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], [31] 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 version 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 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.

This paper was partially supported by the Pastramy project (PTDC/EIA/72405/2006) funded by the Portuguese Fundação para a Ciência e Tecnologia.

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.

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,
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

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.

and tolerating the failure of a minority of process. In Section 7 we show that $\diamondsuit$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 $\diamondsuit P$ 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 predeclared 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 predeclare 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 $<try_i, crit_i[CS\_id]>$ 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, append zero or more $invoke_i[op]$, $response_i[op, res]$ and $outcome_i[CS\_id, op, res]$ 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 $<exit_i[id], rem_i[id]>$, or the $rem_i[id]$ 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. $\forall\{e_0, e_1\} \in H^{SR}, e_0 <_H^{\mathcal{T}} e_1 \Leftrightarrow \mathcal{B}(e_0) <_H^{\mathcal{T}} \mathcal{B}(e_1)$.

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:

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 the user-stub subhistory 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 i that volunteers eventually enters the critical section, if no other process stays forever in its critical section.

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

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

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 reordered 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 "preemptible 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 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.

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 $\diamondsuit$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.

The proof of the algorithms correctness with respect to the specification of the Weak Mutual Exclusion problem is omitted due to space constraints but can be found in the extended technical report [30].

7. On the weakest failure detector for WME

We have just shown that the WME problem is solvable in an asynchronous system using the $\diamondsuit P$ failure detector with

```
Set decisions;          // already decided values
Queue props;            // proposals queue
int rn=0;               // current consensus round
int currCS_ID=⊥;        // 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);

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", currCS_ID);

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=currCS_ID do
13.    propose(rn++, ["Exit_req", currCS_ID]);
14.    wait=true; exiting=true;

15. upon ¬wait ∧ ∃["eject",id]∈props s.t. id=currCS_ID do
16.    propose(rn++, ["Eject_req", currCS_ID]);
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 ∧ currCS_id≠ ⊥ ∧
        ∃["issue",op,id]∈props s.t. id =currCS_id do
22.    propose(rn++,
        first ["issue", op,id]∈props s.t. id =currCS_id]);
23.    wait=true;

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

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

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

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

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

a majority of correct processes. Here we complement this result by showing that the eventually perfect failure detector $\diamondsuit P$ is *necessary* to solve WME in an asynchronous system independently from the number of correct processes. Hence, $\diamondsuit P$ 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 $\mathcal{R}_{\mathcal{D}\rightarrow\diamondsuit P}$ that uses $\mathcal{A}$ to emulate $\diamondsuit P$. In other words, we show that if a failure detector $\mathcal{D}$ solves WME, then $\mathcal{D}$ is not *strictly weaker* than $\diamondsuit P$ [5].

We show such a reduction algorithm $\mathcal{R}_{\mathcal{D}\rightarrow\diamondsuit P}$ 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 with an additional j superscript, 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

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 ($\neg$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).

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 $crit_j^i[id']$ 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 $H_{id'}^{CS}$ after H_{id}^{CS}. This is impossible since within H_{id}^{CS}, 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_{id}^{CS} after $H_{id'}^{CS}$. But, by assumption, in H $invoke_i^i[increment^i] <^{\mathcal{T}} invoke_j^i[increment^i]$. Whereas, by serializing H_{id}^{CS} after $H_{id'}^{CS}$, we get $issue_j^i[id', increment^i] <^{\mathcal{T}} issue_i^i[id, increment^i]$, 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 $\Diamond P$, 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 $\Diamond P$. □

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.

Acknowledgments

The authors wish to sincerely thank the anonymous reviewers for their insightful comments and valuable suggestions.

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] P. Romano, L. Rodrigues, and N. Carvalho. The weak mutual exclusion problem. Technical Report 52/2008, INESC-ID, Nov. 2008.

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