The Weak Mutual Exclusion Problem*

Paolo Romano, Luis Rodrigues and Nuno Carvalho
INESC-ID/IST

Abstract

In this paper we define the Weak Mutual Exclusion (WME) problem. Analogously to classical Distributed Mutual Exclusion (DME), WME serializes the accesses to a shared resource. Differently from DME, however, the WME abstraction regulates the access to a *replicated* shared resource, whose copies are locally maintained by every participating process. Also, in WME, processes suspected to have crashed are possibly ejected from the critical section

We prove that, unlike DME, WME is solvable in a partially synchronous model, i.e. a system where the bounds on communication latency and on relative process speeds are not known in advance, or are known but only hold after an unknown time.

Finally we demonstrate that $\Diamond P$ is the weakest failure detector for solving WME, and present an algorithm that solves WME using $\Diamond P$ with a majority of correct processes.

1 Introduction

Problem Statement and Motivations. The distributed mutual exclusion problem (DME) [10, 11, 21] is considered a fundamental challenge associated with parallel and distributed programming [28]. It captures the coordination required for resolving conflicts resulting from several concurrent processes accessing a *single, indivisible* resource, that can only support one user at a time. The user accessing the resource is said to be in its critical section (CS), and the (safety) property guaranteeing the existence of at most one process in its CS at any time is known as *mutual exclusion*.

Over the years, the mutual exclusion problem has been investigated both in the failure-free model [24, 30] and under the assumption that the processes accessing the shared resource can fail according to the crash-failure model [1, 26]. However, in real life, failures do not only affect the processes contending for the CS, but clearly also the shared resource to be accessed in mutual exclusion. It is therefore very often the case that the shared resource to be accessed in mutual exclusion appears as a single and indivisible object only at a logical level, while instead being physically replicated for fault-tolerance, as well as for scalability purposes. This class of systems is indeed extremely popular: consider, for instance, the wide range of applications, spanning from the world wide web [13], to databases [23] and distributed file systems [15], that rely on various lease mechanisms aimed at simplifying and/or optimizing the consistency mechanisms used for accessing replicated versions of the same logical (collection of) data.

The aforementioned lease based replication mechanisms are clearly closely related to the DME problem as their base goal is to provide to a single process exclusive access to a replicated resource for a given period of time. On the other hand, one key differentiation point between traditional lease based mechanisms and the mutual exclusion problem lies in that leases, being de facto time-based contracts, are tightly coupled to the notion of real-time. Perhaps unsurprisingly, lease schemes have in fact been traditionally designed and implemented assuming strong, and hence restrictive, synchrony levels (such as bounded communication delay and clock skew across processes). Conversely, in the DME problem, processes exit from the CS only by explicitly releasing it (one relevant exception to this rule being failures, which implicitly determine an eventual exit from the CS). In this sense, the DME problem might appear as if it was not directly bound to the notion of time. Unfortunately, this is not the case, since the mutual exclusion property is classically

*Selected sections of this report will be published in the Proceedings of the International Parallel and Distributed Processing Symposium, Rome, Italy, May 2009. This paper was partially supported by the Pastramy project (PTDC/EIA/72405/2006) funded by the Portuguese Fundação para a Ciência e Tecnologia.

specified by explicitly referring to global time, i.e., from [10]:

Mutual exclusion: No two different processes are in their CSs *at the same time*.

As a direct consequence of this property, DME is known to be solvable [10], in the presence of failures, only if the underlying system model encapsulates sufficient synchrony assumptions permitting to accurately (i.e. without the possibility of any false failure suspicion[1]) distinguish crashed processes from slow, but correct, ones. Unfortunately, this is possible only in the presence of strong synchrony assumptions, such as those guaranteed by the classical synchronous system model [17], where communication latencies and relative process speeds are a priori bounded. This is a rather restrictive requirement which has significant implications, both of theoretical and practical nature, as discussed below.

From a theoretical standpoint, the requirement of accurate failure detection makes the DME problem harder than another fundamental problem of distributed computing, namely consensus [17]. Consensus is in fact deterministically solvable even if the failure detector module stops outputting false failure suspicions only eventually, namely after a finite but not a priori known time[2]. This makes the consensus problem solvable in more relaxed, and hence general, system models with respect to DME, such as, e.g., in an eventually synchronous environment [5, 14] where bounds on communication latency and relative process speed are either unknown or are only guaranteed to hold starting at some unknown time. With respect to the synchronous model, partially synchronous models are hence much less restrictive and better suited to capture phenomena such as congestions or performance failures that represent not negligible aspects of realistic systems.

On the other hand, from a more practical perspective, the correctness of *any* algorithm solving the DME problem cannot be guaranteed when deployed in a real-life distributed system where temporary network partitions or overloads of the processing nodes (caused, e.g., by unexpected workload's fluctuations) may give raise to false failure suspicions. In such scenarios, for instance, the algorithm proposed in [10] to solve the DME problem could violate the mutual exclusion property by allowing multiple processes to simultaneously enter the critical section.

Note, however, that the above impossibility result holds for the traditional specification of the DME abstraction, namely the one used to regulate the concurrent access to a *single and indivisible* shared resource, whose mutual exclusion property is defined as above.

We can then pose the following question: is there a meaningful definition of the mutual exclusion problem for a *replicated* shared resource that is solvable under more relaxed assumptions on the system's synchrony?

Contributions. In this paper, we show that the answer to this question is *yes*.

First, we introduce the Weak Mutual Exclusion (WME) problem, which is derived by extending the classical DME specification [10] in a twofold direction.

On one hand, we explicitly model the interactions with the replicated shared resource associated with the CS, which we describe as a deterministic state machine which interacts by exchanging operations' invocations and responses events.

On the other hand, we relax the classical mutual exclusion property in order to detach it from the notion of real-time (which represents the crucial reason underlying the constraining requirements on systems' synchrony characterizing the DME problem) and bind it to the notion of logical time, captured by the concept of critical section *instance*. Unlike the classical DME, in the WME problem a CS instance can be granted not only in case there is currently (i.e. at the same time) no other process in their CS, but rather as long as the whole sequence of established CS instances can be reordered to yield a sequential history in which:

1. no two CS instances overlap over time,
2. the order of establishment of the CS instances and of the operations executed on the (replicated) shared resource does not contradict the (partial) order in the original history,

[1] The work in [10] has actually shown that DME is solvable even if a correct process p is falsely suspected during the initial phase by some other process q, i.e. before p is "trusted" for the first time by q. This however does not significantly relax the synchrony requirements of DME.

[2] In other terms, DME is harder than consensus since the latter is solvable with a failure detector that is strictly weaker [5] than the weakest failure detector required for solving DME.

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 shared resource.

Also, unlike the classical DME, the specification of the WME problem allows aborting already established CS instances. In this case, we say that the process is ejected from its CS and require that any pending operation fails (i.e. is not executed on any replica of the shared resource) and that the associated application is explicitly notified via the delivery of an apposite call-back event.

We then show that the WME problem is solvable in an eventually synchronous system [16], i.e. a system in which bounds on communication latency and relative process speed are either unknown or are only guaranteed to hold starting at some unknown time. This result is achieved by proposing an algorithm, modularly layered on top of the consensus abstraction, which solves the WME problem in an asynchronous system with a majority of correct processes and an *eventually perfect* failure detector, namely $\Diamond$P [5]. An eventually perfect failure detector ensures that all faulty processes are eventually detected (Strong Completeness) and that eventually no correct process is suspected by any correct process (Eventual Strong Accuracy). $\Diamond$P was shown in [5] to be implementable in an eventually synchronous system.

Finally, we prove than no algorithm can solve the problem using a failure detector that is strictly weaker than $\Diamond$P [5]. In other words, we identify in $\Diamond$P the minimum failure detector for solving the WME problem in an asynchronous system with a majority of correct processes. This result also implies that WME is strictly harder than the consensus problem, since the latter is known to be solvable with an eventually strong failure detector, $\Diamond$S, which is strictly weaker than $\Diamond$P, according to the failure detector's hierarchy defined in [5].

Paper organization. The remainder of this paper is structured as follows. In Section 2 we discuss related research. Section 3 describes the system model, and Section 4 introduces terminology and notions which we use in the remainder of the paper to reason about the correctness of distributed algorithms. In Section 5 the specification of the WME problem is provided. Section 6 presents an algorithm that solves the WME problem using the $\Diamond$P failure detector and tolerating the failure of a minority of process. In Section 7 we show that $\Diamond$P is the weakest failure detector for solving the WME problem. Section 8 concludes the paper.

2 Related Research

There is a large body of research related to the mutual exclusion problem. Traditional DME solutions coping with the possibility of process crashes, e.g. [1, 26] assume perfect information about failures, i.e. the ability to distinguish slow processes from crashed ones without making *any* mistake. In other terms, these solutions either assume a synchronous system or encapsulate the required synchrony assumptions within a perfect failure detector, also called P, [5] ensuring that all faulty processes are eventually detected (Strong Completeness) and that no correct process is (falsely) suspected to have crashed by any other process (Strong Accuracy). More recently, [10] has shown that the DME problem can be actually solved with a failure detector that is strictly weaker than P, namely the *trusting* failure detector, T. Informally, T guarantees to eventually and permanently 1.a) trust (consider to be up) every correct process, 1.b) not trust any crashed process, and that 2) if T stops trusting a process, then the process must be actually crashed. On the other hand, it is crucial to highlight that, despite being strictly weaker than a perfect failure detector, T, analogously to P, cannot be implemented in an eventually synchronous system. In fact, any algorithm implementing T has to be able to determine a (bounded) time after which any trusted process that stops responding (e.g., to heartbeat messages) can be certainly considered as crashed, without the possibility to make *any* mistake (i.e. false failure suspicions). Since [10] proves that T is the weakest failure detector for DME, it follows that the DME problem can not be solved in partially synchronous systems where either the bounds bounds on communication latency and on relative processors' speeds exist but are not known in advance, or are known but only start to hold after an *unknown* time [14]. In this paper we relax the specification of the classical DME problem, deriving an abstraction, the Weak Mutual Exclusion, aimed at regulating the concurrent access to a replicated shared resource through an interface very similar to the one provided by conventional DME but that, unlike DME, is implementable even in a partially synchronous system, or, equivalently, in an asynchronous system equipped with a $\Diamond P$ failure detector.

The DME problem is also at the core of a number of well-studied process synchronization problems, such as the allocation of a set of distributed resources [12, 6, 7]. A common characteristic of this class of problems

is that they require to ensure mutual exclusion and starvation freedom in the access to a finite set of (not replicated) resources by some competing processes. Conflict relations in the access of processes to resources are normally captured via a *c*onflict graph [12] and a conventional measure of the failure resiliency of a solution algorithm is *failure-locality* [8]. Failure locality measures the impact of faults as the radius in the conflict graph of the worst-case set of processes that are blocked by a given fault, thus demarcating a halo outside of which faults are masked. For instance, the (crash) failure locality of the dining philosopher problem [12], i.e. the archetype of distributed resource allocation problems, is known to be 2 in an asynchronous system [8] and 1 in an asynchronous system augmented with a $\diamondsuit$P failure detector [27]. As it will be later shown in this paper, the WME problem can be solved in the latter system model, ensuring starvation freedom of *any* participant process despite failures. In other words, the (crash) failure locality of the WME problem is 0 in an asynchronous system augmented with a $\diamondsuit$P failure detector.

The notion of lease [13, 15] is closely related to mutual exclusion. Unfortunately, as already noted in Section 1, most lease based approaches are explicitly tied to the notion of physical time. Indeed, the only lease based solution we are aware of that is designed for employment in an asynchronous system is the one in [3]. This solution allows processes to setup *Asynchronous Leases* over an a priori pre-declared sequence of logical intervals, which can be used to execute arbitrary operations. The motivations underlying this approach, as well as its applications, are, in some sense, common to those of WME: providing distributed users with a tool that ensures the absence of conflicts while issuing operations. On the other hand, there are a number of significant differences between our Weak Mutual Exclusion and the Asynchronous Lease abstraction in [3]. First, upon crash of a process p that has successfully established an asynchronous lease, in [3] the remaining processes are forced to block until p recovers and "uses" all the intervals over which it has acquired a lease. Also, the success in the acquisition of an Asynchronous Lease is conditioned to the fact that there are no contending requesters. The specification of the WME problem, conversely, provides stronger liveness guarantees which rule out the above blocking scenarios. Furthermore, the Asynchronous Lease mechanism requires users to pre-declare the number of logical intervals to allocate, whereas the WME, deriving from the formulation of classical DME, exposes an interface which closely resembles the one of a pre-emptible lock.

Finally, since the proposed WME abstraction is, de facto, a tool to ensure the consistent evolution of a replicated shared resource, it is strongly related also to the vast literature on replication. The WME can be viewed as a higher level abstraction, which can be implemented by leveraging various well known techniques/building blocks developed by previous research, such as consensus [18], atomic broadcast [9] etc. The main practical benefits of using WME abstraction for maintaining the consistency of replicated resources are twofold. On one hand, analogously to classical centralized lock schemes, the WME abstraction allows to mitigate the drawbacks related to the concurrent execution of conflicting user level operations. In this sense the WME abstraction can be applied to support database replication schemes such as [20, 29], where it may be used to reduce the frequency of aborts caused by conflicting data accesses. Further, the ability of the WME abstraction to serialize the sequences of operations issued by each user within the CS can provide benefits for what concerns the performances of some of the typical building blocks used by replication schemes. For instance, it is known that the performance of consensus can be strongly optimized (i.e. its decision latency can be reduced to a single communication step [4, 22]) if there are no two processes simultaneously proposing different values. This is exactly the kind of guarantees that the WME abstraction aims at providing.

3 System Model

We consider in this paper a crash-prone asynchronous message passing system model augmented with the failure detector abstraction [5]. The terminology used in this section closely resembles the one in [10, 25].

System. The system consists of a set of n processes $\Pi = \{1, \ldots, n\}$ $(n > 1)$, communicating over reliable channels, guaranteeing that messages are eventually delivered by the intended receiver, unless the sender or the receiver crashes. The asynchronous communication channels are modeled as a message buffer which contains messages not yet received by their destinations. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is a fictional device, as the processes have no direct

access to it. We take the range $\mathbb{N}$ of the clock's ticks to be the set of natural numbers and denote the time instant in which an event e is generated by a process as $\mathcal{T}(e)$.

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*. The system is augmented with a distributed failure detector oracle $\mathcal{D}$ in the sense that every process i has access to a local failure detector module $\mathcal{D}_i$ that provides i with information about the failures in the system.

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, which can be viewed as an application program, interacts exclusively with its local stub s_i to request exclusive access to the shared resource and to 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 on a single copy of the shared resource that is consistent with the order of establishment of the mutual exclusion. Users, stubs and replica of the shared resources are modeled as deterministic (possibly infinite state) automata that communicate by exchanging input and output events.

A stub s_i interacts with the local replica r_i of the replicated resource through the following classes of events from the domain $SRevents$:

- $invoke_i[op]$ is an input event of r_i (resp. output event of 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 simply associating an additional unique id with each operation, which we omit to simplify presentation).

- $response_i[op, res]$ is an output event of r_i (resp. input event of s_i) which notifies the stub about the result $res \in Results$ returned 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 output.

The interaction with a replica r_i is assumed to be non-blocking, i.e. if r_i receives a $invoke_i[op]$ event it eventually generates the corresponding $response_i[op, res]$.

A user u_i and its local stub s_i interact using the following six classes of events from the domain $USevents$:

- 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.

- $crit_i[CS\_id]$, 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.

- $issue_i[CS\_id, op]$, 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.

- $outcome_i[CS\_id, op, res]$, 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.

- $exit_i[CS\_id]$ 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.

- $rem_i[CS\_id]$ 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.

- $ejected_i[CS\_id]$ 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 (due to a failure suspicion).

An operation that was issued by a user u_i through a $issue_i[CS\_id, op]$ event, and which is not 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* if and only if i) e is an event exchanged between a user and a stub (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 $invoke_i[op']$, respectively $outcome[CS\_id, op, res]$ and $response_i[op', res]$, are said to be *correlated* if and only if $op = op'$, i.e. they are associated with the same operation *op* (recall we are assuming that each operation is univocally identified).

Algorithms, runs and solvability of problems. An *algorithm* $\mathcal{A}$ is a collection of n (possibly infinite state) deterministic automata, one for each of the stubs s_i in the system. $\mathcal{A}(i)$ denotes the stub automaton running on process i. Computation proceeds in steps of the given algorithm $\mathcal{A}$. In each step of $\mathcal{A}$, process i performs atomically the following three actions: (1) s_i processes one of the following three input events, a) receives a single message addressed to process i from the message buffer, or b) a null message, denoted as λ, or c) an input event from either r_i or u_i; (2) s_i queries and receives a value from its failure detector module; (3) s_i changes its state and either sends a message to a single process or generates an event for u_i or r_i, according to the automaton $\mathcal{A}(i)$ and based on its state at the beginning of the step, the behavior of s_i during phase (1) of the execution step, and the value that i sees in the failure detector query phase. Note that the input event chosen in phase (1) of each execution step is chosen non-deterministically among those currently enabled.

A *configuration* defines the current state of each process in the system and the set of messages currently in the message buffer. Initially, the message buffer is empty. A step (i, e, d, A) of an algorithm $\mathcal{A}$ is uniquely determined by the identity of the process i that takes the step, the event processed during the step, and the failure detector value d seen by i during the step. We say that a step (i, e, d, A) is *applicable* to the current configuration if and only if the incoming event e is enabled in the current configuration. A *schedule* S of an algorithm $\mathcal{A}$ is a (finite or infinite) sequence of steps of $\mathcal{A}$. $S_\top$ denotes the empty schedule. A schedule S is applicable to a configuration C if and only if (a) $S = S_\top$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc.

A *run* of an algorithm $\mathcal{A}$ is a infinite schedule applicable to an initial configuration of $\mathcal{A}$. A *problem* M is a set of properties P_M that determines a set of legal runs. An algorithm $\mathcal{A}$ solves a problem M, defined by the set of properties P_M, using a failure detector $\mathcal{D}$, if all the runs of $\mathcal{A}$ satisfy the properties P_M. A failure detector $\mathcal{D}$ is said to solve problem M if an algorithm $\mathcal{A}$ exists that solves M using $\mathcal{D}$.

4 Histories, subhistories and history equivalences.

In this section we provide a set of definitions aimed at formalizing the properties of sequences of events in our distributed system model. Our aim is to define precise formal foundations which will be used in the following sections to specify the WME problem as well as to reason on the correctness of algorithms that either implement a WME abstraction or use such abstraction to solve different problems. The notation used in this section is analogous to the one in [19], but is adapted and extended to fit our system model.

Histories and Sub-Histories. A history H is the (possibly infinite) sequence of 1) events produced by the automata of the system (i.e. users, processes and replicas of the shared resource) and of, 2) all the process crash events. 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 pairs of events $< issue_i[CS\_id, op], outcome[CS\_id, op, res] >$ 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-stub 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 $\{rem_i[CS\_id], eject_i[CS\_id], crash_i\}$ occurring 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:

$$\mathcal{S}^1 := (\ try_i\ crit_i[id]\ \texttt{OPS}\ exit_i[id]\ rem_i[id]\ )$$

`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:

$$\mathcal{S}^2 := (\ try_i\ crit_i[id]\ \texttt{INT\_OPS}\ )$$

`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:

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

Informally, any well-formed CS instance subhistory H_{id}^{CS} starts with the establishment of a new critical section instance, through the $\langle try_i, crit_i[CS\_id]\rangle$ pair of events. Once entered the critical section instance, u_i can issue an arbitrary number (possibly null) of operations, through the $issue_i[CS\_id, op]$ events. We require that operations are issued sequentially, and that each issued operation is followed, unless process i crashes, either by the corresponding outcome or by an eject event, which implicitly notifies u_i about the failure of the pending operation and its exit from the CS. While it would be feasible to extend our model in order to support "pipelining" of operations within a CS, this is out of the scope of this paper and represents subject of future work. 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.

Finally, a user u_i is called a *well-formed user* if it does not violate the cyclic order of events defined by $\mathcal{S}^1$ and $\mathcal{S}^2$.

Complete CS instance subhistories. A well-formed CS instance subhistory H_{id}^{CS} is *complete* if:

1. it has no pending operations, and
2. 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 try_i, deleting the whole CS instance subhistory.
2. for any pending operation *op* issued by user u_i within CS instance CS_id: i) if in H there exists no $invoke_j[op]$, where $j \in \Pi$, delete the corresponding $issue_i[CS\_id, op]$ in H_{id}^{CS}, or ii) append zero or more $invoke_j[op]$, $response_j[op, res]$ and $outcome_i[CS\_id, op, res]$, where $j \in \Pi$, correlated events, preserving H_{id}^{CS}'s 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 $\langle exit_i[id], rem_i[id]\rangle$, or the $rem_i[id]$ event or a $crash_i$ event so to complete it while preserving its well-formedness.

Equivalent and Isomorphic Histories. Two histories H and H' are said *equivalent* if and only 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 $H^{US|op}$ if and only if:

1. there exists a bijection $\mathcal{B}$ between H^{SR} and $H^{US|op}$ such that $\forall e \in H^{SR}$ and $\forall e' \in H^{US|op}, \mathcal{B}(e) = e' \Leftrightarrow$ e and e' are correlated events, and
2. $\mathcal{B}$ is an *order isomorphism* with respect to $<_H^{\mathcal{T}}$, i.e. $\forall \{e_0, e_1\} \in H^{SR}, e_0 <_H^{T} e_1 \Leftrightarrow \mathcal{B}(e_0) <_H^{\mathcal{T}} \mathcal{B}(e_1)$.

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 1 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 2 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:

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

A well-formed history H is *CS-sequential* if and only if $<_H^{CS}$ 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 [10] (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-emptied by the delivery of an *ejected* event.

5 The Weak Mutual Exclusion Problem

Based on the terminology introduced in Section 3 and Section 4 we now provide the specification of the Weak Mutual Exclusion (WME) problem.

We say that 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, organized in two categories:

The safety properties provide joint consistency guarantees on the state of the distributed mutual exclusion protocol and of the replicated shared resource, as well as on the well-formedness of the interactions between users, stubs and replicas of the shared resources. Termination properties, on the other hand, ensure the non-blocking evolution of the system.

Informally, the *Weak Mutual Exclusion* property requires that the CS instance subhistories can be re-ordered 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*). Further, *WME3* constrains the order of execution of the operations on the replicas of the shared resource to be consistent with the execution order viewed by the user while interacting with its stub.

Extending H to H_* has a twofold purpose. On one hand it allows us to consistently apply the $<_H^{CS}$ ordering on both complete and incomplete CS instance subhistories of H, so to extend the correctness criteria even on incomplete CS instance subhistories. On the other hand, it captures the notion that some pending operation issued by user u_i may have taken effect (e.g. being observed by some other process) even though the local stub s_i has not invoked the operation on r_i, yet.

Roughly speaking, one could say that *WME1* and *WME2* properties require the linearizability of a replicated "pre-emptible exclusive lock" (see our discussion on CS-sequential histories in Section 4 for an informal definition of such a lock); applications use such a lock to regulate the concurrent access to the replicated shared resource, or, more precisely, to the local replica of the shared resource. WME3, on the other hand, forces the order of execution of the operations on each independent local replica to adhere to the serial order imposed by the acquisition of the mutual exclusion. Note that *WME3* does not force processes

Safety.

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 S.

1CS: The history of the replicated shared resource (i.e. 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.

Starvation-Freedom A correct process 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 the operation either fails or succeeds, and eventually all the operations it issues succeed.

to exchange mutual information on the state of the local copies of the shared resource, r_i, whose state trajectories would be therefore allowed to diverge arbitrarily.

Such runs are ruled out by the *1CS* property which guarantees that the history of the replicated shared resource is 1-copy serializable [2]. Note that, by property WME3, the stub-resource subhistory (H_*^{SR}) reflects the same ordering of the correlated sequential user-stub successful operations subhistory ($S^{US|op}$), and that the ordering of the operations issued in S^{US} is imposed by the total order relation $<_S^{CS}$ (determining the equivalent serial acquisition order of the CS). Since, the ordering of operations in S^{US} is, by property WME2, consistent with the time-based CS instance acquisition order in H_* (defined by the $<_{H_*}^{CS}$ relation), it directly follows that the stub-resource subhistory is indeed linearizable [19].

Concerning the liveness properties, the *CS-Release Progress* simply ensures that a process does not block while exiting from the CS. *Starvation-Freedom* and *Operation Progress* are, on the other hand, more subtly intertwined. On the one hand, the *Operation Progress* property requires that each operation issued by a correct process eventually either fails or succeeds, as well as that a time exists after which all the operations issued by a correct process succeed. The Starvation-Freedom property, on the other hand, encodes a fairness property requiring that, independently of the frequency or timing with which processes contend for the CS, any process that does not crash is eventually able to enter its CS, unless there is some other correct process that acquires the CS and never resigns. Note that, while conditioning the liveness of the CS acquisition to the assumption that processes behave "altruistically" (i.e. that they eventually resign) could apparently seem overly restrictive, this is actually required in an asynchronous system if one wants to jointly provide progress guarantees on the successful execution of operations, as required by the *Operation Progress* property. In fact, if we allowed an "impatient" process to eject a process p that appears to be correct but has not yet resigned, we could risk to constantly cause the failures of p's operations whose execution time, being the system asynchronous, cannot be a priori bound. This would clearly determine a violation of the *Operation Progress* property.

6 Solving WME using $\diamondsuit$P

We now provide an algorithm that illustrates how WME can be solved in an asynchronous system augmented with a $\diamondsuit$P failure detector and a majority of correct processes. Our solution uses a FIFO reliable broadcast and an uniform consensus service [17] as building blocks. These services (which are implementable under our assumptions [5, 14, 18]) are defined below.

```
Set decisions;          // already decided values
Queue props;            // proposals queue
int rn=0;               // current consensus round
int curCSid=⊥;          // ID of current CS instance
PID CSOwner=⊥;          // process in current CS instance
bool wait=false;        // signals an ongoing consensus round
bool exiting=false;     // true after an exit_i and before a rem_i

1 upon try_i do
2     int id = getUniqueID();
3     RB-Send ("CS_req", i, id);

4 upon exit_i[CS_ID] do
5     RB-Send ("CS_release", CS_ID); exiting=true;

6 upon issue_i[CS_ID, op] do
7     RB-Send ("issue", op,CS_ID);

8 upon CSOwner ∈ ◇P_i do
9     RB-Send ("eject", curCSid);

10 upon RB-Deliver(msg) from p ∈ Π do
11    if (msg ∉ decisions) props.enqueue (msg);

12 upon ¬wait ∧ ∃ ["CS_release",id]∈props s.t. id=curCSid do
13    propose(++rn, ["Exit_req", curCSid]);
14    wait=true;

15 upon ¬wait ∧ ∃["eject",id]∈props s.t. id=curCSid do
16    propose(++rn, ["Eject_req", curCSid]);
17    wait=true;

18 upon ¬wait ∧ CSOwner=⊥ ∧ ∃["CS_req", p, id]∈props do
19    propose(++rn, ["CS_req", first ["CS_req", p, id]∈props]);
20    wait=true;

21 upon ¬wait ∧ curCSid≠ ⊥
          ∧ ∃["issue",op,id]∈props s.t. id=curCSid do
22    propose(++rn, first ["issue",op,id]∈props s.t. id=curCSid]);
23    wait=true;

24 upon decide(rn,["CS_req", p, id]) do
25    decisions = decisions ∪ {["CS_req",p,id]};
26    props=props\{["CS_req",p, id]};
27    CSOwner=p; curCSid=id;
28    if (CSOwner=i) crit_i[curCSid];
29    wait=false;

30 upon decide(rn,["issue", op, id]) do
31    decisions = decisions ∪ {["issue",op,id]};
32    props=props\{["issue",op, id]};
33    invoke_i[op];
34    wait result_i[op, res];
35    if (CSOwner=i) outcome_i[curCSid, op, res];
36    wait=false;

37 upon decide(rn,["Exit_req",id]) do
38    decisions = decisions ∪ {["exit",id]};
39    props=props\{["exit", id]};
40    if (CSOwner=i) rem_i[id];
41    CSOwner=⊥; curCSid=⊥;
42    wait=false; exiting=false;

43 upon decide(rn,["Eject_req", id]) do
44    decisions = decisions ∪ {["Eject_req", id]};
45    ∀x = [·, CSid] ∈props s.t. CSid=id do props=props\{x};
46    if (CSOwner=i ∧ ¬exiting) eject_i[id];
47    else if (CSOwner=i ∧ exiting) rem_i[id]; exiting=false;
48    CSOwner=⊥; curCSid=⊥;
49    wait=false;
```

Figure 1: Solving WME using the $\diamondsuit\mathcal{P}$ with a majority of correct processes (proc. i).

The uniform consensus problem is specified by the following properties: i) *Validity:* Any value decided is a value proposed; ii) *Uniform Agreement:* No two processes decide differently; iii) *Termination:* Every correct process eventually decides, iv) *Integrity:* No process decides twice.

The FIFO reliable broadcast problem is defined by the following properties: i) *Validity:* If a correct process broadcasts a message m, then it eventually delivers m, ii) *Integrity:* For any message m, every process delivers m at most once and only if m was previously broadcast by some process; iii) *Agreement:* If a correct process delivers m, then all correct processes deliver m; iv) *FIFO:* Let m and m' be two messages sent by a given process p. If a process q delivers message m before message m', then p has sent m before m'.

The core of the algorithm consists of a sequential execution of multiple instances of consensus, which we call consensus rounds in the following to avoid ambiguity with the notion of critical section instance. In each consensus round, one of the following events can be proposed/decided: i) to assign a CS instance to a given process; ii) to execute an operation; iii) to exit a process from a CS instance or; iv) to eject a process from a CS instance. Each consensus round is identified by a sequence number *rn*. The global total order defined by the sequence of consensus decisions defines a linearization of the distributed execution, where processes, in sequence, gain access to the CS, execute zero or more operations, and exit (or are ejected from) the CS.

We start by illustrating a non-concurrent, failure-free, execution of the algorithm in stable conditions. Assume that the CS is free and a process p tries to enter in the CS. The algorithm starts by having p RB-broadcast a "CS_req" message to every other process, including itself (l.3). This request is eventually RB-delivered to every correct process and inserted in a proposal list (l.11). Since the CS is free and there is a pending proposal with the request, a consensus is initiated to decide the owner of the next CS instance. In this example, all correct processes propose p. The *wait* flag is just used to prevent a new consensus instance to be initiated before the previous instance has been decided. The *decisions* queue, on the other hand, is used to filter out obsolete incoming messages which have already been decided in a previous consensus instance. Since all correct processes propose p (and failed processes do not propose), the CS is attributed to p as result of the consensus decision (l.24-29). Basically, the same algorithm is executed for deciding the outcome of an issued operation (l.30-36) or to decide to have p exit the CS (l.37-42).

Consider now an execution where two processes p and q try to enter the CS concurrently. The corresponding *CS_req* messages may be received by different processes in different orders. Thus, different processes may propose a different owner for the next CS instance. Still the Uniform Agreement property of consensus ensures that a single value will be decided and a single process will be granted access to the CS.

Consider also that a process p, that owns a CS instance, fails. Eventually, the $\Diamond$P failure detector at some correct process q will mark p as failed (l.8). As a result, an "eject" message is RB-broadcast. Therefore, eventually all correct processes will have the "eject" message in the proposal list (l.11), and will propose an "Eject_req" to the same consensus instance, ejecting p when consensus decides (l.43-50).

Finally, note that due to the asynchrony of the system, process p may be erroneously suspected while issuing an operation, or while exiting the CS. In this case, the corresponding "issue"/"exit" and "eject" messages may be received by different processes in different orders. Hence, some processes may propose to eject p while other may propose to commit the issued operation/allow p to exit the CS. Again, consensus ensures that all processes consistently decide. Once a process is ejected (due to a failure or a false suspicion), all subsequent operations issued in the CS instance are simply discarded (l. 43-50), including any pending exit request. In order to ensure that after a user u_i generates an $exit_i$ event, its stub s_i does not deliver an $eject_i$ event (which would violate well-formedness of the interaction between s_i and u_i), the boolean *exiting* variable is flagged upon the reception of an $exit_i$ even, and its value is then checked upon the delivery of an eject decision from consensus (l.46-47). If the ejected process was resigning, rather than outputting an $eject_i$ event, s_i generates a rem_i event and sets back the exiting variable to false.

6.1 Correctness Proof

In this section we prove that the algorithm in Figure 1 solves the WME problem defined in Section 5. We start by achieving a preliminary result which will be functional to the development of the rest of the proof.

Let us denote with $\mathcal{DT}$=$\{\ t_1^{i_1}, \ldots, t_j^{i_j}, \ldots, t_{rn}^{i_{rn}}\}$ the sequence of time values $t_j^{i_j} \in \mathbb{N}$ in which, for each consensus round $j \in [1, rn]$, process $i_j \in \Pi$ is the *first* to deliver a decision (formally, $t_j^{i_j} = min_{k \in \Pi}(t_j^{dec_k})$ where $t_j^{dec_k} \in \mathbb{N}$ is the time in which process k delivers a decision for consensus round j). We can then prove the following lemmata:

Lemma 1 *$<_{\mathcal{DT}}^{\mathcal{T}}$ is a total order relation.*

Proof Let $t_j^{i_j} \in \mathbb{N}$, resp. $t_{j+1}^{i_{j+1}} \in \mathbb{N}$, where $j \in \mathbb{N}$ and $i_j, i_{j+1} \in \Pi$, be the earliest time in which a process i_j, resp. i_{j+1}, decides for consensus round j, resp. $j+1$. We need to prove that $\forall j \in [1, rn-1] : t_j^{i_j} < t_{j+1}^{i_{j+1}}$. There are two cases to consider: (i) $i_j = i_{j+1}$ and (ii) $i_j \neq i_{j+1}$.

i. case $i_j = i_{j+1}$: when a process i_j proposes a value for a consensus round j, it sets the *wait* variable to false and the rn variable to the j value (see lines 12-13,15-16,18-19,21-22). This prevents i_j from updating the rn variable before a decision value is delivered for the j-th consensus round, hence ensuring that i_j can decide for consensus round $j+1$, only after it has decided for consensus round j. Hence we obtain that $t_j^{i_j} < t_{j+1}^{i_{j+1}}$.

ii. case $i_j \neq i_{j+1}$: given that a process decides for consensus round j only after it has decided for consensus round $j-1$ and since process i_{j+1} decides for consensus round $j+1$, it means that process i_{j+1} must have decided for consensus round j at some time $t' < t_{j+1}^{i_{j+1}}$. Since $i_j \neq i_{j+1}$, process i_{j+1} does not decide for consensus round j before than process i_j, hence it follows that $t_j^{i_j} \leq t' < t_{j+1}^{i_{j+1}}$.

□

Lemma 2 *If a value v is decided in consensus round c, then v is not proposed in any subsequent consensus round $c' > c$.*

Proof Assume by contradiction that the value v is decided in consensus round c and proposed in consensus round c', where $c' > c$. Every process i that decides v in consensus round c, see lines 24-26, 30-32, 37-39,

43-45, adds v to the decisions set and removes it from the props queue, if any. Assume, without loss of generality, that these actions have been completed by process i at time t_i. In order for v to be proposed by i in consensus round $c' > c$, since v is not in props, it means that s_i must have received v via the reliable broadcast at some time $t > t_i$. But after time t, s_i takes no action upon the delivery of v, as it finds v in the decision set at line 11. □

As a corollary of the latter lemma we have that:

Lemma 3 *Two different consensus rounds can never decide the same value.*

Theorem 4 (Well-formedness) *For any $i \in \Pi$, if u_i is well formed, the history describing the interaction between u_i and s_i is well-formed.*

Proof It is enough to show that the cyclic order defined in Section 4 over the events $e \in \mathcal{US}$ is never violated by any run H of the algorithm in Figure 1. More precisely, $\forall i \in \Pi, \forall H^{US}|i \in H$ we need to show that:

1. the preceding event of a $crit_i[CS\_id]$ event is a try_i. A stub s_i generates a $crit_i[CS\_id]$ at time t event only if it decides the ["CS_req",i,CS_id] value in some consensus round rn. By Validity property of consensus this is possible only if some process proposed the ["CS_req",i,CS_id] value which, by the Integrity property of FIFO reliable broadcast, is, in its turn, only possible if, at some time $t' < t$, u_i had generated a try_i event. Now assume by contradiction that at some time t'', with $t' < t'' < t$, an event e is generated that follows a try_i and precedes a $crit_i[CS\_id]$. By user well-formedness, if a user u_i issues a try_i, it can not generate any try_i, nor $exit_i$, nor $issue_i$ events before s_i generates a $crit_i$ event. Hence, by exclusion, e should be either an $outcome_i$ or an $eject_i$ or a rem_i event. We are going to show that none of the above cases is possible.

   By user well-formedness, a user u_i issues a try_i only if it is out of any CS instance. This means that, if u_i had established any previous CS instance CS_ID, then s_i must have already generated either a $rem_i[CS\_ID]$ or an $ejected_i[CS\_ID]$ and set its local $CSOwner$ variable to the $\perp$ value. On the other hand, for s_i to generate either an $outcome_i$ or an $eject_i$ or a rem_i event, the local $CSOwner$ variable must store the i value (see lines 28, 34, 40, 46-47). But the only way for the $CSOwner$ variable to be set to the i value at s_i is that s_i generates a $crit_i$ event (see lines 27-28). Hence we obtain a contradiction and the claim follows.

2. the preceding event of a $rem_i[CS\_id]$ event is an $exit_i[CS\_id]$ event. A stub s_i generates a $rem_i[CS\_id]$ at time t event only if it decides the ["Exit_req",CS_id] value in some consensus round rn. By Validity property of consensus this is possible only if some process proposed the ["Exit_req",CS_id] value which, by the Integrity property of FIFO reliable broadcast, is, in its turn, only possible if at some time $t' < t$ u_i had generated a $exit_i$ event. Now assume by contradiction that at some time t'', with $t' < t'' < t$, an event e is generated that follows an $exit_i[CS\_id]$ and precedes a $rem_i[CS\_id]$. By user well-formedness, if a user u_i issues a $exit_i[CS\_id]$, it can not generate any try_i, or $exit_i$, or $issue_i$ events before s_i generates a $rem_i[CS\_id]$ event. Hence, by exclusion, e should be either an $outcome_i$ or an $eject_i$ or a $crit_i$ event. We are going to show that none of these cases is possible.

   By lines 4-5, if u_i generates an $exit_i[CS\_ID]$ event, s_i sets its local variable $exiting$ to true. Also, by user's well-formedness, a try_i event and a $crit_i[CS\_ID]$ event must be already in $H^{US}|i$. This also implies that the $CSOwner$ variable at s_i must be equal to i.

   Now assume by contradiction that a $crit_i[x]$ event follows a $exit_i[CS\_id]$ event where $x \neq CS\_id$. For this to happen, s_i must have decided the ["CS_req",i,x] value in some consensus round at line 24, which is only possible if some process has proposed ["CS_req",i,x] at line 18 for the same consensus round. This is possible only if s_i had reliably broadcast the ["CS_req",i,x] message at line 3, upon generation of a try_i event by u_i. But, by user's well-formedness, if u_i generates an $exit_i$ event, u_i must have already received a $crit_i$ event for any try_i event that it had previously generated. Hence, if a $crit_i[x]$ event followed a $exit_i[CS\_id]$, then the [CS_req,i,x] value should have been decided in two different consensus rounds, which is impossible by Lemma 3. A contradiction.

   Let us now consider the case in which an $outcome_i$ event follows an $exit_i[CS\_id]$ event. By user well-formedness a user u_i can generate a $exit_i$ event only if i) u_i has already established critical section

instance id, which implies that a ["CS_Req", i, id] value was decided in some consensus round rn, and ii) u_i has successfully executed any operation it issued, i.e. for each operation op issued by u_i a ["issue",op,id] value was decided in some consensus round $rn' \geq rn$. Also, any process that ends consensus round rn sets its local variables CSOwner and curCSid, respectively, to the i and id values, and keeps on storing these values until round rn'. Further, following the generation of the $exit_i[CS\_id]$ event by u_i, s_i does not initiate any new consensus round until it does not deliver the ["CS_Release",id] message that it had broadcast upon the generation of the $exit_i[id]$ event by u_i. At round $rn'+1$, hence, i proposes the ["Exit_req",id] value, but decides ["issue", op, id']. By Validity property of consensus, this is possible only if some process $j \in \Pi$ proposes the ["issue", op, id'] value for consensus round $rn'+1$. However, being the curCSid variable equal to id at any process that initiates consensus round $rn'+1$, it is not possible that the value ["issue",op,id'], where $id' \neq id$, is proposed in round $rn'+1$ (see line 21). It is also impossible that the value ["issue",op,id] is proposed in round $rn'+1$, given that all the processes that end consensus round rn' do not store any operation issued by u_i in their local props queue. In fact, any process ending consensus round rn' has already added any operation issued by u_i to the decision queue and removed it from the props queue (see lines 31,32). Hence we obtain a contradiction and the claim follows.

The only case that is left to prove is that an $exit_i[CS\_id]$ event is not followed by an $eject_i$ event. This is possible only if s_i decides ["Eject_req",$\cdot$] in line 43 and later executes line 46 while the *exiting* variable is set to false. However, after the generation of an $exit_i$ event at line 5, the *exiting* variable is set to true at s_i, and is re-set to false only after generating a $rem_i[CS\_id]$ event. Hence the claim follows.

3. an $issue_i[CS\_id, op]$ event is followed either by an $outcome_i[CS\_id, op, res]$ event or by an $eject_i[CS\_id]$.

   By user's well-formedness, we can exclude that an $issue_i[CS\_id, op]$ event is followed either by any try_i, or by $issue_i$, or by $exit_i$ events. Since we've shown that a $crit_i$ can only be preceded by a try_i we can exclude that an $issue_i[CS\_id, op]$ event is followed by a $crit_i$. Analogous considerations hold for rem_i events.

   Also, by user well-formedness we have that for any operation previously issued by u_i via a $issue_i[CS\_ID', op']$ event, the corresponding $outcome_i[CS\_ID', op', res']$ must have been already generated by s_i before u_i can issue a new operation op in CS instance CS_id. Hence, by Lemma 3, we can exclude that the preceding event of a $outcome_i[CS\_id, op, res]$ is an $issue_i[CS\_id', op']$, with $CS\_id' \neq CS\_id$ or $op \neq op'$, event associated either to a differ CS instance, or to a different operation.

4. the preceding event of an $eject_i[CS\_id]$ is neither a try_i, nor a $rem_i[CS\_id]$, nor an $exit_i[CS\_id]$: Assume by contradiction that the preceding event of an $eject_i[CS\_id]$ is a try_i. By user well-formedness, a try_i event is generated only if u_i has already received either a $term_i$ or an $eject_i$ event for any CS instance that it had previously established. In both cases the CSOwner variable is set to the $\perp$ value (see lines 41,48). Since an $eject_i$ event is generated only if the variable CSOwner is set to the value i at process i (see line 46), it means that after u_i has generated the try_i event, the local CSOwner variable must have been set to the i value. This is only possible if s_i decides ["CS_req",i,id] at line 24, sets CSOwner to the i value at line 27. In this case, however s_i would have generated a $crit_i$ event (at line 28) after the try_i and before the $eject_i$. Hence we obtain a contradiction.

   Now assume by contradiction that the preceding event of an $eject_i[CS\_id]$ is a rem_i. After a rem_i is generated the variable CSOwner is set to the $\perp$ value at s_i. Hence, the test in line 46, enabling the generation of the $eject_i$ event would fail if there wasn't at least one $crit_i$ event between a rem_i and an $eject_i$. A contradiction.

   Finally, assume by contradiction that the preceding event of an $eject_i[CS\_id]$ is a $exit_i[CS\_id]$ event. After a $exit_i[CS\_id]$ is generated the variable exiting is set to the true value (see line 5). Hence, the test in line 46, enabling the generation of the $eject_i$ event would fail if there wasn't at least one $rem_i[CS\_id]$ event between a $exit_i$ and an $eject_i$. A contradiction.

□

6.1.1 Constructing a Legal Completion of H

After having shown that the interactions between users and stubs are well-formed (assuming that users are well-formed), in this section we show how to construct a legal completion H_* of any history H produced by a run of the algorithm in Figure 1.

Any incomplete well-formed CS instance subhistory H_{id}^{CS} initiated by a process i can be legally completed by using the following rules:

1. if $H_{id}^{CS} = \{try_i\} \Rightarrow$ remove the CS instance subhistory H_{id}^{CS}.
2. if $H_{id}^{CS} = \{try_i, crit_i[id]\} \Rightarrow$ append a $crash_i$ event at time $\mathcal{T}(crit_i[id]) + 1$.
3. if $H_{id}^{CS} = \{try_i, crit_i[id]\}$, followed by a number of operations, one of which, say op, is pending $\Rightarrow$
   (a) if there exists no consensus round whose decision value is ["issue",op, id], just remove $issue[id, op]$ from H_{id}^{CS}; otherwise, denoting with rn the consensus round in which the ["issue",op, id] value was decided, append to H the events $invoke_j[op]$, $result_j[op, res]$, $outcome_j[id, op, res]$, for all $j \in \Pi$, unless these are not already present in H. Set the occurrence time of each of these added events at consecutive time instants starting from $\mathcal{DT}(rn)$;
   (b) let e be the last event in the resulting CS instance subhistory H_{id}^{CS}, add the $crash_i$ event at $\mathcal{T}(e) + 1$.
4. if $H_{id}^{CS} = \{try_i, crit_i[id]\}$, followed by a number of operations, none of which is pending $\Rightarrow$ let e be the last event in H_{id}^{CS}, add the $crash_i$ event at $\mathcal{T}(e) + 1$.
5. $H_{id}^{CS} = \{try_i, crit_i[id]\}$, followed by a number of operations, none of which is pending, and finally by an $exit_i[id]$ event $\Rightarrow$ append a $crash_i$ at $\mathcal{T}(exit_i[id]) + 1$.

It is straightforward to see that, by applying the above transformation rules to any not complete well-formed CS instance subhistory H_{id}^{CS}, a legal completion of H_{id}^{CS} is obtained. Also, by completion rule 3.a, any operation op that is either pending or failed in H, no longer appears in H_*: in other words, the operations in H_* are all and only those that were successfully executed in H.

6.1.2 Deriving an Equivalent Sequential History

Let us now denote with $\mathcal{CS} \subset \mathbb{N}$ the set of identifiers of the CS instance subhistories in H_*, with $\mathcal{OP} \subseteq Operations$ the set of operations issued in H_*, and with $\mathcal{CR} \subset \mathbb{N}$ the set of identifiers of the consensus rounds for which at least one process decided.

We can define the following functions:

1. $initCR$: $\mathcal{CS} \rightarrow \mathcal{CR}$ which takes as input a CS instance identifier CS_id and returns the consensus round in which a ["CS_req", $\cdot$, CS_id] value was decided.
2. $invokeCR$: $\mathcal{OP} \rightarrow \mathcal{CR}$ which takes as input a successfully executed operation op and returns the consensus round in which the ["issue",op,$\cdot$] value was decided.
3. $finalCR$: $\mathcal{CS} \rightarrow \mathcal{CR} \cup \{\infty\}$ which takes as input a CS instance identifier CS_id and returns the consensus round in which a ["Exit_req", CS_id] or a ["Eject_req", CS_id] value was decided. If no such round exists (e.g. because the process is still issuing operations and is not suspected to have crashed), the ∞ value is returned.

In the following we use $initCR$, $finalCR$ and $invokeCR$ to index the previously defined $\mathcal{DT}$ sequence. For instance, we use the notation $\mathcal{DT}[initCR(CS\_ID)]$ to denote the earliest time in which the $\mathsf{decide}(initCR(CS\_ID)$,["CS_req", $\cdot$, CS_ID]) is triggered on any process $i \in \Pi$.

We can now construct a well-formed CS-sequential history S, which we will prove to be equivalent to H_*, by applying the following transformations rules to 1) the events $e \in USevents$ exchanged within each $H_{*id}^{CS} \in H_*$ by u_i and s_i (where i is the process that initiated H_{*id}^{CS}), and to 2) any correlated events $e' \in SRevents \in H_*$ exchanged between s_j and r_j, with $j \in \Pi$, to reflect the operations issued by u_i on the local copy of the resource r_j. To simplify the notation of the transformation rules we here omit the subfix i of the events $e \in USevents$ which are generated exclusively by the process that initiated H_{*id}^{CS}.

S1 for each $CS\_id \in \mathcal{CS}$ set the occurrence of the corresponding $crit[CS\_id]$ event and of its associated try event in H_*, respectively, at the following time instants:

(a) $\mathcal{T}(try) = \kappa \cdot \mathcal{DT}[initCR(CS\_id)]$

(b) $\mathcal{T}(crit[CS\_id]) = \kappa \cdot \mathcal{DT}[initCR(CS\_id)]+1$

S2 for each operation $op \in \mathcal{OP}$ set the occurrence of the corresponding $issue[CS\_id, op]$, $invoke_j[op]$, $response_j[op, res]$, $outcome[CS\_id, op, res]$ events in H_* at the following time instants:

(a) $\mathcal{T}(issue[CS\_id, op]) = \kappa \cdot \mathcal{DT}[invokeCR(op)]$

(b) $\mathcal{T}(invoke_j[op]) = \kappa \cdot \mathcal{DT}[invokeCR(op)]+1$

(c) $\mathcal{T}(response_j[op, res]) = \kappa \cdot \mathcal{DT}[invokeCR(op)]+2$

(d) $\mathcal{T}(outcome[CS\_id, op, res]) = \kappa \cdot \mathcal{DT}[invokeCR(op)]+3$

S3 for each $CS\_id \in \mathcal{CS}$, such that $finalCR(CS\_id) \neq \infty$, set the occurrence of any $eject_j[CS\_id]$, $exit[CS\_id]$, $rem[CS\_id]$ events in H_* at the following time instants:

(a) $\mathcal{T}(exit[CS\_id]) = \kappa \cdot \mathcal{DT}[finalCR(CS\_id)]$

(b) $\mathcal{T}(rem[CS\_id]) = \mathcal{T}(eject_j[CS\_id]) = \kappa \cdot \mathcal{DT}[finalCR(CS\_id)]+1$

S4 if a $crash_i$ event exists in $H|i$, let t be the time of the last event in the $S|i$ history obtained after applying the above transformation rules to all the events in $H_*|i$, append $crash_i$ to $S|i$ at time $t+1$.

In the above transformation rules, $\kappa \in \mathbb{N}$ is an integer scale factor aimed at dilating the time gap between the instants in which a decision for two consecutive consensus round is first delivered by any process $i \in \Pi$, and can be arbitrarily chosen as long it is ensured that:

$$\forall rn \in \mathcal{CR} : \kappa \cdot \mathcal{DT}[rn] + 3 < \kappa \cdot \mathcal{DT}[rn+1] \tag{1}$$

Note that since the $\mathcal{DT}$ sequence is totally ordered, a value κ ensuring condition (1) necessarily exists.

Lemma 5 *S and H_* are equivalent.*

Proof First we note that, for any event $e \in H_*$, there exists a transformation rule that defines the position in time of the corresponding event $e \in S$, i.e. $e \in H_* \Leftrightarrow e \in S$. We now prove that the relative order of events in the process sub-histories of S and H_* is identical, or, formally, that $\forall i \in \Pi, \forall e_0, e_1 \in H_*|i$ such that $e_0 <^{\mathcal{T}}_{H_*|i} e_1$, then $e_0 <^{\mathcal{T}}_{S} e_1$.

Our proof is based on showing that any two pairs of adiacent events $e_0, e_1 \in H_*|i$ (i.e. we say that two events e_0 and e_1 are adiacent in $H_*|i$ if no event is interleaved between them in $H_*|i$) are also adiacent in $S|i$, or formally:

$$\forall e_0, e_1 \in H_*|i : e_0 <^{\mathcal{T}}_{H_*|i} e_1 \wedge (\nexists e_2 \in H_*|i : e_0 <^{\mathcal{T}}_{H_*|i} e_2 <^{\mathcal{T}}_{H_*|i} e_1) \Rightarrow e_0 <^{\mathcal{T}}_{S|i} e_1 \wedge (\nexists e_2 \in S : e_0 <^{\mathcal{T}}_{S|i} e_2 <^{\mathcal{T}}_{S_*} e_1))$$

By H_*'s well-formedness and completeness we have that in $H_*|i$: (i) a try_i event is always followed by a $crit_i[CS\_id]$, (ii) an $exit_i[CS\_id]$ event is always followed by a $rem_i[CS\_id]$ event, and (iii) given that all the operations in $H*$ are successfully executed and by lines 30-35 in Figure 1, an $issue_i[CS\_id, op]$ event is followed by the $invoke_i[op]$, $result_i[op, res]$, and $outcome[CS\_id, op, res]$ events. It is straightforward to observe that the same relative order of events is, respectively, guaranteed by rules S1, S3 and S2. Also, any $crash_i$ is, by defintion, the last event both in $H_*|i$ for all $i \in \Pi$, and, by construction, in S. Hence, given two any adiacent events in $H_*|i$, such that $e_0 <^{\mathcal{T}}_{H_*|i} e_1$, the only cases left to examine are:

- $e_0 = crit_i[CS\_id]$ and $e_1 = issue_i[CS\_id, op]$: this case is only possible if, after the establishment of CS instance CS_id, u_i issues an operation op. Denote with $rn = initCR(CS\_id]$ the consensus round in which the ["CS_req", i, CS_id] was decided. Since all the operations appearing in H_* are successfully executed, and since e_0 and e_1 are adiacent events, it means (see lines 21-23) that in consensus round $rn+1$ the ["issue",op,CS_id] was decided (i.e. $invokeCR(op) = rn+1$). Hence, by transformation rules S1 and S2, in S $\mathcal{T}(\rceil\prime) = \kappa \cdot \mathcal{DT}[rn] + 1$ and $\mathcal{T}(\rceil\infty) = \kappa \cdot \mathcal{DT}[rn+1]$, which, by condition 1, implies that $e_0 <^{\mathcal{T}}_{S_*|i} e_1$. Further, being $e0$ and $e1$ associated with two consecutive consensus rounds, and by transformation rules S1 and S2, e_0 and e_1 are also adjacent.

- $e_0 = crit_i[CS\_id]$ and $e_1 = eject_i[CS\_id]$: this case is only possible if after the establishment of CS instance CS_id, u_i issues an operation op which fails. In this case the $issue_i[CS\_id, op]$ event is removed from H when deriving its completion H_* (see completion rule 3.a). Denote with $rn = initCR(CS\_id]$ the consensus round in which the ["CS_req", i, CS_id] was decided. If op fails, it means that the decision value for consensus round $rn + 1$ was ["Eject_req",CS_id]. Hence, by transformation rules S1 and S2, in S $\mathcal{T}(\rceil\prime) = \kappa \cdot \mathcal{DT}[rn] + 1$ and $\mathcal{T}(\rceil\infty) = \kappa \cdot \mathcal{DT}[rn + 1]$, which, by condition 1, implies that $e_0 <^{\mathcal{T}}_{S_*|i} e_1$. Further, being $e0$ and $e1$ associated with two consecutive consensus rounds, and by transformation rules S1 and S3, e_0 and e_1 are also adjacent.

- $e_0 = crit_i[CS\_id]$ and $e_1 = exit_i[CS\_id]$: this case is possible if after the establishment of CS instance CS_id, u_i requests to exit the CS instance without issuing any operations. Denoting with $rn = initCR(CS\_id)$ the consensus round in which the ["CS_req", i, CS_id] value was decided, we have that $\mathcal{T}(\rceil\prime) = \kappa \cdot \mathcal{DT}[rn] + 1$. At consensus round $rn + 1$, s_i will propose necessarily ["Exit_req",CS_id] (see lines 4-5, 10-11, 12-14). On the other hand, any other process that decides for consensus round rn and proposes a value for consensus round $rn + 1$ can propose either ["Exit_req",CS_id] or ["Eject_req",CS_id]. Hence, by Validity property of consensus, the decision value of consensus round $rn + 1$ must be one of these two values. In both cases $finalCR(CS\_id) = rn + 1$. Hence, by transformation rule S3.a, $\mathcal{T}(\rceil\infty) = \kappa \cdot \mathcal{DT}[rn + 1]$, which, by condition 1, implies that $e_0 <^{\mathcal{T}}_{S_*|i} e_1$. Further, being $e0$ and $e1$ associated with two consecutive consensus rounds, and by transformation rules S1 and S3, e_0 and e_1 are also adjacent.

- $e_0 = outcome_i[CS\_id, op, res]$ and $e_1 = issue_i[CS\_id, op']$, where $op \neq op'$: this case is possible if after the successfull execution of an operation op within CS instance CS_id, u_i issues a new operation op'. Denoting with $rn = invokeCR(op)$ the consensus round in which the ["issue", op, CS_id] value was decided, we have that $\mathcal{T}(\rceil\prime) = \kappa \cdot \mathcal{DT}[rn] + 3$. At consensus round $rn + 1$, s_i will propose necessarily ["issue",op',CS_id] (see lines 6-7, 10-11, 21-23). On the other hand, any other process that decides for consensus round rn and proposes a value for consensus round $rn + 1$ can propose either ["issue",CS_id] or ["Eject_req",CS_id]. Hence, by Validity property of consensus, the decision value of consensus round $rn + 1$ must be one of these two values. However, since H_* only contains successfully executed operations, the decision value of consensus round $rn+1$ must be ["issue",op,CS_id]. Hence, by transformation rule S2.a, in $\mathcal{T}(\rceil\infty) = \kappa \cdot \mathcal{DT}[rn + 1]$, which, by condition 1, implies that $e_0 <^{\mathcal{T}}_{S_*|i} e_1$. Further, being $e0$ and $e1$ associated with two consecutive consensus rounds, and by transformation rules S2.a and S2.d, e_0 and e_1 are also adjacent.

- $e_0 = outcome_i[CS\_id, op, res]$ and $e_1 = exit_i[CS\_id]$: this case is possible if after the successfull execution of an operation op within CS instance CS_id, u_i requests to exit the CS. Denoting with $rn = invokeCR(op)$ the consensus round in which the ["issue", op, CS_id] value was decided, we have that $\mathcal{T}(\rceil\prime) = \kappa \cdot \mathcal{DT}[rn] + 3$. At consensus round $rn + 1$, s_i will propose necessarily ["Exit_req",CS_id] (see lines 4-5, 10-11, 12-14). On the other hand, any other process that decides for consensus round rn and proposes a value for consensus round $rn + 1$ can propose either ["Exit_req",CS_id] or ["Eject_req",CS_id]. Hence, by Validity property of consensus, the decision value of consensus round $rn + 1$ must be one of these two values. In both cases $finalCR(CS\_id) = rn + 1$. Hence, by transformation rule S3.a, $\mathcal{T}(\rceil\infty) = \kappa \cdot \mathcal{DT}[rn + 1]$, which, by condition 1, implies that $e_0 <^{\mathcal{T}}_{S_*|i} e_1$. Further, being $e0$ and $e1$ associated with two consecutive consensus rounds, and by transformation rules S1 and S3, e_0 and e_1 are also adjacent.

□

Being H_* well-formed, and as a direct consequence of Lemma 5 we obtain that:

Lemma 6 *S is well-formed.*

Lemma 7 *S is a CS-sequential history.*

Proof By the transformation rules S1, S2 and S3, for any CS instance subhistory S^{CS}_{id} we have: $\mathcal{T}(\mathcal{I}(id)) = \kappa \cdot \mathcal{DT}[initCR(id)]$ and $\mathcal{T}(\mathcal{F}(id)) = \kappa \cdot \mathcal{DT}[finalCR(id)]+1$. By Lemma 6 it must also

be that $initCR(id) < finalCR(id)$. Now assume by contradiction that two non-sequential CS instance subhistories S^{CS}_{id}, $S^{CS}_{id'}$ exist such that $initCR(id) < initCR(id')$ and $finalCR(id) > initCR(id')$, i.e. $initCR(id) < initCR(id') < finalCR(id)$. This implies that all the processes that complete consensus round $initCR(id)$ decide ["CS_req",i,id], where $i \in \Pi$ setting the local CSOwner variable to the i value (see lines 24-29). Also, if a new CS instance is established as a result of consensus round $initCR(id') > initCR(id)$, it means that at least some process k must have proposed a ["CS_req",j,id'] value for consensus round $initCR(id')$ (see line 19). This is possible only if at process k the CSOwner variable, which was holding the i value at time $\mathcal{DT}[initCR(id)]$, must have been subsequently set to the $\perp$ value as a consequence of the delivery of a decision value in the set {["Exit_req",id], ["Eject_req",id]} in some consensus round rn, where $initCR(id) < rn < initCR(id')$. But, by transformation S3.b, this would imply that $\mathcal{T}(\mathcal{F}(id)) = \kappa \cdot \mathcal{DT}[rn] + 1 \neq \kappa \cdot \mathcal{DT}[finalCR(id)] + 1$. Hence we obtain a contradiction and the claim follows. □

By Lemmata 5, 6 and 7 we obtain the following result:

Theorem 8 *The algorithm in Figure 1 satisfies property* **WME1**.

Theorem 9 (WM2) $<^{CS}_{H_*} \subseteq <^{CS}_{S}$

Proof Assume by contradiction that two CS instance subhistories of H_* exist, namely H^{CS}_{*id} and $H^{CS}_{*id'}$ such that $H^{CS}_{*id} <^{CS}_{H_*} H^{CS}_{*id}$ and that either $S^{CS}_{id'} <^{CS}_{S_*} S^{CS}_{id}$ or that $S^{CS}_{id'}$ and S^{CS}_{id} are concurrent according to $<^{CS}_{S_*}$. By Lemma 7 the latter case is not possible. On the other hand, if $S^{CS}_{id'} <^{CS}_{S_*} S^{CS}_{id}$ then it should be in H_* that $initCR(id') < initCR(id)$. But if $H^{CS}_{*id} <^{CS}_{H_*} H^{CS}_{*id'}$ then $\mathcal{T}(\mathcal{F}(id)) < \mathcal{T}(\mathcal{I}(id'))$, which implies that, in H_*, $initCR(id) < finalCR(id) < initCR(id')$. Hence we get a contradiction and the claim follows. □

Theorem 10 (WM3) *The stub-resource subhistory H^{SR}_* is isomorphic to the user-stub successful operations subhistory of S.*

Proof Rule 3.a used to derive the legal completion of H, namely H_*, ensures that there are no pending successful operations in H_*, thus allowing to establish a bijective function between each $issue_i[CS\_id, op]$ and the correlated $invoke_i[op]$. and between each $outcome_i[CS\_id, op, res]$ and the correlated $response_i[op, res]$. Also, FIFO property of reliable broadcast ensures that if $\mathcal{T}(issue_i[CS\_id, op]) < \mathcal{T}(issue_i[CS\_id, op'])$, then $invokeCR(op) < invokeCR(op')$. Hence, by Lemma 1 and transformation rule S2, the relative order of the correlated $issue_i[CS\_id, op]$ and $outcome_i[CS\_id, op, res]$ output events in $S^{US|op}$ is consistent with the order of the $invoke_i[op]$, $response_i[op, res]$ events in H^{SR}_*. □

Theorem 11 (1CS) *The history of the replicated shared resource (i.e. H^{SR}_*), is equivalent to a serial execution on a single replica of the shared resource.*

Proof Since an $invoke_i[op]$ event is generated by a stub s_i, with $i \in \Pi$, only if a consensus round having number rn has ["issue",op, CS_ID] as its decision value, by Lemmata 1, 3 and the Uniform Agreement property of consensus, every resource r_i executes the same set of deterministic operations, say $invokedOps \in Operations$. Also, the operations in $invokedOps$ are executed in the same order, namely the one associated with the sequential execution of consensus rounds. Hence, the claim follows. □

Lemma 12 *If a correct process proposes a value in a consensus round, it eventually delivers a decision for that consensus round.*

Proof Assume by contradiction that a correct process i proposes a value v in a consensus round rn and never decides. A process can propose a value in a consensus round rn only if it has delivered a decision for all the previous consensus rounds, i.e. consensus rounds $rn' \in [1, rn-1]$. Then, by Termination properties of consensus, if i decides for consensus round $rn-1$, then all correct processes must eventually deliver a decision value for round $rn-1$. Also, if i never decides a value for consensus round rn, it means that there

exists a majority of correct processes Δ which completes consensus round $rn-1$ but never proposes a value for consensus round rn. But, given that a value proposed by a process must have been previously reliably broadcast (lines 3,5,7,9,11,12,15,18,21), then, by Validity and Agreement properties of Reliable Broadcast, it means that the value v proposed by i (which is a correct process) must be received also by all the processes in Δ, since they are also correct. There are three cases for what concerns the timing of delivery the value v at a process $j \in \Delta$:

1. when j received the value v, this was already present in the decision set, and hence was discared. However, by the code, a value is added to the decision set only if this has been decided in a round. Hence the value v that i proposes in rn was already decided in a previous consensus round. But this is impossible by Lemma 2. Hence we get a contradiction.

2. when j received the value v, it appended it to the props queue, but then removed it from there before completing round $rn-1$. By the code, a value v can be erased from the proposal queue only if it is the decision value of a consensus round. Hence the value that i proposes in round rn was already decided in a previous consensus round. But this is impossible by Lemma 2.

3. when j received the value v, it appended it to the props queue, and after the completion of round rn-1, v is still there, but the conditions enabling its proposal are never enabled henceforth. Note that, since j completes round rn-1 and does not propose a value for round rn, it must have the wait flag set to false. Hence, if j has the value that i proposed in round rn in the props queue but does not propose it means that, at the end of round $rn-1$, processes i and j are storing different values either for the variable CSOwner or currCS_ID. However, both these variables are only updated at the end of a consensus round depending on its corresponding decision value. Hence by Agreement property of consensus i and j, at the end of any consensus round that they complete, store the same values for both CSOwner and currCS_ID. Hence we obtain a contradiction, and the claim follows.

□

Theorem 13 (CS-Release Progress) *If a correct process resigns, it enters its remainder section.*

Proof Assume by contradiction that a correct user u_i generates a $exit_i[id]$ event but never enters its remainder section. A stub s_i that receives an $exit_i[id]$ event, reliably broadcasts a ["CS release",id] message, which by validity of Reliable Broadcast, it eventually self-delivers. Since, by user's well-formedness, a user u_i generates a $exit_i[id]$ event only if it is in the critical section having id as CS identifier. This implies that a consensus round number y exists whose decision value was ["CS_req",i,id], and that s_i completed this consensus round and set its local currCS_ID variable to the value id. Let x be the value of the rn variable at stub s_i when the user generates $exit_i[id]$. When s_i self-deliveres the "CS release",id] message it proposes the ["Exit_req",id] value for consensus round $x+1$. By Lemma 12 s_i eventually delivers a decision value for consensus round $x+1$. Also, by Lemma 4, we have that $x > y$. The set $\pi \in \Pi$ of all processes that complete consensus round x, which by Termination property of consensus include all correct processes, have the local variables $CSOwner$ and $currCSid$ set, respectively, to the i and id values. Hence, all processes that enter consensus round $x+1$ can only propose either the value ["Exit_req",id] (in case they deliver the reliable broadcast triggered by s_i at line 5) or ["Eject_req",id] (in case they suspect i to have crashed). Hence, by Validity property of consensus, the decision of consensus round $x+1$ must be one of these two proposed values. In both cases, u_i enters the remainder section (see lines 37-42, 43-49). Hence, we obtain a contradiction and the claim follows. □

Theorem 14 ***(Operation Progress)*** *If a correct process issues an operation, eventually the operation either fails or succeeds, and eventually all the operations it issues succeed.*

Proof We first prove that if a correct process issues an operation, then it delivers an outcome for it.

By user's well-formedness an operation *op* is issued by i in CS instance id only if i it delivered a $crit_i[id]$, never issued $exit_i[id]$ and did not delivered neither an $eject_i[id]$. Then, since H is well-formed by Lemma 4, it can not have delivered a $rem_i[id]$. Let c be the value of the variable rn, i.e. the identifier of the just

completed consensus round identifier, at i when it issues *op*. As round c terminates, the variable CSOwner and currCS_ID must be set respectively to i and id at process i. Hence, by the Uniform Agreement property of consensus, all processes that complete round c must have the value of the variable CSOwner set to i. Also, by the Termination property of consensus and since there a majority of correct processes, eventually a majority of correct processes complete round c, after which their local CSOwner and currCS_ID variables are set, respectively, to the i and id values. As s_i issues *op*, it reliably broadcast it. Since i is correct, it eventually receives its broadcast and adds ["issue",op,id] to the proposal queue. (["issue",op,id] can not be already found by s_i in the decision set as s_i is the only one that can issue this operation - recall we are assuming operations to be uniquely identified - and it still has not done so). Hence, s_i proposes ["issue",op,id] as the value of the consensus round c+1. By Lemma 12, s_i eventually decides for round c+1. As the history H is well formed, the only legal decision values for round c+1 are either ["eject_req",id] or ["issue",op,id], which lead to a delivery of an outcome for *op*.
Now we prove that eventually every correct process successfully executes all the operations it issues.

By eventual strong accuracy, a time t exists after which every correct process is no longer suspected by any process. Hence, if a user u_i requests the CS instance id at time $t' > t$ and issues an operation, proposing it in consensus round rn, it is not possible that an ["eject_req",id] value is proposed by any other process. Finally, by the previous result, i delivers an outcome for that operation, and, in absence of ["eject_req",id] proposal, the only possible outcome is successful. Hence the operation is invoked at i and since interactions with r_i are non-blocking eventually an $outcome[id, op, res]$ is generated by s_i. □

Theorem 15 ***(Starvation-Freedom)*** *A correct process that volunteers eventually enters the critical section, if no other process stays forever in its critical section.*

Proof Assume that no process stays forever in its critical section, and that a correct process i that requests the critical section never enters it. Upon generation of a try_i event by u_i, s_i reliably broadcasts a ["CS_req",i,id] which is eventually appended (in FIFO order) to the props queue by all the other correct processes. Since values of type ["CS_req",-,-] are extracted from the props queue in FIFO order, given that we are assuming that all the critical sections that are entered are eventually exited, we have that, independently of the order in which the ["CS_req",i,id] value is enqueued in the props queue of any correct process, ["CS_req",i,id] will eventually be selected as the proposal value of consensus round rn by some correct process $j \in \Pi$. Further, by Lemma 12, since j is correct and proposes a value in consensus round rn, then it eventually decides a value for round rn. This implies that all correct processes propose a value for round rn. Given that at the end of round rn-1 all correct processes must have set the loval currCS_ID variable to $\perp$, the only possible proposal, and hence decision value, for round rn is of type ["CS_req",-,$id*$]. If the decision of round rn is ["CS_req",i,id], we obtain a contradiction and the claim follows. Otherwise, i.e. $id* \neq id$, we have that the ["CS_req",i,id] proposal has advanced in the props queue of a position in at least one process. Hence, eventually, the ["CS_req",i,id] will become first in the props queue at each process that is not crashed. It follows by the Validity property of consensus, and by Lemma 12 that the ["CS_req",i,id] value will be eventually decided by s_i, which will generate a $crit_i$ event and allow i enter the CS (see line 29). A contradiction. □

7 On the weakest failure detector for WME

We have just shown that the WME problem is solvable in an asynchronous system using the $\Diamond P$ failure detector with a majority of correct processes. Here we complement this result by showing that the eventually perfect failure detector $\Diamond P$ is *necessary* to solve WME in an asynchronous system independently from the number of correct processes. Hence, $\Diamond P$ is the weakest failure detector for solving the WME problem. This result is achieved by showing that, given an algorithm $\mathcal{A}$ that solves WME using a failure detector $\mathcal{D}$, it is possible to construct a reduction algorithm $\mathcal{R}_{\mathcal{D}\rightarrow\Diamond P}$ that uses $\mathcal{A}$ to emulate $\Diamond P$. In other words, we show that if a failure detector $\mathcal{D}$ solves WME, then $\mathcal{D}$ is not *strictly weaker* than $\Diamond P$ [5].

We show such a reduction algorithm $\mathcal{R}_{\mathcal{D}\rightarrow\Diamond P}$ in Figure 2. The processes run n instances of the algorithm $\mathcal{A}$, denoted as $f^1, \ldots, f^n$. The events defining the interactions of process i within instance f^j are tagged

Set output$_i$=$\emptyset$; // Set of processes suspected to have crashed

$\forall j \in \Pi$ **do** try_i^j;

upon $crit_i^i[CS\_id]$
boolean ejected=$false$;
while (¬ejected) **do**
send [Alive] to any process $k \in \Pi$;
$issue_i^i[CS\_id, increment^i]$;
wait $(outcome_i^i[CS\_id, increment^i, currValue^i] \vee ejected_i^i[CS\_id])$
if $ejected_i^i[CS\_id]$) ejected=$true$;
try_i^i;

upon $crit_i^j[CS\_id]$
$issue_i^j[CS\_id, increment^j]$;
wait $(outcome_i^j[CS\_id, increment^j, currValue^j] \vee ejected_i^j[CS\_id])$
if $(outcome_i^j[CS\_id, increment^j, currValue^j])$ output$_i$=output$_i \cup \{j\}$;
$exit_i^j[CS\_id]$;
wait $rem_i^j[CS\_id]$;
try_i^j;

upon receive[Alive] from process $k \in \Pi$
output$_i$=output$_i - \{k\}$;

Figure 2: Reduction algorithm $\mathcal{R}_{\mathcal{D}\rightarrow\Diamond P}$ (proc. i).

with an additional j super-script, e.g. try_i^j, $crit_i^j$, $issue_i^j$ and so on. Each instance f^i is associated with an independent replicated shared resource r_i, i.e. an independent integer counter initially set to 0 and which exports a single operation $increment^i$, returning $currValue^i$, i.e. the current value of the counter.

Every process initially attempts to enter the CS associated with all the n different algorithm instances. However, once a process i enters the CS associated with the algorithm instance f^j, with $j \in [1, n]$, it behaves differently depending on the relative values of i and j.

Specifically, every time that i enters the CS associated with f^i, it cyclically 1) sends *Alive* heartbeat messages to all the other processes, and 2) issues an $increment^i$ operation, without *ever* exiting from the CS. If, in the meanwhile, i is ejected from the CS, it attempts to enter it again, repeating unmodified its behavior upon any subsequent entry in this CS.

Instead, if process i enters the CS associated with f^j, where $i \neq j$, it issues a single $increment^j$ operation and, if this successfully executes, it adds j to its local *ouput* variable, i.e. to the set containing the identities of the suspected processes which is used to emulate the $\Diamond P$ failure detector. Independently of the success of the issued operation, i then resigns from the CS, and immediately volunteers again.

Finally, a process i removes a process j from his *output* set of suspected processes only upon reception of a heartbeat message from j.

Lemma 1. The output of the reduction algorithm of Figure 2 satisfies the properties of the eventually perfect failure detector $\Diamond P$.

Proof: An eventually perfect failure detector must ensure the *Eventual Strong Accuracy* and the *Strong Completeness* properties.

Assume by contradiction that the *Eventual Strong Accuracy* property of $\Diamond P$ is violated:

$$\exists \{i,j\} \in Correct(\Pi), \nexists t \in \mathbb{N} : \forall t' > t \;\; i \notin output_j(t')$$

However, since i is the only process that, once entered the CS associated with f^i never resigns, by the Starvation-Freedom property, whenever i requests the CS for f^i, there is a time at which it enters the corresponding CS, and sends an *Alive* message to all processes. Also, for the Operation Progress property to hold, there exists a time t^{lastCS} after which i has already successfully executed at least one operation in its CS, and also successfully executes all the subsequent operations it issues without ever being ejected from the CS.

Now, assume by contradiction that at time $t > t^{lastCS}$ some other process j enters the CS instance associated with f^i by generating a $crit^i_j[id']$ event and successfully executes an $increment^i$ operation which returns the value k'. Let k be the $currValue^i$ returned as result of the last $increment^i$ operation successfully executed by i at time t^{lastCS}. We can distinguish three cases:

1. $k' = k$: which is impossible, by the 1CS property, as in this case we would incur in a violation of the sequential behavior of the counter.

2. $k' > k$: which corresponds to serializing $H^{CS}_{id'}$ after H^{CS}_{id}. This is impossible since within H^{CS}_{id}, by the Operation Progress property, i can successfully execute an arbitrary number of $increment^i$ operations which, by WME3 and WME1, must return all the successors of k (with no gaps), k' included. But, then the same value k' would be output by replica r_i and r_j as a result of two independent invocations, leading to a violation of the 1CS property.

3. $k' < k$: which corresponds to serializing H^{CS}_{id} after $H^{CS}_{id'}$. But, by assumption, in H $invoke^i_i[increment^i] <^{\mathcal{T}} invoke^i_j[increment^i]$. Whereas, by serializing H^{CS}_{id} after $H^{CS}_{id'}$, we get $issue^i_j[id', increment^i] <^{\mathcal{T}} issue^i_i[id, increment^i]$, violating property WME3.

Thus after time t^{lastCS} no process $j \neq i$ adds i to the set of suspected processes. Also, i keeps periodically sending *Alive* of messages to all other processes. By the reliability of channels these messages are eventually delivered by all correct processes which will remove i from the set of suspected processes. Denote with t^* the maximum time in which a correct process receives the first of the *Alive* messages sent by process i after time t^{lastCS}. After time t^* no correct processes ever suspects i, hence we get a contradiction.

Now we prove that the algorithm in Figure 2 ensures the *Strong Completeness* property of $\diamondsuit P$, i.e. if a process i crashes, then every correct process eventually suspects i. If process i crashes there must be a time t after which no process delivers any *Alive* message that i has ever sent out before $Crash(i)$. On the other hand, after $Crash(i)$, any correct process $j \neq i$ that enters the CS associated with f^i eventually resigns, and immediately retries to enter the CS. Hence, by the Starvation-Freedom property, every correct processes enters the CS associated with f^i and issues an increment operation on r_i an infinite number of times. By Operation Progress property, we get that eventually all of these issued operations will succeed, each time causing j to add i to its set of suspected processes. Hence, there exists a time $t' > t$ after which all correct processes i) have already added at least once i to their set of suspected processes, and ii) never remove i from the set of suspected processes. Thus, the claim follows. □

As a corollary of this last lemma we get:
Theorem 1. If a failure detector $\mathcal{D}$ solves WME, then $\mathcal{D}$ is not strictly weaker than $\diamondsuit P$. □

8 Concluding Remarks

In this paper we introduced the Weak Mutual Exclusion problem, a variant of the classical Distributed Mutual Exclusion problem in which users access a shared resource which logically appears as single and indivisible, but that is physically replicated at each participating process for both fault-tolerance and performance reasons.

We have shown that, unlike the Distributed Mutual Exclusion problem, that is only solvable in a synchronous system, the Weak Mutual Exclusion abstraction is implementable even in presence of partial synchrony. More in detail, we have shown that the Eventually Perfect failure detector, $\Diamond P$, is the weakest failure detector for solving the Weak Mutual Exclusion problem, and a presented solution that uses $\Diamond P$ and tolerates the crash of a minority of processes.

Relying on the WME abstraction to regulate the access to replicated resources has the following practical benefits:

Robustness: pessimistic concurrency control is widely used in commercial off the shelf systems, e.g. DBMSs and operating systems, because of its robustness and predictability in presence of conflict intensive workloads. The WME abstraction lays a bridge between these proven contention management techniques and replica control schemes. Analogously to centralized lock based concurrency control, WME reveals particularly useful in the context of conflict-sensitive applications, such as transactional or interactive systems, where it may be preferable to bridle concurrency rather than incurring the costs of application level conflicts, such as transactions abort or re-submission of user inputs.

Performance: the ability of the WME abstraction to serialize the sequences of operations issued by each user within the CS can also provide benefits for what concerns the performances of typical building blocks used by replication schemes. For instance, it is known that the performance of consensus can be significantly enhanced (i.e. its decision latency can be reduced to a single communication step [4, 22]) if there are no two processes simultaneously proposing different values. This is exactly what happens in nice runs of the WME algorithm presented in this paper: once a process p establishes a CS instance, and as long as it does not resign, any other process proposes as input value to the consensus the same sequence of operations, namely those issued by p within its CS. Quantitatively evaluating the performance benefits from the employment of WME in a realistic distributed system is part of our future work.

Simplicity: finally, the WME abstraction exposes a simple lock-like interface that is familiar even to developers with no experience with distributed programming.

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] R. Boichat, P. Dutta, and R. Guerraoui. Asynchronous leasing. In *Proc. of the The International Workshop on Object-Oriented Real-Time Dependable Systems*, page 180, Washington, DC, USA, 2002. IEEE Computer Society.

[4] F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *Proc. of the International Conference on Parallel Computing Technologies*, pages 42–50. Springer-Verlag, 2001.

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

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

[7] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley Longman Publishing Co., Inc., 1988.

[8] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, pages 593–602, New York, NY, USA, 1992. ACM.

[9] D. Powell (ed.). Special issue on group communication. 39(4):50–97, 1996.

[10] 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.

[11] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Com.ACM*, 8(9):569, 1965.

[12] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[13] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1266–1276, 2003.

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

[15] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the Symposium on Operating Systems Principles*, pages 202–210. ACM, 1989.

[16] R. Guerraoui and M. Raynal. A leader election protocol for eventually synchronous shared memory systems. In *Proc. of the Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 75–80, Washington, DC, USA, 2006. IEEE Computer Society.

[17] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[18] 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.

[19] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[20] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the The 18th International Conference on Distributed Computing Systems*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[21] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

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

[23] B. Liskov, M. Day, and L. Shrira. Distributed object management in thor. In *Distributed Object Management*, pages 79–91. Morgan Kaufmann, 1993.

[24] S. Lodha and A. Kshemkalyani. A fair distributed mutual exclusion algorithm. *IEEE Trans. Parallel Distrib. Syst.*, 11(6):537–549, 2000.

[25] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[26] 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.

[27] S. M. Pike and P. Sivilotti. Dining philosophers with crash locality 1. In *Proc. of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS*, pages 22–29. IEEE, 2004.

[28] M. Raynal and D. Beeson. *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA, 1986.

[29] L. Rodrigues, H. Miranda, R. Almeida, a. M. Jo and P. Vicente. The globdata fault-tolerant replicated distributed object database. In *Proc. of the First EurAsian Conference on Information and Communication Technology*, pages 426–433, London, UK, 2002. Springer-Verlag.

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