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.

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 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 and bind it to the concept of logical time. Unlike the classical DME, 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 [17, 14]: 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 ♠P [4]. The ♠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 specification of the WME problem 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. Finally, section 6 concludes the paper.

2 System Model

We consider a classical crash-prone asynchronous message passing system model consisting of a set of n processes Π = {1, . . . , 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 crashi. 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 and take the range N of the clocks ticks to be the set of natural numbers. The time instant in which an event e is generated is denoted as 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 (EP), 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 ui, a stub si, and a replica of the shared resource ri. A user ui only interacts with its local stub si to request exclusive access to the shared resource and issue operations on it. The stub si acts as a wrapper on the local replica of the shared resource ri and coordinates with the other processes to ensure that the operations executed on ri 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 replica of the shared resources are modeled as deterministic automata that communicate by exchanging input and output events [5]. A stub si interacts with its local replica ri of the replicated resource through the following classes of events from the domain SRevents:

i) invokei[op] is an input event of ri (resp. output event for si), which triggers the execution of the operation op ∈ 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) responsei[op, res] is an output event of ri (resp. input event for si) which notifies the stub about the result res ∈ Results produced by the execution of a previously issued operation op on ri, where Results is the set of possible results that the shared resource automaton can produce in output.

The interaction with a replica ri is assumed non-blocking, i.e. if ri receives a invokei[op] event it eventually generates the corresponding responsei[op, res]. A user ui and its local stub si interact using the following six classes of events from the domain USevents:

i) tryi is an input event of si (resp. output event of ui) which indicates the wish of ui to enter its CS. In this case we say that i volunteers.

ii) criti[CSi]d, where CSi ∈ N, is an input event of ui (resp. output event of si) which is used by si to grant ui access to the critical section instance CSi. 

iii) issuei[CSi, op], where CSi ∈ N and op ∈ Operations, is an input event of si (resp. output event of ui), which is used by ui to issue an operation op on the shared resource.

iv) outcomei[CSi, op, res], where CSi ∈ N, op ∈ Operations and res ∈ Results, is an input event of ui (resp. output event of si) which notifies the result res of the execution of operation op by ri.

v) exiti[CSi]d is an input event of si (resp. output event of ui) which indicates the wish of ui to leave the critical section instance CSi. In this case we say that i resigns.

vi) remi[CSi]d is an input event of ui (resp. output event of si) which notifies ui that it can continue its work out of its critical section instance.

vii) ejectedi[CSi]d is an input event of ui (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. $outcome_{i}[CS_{id}, op, res]$ 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 automata 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 $<_{H}$ on its events:

$$e_0 <_{H} e_1 \Leftrightarrow T(e_0) < T(e_1)$$

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-subst subhistory $H_{U}^{S}$ as the subsequence of the history $H$ restricted to the events exchanged between stubs and users, i.e. $H_{U}^{S} = H \cap USevents$. Analogously, the stub-resource subhistory $H_{SR}^{S}$ is defined as the subsequence of the history $H$ restricted to the events exchanged between stubs and replicas of the shared resource, i.e. $H_{SR}^{S} = H \cap SRevents$. The subsequence of the user-subst subhistory $H_{U}^{S}$ restricted to the $issue_{i}[CS_{id}, op]$ and $outcome_{i}[CS_{id}, op, res]$ events is called the user-subst successful operations subhistory and denoted as $H_{US}^{op}$. We define a CS instance subhistory, $H_{CS}^{S}$, as the subsequence of the user-process subhistory $H_{U}^{S}$ restricted to the events associated with CS instance $id$. We define the init event of a CS instance subhistory $H_{CS}^{S}$, as the try event in $H$ that determined the establishment of CS instance $id$, and denote it as $I(H_{CS}^{S})$. The final event of a CS instance subhistory $H_{CS}^{S}$, denoted with $F(H_{CS}^{S})$, 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, $H_{CS}^{S}$, is said to be well-formed if and only if it is a prefix of the cyclically ordered sequences $S^1$ or $S^2$, where $S^1$ is defined as $S^1 := \{try_{i}, crit_{i}[id] \text{ OPS } exit_{i}[id] \text{ rem}_{i}[id]\}$, $OPS$ being any sequence of $issue_{i}[CS_{id}, op]$ and $outcome_{i}[CS_{id}, op, res]$ events generated by the following context free grammar:

$$OPS := (\text{issue}_{i}[id, op] \text{ outcome}_{i}[id, op, res] \text{ OPS } | \varepsilon$$

and $S^2$ is defined as $S^2 := (\text{try}_{i}, \text{crit}_{i}[id] \text{ INT}_{OP}$, $\text{INT}_{OPS}$ 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:

$$\text{INT}_{OPS} := (\text{issue}_{i}[id, op] \text{ outcome}_{i}[id, op, res] \text{ INT}_{OPS} | \text{issue}_{i}[id, op] \text{ ejected}_{i}[id] | \text{ejected}_{i}[id])$$

Informally, any well-formed CS instance subhistory $H_{CS}^{S}$ starts with the establishment of a new critical section instance, through the $\text{try}_{i}, \text{crit}_{i}[CS_{id}]$ 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 $H_{CS}^{S}$ 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 $T(H_{CS}^{S}) \neq \emptyset$.

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 $H_{CS}^{S}$ by adding or removing events from $H$ according to the following rules:

1) if $H_{CS}^{S} = \{\text{try}_{i}\}$ then either append a $\text{crit}_{i}[id]$ event or remove $\text{crit}_{i}$, 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 $\text{invoke}_{i}[op]$, $\text{response}_{i}[op, res]$ and $\text{outcome}_{i}[CS_{id}, op, res]$ correlated events, preserving its well-formedness,

3) if, after applying rules 1 and 2, $H_{CS}^{S}$ is not empty, append either an $\text{ejected}_{i}[id]$ event, or the pair of events $\text{exit}_{i}[id] - \text{rem}_{i}[id]$, or the $\text{rem}_{i}[id]$ event so to complete it 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 $H_{SR}^{S}$ is isomorphic to a user-subst successful operations subhistory $H_{US}^{op}$ iff:

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

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

Informally, a stub-resource subhistory and a user-subst successful operations subhistory are isomorphic if each event in $H_{SR}^{S}$ 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 \(<_H^{CS}\) on well-formed, complete CS instance subhistories of the history \(H\) as follows:

\[
H^C_S <_H^{CS} H^C_T \Leftrightarrow \mathcal{F}(H^C_S) <_T \mathcal{I}(H^C_T)
\]

A well-formed history \(H\) is *CS-sequential* iff \(<_H^{CS}\) 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 correctness properties [16]:

**Safety Properties**

**Weak Mutual Exclusion:** For every history \(H\) there exists a legal completion \(H^*_c\), such that:

WME1: \(H^C_S <_H^{CS} H^C_T \Leftrightarrow \exists \mathcal{F}(H^C_S) <_T \mathcal{I}(H^C_T)\)

WME2: \(<_H^{CS} \subseteq <_S^{CS}\)

WME3: the stub-resource subhistory \(H^R_S\) is isomorphic to the user stub subhistory of \(S\).

1CS: The stub-resource subhistory \(H^*_S\) is equivalent to a serial execution on a single replica of the shared resource.

**Well-formedness:** For any \(i \in I\), 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 at after some unknown, but finite, time any issued operation is successfully executed.

4 The Algorithm

In this section we present a token-based algorithm which solves the WME problem using a DP failure detector and tolerating a majority 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.

\(^1\)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.
Figure 1: Pseudo-code during nice-runs (stub $s_1$)

**Local variables.** Each stub $s_1$ 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, which inhibits 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 $s_1$ needs to first establish the ownership of token. If $s_1$ 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 $try$ event. Otherwise, $s_1$ 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 $s_1$, 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 \( \text{reqs} \) 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 \textit{TOKEN HELD} state, it updates its state to the \textit{IDLE} value and transfers the token by broadcasting a \textit{GRANTED} message tagged with the \( \text{curSN} + 1 \) sequence number, the identity of the new token owner, and the \( \text{reqId} \) value associated with the enabled CS request. The reception of a (not obsolete) \textit{GRANTED} message at stub \( s_k \) triggers the update of the \( k \)-th entry of the \textit{granted} vector, the increase of the global sequence number, as well as the removal of the corresponding CS request from the \( \text{reqs} \) queue (if this is already present). Furthermore, the requesting stub \( s_k \) enters the CS and updates its state variable to the \textit{CS IDLE} value.

As \( u_i \) generates an \textit{exit} event, \( s_i \) immediately responds with a \textit{rem} event and, depending on whether there are pending requests for acquiring the CS in the \( \text{reqs} \) 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 \textit{TOKEN HELD}.

Upon the issuing of an operation, a stub enters the \textit{CS ISSUING} state, records the operation in \textit{lastIssuedOp} and broadcasts an \textit{INVOKE} message carrying the operation along with the \( \text{curSN} + 1 \) sequence number. Upon reception of an \textit{INVOKE} message tagged with the \( \text{sn} = \text{curSN} + 1 \), a stub appends the operation to \textit{opHist}, increases \( \text{curSN} \) and broadcasts back an \textit{ACK} message tagged with the updated \( \text{curSN} \) 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 \textit{ACK} messages tagged with a sequence number equal to the local value of \( \text{curSN} \).

**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 \textit{block} variable to false and broadcasts a \textit{NEWEP} message conveying its CS requests queue \( \text{req} \), the \textit{granted} vector, the \( \text{curSN} \), the sequence of operations stored in \textit{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 \textit{NEWEP} message at \( s_j \) triggers, in its turn, the broadcast of a \textit{NEWEP} message. In this case, however, if \( s_j \) is not expecting the current token owner, say \( s_k \), \( s_j \) specifies \( s_k \)’s identity, rather than his own, in the \textit{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 \textit{NEWEP} message, the stub’s behavior is identical. First, the stub waits for a majority of \textit{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 \( \text{curSN} \), \textit{granted} and \( \text{reqs} \). Then, it determines whether there are operations included in the \textit{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 \textit{CS IDLE} value. If \( s_i \) has not become the token owner and his CS request was not included in the \( \text{reqs} \) 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 \textit{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 \textit{lastIssuedOp}, the corresponding \textit{outcome} event is delivered to \( u_i \) and the \textit{CS IDLE} state is entered. On the other hand, if there are no operations in \textit{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 \( s_j \) 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 \textit{eject} event towards \( u_i \) (this happens in case \( s_i \) was previously in the CS and was taken away the token ownership during the epoch change), or ii) immediately transferring the token ownership to the first process in the \( \text{reqs} \) queue (in case \( s_j \) was not requesting the CS in the former epoch, it has now been assigned the token ownership, and \( \text{req} \) is not empty), or iii) just a simple update of the \textit{state} variable to the \textit{TOKEN HELD} value (if \( s_i \) was not requesting the CS in the former epoch, it has now become the token owner, and \( \text{req} \) 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 \( H_e \) 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 \( \text{crash}_i \) event to \( H \); ii) for any stub \( s_j \) that is in the \( REQUESTING \) state, delete the last \( \text{try}_j \) 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 just 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 \( H_e \) must be CS-sequential (hence WME1 holds). Also, since \( H_e \) 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 \( \text{curSN} \), 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 actual order of invocation 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 \( \Diamond P \), 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 \( \mu' \) 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 \( \text{INVOKE} \) 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 execution of 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 \( \Diamond P \), we get that eventually the whole set of correct
5 Performance analysis

In Table 1 we discuss the performances of the presented WME algorithm and contrast them with those of the WME solution 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 presented algorithm, 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 correspondingly exchanges either 0 or \( O(N) \) messages. The release of the CS is immediate (if \( s_i \) receives an \( \text{exit} \) event, the \( \text{rem} \) event can be generated right away) 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 \( \text{Invoke} \) message, followed by a distributed acknowledgment phase in which each process broadcasts back an \( \text{Ack} \) message).

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]. In both cases, the performance figures shown in Table 1 clearly highlight the sensible advantages arising from the algorithm presented in this work.

6 Conclusions

This work presented a token-based algorithm solving the Weak Mutual Exclusion problem, a recently introduced abstraction which captures the problem of how to ensure mutual exclusion in the access to a replicated shared resource.

The proposed algorithm sensibly outperforms previous WME solutions in terms of both communication steps delay and message complexity.

References