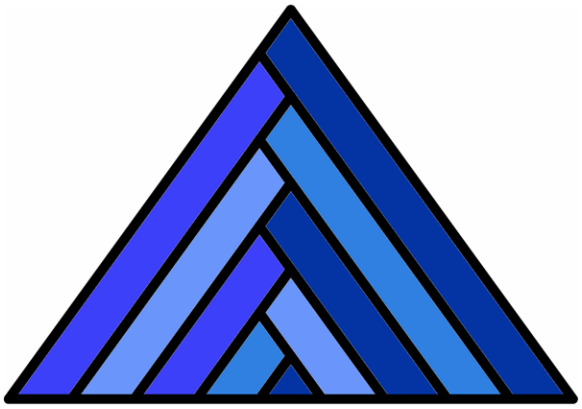
The Weakest Failure Detector for Wait-Free, Eventually Fair Mutual Exclusion

Yantao Song, Scott M. Pike, and Srikanth Sastry

Texas A&M University
Department of Computer Science
College Station, TX 77843-3112, USA
{yantao, pike, sastry}@tamu.edu

Technical Report: TAMU-CS-TR-2007-2-2



Delta Computing Lab

The Weakest Failure Detector For Wait-Free, Eventually Fair Mutual Exclusion

Yantao Song, Scott M. Pike, and Srikanth Sastry
Texas A&M University

We establish the necessary conditions for solving wait-free, eventually fair mutual exclusion in message-passing environments subject to crash faults. Wait-freedom guarantees that every correct hungry process eventually enters its critical section. Eventual fairness guarantees that every run has an infinite suffix during which no correct hungry process is overtaken more than b times. Previously, we showed that the eventually perfect failure detector ($\diamond\mathcal{P}$) is *sufficient* to solve wait-free, eventually fair mutual exclusion. The present paper completes this reduction by proving that $\diamond\mathcal{P}$ is also *necessary*, and hence is the weakest oracle to solve this problem. Our construction uses wait-free, eventually fair exclusion to build an elastic clock that provides an eventually reliable time-out mechanism for detecting crashed processes. This lease-based implementation of $\diamond\mathcal{P}$ uses bounded-capacity, non-FIFO channels, and is crash-quiescent. The construction itself may be of independent interest, insofar as it demonstrates how fairness properties can be sufficient to encapsulate temporal assumptions about partial synchrony.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *mutual exclusion*; *scheduling*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; *verification*

1. INTRODUCTION

It is well-known that many fundamental problems in distributed computing are unsolvable by deterministic, asynchronous systems subject to crash faults [Fich and Ruppert 2003]. Such impossibility results are based on the intrinsic difficulty of reliable fault detection in asynchronous environments. Consequently, crash-tolerant algorithms require timing assumptions (either *explicit* or *implicit*) in order to detect and react to crash faults.

Explicit timing assumptions result in concrete models of partial or full synchrony, which consider the existence and/or knowledge of timing parameters. For example, the partially synchronous model $\mathcal{M}_1$ assumes the existence of unknown upper-bounds on maximum message delay and relative process speeds [Dwork et al. 1988]. A drawback of explicit temporal assumptions, however, is that they tend to obscure the fundamental properties of fault detection necessary or sufficient to solve a given problem.

An alternative method for circumventing impossibility results is to augment asynchronous environments with abstract oracles that encapsulate *implicit* timing as-

This work was supported by the Advanced Research Program of the Texas Higher Education Coordinating Board under Project Number 000512-0007-2006. Author addresses: Department of Computer Science, Texas A&M University, College Station, TX 77843-3112, USA. Email: {yantao, pike, sastry}@cs.tamu.edu.

Submitted to the ACM Symposium on Principles of Distributed Computing, Feb. 2007.

sumptions necessary to support assertional properties of fault detection. This approach was advocated by Chandra and Toueg in their pioneering work on unreliable failure detectors [Chandra and Toueg 1996]. Briefly, an unreliable failure detector can be viewed as a distributed oracle that can be queried for (potentially incorrect) information about process crashes.

Most oracles cannot be implemented in asynchrony. Instead, they encapsulate *implicit* assumptions about time necessary to implement abstract detection properties. As such, failure detectors provide an essential separation of concerns between fault detection properties and their underlying implementation mechanisms. From a broader perspective, however, failure detectors can be viewed as a unifying abstraction for understanding the relative solvability of crash-tolerant problems [Fromentin et al. 1999].

1.1 Contributions

This paper considers mutual exclusion, a fundamental problem that has been studied for decades in various models of distributed computing [Raynal 1986; Anderson et al. 2003]. In particular, we explore a new dimension of this classic problem with the following result: *wait-free, eventually fair exclusion* [Song and Pike 2007] *is equivalent to the class of eventually perfect failure detectors* ($\diamond\mathcal{P}$) from the Chandra-Toueg hierarchy [Chandra and Toueg 1996].

Subsequent sections will explain the foregoing terms in greater detail. For now, we motivate these concepts informally. The variant of mutual exclusion we consider guarantees that every correct hungry process eventually eats (wait-freedom), and that every run has an eventually fair suffix where no correct hungry process is overtaken more than b times. The bound b is unknown and may vary from run to run. Similarly, the time to convergence for eventual fairness is also unknown and may vary from run to run. As such, these parameters represent different dimensions of temporal uncertainty.

The oracle $\diamond\mathcal{P}$ encapsulates temporal uncertainty in terms of the (un)reliability of the suspect list it provides. Informally, $\diamond\mathcal{P}$ always suspects crashed processes, but only *eventually* stops suspecting correct processes. Thus, $\diamond\mathcal{P}$ is allowed to make mistakes by wrongfully suspecting correct processes finitely many times during any run. As before, the time to convergence is unknown and may vary from run to run. Still, $\diamond\mathcal{P}$ is sufficiently powerful to solve many crash-tolerant distributed problems such as consensus [Chandra and Toueg 1996], stable leader election [Aguilera et al. 2001], and crash-locality-1 dining philosophers [Pike and Sivilotti 2004], among others.

In previous work, we demonstrated the sufficiency of $\diamond\mathcal{P}$ for solving wait-free, eventually fair dining philosophers [Song and Pike 2007]. Dining is a generalization of mutual exclusion where the conflict graph can be an arbitrary connected topology. As such, mutual exclusion is just a special case of dining where the conflict graph happens to be a clique. It was previously known that $\diamond\mathcal{P}$ was insufficient to solve wait-free dining (and hence wait-free mutual exclusion) under *perpetual weak exclusion* [Pike and Sivilotti 2004]. This safety model requires that no two live neighbors eat simultaneously. The primary bottleneck is that $\diamond\mathcal{P}$ provides insufficiently reliable information about crashes to guarantee both wait-freedom and perpetual safety.

To circumvent this impossibility result, we considered wait-free, eventually fair dining under an alternative model known as *eventual weak exclusion* ($\diamond\mathcal{WX}$) [Song and Pike 2007]. This more permissive model of exclusion requires only that every run must converge to an infinite suffix where no two live neighbors eat simultaneously. Intuitively, $\diamond\mathcal{WX}$ defines a class of unreliable schedulers that can make mistakes by wrongfully scheduling neighbors to eat simultaneously at most finitely many times during any run. Interestingly, $\diamond\mathcal{P}$ is actually *sufficient* to solve wait-free, eventually fair mutual exclusion under $\diamond\mathcal{WX}$.

A natural follow-up to the foregoing result is whether $\diamond\mathcal{P}$ is also *necessary*. This question seeks to understand whether or not $\diamond\mathcal{P}$ encapsulates the minimal temporal assumptions sufficient to solve this problem. We answer this question affirmatively by proving that $\diamond\mathcal{P}$ is, in fact, the *weakest* failure detector to solve wait-free, eventually fair mutual exclusion subject to $\diamond\mathcal{WX}$.

The primary technical challenge is demonstrating how the uncertainty implied by an unreliable (but eventually accurate) oracle can be reduced to an unreliable (but eventually fair) scheduler. A corollary of our work is that both problems encapsulate equivalent assumptions about partial synchrony. Like oracles, we conjecture that *fairness models* may serve as an alternative abstraction for unifying implicit assumptions about time.

2. BACKGROUND AND TECHNICAL FRAMEWORK

We consider an asynchronous environment where message delay, clock drift, and relative process speeds are unbounded. A system is modeled by a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that communicate only by asynchronous message passing. Specifically, we assume that processes communicate through reliable non-FIFO channels; every message sent to a correct process is eventually received by that process, and messages are neither lost, duplicated, nor corrupted. Additionally, we posit a discrete global clock $\mathcal{T}$ whose range of clock ticks is the set of natural numbers $\mathbb{N}$. $\mathcal{T}$ is merely a conceptual device and inaccessible to processes in Π.

Fault Patterns. Processes may fault only by crashing. Processes crash only as the result of a crash fault, which occurs when a process ceases execution without warning and never recovers [Cristian 1991]. A *fault pattern* F models the occurrence of crash faults in a given run. Specifically, F is a function from the global time range $\mathcal{T}$ to the powerset of processes 2^{Π}, where $F(t)$ denotes the subset of processes that have crashed by time t. Since crash faults are permanent, F is monotonically non-decreasing. That is, $F(t) \subseteq F(t+1)$. We say that i *is faulty in* F if $i \in F(t)$ at any time t; otherwise, we say that *a process* i *is correct in* F. Additionally, a process i is *live at time* t if $i \notin F(t)$. Thus, correct processes are always live, and faulty processes are live only prior to crashing.

Failure Detectors. An unreliable failure detector can be viewed as a distributed oracle that can be queried for (possibly incorrect) information about process crashes [Chandra and Toueg 1996]. Each process has access to its local detector module that outputs a set of processes currently suspected of having crashed. Unreliable failure detectors are characterized by the kinds of *mistakes* they can make.

Mistakes include false-negatives (i.e., not suspecting crashed processes) and false-positives (i.e., wrongfully suspecting correct processes). Each failure detector class is defined by two properties: *completeness* (which restricts false negatives) and *accuracy* (which restricts false positives). The eventually perfect failure detector $\diamond\mathcal{P}$ satisfies [Chandra and Toueg 1996]:

—**Strong Completeness:** *Every crashed process is eventually and permanently suspected by all correct processes.*

—**Eventual Strong Accuracy:** *For every execution, there exists an unknown time after which no correct process is suspected by any correct process.*

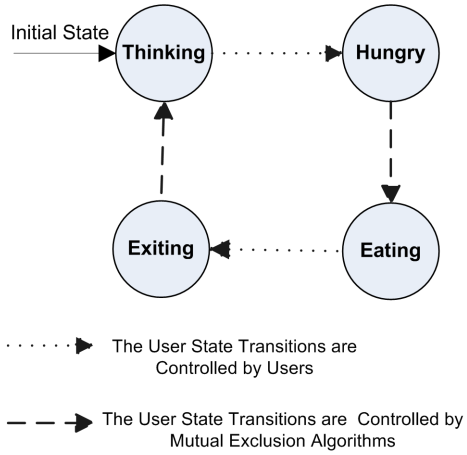


Fig. 1. State Transition Diagram for Mutual Exclusion

Thus, $\diamond\mathcal{P}$ may suspect correct processes finitely many times in any execution. However, $\diamond\mathcal{P}$ must *converge* to at some point, after which the oracle provides reliable information about process crashes. Unfortunately, the time to convergence is unknown and may vary from run to run. In our reduction, we will implement the completeness and accuracy properties of $\diamond\mathcal{P}$ using a black-box solution to wait-free, eventually fair mutual exclusion under $\diamond\mathcal{WX}$ (defined formally below).

Mutual Exclusion. This fundamental problem has been studied extensively since its original presentation in 1965 by Dijkstra [1965]. Mutual exclusion is essentially a problem in process synchronization for coordinating safe access to exclusive *critical sections* of code. Processes cycle among four states: thinking, hungry, eating, and exiting (Figure 1). Initially, every process is thinking. This state represents independent execution. Processes are permitted to think forever, but may also transit to the hungry state at any time; this happens when the processes attempt to enter their critical section. Hungry processes are said to be in conflict, because they compete for exclusive access to critical sections. An eating process is executing its critical section, which is assumed to be finite in duration. An exiting process merely returns to the thinking state after each critical section has completed. The duration

of a process' execution beginning from its transition to being hungry, through its transition to exiting state is referred to a *hungry-eating* session.

Variations on mutual exclusion can be defined by different specifications for safety, progress, and fairness. This paper is based on wait-free, eventually fair mutual exclusion, subject to the safety constraint of eventual weak exclusion. These terms are defined next.

• **Eventual Weak Exclusion $\diamond\mathcal{WX}$:** For every run, there exists an unknown time after which no two live neighbors eat simultaneously. $\diamond\mathcal{WX}$ allows finitely many scheduling mistakes during any run, but eventually converges to an infinite suffix during which live neighbors never eat simultaneously. Thus, $\diamond\mathcal{WX}$ can be viewed as eventual safety [Dolev 2000].

• **Wait-Freedom:** Every correct hungry process eventually eats, regardless of how many processes crash. Wait-freedom [Herlihy 1991] guarantees individual progress in the presence of crash faults. As such, wait-free exclusion algorithms never starve correct hungry processes.

• **Eventual Bounded Waiting $\diamond\mathcal{BW}$:** For every run, there exists an unknown time t and an unknown bound b such that no correct process p which becomes hungry after time t can be overtaken more than b times by any neighbor q. We say that a hungry process p is overtaken by a neighbor q each time that q gets scheduled to eat while p remains continuously hungry. Thus, $\diamond\mathcal{BW}$ is a form of eventual fairness among live hungry processes. Whereas wait-freedom *guarantees* progress with respect to crashed neighbors, however, $\diamond\mathcal{BW}$ *restricts* progress with respect to live neighbors.

3. METHODOLOGY AND DESIGN OVERVIEW

Elsewhere in the paper, we will refer to the problem of wait-free, eventually fair mutual exclusion for $\diamond\mathcal{WX}$ simply as "$\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion". As discussed previously, $\diamond\mathcal{P}$ is already known to be sufficient to solve this problem [Song and Pike 2007]. To show that $\diamond\mathcal{P}$ is also necessary (and hence the weakest oracle), we present a reduction of $\diamond\mathcal{P}$ to the problem of $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion.

The proof technique works as follows. Suppose there exists a failure detector $\mathcal{D}$ which is strictly weaker than $\diamond\mathcal{P}$ and yet can also solve $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion. Using $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion as a black box, we construct a failure detector that implements the strong completeness and eventually strong accuracy properties of $\diamond\mathcal{P}$. By hypothesis, $\mathcal{D}$ can implement $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion, but (by construction) $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion can, in turn, implement $\diamond\mathcal{P}$. By transitivity, $\mathcal{D}$ can also implement $\diamond\mathcal{P}$, thereby contradicting the assumption that $\mathcal{D}$ is strictly weaker than $\diamond\mathcal{P}$. We conclude that $\diamond\mathcal{P}$ is, in fact, the weakest oracle to solve $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion.

3.1 The Preliminary Construction

The key idea of our reduction is to convert wait-freedom and $\diamond\mathcal{BW}$ into an eventually reliable time-out mechanism for detecting crash faults. Consider two processes p and q, where p is correct and does not remain thinking forever. If q is faulty, then wait-freedom guarantees that p will eat infinitely often after q crashes. If q is correct, however, then $\diamond\mathcal{BW}$ guarantees an infinite suffix during which p cannot overtake q more than b times while q is hungry. Although b is unknown and only

holds eventually, p can adaptively estimate its value.

A simplified overview of our construction works as follows. Processes p and q participate in a single instance of $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion to implement the local failure detection module at p. In this instance, p is called the *witness* and q is called the *subject*. Both processes cooperate to maintain an adaptive lease that estimates the eventual fairness bound b.

Process p maintains two integer-valued variables: *term* (which denotes the current lease duration) and *counter* (which denotes the number of times p has eaten during the current lease). Every time q becomes hungry, q sends a *renew* message to p. Upon receiving this message, p renews the lease for q by resetting the *counter* to 0. Every time p eats, however, *counter* gets incremented by 1. If *counter* ever exceeds *term* (the duration of the lease), then the lease expires and p adds q to the suspect list.

Assume for the moment that p actually overtakes q every time that p eats; that is, p never gets scheduled to eat unless q is currently hungry.[1] Now consider two executions: one where p is correct but q is faulty, and the other where both processes are correct. In the first run, q eventually crashes. By wait-freedom, every time p becomes hungry, p eventually eats and *counter* increases. After finitely many eating sessions, *counter* exceeds *term* and the lease expires. As such, p adds q to the suspect list and establishes strong completeness for $\diamond\mathcal{P}$.

In the second run, q is correct and becomes hungry infinitely often. If $\diamond\mathcal{BW}$ has not yet converged, or if the current value of $term < b$, then p may legitimately overtake q more than *term* times. As such, the lease will expire prematurely and p will mistakenly suspect q. By wait-freedom, however, q will eventually eat and then become hungry again, whereupon q sends a fresh *renew* message to p.

Upon receiving this message, p removes q from the suspect list and renews the lease by resetting *counter*. Additionally, p increases the duration of the lease *term* in response to the false-positive mistake. Consequently, p cannot repeatedly suspect q without the value of *term* eventually exceeding the *de facto* fairness bound b. For the infinite suffix where the fairness bound holds and $term > b$, process p will never suspect q, provided that q continually renews the lease before it expires. Under such circumstances, the adaptive lease establishes the eventual strong accuracy of $\diamond\mathcal{P}$.

3.2 Additional Considerations

The foregoing analysis is an over-simplification of our actual construction. To streamline the intuitive basis for accuracy, we assumed that the witness thread at process p only gets scheduled to eat when q is actually hungry (and hence, p overtakes q every time p eats). We dispatch this assumption next.

In any run where q is correct, wait-freedom guarantees q will not remain hungry forever. Consequently, when p gets scheduled to eat, p does not necessarily overtake q (because q could be thinking, exiting, or even eating[2]). Nonetheless, the *counter* at p increases every time p eats. This precipitates a potential problem for accuracy if the lease continually expires due to false-overtaking.

[1] This assumption will be dispatched in the next subsection.

[2] Recall that $\diamond\mathcal{WX}$ may schedule p and q to eat *simultaneously* at most finitely many times, but such cases do not constitute overtaking.

To resolve this problem, process q executes two parallel threads, where each thread participates as a separate *logical process* in the same mutual exclusion instance with p. Each thread at q functions as a separate *subject*. As such, the mutual exclusion instance has three participants: the witness thread at p and two subject threads at q. We model each subject thread as a set of actions, the union of which is executed by the single physical process q. Consequently, their failure semantics are correlated; that is, both subject threads crash whenever q crashes.

The subject threads at q coordinate their transitions to establish the following suffix invariant: *for every run, there exists a time t, such that for every later time $t' > t$, some subject thread is hungry or eating at time t'.*

This suffix invariant is based on a simple ping-ack protocol between each subject at q and the witness at p. In this protocol, both subjects cooperate to continually renew a common *shared lease* with the witness. First, each subject thread has a local Boolean variable called *watched*, which is initially false. As before, each subject thread sends a *renew* message to p upon becoming hungry. This message serves as the ping. The witness at p handles the lease renewal as usual, but it also sends an *ack* message back to the original subject thread. This ack serves as confirmation that the lease has been renewed.

Upon receiving this *ack*, the subject sets its *watched* variable to true. By wait-freedom, the current subject eventually gets scheduled to eat, but it will not exit until the *watched* variables of both subjects are true.[3] Upon exiting, the subject sets its own *watched* variable to false, thereby disabling the exit guard for the sibling subject. This simple synchronization scheme guarantees that no eating subject can exit its critical section until *both* subject threads have set their *watched* variables by acknowledged lease renewals. This implies that the sibling subject must be hungry or also eating,[4] so the suffix invariant is maintained if either subject exits.

It is worth noting that the suffix invariant is actually implied by an even stronger stability property established by the ping-ack protocol along with the exit action: once either *watched* variable is set to true, at least one of them is also true at every point in the computation thereafter. The *watched* variables are set to true only after subjects becomes hungry, and are set back to false only upon exiting. But neither subject can exit unless *both* watched variables are true; thus, the hungry-eating sessions (defined in Section 2) of both subjects always overlap each other.

The ping-ack protocol establishes the eventual strong accuracy of $\diamond\mathcal{P}$ by implementing an elastic clock that eventually synchronizes to regulate the relative progress of the witness and subject threads. For each run, consider the infinite suffix β during which (1) the suffix invariant holds, (2) the fairness bound b of $\diamond\mathcal{BW}$ holds, (3) the lease duration *term* is greater than b at the witness thread, and (4)

[3] Although subjects can read (and write) both *watched* variables, we do not require auxiliary support for read/write atomicity. Recall that each subject is simply a set of actions, the union of which is executed in some non-deterministic interleaving order by process q. Since only one enabled action is executed by q at any given time, access to *watched* variables is temporally exclusive.

[4] Again, this latter case is only possible because $\diamond\mathcal{WX}$ may schedule both subjects threads to eat simultaneously finitely many times. Since one exiting subject disables the other subject from exiting, however, the *watched* variables enforce the suffix invariant by preventing both subjects from exiting simultaneously.

$\diamond\mathcal{WX}$ no longer schedules live processes to eat simultaneously. During the suffix β, the subject threads of any correct process q will always renew the shared lease before it expires. Consequently, accuracy is satisfied because p never suspects q during β.

The key idea (proved formally in Section 5) is that, during β, the interval between consecutive lease renewals is always encompassed by the hungry-eating session of some subject thread. By $\diamond\mathcal{BW}$, the witness process can eat at most b times during this interval. Since $term > b$, however, the lease is always renewed before it expires.

4. THE REDUCTION ALGORITHM

Let Π be a finite system of processes. For each pair of processes, p and q, we implement an eventually perfect failure detector $\diamond\mathcal{P}$ using two instances of the wait-free, eventually fair mutual exclusion problem for eventual weak exclusion (or simply $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion, for short). One instance, $\mathcal{M}_{pq}$, implements the local detection module at p, while the other instance, $\mathcal{M}_{qp}$, implements the local detection module at q. Since both instances are symmetric, we restrict our presentation only to the instance $\mathcal{M}_{pq}$. For clarification, this is the module that provides process p with $\diamond\mathcal{P}$ information about process q.

The instance $\mathcal{M}_{pq}$ consists of three logical processes: $p.w$, $q.s_0$, and $q.s_1$, where $p.w$ is a *witness thread* executed at process p, and $q.s_0$ and $q.s_1$ are *subject threads* executed at process q. Note that in the symmetric instance $\mathcal{M}_{qp}$, the witness versus subject roles of p and q would be reversed. See Figure 2.

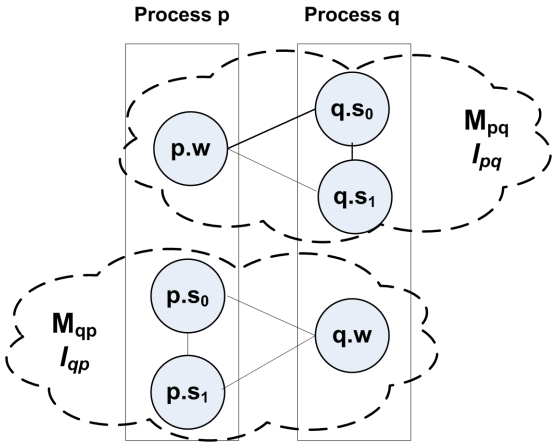


Fig. 2. An eventually perfect failure detector between processes p and q is implemented with two symmetric instances of the wait-free, eventually fair mutual exclusion problem for eventual weak exclusion.

Although logically distinct with respect to $\mathcal{M}_{pq}$, the subject threads $q.s_{i\in\{0,1\}}$ are implemented as a single stream of execution. More specifically, each subject thread is a distinct set of actions, the union of which is executed under interleaving semantics by process q. As such, the failure semantics are also correlated; thus,

if process q crashes, then both subject threads $q.s_0$ and $q.s_1$ also crash. Similar remarks apply to the witness thread $p.w$ executed by process p.

Actions for the subject threads and the witness thread are presented in Algorithms 1 and 2, respectively. These threads participate in the mutual exclusion instance $\mathcal{M}_{pq}$ of our weakest-oracle reduction for $\diamond\mathcal{P}$.

	Actions of the two subject threads $q.s_{i\in\{0,1\}}$ executed at process q	
1 :	$\{\mathsf{state}_i = \mathsf{thinking}\} \longrightarrow$	*Action $\mathcal{S}_h$*
2 :	$\mathsf{state}_i := \mathsf{hungry};$	*Becomes Hungry*
3 :	send $\langle\mathsf{renew}\rangle$ **to** witness $p.w$;	*Sends renew Message to $p.w$*
4 :	$\{$**upon** receiving $\langle\mathsf{ack}\rangle$ **from** witness $p.w\} \longrightarrow$	*Action $\mathcal{S}_a$*
5 :	$\mathsf{watched}_i := \mathsf{true};$	
6 :	$\{(\mathsf{state}_i = \mathsf{eating}) \wedge \mathsf{watched}_i \wedge \mathsf{watched}_j\} \longrightarrow$	*Action $\mathcal{S}_x$*
7 :	$\mathsf{watched}_i := \mathsf{false};$	
8 :	$\mathsf{state}_i := \mathsf{exiting};$	*Exits eating*

Algorithm 1. Actions for subjects $q.s_{i\in\{0,1\}}$ in mutual exclusion instance $\mathcal{M}_{pq}$, where $q.s_j$ denotes the other subject in $\mathcal{M}_{pq}$, and $p.w$ denotes the witness thread at process p.

	Actions for the single witness thread $p.w$ executed at process p	
1 :	$\{\mathsf{state}_w = \mathsf{thinking}\} \longrightarrow$	*Action $\mathcal{W}_h$*
2 :	$\mathsf{state}_w := \mathsf{hungry};$	*Becomes Hungry*
3 :	$\{\mathsf{state}_w = \mathsf{eating}\} \longrightarrow$	*Action $\mathcal{W}_x$*
4 :	**if** $(q \notin \mathsf{suspect}(p))$	*Exit Eating*
5 :	$\mathsf{counter}_{pq} := \mathsf{counter}_{pq} + 1;$	*Increments Counter by 1*
6 :	$\mathsf{state}_w := \mathsf{exiting};$	
7 :	$\{$**upon** receiving $\langle\mathsf{renew}\rangle$ **from** subject $q.s_i\} \longrightarrow$	*Action $\mathcal{W}_s$*
8 :	$\mathsf{counter}_{pq} := 0;$	*Renews the lease*
9 :	$\mathsf{suspect}(p) := \mathsf{suspect}(p) - \{q\};$	*removes q from* $\mathsf{suspect}(p)$
10 :	send $\langle\mathsf{ack}\rangle$ **to** subject $q.s_i;$	
11 :	$\{\mathsf{counter}_{pq} > \mathsf{term}_{pq}\} \longrightarrow$	*Action $\mathcal{W}_{sus}$*
12 :	$\mathsf{suspect}(p) := \mathsf{suspect}(p) \cup \{q\};$	*Lease expires*
13 :	$\mathsf{term}_{pq} := \mathsf{counter}_{pq};$	*Increases the term*

Algorithm 2. Actions for the witness $p.w$ in mutual exclusion instance $\mathcal{M}_{pq}$. Subjects $q.s_{i\in\{0,1\}}$ denote threads at process q.

The implementation of $\diamond\mathcal{P}$ in our reduction is based on adaptive leases. The witness at p (the lessor) maintains a shared lease with the subjects at q (the lessees). Process p only suspects q if the lease expires. In an adaptive lease, the duration (also known as the *lease term*), increases with each false-positive suspicion until it either stabilizes or surpasses some *de facto* timing bound.

In our case, each exclusion instance $\mathcal{M}_{pq}$ implements an adaptive lease $l_{pq} = (p, q, term_{pq})$ between processes p and q, where the lease duration, $term_{pq}$, adaptively estimates the unknown upper-bound on eventual fairness guaranteed by the $\diamond\mathcal{BW}$ property of $\mathcal{M}_{pq}$. As such, our leases are not based on physical time; rather, each lease duration is denominated in terms of *counted events*.

A counted event occurs whenever the witness transitions from eating to exiting. The local variable $counter_{pq}$ is incremented once for each counted event. This provides a basis for counting the number of times the witness overtakes its neighbors. The witness becomes hungry infinitely often in order to count the number of times it can overtake the subject neighbors. Whenever $counter_{pq}$ exceeds $term_{pq}$, process p adds q to the suspect list.

The subjects, meanwhile, continually attempt to extend the lease by sending a *renew* message upon becoming hungry. If p currently suspects q, the receipt of a *renew* message indicates that the suspicion was false. In the event of such a false suspicion, the witness takes the subjects' process off the suspect list, increases the lease duration, and resets the local counter.

4.1 Local Variables

Each instance $\mathcal{M}_{pq}$ in the reduction uses six types of local variables. These are partitioned into two sets: witness variables at process p and subject variables at process q.

Witnesses at Process p have four kinds of local variables: state, term, $\mathsf{counter}$, and $\mathsf{suspect}$.

Each witness $p.w$ has one local variable state_w to express its current state. Variable state_w could be *thinking, hungry, eating* or *exiting*. Initially variable state_w is *thinking*.

As mentioned before, process p has one integer variable $\mathsf{counter}_{pq}$ for process q. Initially, $\mathsf{counter}_{pq}$ is 0. The variable $\mathsf{counter}_{pq}$ counts the number of exiting events of the witness $p.w$ during the current duration of lease l_{pq}. Every time the lease l_{pq} gets renewed, $\mathsf{counter}_{pq}$ is reset to 0.

Every lease has a duration. Hence p has an integer variable term_{pq} for lease l_{pq}. Initially term_{pq} is one. Variable term_{pq} is monotonically non-decreasing. Every time lease l_{pq} expires, witness $p.w$ increases term_{pq}.

Every process p has a set variable $\mathsf{suspect}(p)$, which denotes the set of processes that are currently suspected of being crashed by process p. Initially variable $\mathsf{suspect}(p)$ could include any process except p itself. Note that every process has only one set variable $\mathsf{suspect}(p)$, which could be updated by every witness at p.

Subjects at Process q have two kinds of local variables: state and $\mathsf{watched}$. Instance $\mathcal{M}_{qp}$ includes two subjects $q.s_0$ and $q.s_1$ at q. Each subject $q.s_i$, where $i \in \{0, 1\}$, has two local variables: state_i and $\mathsf{watched}_i$.

Local variable state_i is used to express the current state of subject $q.s_i$. The variable state_i could be *thinking, hungry, eating* or *exiting*. Initially subject $q.s_i$ is

thinking.

As described before, each subject $q.s_i$ has a boolean local variable $\mathsf{watched}_i$, initially set to false. The variable $\mathsf{watched}_i$ is used to implement the *ping-ack* protocol described in Section 3.2. Variable $\mathsf{watched}_i$ is set to true only when subject $q.s_i$ receives an *ack* message, which is sent by witness $p.w$ as a confirmation of the lease renewal requested by $q.s_i$. Meanwhile, variable $\mathsf{watched}_i$ will remain true until subject $q.s_i$ exits eating. Thus, while variable $\mathsf{watched}_i$ is true, the corresponding subject $q.s_i$ is in its hungry-eating (defined in Section 2) session; but not vice versa. While subject $q.s_i$ is thinking or exiting, variable $\mathsf{watched}_i$ is false; but not vice versa. Before a subject $q.s_i$ exits eating, watched variables of the two subjects in $\mathcal{M}_{qp}$ must be true simultaneously. This ensures that every lease renewal interval is encompassed by some hungry-eating session of some subject.

4.2 Actions

Subject Actions. As shown in Algorithm 1, a correct thinking subject eventually becomes hungry by executing Action $\mathcal{S}_h$. Immediately after a subject $q.s_i$ becomes hungry, where $i \in \{0, 1\}$, $q.s_i$ sends a *renew* message to request renewing lease l_{pq}. Subjects receive *ack* messages in Action $\mathcal{S}_a$. As a result, subject $q.s_i$ sets its $\mathsf{watched}_i$ variable to true, and $q.s_i$ is being *watched* by the corresponding witness $p.w$.

Action $\mathcal{S}_x$ coordinates the hungry-eating sessions of the two subjects in the instance $\mathcal{M}_{pq}$. When Action $\mathcal{S}_x$ is enabled at an eating subject $q.s_i$, both subjects in $\mathcal{M}_{pq}$ must be *watched* (watched variables are true). In Action $\mathcal{S}_x$, subject $q.s_i$ sets its own $\mathsf{watched}_i$ variable to false before $q.s_i$ exits eating. Therefore, after $q.s_i$ exits eating, the exiting action is disabled at the other subject $q.s_j$, and $q.s_j$ must stay in its current hungry-eating session until variable $\mathsf{watched}_i$ becomes true again. Thus, we guarantee that every lease renew interval is encompassed by some hungry-eating session of some subject.

Witness Actions. As shown in Algorithm 2, a correct thinking witness eventually becomes hungry by executing Action $\mathcal{W}_h$. Eating witnesses eventually exit eating by executing Action $\mathcal{W}_x$.

Executions of Action $\mathcal{W}_x$ (exiting events of witness $p.w$) serve as an abstract time measurement mechanism in our lease-based algorithm. Every time witness $p.w$ exits eating, if p does not suspect q, then variable $\mathsf{counter}_{pq}$ increases by one.

Action $\mathcal{W}_s$ is enabled and executed when witness $p.w$ receives a *renew* message. By executing Action $\mathcal{W}_s$, $p.w$ renews lease l_{pq}. As a result, variable $\mathsf{counter}_{\mathsf{pq}}$ is reset to zero, and q is removed from the suspect list $\mathsf{suspect}(p)$. Also in Action $\mathcal{W}_s$, $p.w$ sends an *ack* message to the subject that sent the *renew* message.

Action $\mathcal{W}_{sus}$ is enabled when the lease l_{pq} expires. When l_{pq} expires, p suspects q and puts q into the suspect list $\mathsf{suspect}(p)$. The term of the lease increases to the current value of variable $\mathsf{counter}_{\mathsf{pq}}$.

5. PROOF OF CORRECTNESS

In this section, we prove correctness by showing that reduction satisfies the *Strong Completeness* and *Eventual Strong Accuracy* properties of $\diamond\mathcal{P}$.

5.1 Strong Completeness

Theorem 1: *Every crashed process is eventually and permanently suspected by all correct processes.*

Proof Concept. For each pair of processes p and q, consider any fault pattern in which p is correct and q is faulty. To prove strong completeness, we need to show that if process q crashes, p suspects q eventually and permanently. In our proof, we only consider the instance $\mathcal{M}_{pq}$ and the lease l_{pq}, where process p monitors process q. The instance $\mathcal{M}_{pq}$ consists of three threads: witness $p.w$, subjects $q.s_0$ and $q.s_1$.

Theorem 1 is proved by direct construction. Process q is faulty and eventually crashes, therefore there exists a time t_{rs} after which p stops receiving *renew* messages from q. Consequently, after time t_{rs}, Action $\mathcal{W}_s$ is disabled at witness $p.w$, and lease l_{pq} is never renewed. As a result, lease l_{pq} eventually expires, and p eventually and permanently suspects q.

Proof: For any execution α in which process q is faulty, after q crashes, subjects $q.s_0$ and $q.s_1$ do not send *renew* messages. Consequently, in execution α, process q only sends a finite number of *renew* messages to p, and p receives a finite number of *renew* messages from q. Let time t_{rs} be the earliest time after which p stops receiving *renew* messages from q.

Action $\mathcal{W}_s$ is permanently disabled at $p.w$ after time t_{rs}, because witness $p.w$ stops receiving *renew* messages. Therefore, the lease l_{pq} is never renewed after time t_{rs}. As a result, after time t_{rs}, variable $\mathsf{counter}_{pq}$ never decreases. If p suspects q after time t_{rs}, p suspects process q permanently.

By Action $\mathcal{W}_h$, correct thinking witnesses eventually become hungry. Wait-freedom guarantees that correct hungry witnesses eventually eat. Correct eating witnesses eventually exit eating by executing Action $\mathcal{W}_x$. Thus, witness $p.w$ becomes hungry infinitely often, and exits eating infinitely often. Every time $p.w$ exits eating, if p is not suspecting q, $\mathsf{counter}_{pq}$ increases by one. Consequently, $\mathsf{counter}_{pq}$ eventually exceeds term_{pq}; lease l_{pq} expires, and p suspects q (Action $\mathcal{W}_{sus}$). We have already shown that after time t_{rs}, if q is suspected by p, q is suspected permanently. Thus, Theorem 1 holds. □

5.2 Eventual Strong Accuracy

Theorem 2: *For each execution, there exists a time after which no correct process is suspected by any correct process.*

For each pair of processes p and q, consider any execution α in which p and q are correct. To prove eventual strong accuracy, we need to show that for execution α, there exists a time after which p does not suspect q. In this subsection, we only consider the instance $\mathcal{M}_{pq}$ and the lease l_{pq}, in which process p monitors process q. By generality, the proof holds for $\mathcal{M}_{qp}$ as well.

Theorem 2 is proved in two steps. First, we prove Lemma 1 which states that every lease renewal interval is encompassed by some hungry-eating session of some subject in $\mathcal{M}_{pq}$. Next, Theorem 2 is formally proved using Lemma 1.

We now introduce the notations used in the proofs that follow. An *event* denotes an execution of some action. In an execution α, event $\mathcal{A}^n$ denotes the n^{th} execution of Action $\mathcal{A}$, where $n \in \mathbb{N}$. For example: event $\mathcal{W}_s^n$ denotes the n^{th} execution of Action $\mathcal{W}_s$ by witness $p.w$, and event $\mathcal{S}_{x,i}^n$ denotes the n^{th} execution of Action $\mathcal{S}_x$

by subject $q.s_i$, where $i \in \{0, 1\}$. Notation $\alpha[\phi_1, \phi_2]$ denotes the execution segment of α, which starts from event ϕ_1 and ends at event ϕ_2 (including ϕ_1 and ϕ_2). Therefore, $\alpha[\mathcal{W}_s^n, \mathcal{W}_s^{n+1}]$ denotes the n^{th} lease renewal interval in execution α.

Lemma 1: *Consider each pair of processes p and q, and every execution α in which p and q are correct. For lease $l_{pq} = (p, q, term_{pq})$, every lease renewal interval $\alpha[\mathcal{W}_s^n, \mathcal{W}_s^{n+1}]$ is encompassed by some hungry-eating session of the subject which does not request the $n + 1^{st}$ lease renewal.*

Generally, we say that action $\mathcal{W}_s$ is enabled when it receives a *renew* message from a subject. This implies that the action $\mathcal{W}_s$ is enabled for the n^{th} time when it receives the n^{th} *renew* message from a subject. But to simplify the presentation, we say that event $\mathcal{W}_s^n$ is enabled by a subject. Let t_n be the time when the n^{th} lease renewal (event $\mathcal{W}_s^n$) is executed by witness $p.w$. For example: events $\mathcal{W}_s^1$ and $\mathcal{W}_s^2$ are executed at times t_1 and t_2, respectively. Note that $t_n < t_{n+1}$.

Note that a subject is allowed to become hungry more than once while the other subject in instance $\mathcal{M}_{pq}$ stays in its hungry session. Therefore, the consecutive lease renewals could be requested by the same subject.

Proof: Lemma 1 is proved by induction on the sequence number of lease renewal intervals.

Base Case. We will show that the first lease renewal interval $\alpha[\mathcal{W}_s^1, \mathcal{W}_s^2]$ is encompassed by the first hungry-eating session of the subject that requests the first lease renewal (event $\mathcal{W}_s^1$), and this subject does not request the second lease renewal (event $\mathcal{W}_s^2$). The base case is shown in figure 3a.

event $\mathcal{W}_s^n$ is enabled by a *renew* message sent by a subject $q.s_i$, where $i \in \{0, 1\}$. When subject $q.s_i$ sends the *renew* message, $q.s_i$ is hungry (Action $\mathcal{S}_h$). Also, $q.s_i$ cannot exit eating until it receives the *ack* message sent in event $\mathcal{W}_s^n$. Thus, if a subject enables event $\mathcal{W}_s^n$ at time t_n, the subject must be hungry or eating at time t_n.

Without loss of generality, assume that event $\mathcal{W}_s^1$ is enabled by subject $q.s_0$. Therefore, $q.s_0$ is hungry or eating at time t_1. Next, we will show that $q.s_0$ stays in its hungry-eating session through time t_2.

By Action $\mathcal{S}_x$, subject $q.s_0$ cannot exit eating until variable $\mathsf{watched}_1$ of subject $q.s_1$ is set to true. Initially, $\mathsf{watched}_1$ is false. Also $\mathsf{watched}_1$ can be set to true only after $q.s_1$ enable Action $\mathcal{W}_x$ and receives an *ack* message from $p.w$. Thus, event $\mathcal{W}_s^2$ must be enabled by $q.s_1$, and subject $q.s_0$ cannot exit its eating session up through time t_2.

Since subject $q.s_0$ cannot exit its eating session up through time t_2, the first lease renewal interval is encompassed by the first hungry-eating session of $q.s_0$, and $q.s_0$ does not enable event $\mathcal{W}_s^2$. Lemma 1 holds for the base case.

Inductive Hypothesis. If Lemma 1 holds for the n-1^{st} lease renewal interval $\alpha[\mathcal{W}_s^{n-1}, \mathcal{W}_s^n]$, then Lemma 1 also holds for the n^{th} lease renewal interval $\alpha[\mathcal{W}_s^n, \mathcal{W}_s^{n+1}]$. The inductive step is shown in figure 3b.

Suppose that subject $q.s_i$, where $i \in \{0, 1\}$, does not enable event $\mathcal{W}_s^n$. By inductive hypothesis, the n-1^{st} lease renewal interval is encompassed by some hungry-eating session of $q.s_i$. Thus, $q.s_i$ must be hungry or eating at time t_n when event $\mathcal{W}_s^n$ occurs. Also in the base case, we have shown that the subject which enables event $\mathcal{W}_s^n$ must be hungry or eating at time t_n. Therefore, both subjects in $\mathcal{M}_{pq}$

are hungry or eating at time t_n.

Since both subjects are either hungry or eating at time t_n, at least one subject must exit eating in the n^{th} lease renewal interval $\alpha\left[\mathcal{W}_s^n, \mathcal{W}_s^{n+1}\right]$ to enable $\mathcal{W}_s^{n+1}$. Suppose that time t_{sx} is the earliest time when a subject $q.s_0$ exits its eating session after time t_n, then at time t_{sx}, variable $\mathsf{watched}_0$ is set to false (by Action $\mathcal{W}_x$). Hence, Action $\mathcal{W}_x$ is disabled at the other subject $q.s_1$ from time t_{sx} until variable $\mathsf{watched}_0$ becomes true again. Variable $\mathsf{watched}_0$ cannot become true until $q.s_0$ receives an *ack* message, which is sent in an event of $\mathcal{W}_s$ enabled by $q.s_0$. Thus, event $\mathcal{W}_s^{n+1}$ must be enabled by subject $q.s_0$. Subject $q.s_1$ stays in its hungry-eating session from time t_{sx} until time t_{n+1}.

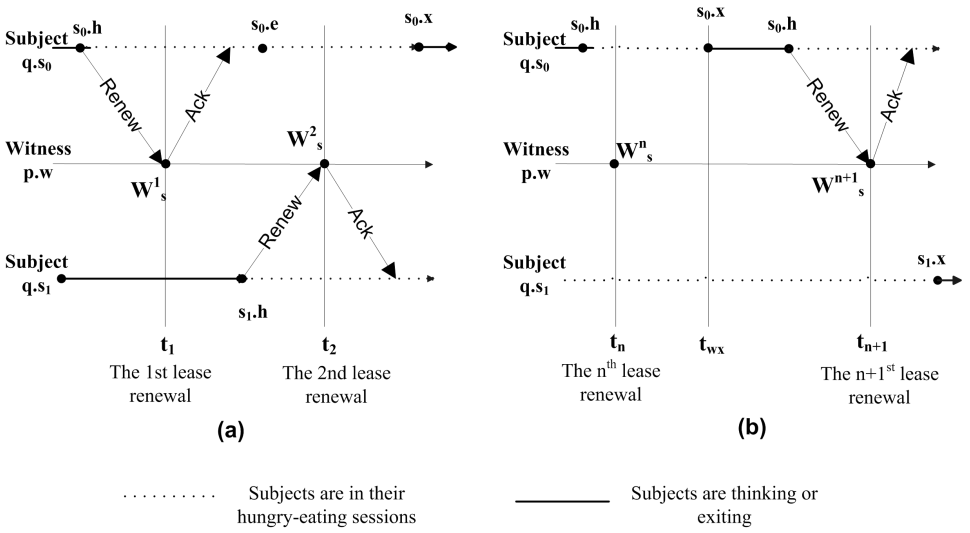


Fig. 3. The n^{th} lease renewal interval is encompassed by some hungry-eating session of the subject which does not request $(n+1)^{st}$ lease renewal, where $s_i.h$, $s_i.e$, $s_i.x$, $s_i.t$ denote that the subject $q.s_i$ transits to hungry, eating, exiting thinking states respectively.

By the assumption that time t_{sx} is the earliest time when a subject exits eating after time t_n, subject $q.s_1$ cannot exit eating from time t_n through t_{sx}. Action $\mathcal{W}_x$ is disabled at subject $q.s_1$ from time t_{sx} up through t_{n+1}. Thus, subject $q.s_1$ cannot execute exiting action (Action $\mathcal{W}_x$) in the n^{th} lease renewal interval. As shown previously, both subjects must be hungry or eating at time t_n. Thus, the n^{th} lease renewal interval is encompassed by some hungry-eating session of subject $q.s_1$, and $q.s_1$ does not request the $n+1^{st}$ lease renewal. Thus, Lemma 1 holds for the inductive step.

From the base case and the inductive step, we conclude that Lemma 1 holds. □

Theorem 2: *For each execution, there exists a time after which no correct process is suspected by any correct process.*

Proof Concept. Theorem 2 is proved in two steps. In the first step, we show that for each execution α, there exists a natural number k, and a time t_1 after which

witness $p.w$ can execute the exiting action at most $k+1$ times in each lease renewal interval. Consequently, the variable $\mathsf{counter}_{pq}$ is eventually bounded by $k+1$ in execution α. In the second step, we show that for execution α, there exists a time t_2 after which the variable $\mathsf{counter}_{pq}$ never exceeds the variable term_{pq}. Therefore, after time t_2, lease l_{pq} never expires, and p never suspects q.

Proof: For each execution α of instance $\mathcal{M}_{pq}$, there exists a natural number k such that after the $\diamond\mathcal{WX}$ algorithm converges at some time t_1, no thread can go to eat more than k times while any other thread is hungry. Thus, after time t_1, witness $p.w$ goes to eat at most k times while any subject ($q.s_0$ or $q.s_1$) is hungry. On the other hand, after the $\diamond\mathcal{WX}$ algorithm converges, witness $p.w$ cannot eat while subjects are eating. Thus, after time t_1, witness $p.w$ goes to eat at most k times while any subject is in its hungry-eating session. Note that when a subject becomes hungry, the witness may already be in its eating session. Thus, after time t_1, a hungry-eating session of a subject may encompass $k+1$ (partial) eating sessions of witness $p.w$.

By Lemma 1, every lease renewal interval is encompassed by some hungry-eating session of some subject. Thus, after time t_1, each lease renewal interval contains at most $k+1$ (partial) eating sessions of witness $p.w$. Consequently, after time t_1, witness $p.w$ can execute exiting action (Action $\mathcal{W}_x$) at most $k+1$ times in each lease renewal interval. Therefore, after time t_1, variable $\mathsf{counter}_{pq}$ is bounded by $k+1$ in execution α.

Since time t_1 is finite, there must exist a maximum value c_m for $\mathsf{counter}_{pq}$ before time t_1. Therefore, in execution α, there exists an upper-bound $k' = max(c_m, k+1)$ for the variable $\mathsf{counter}_{pq}$.

Every time $\mathsf{counter}_{pq}$ exceeds term_{pq}, term_{pq} is increased to the value of $\mathsf{counter}_{pq}$ (Action $\mathcal{W}_{sus}$). Thus, in execution α, term_{pq} is also bounded by k'. Otherwise, $\mathsf{counter}_{pq}$ cannot be bounded by k'. Suppose the variable term_{pq} reaches to the bound k' at time t_2. Hence, after time t_2, lease l_{pq} never expires, and p never puts q into the suspect list $\mathsf{suspect}(p)$. Time t_2 may or may not be later than t_1.

In our algorithm, p may wrongfully suspect q during some finite prefix of execution α. However, p eventually stops suspecting q. Given that communication channels are reliable, and p and q are correct, no subject can be blocked in eating sessions forever. Thus, subjects $q.s_0$ and $q.s_1$ become hungry infinitely often and send an infinite number of *renew* messages (Action $\mathcal{S}_h$). If p suspects q, $p.w$ eventually receives a *renew* message, and p stops suspecting q.

Therefore, any wrongfully suspected process is eventually removed from the suspect list. After time t_2, p never suspects q. Thus, Theorem 2 holds. □

6. DISCUSSION

6.1 Properties of the algorithm

Finite Space Complexity. Every process p has six types of local variables, among which only variables term and counter are only ones unbounded. Although variables term and counter are bounded by some value in each execution of every exclusion instance, there does not exists an upper-bound for all executions of all instances. Thus, variables term and counter need finite but unbounded space. Other variables at process p are bounded. Therefore, our algorithm needs finite space.

Bounded Capacity of Channels. At any time, the number of messages in transit between each pair of processes is bounded. Between a subject and a witness, at most one message can be in transit at any time. The message in transit is either a *renew*, or an *ack* message. There are two exclusion instances for each pair of processes, and each instance includes two subjects and one witness. Therefore, at most four messages can be in transit at any given time between each pair of processes.

The size of messages is bounded. The process id information needs to be encoded into the message and requires $log_2(|\Pi|)$ bits. Also the message type needs to be specified in all messages. The message type requires a constant number of bits. Thus, every message needs $log_2(|\Pi|) + c$ bits, where c is a constant.

Crash-Quiescence. In our algorithm, correct processes eventually stop communicating with crashed processes. This property is called crash-quiescence. It follows from the following observation.

For each pair of processes p and q, consider any execution in which p is correct and q is faulty. Let q crash at time t_c. We now examine the messages send by the witness and subject threads in process p after q crashes.

Given that q crashes at time t_c, there exists a time t_r after which p never receives messages from q (note that t_r may be earlier than t_c). Since witness $p.w$ does not receive a renew message from subjects $q.s_0$ and $q.s_1$ (when q crashes, so do the threads $q.s_0$ and $q.s_1$), $p.w$ does not send an *ack* message to q. Therefore, witness $p.w$ stops sending *ack* messages to q after time t_r.

Subjects in p ($p.s_0$, $p.s_1$) send *renew* messages. However, any renew message sent after time t_c are not followed by an *ack* from $q.w$ because process q (and hence thread $q.w$) is crashed. Consequently, subjects $p.s_0$ and $p.s_1$ cannot exit their eating session, and stay in the eating session for the suffix of the execution. Since subjects $p.s_0$ and $p.s_1$ cannot become hungry again, they do not send *renew* messages to $q.w$. Therefore, subject threads $p.s_0$ and $p.s_1$ stop sending *renew* messages to q after time t_r.

From the above arguments, it follows that correct process p eventually stops communicating with every crashed process q. In other words, the algorithm in crash-quiescent.

6.2 Fairness Properties and Partial Synchrony

In this section, we discuss the relationship between fairness properties and different models of partial synchrony. We conjecture that the fairness properties can encapsulate the timing assumptions about partial asynchrony in distributed systems. We base this conjecture on the comparison of three models of partial synchrony $\mathcal{M}_1$, $\mathcal{M}_3$, and $\mathcal{M}_3$ (described in [Chandra and Toueg 1996]) with three fairness properties $\Box\mathcal{BW}$, $\Diamond k$-$\mathcal{BW}$, and $\Diamond\mathcal{BW}$. Each of the foregoing partially synchronous models and fairness properties are described in subsequent paragraphs.

As mentioned earlier, Chandra and Toueg [Chandra and Toueg 1996] described three partially synchronous models:

• **Model $\mathcal{M}_1$:** proposed by Dwork et al. [1988], defines that for each execution, there exists some *unknown* bounds on message delay and relative process speed.

Essentially, these bounds are *unknown and perpetual* with respect to individual executions.

• **Model $\mathcal{M}_2$:** also proposed by Dwork et al. [1988], defines that bounds on message delay and relative process speed are *known*, but hold only after a finite unknown period of a time. In other words, the bounds are *known but eventual* in model $\mathcal{M}_2$.

• **Model $\mathcal{M}_3$:** proposed by Chandra and Toueg [1996], defines that the bounds on message delay and relative process speedare *unknown* and hold only after a finite unknown period of time. Therefore, the bounds are *unknown and eventual* in model $\mathcal{M}_3$.

The fairness properties we consider in this section are:

• $\Box\mathcal{BW}$**:** *Perpetual* bounded waiting with *unknown* bounds. *Perpetual bounded waiting* stipulates that for each execution, there exists an unknown bound b such that no live user goes to eat more than b times while any other user is hungry.

• $\Diamond k$**-**$\mathcal{BW}$**:** *Eventual* bounded waiting with *known* bounds. $\Diamond k$-$\mathcal{BW}$ stipulates that there exists an unbound time t after which each hungry process can be overtaken by its hungry neighbor k times, for a known k. The time t may vary from run to run, but for each run there exists such a bound.

• $\Diamond\mathcal{BW}$**:** *Eventual* bounded waiting with *unknown* bounds. $\Diamond\mathcal{BW}$ stipulates that there exists an unbound time t after which each hungry process can be overtaken by its hungry neighbor k times, for some unknown k. The time t, and bound k vary from run to run, but for each run there exist such bounds.

We now present plausible mutual reductions of partially synchronous models to fairness properties and *vice-versa*.

6.2.1 $\mathcal{M}_1$ *and* $\Box\mathcal{BW}$.

The partially synchronous model $\mathcal{M}_1$ can implement mutual exclusion satisfying $\Box\mathcal{BW}$ using the hygienic solution proposed by Choy and Singh [1995]. Insofar as there is an unknown upper-bound on the round-trip time of a message (as a consequence of unknown upper-bounds on relative process speed and message delay), there is a bound on the number of times a process holding a fork can enter its critical section after its neighbor becomes hungry, and before the fork request (from that neighbor) is received. This serves as an upper-bound on overtaking, and hence satisfies $\Box\mathcal{BW}$.

Given a solution of mutual exclusion with $\Box\mathcal{BW}$, we can simulate $\mathcal{M}_1$ as follows: Consider any action system $\mathcal{A}$ executing in an asynchronous environment augmented with a solution to mutual exclusion satisfying $\Box\mathcal{BW}$. Augment the action system $\mathcal{A}$ such that each action in the $\mathcal{A}$ is an independent critical section; thus, only one process can execute its enabled action at any given time, and after executing an action, the process goes back to being hungry to execute the next enabled action. The $\Box\mathcal{BW}$ ensures that there exists a bound on the number of actions every process p's neighbors may execute before p can execute its enabled action. Let such bound be k. This implies that no process in the system may execute more than k actions without process p executing its enabled action. This effectively imposes an upper-bound on the relative speed of the processes in the system.

In order to simulate an upper-bound on message delay, we implement a local

clock on each process whose ticks are the actions executed by the process itself. In other words, the local clock increments by one every time the process executes a single action. Also, we piggyback application messages to the messages exchanged by the $\Box\mathcal{BW}$ implementation in the system so that application messages arrive with the $\Box\mathcal{BW}$ control messages. Thus, although there may not be an upper-bound on absolute message delay with respect to physical time, there is an absolute bound on the message delay with respect to the process' local clocks. The bound on message delay in unknown, but it exists, and may vary from run to run.

Through the above constructions, we illustrate a plausible mutual reduction of the model $\mathcal{M}_1$ to $\Box\mathcal{BW}$, and *vice-versa*.

6.2.2 $\mathcal{M}_2$ *and* $\Diamond k$-$\mathcal{BW}$.

The partially synchronous model $\mathcal{M}_2$ can implement mutual exclusion with $\Diamond k$-$\mathcal{BW}$ with $\Box\mathcal{BW}$ using the hygienic solution proposed by Choy and Singh [1995]. During the prefix of the execution where $\mathcal{M}_2$ has not converged (*i.e.*, the known bounds on relative process speed and message delay do not hold), there is no upper-bound on message round trip time, therefore processes can enter their critical section unbounded number of time before they receive a fork request from their hungry neighbors. However, after $\mathcal{M}_2$ has converged (the known bounds on relative process speed and message delay hold), there is a known upper-bound on the round-trip time of a message (as a consequence of known upper-bounds on relative process speed and message delay), and hence a known upper-bound on the number of times a process holding a fork can enter its critical section after its neighbor becomes hungry, and before the fork request (from that neighbor) is received. Let this bound be k. This bound k serves as a known upper-bound on overtaking, and hence satisfies $\Diamond k$-$\mathcal{BW}$.

The construction to simulate $\mathcal{M}_2$ using a solution of mutual exclusion with $\Diamond k$-$\mathcal{BW}$ is similar to the construction described in Section 6.2.1. The only difference is that the environment is an asynchronous message passing system augmented with a solution to mutual exclusion satisfying $\Diamond k$-$\mathcal{BW}$, for a known k.

Following a line of argument similar to the one in Section 6.2.1: There exists no upper-bounds on overtaking during the prefix before the $\Diamond k$-$\mathcal{BW}$ module converges, hence any process may execute its actions an unbounded number of time before its neighbors execute any of their actions. This translates to no bounds on relative process speed and no bounds on message delay before the $\Diamond k$-$\mathcal{BW}$ module converges. However, after the $\Diamond k$-$\mathcal{BW}$ module converges, the known bound k on overtaking is in effect, and this translates to known bounds on on relative process speed and message delay. In summary, until such time as the $\Diamond k$-$\mathcal{BW}$ module converges, there are no bounds on relative process speed and message delay, but after the $\Diamond k$-$\mathcal{BW}$ module converges, known bounds on relative process speed and message delay hold. This satisfies the specification for $\mathcal{M}_2$.

We have thus illustrated a plausible mutual reduction of the model $\mathcal{M}_2$ to $\Diamond k$-$\mathcal{BW}$, and *vice-versa*.

6.2.3 $\mathcal{M}_3$ *and* $\Diamond\mathcal{BW}$.

The partially synchronous model $\mathcal{M}_3$ can implement mutual exclusion with $\Diamond\mathcal{BW}$ using the hygienic solution proposed by Choy and Singh [1995]. During the prefix

of the execution where $\mathcal{M}_3$ has not converged (*i.e.*, the unknown bounds on relative process speed and message delay do not hold), there is no upper-bound on message round trip time, therefore processes can enter their critical section unbounded number of time before they receive a fork request from their hungry neighbors. However, after $\mathcal{M}_3$ has converged (the unknown bounds on relative process speed and message delay hold), there is an unknown upper-bound on the round-trip time of a message (as a consequence of unknown upper-bounds on relative process speed and message delay), and hence an unknown upper-bound on the number of times a process holding a fork can enter its critical section after its neighbor becomes hungry, and before the fork request (from that neighbor) is received. This bound serves the upper-bound on overtaking, satisfiying $\diamond\mathcal{BW}$.

The construction to simulate $\mathcal{M}_3$ using a solution of mutual exclusion with $\diamond\mathcal{BW}$ is similar to the construction described in Section 6.2.1. The only difference is that the environment is an asynchronous message passing system augmented with a solution to mutual exclusion satisfying $\diamond\mathcal{BW}$.

Following a line of argument similar to the one in Section 6.2.1: There exists no upper-bounds on overtaking during the prefix before the $\diamond\mathcal{BW}$ module converges, hence any process may execute its actions an unbounded number of time before its neighbors execute any of their actions. This translates to no bounds on relative process speed and no bounds on message delay before the $\diamond\mathcal{BW}$ module converges. However, after the $\diamond\mathcal{BW}$ module converges, an unknown bound on overtaking is in effect, and this translates to unknown bounds on on relative process speed and message delay. In summary, until such time as the $\diamond\mathcal{BW}$ module converges, there are no bounds on relative process speed and message delay, but after the $\diamond\mathcal{BW}$ module converges, unknown bounds on relative process speed and message delay hold. This satisfies the specification for $\mathcal{M}_3$.

We have thus illustrated a plausible mutual reduction of the model $\mathcal{M}_3$ to $\diamond\mathcal{BW}$, and *vice-versa*.

From the above three examples we conjecture that fairness properties can encapsulate the timing properties about partial synchrony in distributed systems

7. RELATED WORK

Several decades of research on mutual exclusion are covered in [Raynal 1986; Anderson et al. 2003]. Many bounded-waiting algorithms have been developed for shared memory [Lamport 1974; Eisenberg and McGuire 1972; Burns et al. 1982] and message passing systems [Lamport 1978; Ricart and Agrawala 1981]. All of these satisfy perpetual bounded-waiting, where the bounds hold for all executions; none are wait-free.

Wait-free algorithms guarantee that every correct process finishes any operation within finitely many steps, regardless of how many processes crash. Wait-freedom was originally explored in the context of shared memory primitives for consensus [Herlihy 1991]. The concept of wait-freedom, however, can be extended to all agreement and coordination problems in distributed computing. Wait-freedom is not achievable in pure asynchrony. In fact, it is unsolvable even in some models of partial synchrony; this result was demonstrated in [Pike and Sivilotti 2004] by showing that wait-free mutual exclusion under perpetual weak exclusion is un-

solvable in asynchronous message passing systems augmented with $\diamond\mathcal{P}$. There are, however, stronger models of partial synchrony in which wait-freedom can be achieved. For instance, partially synchronous systems strong enough to implement the trusting failure detector T [Delporte-Gallet et al. 2005] can achieve wait-free mutual exclusion.

In this paper, we achieve wait-freedom under eventual weak exclusion using the eventually perfect failure detector $\diamond\mathcal{P}$. The oracle $\diamond\mathcal{P}$ is, in fact, the weakest failure detector to solve wait-free eventually fair mutual exclusion under eventual weak exclusion. The oracle $\diamond\mathcal{P}$ is also the weakest to solve two other problems. First, $\diamond\mathcal{P}$ is the weakest oracle with *bounded outputs* to solve quiescent reliable communication [Aguilera et al. 2000]. Interestingly, this problem can be solved in asynchronous systems using an oracle called Heartbeat with unbounded outputs [Aguilera et al. 1997]. Second, $\diamond\mathcal{P}$ is the weakest oracle to solve wait-free contention management in shared-memory systems [Guerraoui et al. 2006]. Wait-free contention managers ensure that every process has enough time to complete its operation, even in the presence of high contention and crash faults.

8. CONCLUSIONS

We established the necessary conditions for wait-free, eventually fair mutual exclusion in message-passing systems subject to crash faults. In conjunction with our previous work, we showed that $\diamond\mathcal{P}$ is the weakest failure detector to solve $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion. Specifically, we presented an adaptive lease-based implementation of $\diamond\mathcal{P}$ using an underlying, black-box solution to $\mathcal{WF}$-$\mathcal{EF}$ mutual exclusion. Our approach used an elastic clock to measure progress indirectly in terms of eventual fairness properties. Additionally, we explored the plausibility of characterizing fairness properties as an alternative abstraction for unifying timing assumptions about partial synchrony in distributed systems.

REFERENCES

Aguilera, M. K., Chen, W., and Toueg, S. 1997. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th International Workshop on Distributed Algorithms*. Springer-Verlag, 126–140.

Aguilera, M. K., Chen, W., and Toueg, S. 2000. On quiescent reliable communication. *SIAM Journal on Computing 29,* 6, 2040–2073.

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. 2001. Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing*. 108–122.

Anderson, J. H., Kim, Y.-J., and Herman, T. 2003. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing 16,* 2-3 (Sep), 75–110.

Burns, J. E., Jackson, P., Lynch, N. A., Fischer, M. J., and Peterson, G. L. 1982. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM 29,* 1 (Jan), 183–205.

Chandra, T. D. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43,* 2 (Mar), 225–267.

Choy, M. and Singh, A. K. 1995. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems 17,* 3 (May), 535–559.

Cristian, F. 1991. Understanding fault-tolerant distributed systems. *Communications of the ACM 34,* 2 (Feb), 56–78.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., and Kouznetsov, P. 2005. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing 65,* 4 (Apr), 492–505.

Dijkstra, E. W. 1965. Solution of a problem in concurrent programming control. *Communnications of the ACM 8,* 9, 569.

Dolev, S. 2000. *Self-Stabilization*. MIT Press.

Dwork, C., Lynch, N. A., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35,* 2 (Apr), 288–323.

Eisenberg, M. A. and McGuire, M. R. 1972. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM 15,* 11, 999.

Fich, F. and Ruppert, E. 2003. Hundreds of impossibility results for distributed computing. *Distributed Computing 16,* 2-3, 121–163.

Fromentin, E., Raynal, M., and Tronel, F. 1999. On classes of problems in asynchronous distributed systems with process crashes. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*. 470–477.

Guerraoui, R., Kapalka, M., and Kouznetsov, P. 2006. The weakest failure detectors to boost obstruction-freedom. In *Proceedings of the 20th International Symposium on Distributed Computing*. 399–412.

Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13,* 1 (Jan), 124–149.

Lamport, L. 1974. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17,* 8 (Aug), 453–455.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21,* 7 (Jul), 558–565.

Pike, S. M. and Sivilotti, P. A. G. 2004. Dining philosophers with crash locality 1. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*. 22–29.

Raynal, M. 1986. *Algorithms for Mutual Exclusion*. MIT Press.

Ricart, G. and Agrawala, A. K. 1981. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM 24,* 1 (Jan), 9–17.

Song, Y. and Pike, S. M. 2007. Eventually k-bounded wait-free distributed daemons. Tech. Rep. TAMU-CS-TR-2007-2-1, Texas A&M University. Feb.