, but not yet invoked, if any; the current epoch number, $curEp$; a boolean flag, $block$, which inhibits

1. The consensus problem, which is solvable in our model, is specified by the following properties [7], [6]: i) *Validity:* Any value decided is a value proposed; ii) *Uniform Agreement:* No two correct processes decide differently; iii) *Termination:* Every correct process eventually decides, iv) *Integrity:* No process decides twice.

```
PID curOwner=s1;     // identity of the token owner
State state=IDLE;    // initialized to TOKEN_HELD only on s1
int reqId=0;         // id of the last issued CS request
int[n] granted={0,...,0}; // last CS granted ∀i ∈ Π
int curSN=0;              // current global sequence number
FIFOQueue reqs=∅;         // stores CS requests
FIFOQueue opHist=∅;       // ordered seq. of invoked operations
Operation lastIssuedOp =⊥;   // last pending operation
int curEp=0;              // current epoch number
boolean block=false; // set to true during termination phases

upon try_i ∧ ¬block do
  if ( state=TOKEN_HELD )
    state=CS_IDLE; crit_i;
  else
    broadcast[REQUEST, curEp, ++reqId];
    state=REQUESTING;

upon receive_i[REQUEST, epoch, reqId] from p_j
      where curEp=epoch ∧¬ block do
  if (granted[j] < reqId)
    if ( state=TOKEN_HELD )
      curOwner=j; state=IDLE;
      broadcast[GRANTED, curEp, j, reqId, curSN+1];
    else
      reqs.push([j,reqId]);

upon receive_i[GRANTED, epoch, newOwner, reqId, sn]
      where curEp=epoch ∧ sn=curSN+1 ∧¬ block do
  granted[newOwner]=reqId; curSN++;
  reqs.remove([newOwner,reqId]);
  curOwner=newOwner;
   if (curOwner=myself)
    state=CS_IDLE; crit_i;

upon exit_i ∧ ¬block do
  rem_i;
  if ( reqs≠ ∅ )
    [newOwner, newCSID] = reqs.pop();
    broadcast[GRANTED, curEp, newOwner, newCSID, curSN+1];
    curOwner=newOwner; state=IDLE;
  else
    state=TOKEN_HELD;

upon issue_i[CSid, op] ∧¬block do
  broadcast[INVOKE, curEp, op, curSN+1];
  state=CS_ISSUING; lastIssuedOp=op;

upon receive_i[INVOKE, epoch, op, sn]
      where curEp=epoch ∧ sn=curSN+1 ∧ ¬block do
    curSN++;
    opHist.push(op);
    broadcast [ACK, curEp, op, sn];

upon ¬block ∧ receive_i[ACK, epoch, op, sn] where
      ( curEp=epoch ∧ sn=curSN) from ⌊N/2⌋ + 1 procs. do
    invoke_i[op];
    wait result_i[op,res];
    if (CS_ISSUING)
      outcome_i[CSid,op,res];
      state=CS_IDLE; lastIssuedOp = ⊥;
```

Figure 1. Pseudo-code during nice-runs (stub s_i)

```
upon receive_i [NEWEP, ep, granted, sn, owner, opHist]
      where ep=curEp ∨ curOwner∈ ◇P_i do
block=true;
if ( curOwner∈ ◇P_i )
  broadcast[NEWEP,curEp,reqs,granted,curSN,s_i,opHist];
else
  broadcast[NEWEP,curEp,reqs,granted,curSN,curOwner,opHist];
    // collect a majority of NEWEP messages
Set S=∅;
while (|S| < ⌊N/2⌋ + 1)
  wait receive_i[NEWEP,ep',reqs',granted',sn',owner',opHist']
      where ep'=curEp;
  S=S∪[NEWEP, ep', reqs', granted', sn', owner', opHist'];
    // propose message with maximum sequence number
Msg m = msg∈S : ∀s ∈ S msg.sn ≥ s.sn;
propose([curEp, m.granted, m.sn, m.owner, m.opHist]);
wait decision([ep*, reqs*, granted*, sn*, owner*, opHist*])
      where ep*=curEp;

    // update local state according to consensus decision
curSN=sn*; granted=granted*; reqs=reqs*;
opHist=opHist*\opHist;
curEp++;
if ( state = REQUESTING )
  if ( owner* = myself )
    crit_i;
    state=CS_IDLE;
  else if ( [myself,reqId] ∉ reqs )
    broadcast[REQ, curEp, reqId];
else if ( state = CS_ISSUING )
  if ( opHist≠ ∅ )
    Operation op* = opHist.pop();
    invoke_i[op*];
    wait result_i[op*, res*];
    if ( op* = lastIssuedOp )
      outcome_i[CSid,op*,res];
      state = CS_IDLE;
      last_issued_op = ⊥;
  else if ( owner*=myself )
    broadcast [INVOKE, curEp, last_issued_op, curSN+1];

    // process remaining operations from the previous epoch
while ( opHist≠ ∅ )
  op† = opHist.pop();
  INVOKE_i[op†];
  wait result_i[op†,res†];

curOwner=owner*;
if (curOwner≠myself) ∧ (state=CS_IDLE ∪ state=CS_ISSUING)
    eject_i;
    state=IDLE;
else if (curOwner = myself) ∧ (state = IDLE)
  if (reqs≠ ∅) // test if there are requests to enter the CS
    [newOwner, newCSID] = reqs.pop();
    broadcast[GRANTED, curEp, newOwner, newCSID, curSN+1];
    curOwner=newOwner;
  else          // if there are no CS requests, retain the token
    state=TOKEN_HELD;

block=false;
```

Figure 2. Termination phase pseudo-code (stub s_i).

message reception during epoch changes; the $state$ variable, storing values in the domain $\{IDLE, REQUESTING, TOKEN\_HELD, CS\_IDLE, CS\_ISSUING\}$, which is initialized to $IDLE$ on all processes except on the initial token owner, s_1, where it is set to the $TOKEN\_HELD$ value. The possible evolutions of the $state$ variable are shown in Figure 3 and described in the following.

Behavior during nice runs. The mechanisms underlying the establishment and the release of the CS are analogous to those employed in [15]. To establish a new CS instance, a stub s_i needs to first establish the ownership of token. If s_i already owns the token, i.e. it is in the $TOKEN\_HELD$ state, a new CS can be immediately established upon the reception of a try_i event. Otherwise, s_i enters the $REQUESTING$ state, increases the $reqId$ value and broadcasts a REQUEST message tagged with the current $reqId$ value. Upon reception of a REQUEST message at s_j, the $granted$ vector is used to determine whether the incoming CS request has already been served. In such a case, if s_j is not the current token owner, the request is just added to the $reqs$ queue. On the other hand, if s_j is the token owner and is currently out of the CS, i.e. s_j is in the $TOKEN\_HELD$ state, it updates its state to the $IDLE$ value and transfers the token by broadcasting a GRANTED message tagged with the $curSN + 1$ sequence number, the identity of the new token owner, and the $reqId$ value associated with the enabled CS request. The reception of a (not obsolete) GRANTED message at stub s_k triggers the update of the k-th entry of the $granted$ vector, the increase of the global sequence number, as well as the removal of the corresponding CS request from the $reqs$ queue (if this is

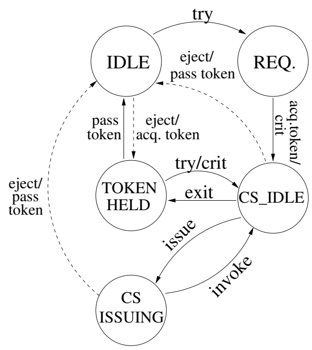

Figure 3. Stub's state machine (transitions occurring in the termination phase shown in dashed lines).

already present). Furthermore, the requesting stub s_k enters the CS and updates its state variable to the CS_IDLE value.

As u_i generates an $exit_i$ event, s_i immediately responds with a rem_i event and, depending on whether there are pending requests for acquiring the CS in the $reqs$ queue or not, the token is, respectively, either transferred (using the same mechanism above described) to the stub whose request is first in queue, or locally retained, setting the state value to $TOKEN\_HELD$.

Upon the issuing of an operation, a stub enters the $CS\_ISSUING$ state, records the operation in $lastIssuedOp$ and broadcasts an INVOKE message carrying the operation along with the $curSN + 1$ sequence number. Upon reception of an INVOKE message tagged with the $sn = curSN + 1$, a stub appends the operation to $opHist$, increases $curSN$ and broadcasts back an ACK message tagged with the updated $curSN$ value. In order for an operation to be invoked on the local copy of the shared resource, a stub waits to gather a majority of ACK messages tagged with a sequence number equal to the local value of $curSN$.

Termination protocol. The termination protocol is activated as soon as the token owner is suspected to have crashed. In this case a stub s_i sets the $block$ variable to false and broadcasts a NEWEP message conveying its CS requests queue req, the $granted$ vector, the $curSN$, the sequence of operations stored in $opHist$, as well his own identity, i.e. i, signaling his intention to candidate itself as the initial token owner for the next epoch. The reception of a NEWEP message at s_j triggers, in its turn, the broadcast of a NEWEP message. In this case, however, if s_j is not suspecting the current token owner, say s_k, s_j specifies s_k's identity, rather than his own, in the NEWEP message (i.e. s_j does not try to eject s_k from the CS).

Henceforth, independently of whether the termination protocol was activated by the suspect of failure of the current token owner, or by the reception of a NEWEP message, the stub's behavior is identical. First, the stub waits for a majority of NEWEP messages. Among the received messages, the message carrying the largest sequence number value is selected (if more than one message is tagged with the same, largest sequence number, one of these messages is randomly selected), and is proposed as input value to the consensus service. Next, the stub waits for consensus termination. As consensus outputs a decision value, s_i accordingly updates his local variables $curSN$, $granted$ and $reqs$. Then, it determines whether there are operations included in the $opHist*$ output by consensus which have not yet been executed on the local copy of the resource.

If s_i was requesting to enter the CS, it first checks whether he has now been assigned the token ownership. In this case the stub allows the user to enter the CS and accordingly updates its state variable to the CS_IDLE value. If s_i has not become the token owner and his CS request was not included in the $reqs$ queue output by consensus, s_i re-broadcasts it with an increased epoch number to make sure that it is considered in the new epoch.

Alternatively, if s_i was issuing an operation while the termination protocol was triggered, it checks whether any of the operations in $opHist*$ has not been locally executed yet. If so, s_i executes the first of such operations and if this coincides with his own pending operation stored in $lastIssuedOp$, the corresponding $outcome$ event is delivered to u_i and the CS_IDLE state is entered. On the other hand, if there are no operations in $opHist*$ to be executed but s_i results to still be the token owner, s_i re-broadcasts his last issued operation in the new epoch.

Next s_i processes any remaining operation and, before completing the termination phase, it verifies if the consensus round has determined an alteration of the token ownership which requires either i) the generation of an $eject_i$ event towards u_i (this happens in case s_i was previously in the CS and lost the token ownership during the epoch change), or ii) immediately transferring the token ownership to the first process in the $reqs$ queue (in case s_i was not requesting the CS in the former epoch, it has now been assigned the token ownership, and req is not empty), or iii) just a simple update of the $state$ variable to the $TOKEN\_HELD$ value (if s_i was not requesting the CS in the former epoch, it has now become the token owner, and req is empty).

As a final remark, note that, for simplicity of presentation, in the algorithm's pseudo-code the local $opHist$ variable is explicitly garbage collected only during the execution of a termination phase. However, in order to timely prune unneeded information from $opHist$, one could rely on classical stability detection schemes, e.g. [8], aimed at informing each stub about the sequence of operations already invoked in the current epoch by all processes. All such operations can in fact be immediately garbage collected from $opHist$.

4.1. Correctness Arguments

For space constraints it is not possible to present a formal correctness proof with respect to the whole set of properties defining the WME problem. However, we provide a sketch of proof aimed at giving some insights on the algorithm's correctness. Our argumentation is structured as follows. We first discuss why the algorithm guarantees the WME properties in nice runs. Then we analyze the algorithm' s dynamics in case of failure suspicions.

Nice runs. First let us derive a legal completion H_* of any incomplete history H generated by a nice run using the following rules: i) if there is any stub s_i in the CS, then append a $crash_i$ event to H; ii) for any stub s_j in the $REQUESTING$ state, delete the last try_j event from H.

Next, let us recall that in absence of failure suspicions the presented algorithm can be viewed as a modular extension of the token-based DME algorithm in [15], extended to include the logic for the management of the operations issued on the replicated resource. Since in nice runs the algorithm in [15] guarantees mutual exclusion, it follows that H_* must be CS-sequential (hence WME1 holds). Also, since H_* is obtained from H without altering the ordering of any event, WME2 trivially follows. Further, each stub s_i issues (the same set of) operations on its local resource r_i according to the (total) order specified by the global sequence number $curSN$, whose advancement is determined exclusively by the stub in CS either when it issues a new operation, or when it grants the token ownership (and hence the CS) to some other process. At the light of these considerations, it can be shown that i) there can be no mismatch between the order of execution of operations observed by a user u_i and the order of invocations on the resource r_i (hence WME3 holds), as well as that ii) the sequence of operations invoked on any resource replica produces the same results that would have been produced if it had been executed on a single copy of the resource (hence 1CS holds).

The fact that, in nice runs, the establishment and the release of the CS are handled using mechanisms analogous to those in [15] explains why the Starvation-Freedom and CS-Release progress properties (which are common to both the WME and DME problems) hold also for our algorithm. Finally, operation progress is ensured since, in nice runs, all messages sent are received by the designated recipient, making the distributed acknowledgment phase associated with operations issuing non-blocking and precluding the possibility for any operation to fail.

Runs including failures (or failure suspicions). It can be easily observed that failures of a process that is not owning the token can not endanger the correctness of the algorithm (as long as a majority of processes is correct). In case of crash of the token owner, by the completeness property of $\diamondsuit$P, we get that eventually some process s_i enters the termination phase, broadcasts a NEWEP message, and waits for a majority of replies before starting consensus. If s_i is correct, since a majority of processes is correct and channels are reliable, s_i will deliver its NEWEP to at least a majority of processes. On the other hand, if s_i were faulty, eventually some correct process would suspect the token owner, leading to the same result: a majority of processes $\mu \subseteq \Pi$ switching from the normal to the termination phase. Hence, some correct process s_k will eventually propose a value to consensus. Further, since s_k is correct, its NEWEP messages will be eventually received by all other correct processes, which will all activate the termination protocol and propose a value to consensus. This suffices to guarantee termination of consensus.

Now let sn'' be the largest sequence number associated with an operation contained in the $opHist''$ selected by a stub s_j after gathering NEWEP messages for epoch e from any majority $\mu' \subseteq \Pi$ of processes. Since all processes in μ' have switched from the normal phase to the termination phase, then no operation with sequence number larger than sn'' can be invoked by any process in the normal phase of epoch e. In fact, at most a minority of processes can still be in the normal phase of epoch e, whereas, for an operation to be invoked, a majority of processes must have first acknowledged the corresponding INVOKE message. Further, since any two majorities of processes necessarily intersect, it follows that $opHist''$ must contain *all* the operations invoked by at least one process in the normal phase of epoch e.

Hence, all processes that complete the termination phase for epoch e execute in the same order the same set of operations. Also, these processes will update their local variables to reflect the common, consistent, global state determined by the consensus decision. Further, as already discussed, upon crash of the current token owner, all correct processes activate and complete the termination phase, entering the following epoch.

In other words, the completion of an epoch and the starting of a fresh new one represents a regeneration point in which the set of alive processes is brought back to a state equivalent to the initial one of epoch 0. Further, by the eventually strong accuracy property of $\diamondsuit$P, we get that eventually the whole set of correct processes will start a new epoch (with a consistent initial state) where no failure suspicions occur. As discussed above, in such conditions (i.e. in nice runs), the algorithm ensures WME.

5. Performance analysis

In this section we discuss the performances of the presented WME algorithm and contrast them with those of the WME algorithm presented in [16]. We consider two classical metrics for evaluating the performance of distributed algorithms, namely latency (in terms of communication steps delay) and message complexity. Our analysis is focused on nice runs as these are the most likely in practice.

In the algorithm presented in Section 4, which we refer to as WME-1, the latency for entering the CS is either 0, if the process is already the token owner, or 2, if the process needs to acquire the token ownership. The algorithm

```
upon ¬block ∧ receive_i[INVOKE, epoch, op, sn]
        where ( curEp=epoch ∧ sn=curSN+1 ) do
    curSN++;
    opHist.push(op);
    send [ACK, curEp, op, sn] to curOwner;

upon ¬block ∧ receive_i[ACK, epoch, op, sn]
        where ( curEp=epoch ∧ sn=curSN ) from ⌊N/2⌋ + 1 procs. do
    broadcast [DOINVOKE, curEp, op, sn];

upon ¬block ∧ receive_i[DOINVOKE, epoch, op, sn] from curOwner
        where ( curEp=epoch ∧ sn=curSN ) do
    invoke_i [op];
    wait result_i [op,res];
    if (CS_ISSUING)
        outcome_i [CSid,op,res];
        state=CS_IDLE; lastIssuedOp = ⊥;
```

Figure 4. A variant of the algorithm in Fig. 1 relying on a centralized acknowledgment scheme for operations' invocation

correspondingly exchanges either 0 or O(N) messages. The release of the CS is immediate and is possibly followed by a broadcast advertising the change of the token ownership. Finally, the invocation of an operation requires 2 communication steps and the exchange of $O(N^2)$ messages (the broadcast of an INVOKE message, followed by a distributed acknowledgment phase in which each process broadcasts back an ACK message). Actually, the handling of the operation invocation in the WME-1 algorithm can be relatively straightforwardly transformed to yield a lower, O(N), message complexity, at the cost of an additional communication step latency. Figure 4 shows exactly such a variant of the WME-1 algorithm, which we refer to as WME-2 (only the code related to the operation invocation is shown as the remaining is unchanged). WME-2 relies on a centralized acknowledegment scheme in which ACKs messages are not broadcast, as in the WME-1 algorithm, but only sent to the current CS Owner. The latter waits to gather a majority of ACKs before broadcasting a DOINVOKE message whose reception triggers the actual execution of the operation on the replicas of the shared resource.

The algorithm in [16] relies on a common scheme for establishing and releasing the CS, as well as to issue operations: the execution of a consensus instance, preceded by a preliminary reliable broadcast aimed at spreading the consensus proposal value. As shown in [10], consensus algorithms can be designed to provide optimal performances either in terms of latency or of message complexity. In the former case, consensus can be solved in 2 communication steps exchanging $O(N^2)$ messages. Alternatively, the message complexity can be linear at the cost of at least an additional communication step. The second and third rows of Table 1 analyze the performances of the algorithm [16] when employing, respectively, a consensus algorithm achieving optimal communication latency, such as the one in [11], rather than linear message complexity, as, e.g., for the solution [9].

Table 1 summarizes the performance of the analyzed WME algorithms, clearly highlighting the significant advantages arising from the algorithm presented in this paper in terms of both communication steps latency and message complexity.

	Enter CS		Exit CS		Invoke Op.	
	Lat.	Msgs	Lat.	Msgs	Lat.	Msgs
WME-1	0/2	O(N)	0	0/O(N)	2	$O(N^2)$
WME-2	0/2	O(N)	0	0/O(N)	3	O(N)
[16]+[11]	3	$O(N^2)$	3	$O(N^2)$	3	$O(N^2)$
[16]+[9]	4	O(N)	4	O(N)	4	O(N)

Table 1. Performance comparison with the algorithm in [16].

References

[1] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, 1991.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[3] O. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Comm. of the ACM*, 26(2):146–147, 1983.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[5] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005.

[6] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[7] R. Guerraoui and A. Schiper. Consensus: The big misunderstanding. In *Proc. of the Workshop on Future Trends of Distributed Computing Systems*, pages 183–188. IEEE Computer Society, 1997.

[8] K. Guo and I. Rhee. Message stability detection for reliable multicast. In *Proc. of IEEE INFOCOM*, pages 814–823, 2000.

[9] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[10] L. Lamport. Lower bounds on consensus, 2000.

[11] L. Lamport. Fast paxos. *Distributed Computing*, 9(2):79–103, 2006.

[12] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[13] D. Manivannan and M. Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *Proc of the Conference on Parallel and Distributed Computing*, pages 525–530, 1994.

[14] M. Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2):47–50, 1991.

[15] G. Ricart and A. Agrawala. Author response to ön mutual exclusion in computer networks̈ by carvalho and roucairol. *Comm. ACM*, 26(2):147–148, 1983.

[16] P. Romano, L. Rodrigues, and N. Carvalho. The weak mutual exclusion problem. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2009.

[17] M. Singhal. A taxonomy of distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 18(1):94–101, 1993.

[18] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4):344–349, 1985.