The Weakest Failure Detector for Wait-Free Dining under Eventual Weak Exclusion*

Srikanth Sastry
Computer Science and Engr
Texas A&M University
College Station, TX, USA
sastry@cse.tamu.edu

Scott M. Pike
Computer Science and Engr
Texas A&M University
College Station, TX, USA
pike@cse.tamu.edu

Jennifer L. Welch
Computer Science and Engr
Texas A&M University
College Station, TX, USA
welch@cse.tamu.edu

ABSTRACT

Dining philosophers is a classic scheduling problem for local mutual exclusion on arbitrary conflict graphs. We establish necessary conditions to solve wait-free dining under eventual weak exclusion in message-passing systems with crash faults. Wait-free dining ensures that every correct hungry process eventually eats. Eventual weak exclusion permits finitely many scheduling mistakes, but eventually no live neighbors eat simultaneously; this exclusion criterion models scenarios where scheduling mistakes are recoverable or only affect performance. Previous work showed that the eventually perfect failure detector ($\diamond\mathcal{P}$) is sufficient to solve wait-free dining under eventual weak exclusion; we prove that $\diamond\mathcal{P}$ is also necessary, and thus $\diamond\mathcal{P}$ is the weakest oracle to solve this problem. Our reduction also establishes that any such dining solution can be made eventually fair. Finally, the reduction itself may be of more general interest; when applied to wait-free *perpetual* weak exclusion, our reduction produces an alternative proof that the more powerful trusting oracle ($\mathcal{T}$) is necessary (but not sufficient) to solve the problem of Fault-Tolerant Mutual Exclusion (FTME).

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications; network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations between models*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*

General Terms

Algorithms, Reliability, Theory

*This work was supported in part by NSF grant 0500265 and by the Texas Higher Education Coordinating Board under grants ARP 000512-0007-2006 and ARP 000512-0130-2007.

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

Keywords

Dining Philosophers, Failure Detectors, Mutual Exclusion, Partial Synchrony, Wait-Freedom

1. INTRODUCTION

We examine the Dining Philosophers problem (or dining, for short), a classic problem in distributed scheduling. First proposed by Dijkstra for a ring topology in [5], dining was later generalized by Lynch in [10] for local mutual exclusion on arbitrary conflict graphs. We explore a recent dimension of this problem with the following primary result: *wait-free* dining under *eventual weak exclusion* [12] is equivalent to the class of *eventually perfect failure detectors* ($\diamond\mathcal{P}$) from the Chandra-Toueg hierarchy [3].

Subsequent sections will explain the foregoing terms in greater detail. For now, we introduce these concepts only informally. The variant of dining we consider guarantees that — in spite of potential process crashes — every correct process competing for exclusive access to its critical section eventually accesses its critical section (wait freedom), and that eventually no two live neighbors are in their critical sections simultaneously (eventual weak exclusion — $\diamond\mathcal{WX}$). The time for convergence to the exclusive suffix may vary for each run and is not assumed to be known.

A *failure detector* can be viewed as a distributed oracle that can be queried for (potentially incorrect) information about process crashes [3]. Failure detectors can encapsulate temporal uncertainties about the underlying system models in which they are implemented in terms of the (un)reliability of the suspect lists provided. The eventually perfect failure detector – $\diamond\mathcal{P}$ – is one such failure detector. 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 for convergence to the perfect suffix may vary for each run and is not assumed to be known. Despite such unreliability, $\diamond\mathcal{P}$ is sufficiently powerful to solve many crash-tolerant problems including consensus [3], stable leader election [1], and crash-locality-1 dining for perpetual exclusion [11].

It is also known that $\diamond\mathcal{P}$ is *sufficient* to solve wait-free dining under $\diamond\mathcal{WX}$ [12, 13]. A natural follow-up question is whether $\diamond\mathcal{P}$ is also *necessary*; that is, does $\diamond\mathcal{P}$ encapsulate temporal assumptions equivalent to those encapsulated by a wait-free scheduling service for eventual weak exclusion in systems subject to crash faults? Our primary result answers this question affirmatively: $\diamond\mathcal{P}$ is also necessary, and hence is equivalent to wait-free dining under $\diamond\mathcal{WX}$.

We establish the necessity of $\diamond\mathcal{P}$ using the standard proof technique of reducing a candidate failure-detection oracle to the target problem itself [2]; that is, we construct an asynchronous transformation that extracts the oracle $\diamond\mathcal{P}$ from an underlying black-box instance of wait-free dining for eventual weak exclusion (denoted hereafter as $\mathcal{WF}$-$\diamond\mathcal{WX}$). As such, the primary technical challenge is demonstrating how the temporal uncertainty implied by an unreliable (but eventually perfect) oracle can be reduced to an unreliable (but eventually exclusive) scheduler. In conjunction with the sufficiency results from [12, 13], our necessity reduction establishes that $\diamond\mathcal{P}$ is, in fact, the weakest failure detector to solve $\mathcal{WF}$-$\diamond\mathcal{WX}$.

A related paper on boosting obstruction-freedom [8] claims that $\diamond\mathcal{P}$ is the weakest failure detector to solve wait-free contention management (a special case of wait-free $\diamond\mathcal{WX}$). Although the equivalence relation between $\diamond\mathcal{P}$ and wait-free contention management happens to be true, the reduction used to establish the equivalence is actually flawed, as is the related proof of correctness. Our paper reveals a subtle, but important, vulnerability in [8], and provides an alternate reduction to remedy the error.

Our reduction also leads to two secondary results which are briefly stated and explained in the next two paragraphs. Both results are tangential to the primary arc of this paper, but we present them for completeness as relevant implications of the present research.

First, any $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining solution can be transformed into an augmented algorithm satisfying *eventual 2-fairness*. As defined in [13], eventual k-fairness ensures that, for each run, there exists a time after which no process can enter its critical section more than k consecutive times while any correct neighbor remains hungry. The construction in [13] demonstrates that asynchronous systems augmented with $\diamond\mathcal{P}$ are sufficient to solve $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining with eventual 2-fairness.[1] Thus, beginning with any solution to $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining, one can apply the reduction in our paper to extract an implementation of $\diamond\mathcal{P}$, which can then be used in the construction from [13] to produce a $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining solution with the stronger service property of eventual 2-fairness. This two-step construction, however, might not be optimal; if a $\diamond\mathcal{P}$-based transformation to eventual 1-fairness exists, for example, then solutions for $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining can be converted into eventually lock-step schedulers by using a clique for the dining conflict graph.

Our other secondary result concerns the actual reduction algorithm itself, which may be of more general interest as a widget for proving the necessity of other failure detectors for related synchronization problems. For example, a paper [4] by Delporte-Gallet et al. proved in 2005 that the Trusting [4] and Strong [3] failure detectors ($\mathcal{T}+\mathcal{S}$) are sufficient to solve wait-free mutual exclusion, a special case of dining where the conflict graph is a clique. Their result considered the stricter criterion of *perpetual weak exclusion* — $\Box\mathcal{WX}$ — for which live neighbors *never* eat simultaneously; as such, ($\mathcal{T}+\mathcal{S}$) can implement wait-free and mistake-free schedulers. The same paper [4] proved that $\mathcal{T}$ is necessary for wait-free mutual exclusion, but it is still unknown whether the more powerful composition of ($\mathcal{T}+\mathcal{S}$) is necessary too. Our paper presents a reduction that extracts the oracle $\diamond\mathcal{P}$ from any instance of $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining. If applied to any wait-free instance of perpetual weak exclusion, however, the reduction extracts the more powerful oracle $\mathcal{T}$ instead. This application of our reduction provides an alternate proof of the necessity of $\mathcal{T}$. As described in Section 9, our reduction can also be amended (under $\Box\mathcal{WX}$) to extract an oracle slightly more powerful than $\mathcal{T}$, which implies the following negative result: *$\mathcal{T}$ is insufficient to solve wait-free mutual exclusion by itself.*

[1]The actual construction is for message-passing systems, but since it only uses bounded variables and bounded-capacity channels, the result also applies to shared-memory systems.

2. ON EVENTUAL WEAK EXCLUSION

Eventual weak exclusion ($\diamond\mathcal{WX}$) has been a frequently overlooked and under-explored safety property. Although a full defense is beyond the scope of this paper, the following examples illustrate the practical utility of $\diamond\mathcal{WX}$. In particular, $\diamond\mathcal{WX}$ provides a basis for crash-tolerant applications in scenarios where safety violations are either recoverable or only affect performance (rather than correctness).

Duty Cycle Scheduling. Consider wireless sensor networks (WSN) that must cover a given surveillance environment. Such WSNs typically have finite power supplies; as such, every node will eventually crash due to power depletion. Ideally, the life-span of the WSN should greatly exceed the life-span of its constituent nodes. With node redundancy, a WSN can employ schedulers to schedule a minimal set of nodes to be *on duty* to cover the environment, while the remainder can sleep to conserve energy.

In this scenario, the resources being shared are the coverage areas for which nodes are on duty. Nodes that volunteer to be on duty can be modeled as contending for exclusive access to this shared resource, and nodes that are on duty can be modeled as being in their critical section. Ideally, schedulers for such WSNs should satisfy wait-freedom and perpetual weak exclusion. Wait-freedom guarantees sensor coverage despite node crashes whereas perpetual weak exclusion maximizes the network life-span by scheduling a minimal set of nodes to be on duty.

The underlying system environment for such WSNs is often *partially synchronous* [7]. It is reasonable to expect the partial synchrony in such environments to be sufficient to implement $\diamond\mathcal{P}$, but insufficient to implement stronger oracles like the *trusting failure detector* ($\mathcal{T}$)[2]. However, earlier work demonstrates that $\diamond\mathcal{P}$ is *insufficient* to achieve both wait-freedom and perpetual weak accuracy [11], whereas $\mathcal{T}$ is *necessary* to achieve the same [4]. So is it possible to achieve wait-freedom *and* maintain an extended network life-span despite such limited synchrony?

We claim that the answer is *yes*. We know that $\diamond\mathcal{P}$ can implement a wait-free scheduler under *eventual* weak exclusion ($\diamond\mathcal{WX}$) [12]. Although such schedulers guarantee complete sensor coverage through wait-freedom, they may make finitely many mistakes by scheduling redundant nodes to go on duty concurrently. These mistakes, however, only result in redundant coverage, which impacts the performance, but not the correctness, of WSNs. Eventually only a minimal set of nodes are scheduled to be on duty, thereby maximizing network life-span.

Recoverable Resources. Recall that wait-free schedulers for perpetual weak exclusion ($\Box\mathcal{WX}$) may be impossible to implement in some systems [11]. However, not all applications in such systems require $\Box\mathcal{WX}$. Consider systems with recoverable resources where applications are scheduled using

[2]The trusting oracle is discussed in detail in Section 8.

a wait-free scheduler with $\diamond\mathcal{WX}$. In such systems, exclusion violations can often be modeled as transient faults. For example, if two processes perform non-atomic updates on shared resources, the result may be viewed as an arbitrary corruption of the shared data. In the presence of such scheduling errors, consistent data values may be restored via data redundancy, rollback recovery, or even self-stabilization [6]. As such, $\diamond\mathcal{WX}$ is still viable when $\Box\mathcal{WX}$ is not possible.

Contention Managers. Contention Managers (CM) [8] are wait-free eventually exclusive ($\diamond\mathcal{WX}$) protocols which can boost the liveness properties of obstruction-free software transactional memory (STM) implementations. Obstruction freedom guarantees progress only if a transaction executes in isolation for a "long enough" period of time. In high-contention systems, where multiple processes may attempt to execute concurrent transactions, there may be no guarantee on progress. However, CMs can eliminate such contention as follows: clients wanting to execute a transaction first request permission from the CM and execute the transaction only upon being permitted by the CM. For a finite prefix, the CM may permit multiple clients to execute their transactions concurrently which may result in unsuccessful transactions. However, eventually, the CM permits just one client to execute its transaction at a time. By obstruction freedom, such transactions are guaranteed to succeed. Insofar as the CM guarantees that every client with a pending transaction is eventually permitted to execute its transaction, the liveness property of the STM implementation is boosted from obstruction freedom to wait freedom. Thus, CMs show the practical utility of $\diamond\mathcal{WX}$ as a mechanism for funneling high-contention systems into contention-free systems with stronger liveness properties.

3. ON CONTENTION MANAGEMENT

As described earlier, contention managers are wait-free eventually exclusive protocols in shared memory systems which boost the liveness properties of software transactional memory (STM) implementations. A result in [8] states that $\diamond\mathcal{P}$ is the weakest failure detector to convert an obstruction free shared-object implementation into a wait-free one.

Although the result from [8] is true, the construction (and the accompanying proof) provided in [8] to extract $\diamond\mathcal{P}$ from a wait-free contention manager do not apply universally (*i.e.*, the transformation is not black-box based). Specifically, there exist certain implementations of wait-free contention managers from which $\diamond\mathcal{P}$ cannot be extracted using the construction in [8]. Therefore, insofar as the construction does not work for any black-box implementation of wait-free contention managers, the construction, the result, and the proof of correctness in [8] are not universal.

The necessity of $\diamond\mathcal{P}$ to implement a wait-free contention manager is demonstrated in [8] by extracting $\diamond\mathcal{P}$ from a black-box implementation of a wait-free contention manager. The construction is as follows: For each ordered pair of processes (p, q) where p is monitoring the liveness of q, the $\diamond\mathcal{P}$ module is implemented by an instance of a wait-free contention manager in which p and q participate. Upon initialization, q sends heartbeats to p at regular intervals and requests permission from the contention manager for obstruction-free access. Upon being granted permission by the contention manager, q enters the section of the code that requires obstruction freedom (called the "critical section") and never exits.

Process p, on the other hand, upon receiving a heartbeat from q, trusts q as being correct and proceeds to request permission from the contention manager for obstruction-free access. Upon being granted permission by the contention manager, p enters its "critical section" and immediately exits its "critical section" (allowing the contention manager to permit q to enter its respective critical section), p suspects q, and waits for another heartbeat from q to start the sequence of actions all over again.

The basis for correctness is the following: If q crashes, then p will gain access to its own critical section, and hence suspect q. Since p will stop receiving heartbeats from q, p will permanently suspect q. On the other hand, if q is correct, then the wait-free contention manager eventually provides exclusive access to a single process, eventually q is permanently in its critical section and p is permanently locked out. Hence, p eventually trusts q forever.

Note that the correctness of the $\diamond\mathcal{P}$ implementation assumes that wait-free contention managers guarantee eventual exclusion even in executions where a process never exits the obstruction-free section of its code. This, however, may not be the case with all wait-free contention manager implementations.

For instance, consider the implementation of wait-free dining under $\diamond\mathcal{WX}$ in [12]. This implementation can be used as a wait-free contention manager (because it satisfies the specifications for a wait-free contention manager). However, the implementation in [12] guarantees exclusive access to the "critical section" in the suffix of an execution *only after (1) the failure detector $\diamond\mathcal{P}$ stops making mistakes, and (2) each process that entered its critical section before $\diamond\mathcal{P}$ stopped making mistakes has also exited.*

Note that if the implementation from [12] is used as the wait-free contention manager for the $\diamond\mathcal{P}$ construction in [8], and the aforementioned (correct) process q enters its critical section during the non-exclusive prefix of the execution and never exits, then the $\diamond\mathcal{P}$ construction in [8] cannot guarantee an exclusive suffix in which p is permanently locked-out. In other words, p could access its critical section infinitely often, and hence suspect q infinitely often, thereby violating the $\diamond\mathcal{P}$ specifications.

This paper addresses this vulnerability and provides a construction which implements $\diamond\mathcal{P}$ in *all* black-box solutions to dining under $\diamond\mathcal{WX}$ (including the solution in [12]). Specifically, this paper demonstrates a more general equivalence result by proving that the wait-free $\diamond\mathcal{WX}$ variant of the dining philosophers problem, which is a generalization of mutual exclusion (which in turn is a generalization of contention management) is equivalent to $\diamond\mathcal{P}$.

4. TECHNICAL FRAMEWORK

We consider asynchronous systems where message delay, clock drift, and relative process speeds are unbounded. Additionally, we posit a discrete global clock $\mathbb{T}$ whose range of clock ticks is the set of natural numbers $\mathbb{N}$. $\mathbb{T}$ is merely a conceptual device and inaccessible to processes in the system.

Processes. The system has a finite set of processes Π. Processes execute actions as atomic steps. In each atomic step, a process receives at most one message from each process, makes a state transition, and sends at most one message to each process.

Channels. Processes send and receive messages to each other through channels. Processes are assumed to be connected to all the processes in the system by 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.

Runs. A run consists of a potentially infinite sequence of enabled steps taken by processes while executing an algorithm. For terminating algorithms, processes are modeled as having reached a final state S_f followed by an infinite sequence of (no-op) steps so that the process loops in state S_f. Note that not all processes may execute an infinite sequence of steps. Only *correct*, *i.e.*, non-faulty, processes execute an infinite sequence of steps.

Faults. In each run, processes are either *correct* or *faulty*. Correct processes execute actions according to their specification, and never fail. *Faulty* processes, on the other hand, fail after finite time. Processes can fail only by *crashing*, which occurs when a process ceases execution without warning and never recovers. Any process that has not crashed is considered to be *live*. Thus, correct processes are always live, and faulty processes are live until they crash.

Failure Detectors. An unreliable failure detector can be viewed as a distributed oracle that can be queried for (possibly incorrect) information about process crashes [3]. 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 [3]:

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

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

Thus, $\diamond\mathcal{P}$ may suspect correct processes finitely many times in any execution. However, $\diamond\mathcal{P}$ must *converge* at some point, after which the oracle provides reliable information about process crashes. The time to convergence may be unknown and may vary from run to run.

Dining. A dining instance is modeled by an undirected conflict graph $DP = (\Pi, E)$, where each vertex $p \in \Pi$ represents a diner, and each edge $(p, q) \in E$ represents a set of resources shared between neighbors p and q. Each diner is either *thinking*, *hungry*, *eating*, or *exiting*. The above four states correspond to the four basic phases of a participating process: executing independently, requesting shared resources, utilizing shared resources in a critical section, and relinquishing its exclusive access to the shared resources, respectively.

Initially, every process (diner) is thinking. Although processes may think forever, they are permitted to become hungry at any time. Upon being scheduled to eat, the process enters its critical section. Eating is always finite (but not necessarily bounded) for correct processes; such processes must transit from eating to *exiting* in finite time. Similarly, a correct dining solution guarantees that exiting is finite too; an exiting process eventually transits to thinking. Hungry neighbors are said to be in *conflict*, because they compete for a set of shared but mutually exclusive resources. A correct solution to wait-free dining under eventual weak exclusion ($\diamond\mathcal{WX}$) is an algorithm that schedules diner transitions from hungry to eating, subject to the following two requirements:

• **Eventual Weak Exclusion $\diamond\mathcal{WX}$:** *For every run, there exists a time after which no two live neighbors eat simultaneously.* During any run, $\diamond\mathcal{WX}$ can make finitely many scheduling mistakes, but eventually converges to an infinite suffix during which live neighbors never eat simultaneously. The time to convergence may vary from one run to another and may be unknown. Thus, $\diamond\mathcal{WX}$ may be viewed as eventual safety [6].

• **Wait-Freedom:** If correct processes eat for finite time, then *every correct hungry process eventually eats, regardless of how many processes crash.* Wait-freedom [9] guarantees individual progress in the presence of crash faults. As such, wait-free exclusion algorithms never starve correct hungry processes.

5. METHODOLOGY AND OVERVIEW

To show that $\diamond\mathcal{P}$ is necessary to solve wait-free dining under $\diamond\mathcal{WX}$ (denoted $\mathcal{WF}$-$\diamond\mathcal{WX}$), we present a reduction of $\diamond\mathcal{P}$ to the problem of $\mathcal{WF}$-$\diamond\mathcal{WX}$.

The proof technique works as follows. Suppose, for the purpose of contradiction, there exists a failure detector $\mathcal{D}$ which is strictly weaker than $\diamond\mathcal{P}$ and yet can also solve $\mathcal{WF}$-$\diamond\mathcal{WX}$. Using a black-box solution to $\mathcal{WF}$-$\diamond\mathcal{WX}$, 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 $\mathcal{WF}$-$\diamond\mathcal{WX}$, but (by construction) $\mathcal{WF}$-$\diamond\mathcal{WX}$ 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 $\mathcal{WF}$-$\diamond\mathcal{WX}$.

5.1 Preliminary Construction

The key idea of our construction is to convert wait-freedom and eventual weak exclusion into an eventually reliable timeout mechanism for detecting crash faults. Consider two processes p and q, where p is correct and p is monitoring q. Let p and q compete for exclusive access to their respective critical sections infinitely often in a given instance of $\mathcal{WF}$-$\diamond\mathcal{WX}$. If q is faulty, then wait-freedom guarantees that p will eat infinitely often after q crashes. If q is correct, however, then $\diamond\mathcal{WX}$ guarantees an infinite suffix during which p does not eat until q has exited eating. Every time q eats, let q send a ping and exit only after receiving an ack from p. Subsequently, when p transits to eating, p has a basis for trusting q if p had received a ping from q since the last time p ate.

However, $\mathcal{WF}$-$\diamond\mathcal{WX}$ does not guarantee fairness insofar as it is possible for p to eat an unbounded number of times between each time q eats; this allows p to suspect q infinitely often. To circumvent this, p and q compete in *two* $\mathcal{WF}$-$\diamond\mathcal{WX}$ instances. For ease of understanding, we view each process as consisting of two *threads* which participate in two dining instances $\mathcal{DX}_1$ and $\mathcal{DX}_2$. The threads $p.w_0$ and $p.w_1$ in p are called *witness* threads, and the threads $q.s_0$ and $q.s_1$ in q are called *subject* threads (simply because p is 'witnessing' the liveness of q).

The threads $p.w_i$ and $q.s_i$ participate in the dining instance $\mathcal{DX}_i$, where $i \in \{0, 1\}$. Hereafter, $\mathcal{DX}_i$ refers to one of $\mathcal{DX}_0$ and $\mathcal{DX}_1$, $q.s_i$ refers to one of the subject threads

var $w_{i\in\{0,1\}}$.state $\leftarrow$ thinking	*Initially, the witnesses are thinking*
boolean switch $\leftarrow 0$	*Witness w_0 will be first to become hungry*
boolean havePing$_{i\in\{0,1\}}$ $\leftarrow$ **false**	*Set to true when a ping is received; used to determine suspicion*
boolean suspect$_q$ $\leftarrow$ **true**	*Initially suspect q*
1 : $\{(w_i$.state = thinking$) \wedge (w_{1-i}$.state = thinking$) \wedge$ (switch $= i)\} \longrightarrow$	*Action $\mathcal{W}_h$*
2 : w_i.state $\leftarrow$ hungry	*Become Hungry in $\mathcal{DX}_i$*
3 : $\{(w_i$.state = eating$)\} \longrightarrow$	*Action $\mathcal{W}_x$*
4 : suspect$_q$ $\leftarrow \neg$havePing$_i$	*Trust q iff ping has been received*
5 : havePing$_i$ $\leftarrow$ **false**	
6 : switch $\leftarrow (1-i)$	*Enable subject $p.w_{1-i}$ to become hungry*
7 : w_i.state $\leftarrow$ exiting	*Exit eating in $\mathcal{DX}_i$*
8 : {**upon** receive $\langle$ping$\rangle$ **from** subject $q.s_i$} $\longrightarrow$	*Action $\mathcal{W}_p$*
9 : havePing$_i$ $\leftarrow$ **true**	
10 : **send** $\langle ack \rangle$ **to** subject $q.s_i$	

Alg. 1: Actions for the witness $p.w_{i\in\{0,1\}}$ in dining instance $\mathcal{DX}_i$. Thread $p.w_{1-i}$ denotes the other witness thread in p, and $q.s_i$ denotes the subject thread at process q participating in $\mathcal{DX}_i$. Process q is suspected if suspect$_q$ is true.

$q.s_0$ and $q.s_1$, and $p.w_i$ refers to one of the witness threads $p.w_0$ and $p.w_1$, respectively. The notations $\mathcal{DX}_{1-i}$, $q.s_{1-i}$, and $p.w_{1-i}$ refer to the peer dining instance, subject thread, and witness thread of $\mathcal{DX}_i$, $q.s_i$, and $p.w_i$, respectively. Conceptually, $\mathcal{DX}_0$ and $\mathcal{DX}_1$ implement the local detection module at p wherein p monitors the liveness of q. Implementing a similar local detection module at q to monitor p requires two more dining instances wherein the witness versus subject roles of p and q are reversed.

Although logically distinct, the subject threads $q.s_0$ and $q.s_1$ are implemented as a single stream of physical execution. More specifically, each subject thread is a distinct set of actions, the union of which is executed under interleaving semantics by process q. Consequently, the variables used by $q.s_0$ and $q.s_1$ are mutually accessible to each other. As such, the failure semantics are also correlated; that is, if process q crashes, then both subject threads $q.s_0$ and $q.s_1$ also crash. Similar remarks apply to the witness threads $p.w_0$ and $p.w_1$ executed by process p.

The witness threads in p take turns becoming hungry, eating, and exiting in their respective dining instances. Witness $p.w_0$ becomes hungry, transits to eating, determines if q was live, and then exits. After $p.w_0$ exits eating, $p.w_1$ becomes hungry, transits to eating, determines if q was live, and then exits. After $p.w_1$ exits eating, $p.w_0$ becomes hungry again, and so on.

The subject threads in process q, on the other hand, coordinate their eating sessions via the following hand-off mechanism. Subject $q.s_0$ becomes hungry, and after it transits to eating, it does not exit until $q.s_1$ becomes hungry and starts eating as well. Similarly, after $q.s_1$ transits to eating, $q.s_1$ does not exit eating until $q.s_0$ (which had exited earlier) becomes hungry and starts eating again. In other words, the beginning and ending of each subject's eating session overlaps with the other subject's eating session. This hand-off mechanism ensures the following: in the suffix where $\mathcal{WF}$-$\diamond\mathcal{WX}$ has stopped making scheduling mistakes, witness $p.w_i$ cannot eat twice consecutively in dining instance $\mathcal{DX}_i$ without $q.s_i$ eating at least once in the same dining instance $\mathcal{DX}_i$. This is illustrated in Fig. 1.

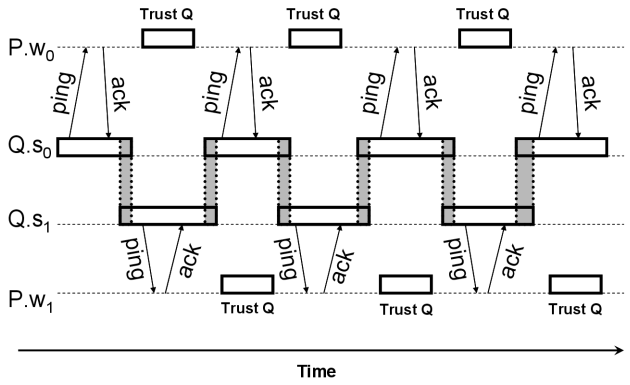

Figure 1: Witness and subject threads in the exclusive suffix. The white boxes denote eating sessions and the gray regions denote the overlap in subjects' eating sessions.

Every time $q.s_i$ eats in any dining instance, it sends a ping to $p.w_i$ and does not exit until an ack has been received. In other words, when $q.s_i$ exits eating, $p.w_i$ has recorded that a ping has been received from $q.s_i$. If q is correct, then in the infinite suffix after $\diamond\mathcal{WX}$ has converged, every time $p.w_i$ eats in its dining instance, $p.w_i$ has already received a ping from $q.s_i$ previously, and hence trusts q for the infinite suffix.

6. THE REDUCTION ALGORITHM

Let Π be a finite system of processes. For each ordered pair of processes, (p, q), we implement an eventually perfect failure detector $\diamond\mathcal{P}$ using two instances of a black-box solution to $\mathcal{WF}$-$\diamond\mathcal{WX}$: $\mathcal{DX}_0$, and $\mathcal{DX}_1$ as described in Section 5.1. Actions for the witness threads and the subject threads are presented in Alg. 1 and Alg. 2, respectively. The witness and subject threads participate in a ping-ack protocol.

Initially, process p suspects q, and the witness threads are initialized to be thinking. The witness thread $p.w_0$ then becomes hungry in $\mathcal{DX}_0$ (Action $\mathcal{W}_h$). After being scheduled

var $s_{i\in\{0,1\}}$.state $\leftarrow$ thinking	*Initially, the subjects are thinking*
boolean trigger $\leftarrow 0$	*Initially, set only subject s_0 to become hungry*
boolean $\text{ping}_{i\in\{0,1\}} \leftarrow$ **true**	*Initially, enable the subject to send a ping upon eating*
1 : $\{(s_i.\text{state} = \text{thinking}) \wedge (\text{trigger} = i)\} \longrightarrow$	*Action $\mathcal{S}_h$*
2 : $s_i.\text{state} \leftarrow \text{hungry}$	*Become hungry in the dining instance $\mathcal{DX}_i$*
3 : $\{(s_i.\text{state} = \text{eating}) \wedge (s_{1-i}.\text{state} \neq \text{eating}) \wedge (\text{ping}_i = \textbf{true})\} \longrightarrow$	*Action $\mathcal{S}_p$*
4 : **send** $\langle ping \rangle$ **to** $p.w_i$	*Send a ping to peer witness at p and wait for an ack*
5 : $\text{ping}_i \leftarrow$ **false**	
6 : { **upon receive** $\langle ack \rangle$ **from** $p.w_i$} $\longrightarrow$	*Action $\mathcal{S}_a$*
7 : trigger $\leftarrow (1 - i)$	*Schedule the other subject to become hungry*
8 : $\{(s_i.\text{state} = \text{eating}) \wedge (s_{1-i}.\text{state} = \text{eating}) \wedge (\text{trigger} = 1 - i)\} \longrightarrow$	*Action $\mathcal{S}_x$*
9 : $\text{ping}_i \leftarrow$ **true**	*Enable sending a ping next time*
10 : $s_i.\text{state} \leftarrow \text{exiting}$	*Exit eating in $\mathcal{DX}_i$*

Alg. 2: Actions for the subject $q.s_{i\in\{0,1\}}$ in dining instance $\mathcal{DX}_i$. Subject thread $q.s_{1-i}$ denotes the other subject thread in q, and $p.w_i$ denotes the witness thread at process p participating in $\mathcal{DX}_i$.

to eat by the underlying $\mathcal{WF}$-$\Diamond\mathcal{WX}$ protocol, $p.w_0$ checks to see if $havePing_0$ is true: if so, p trusts q; otherwise it suspects q. After determining the liveness of q, $p.w_0$ exits $\mathcal{DX}_0$. After $p.w_0$ transits back to thinking, $p.w_1$ becomes hungry in $\mathcal{DX}_1$ (Action $\mathcal{W}_h$). After being scheduled to eat by the underlying $\mathcal{WF}$-$\Diamond\mathcal{WX}$ protocol, $p.w_1$ checks to see if $havePing_1$ is true: if so, p trusts q; otherwise, it suspects q (Action W_x). After determining the liveness of q, $p.w_1$ exits $\mathcal{DX}_1$. Now, $p.w_0$ becomes hungry in $\mathcal{DX}_0$, and so on. Also, each ping is acknowledged by a single ack (Action $\mathcal{W}_p$).

The subject threads are initialized to be thinking initially. The subject thread $q.s_0$ immediately transits to becoming hungry in $\mathcal{DX}_0$ (Action $\mathcal{S}_h$), and is eventually scheduled to eat (by wait-freedom). When $q.s_0$ is scheduled to eat, it sends a ping to the peer witness (Action $\mathcal{S}_p$), waits for an ack, and, upon receiving the ack (in Action $\mathcal{S}_a$), schedules $q.s_1$ to become hungry in $\mathcal{DX}_1$ (by setting trigger to 1). When $q.s_1$ starts eating in $\mathcal{DX}_1$, $q.s_0$ exits $\mathcal{DX}_0$ (Action $\mathcal{S}_x$). The subject thread $q.s_1$ then follows the same sequence of steps as $q.s_0$.

7. CORRECTNESS PROOFS

Proof Structure: We prove strong completeness by showing that if the subjects crash, then the witnesses eat infinitely often (Lemma 11) and their havePing_i variables become continuously false. Therefore, all crashed processes are eventually and permanently suspected.

We prove eventual strong accuracy by showing that if the subjects are correct, then eventually, every time a witness is scheduled to eat, its havePing_i variable is *true*. Therefore, all correct processes are eventually and permanently trusted. In order to prove eventual strong accuracy, we show that witnesses and subjects eat infinitely often (Lemmas 11 and 7, respectively), which enables pings to be sent infinitely often. In fact, we show that pings and acks are sent only when subjects are eating (Lemma 5) and not otherwise (Lemma 3). But note that subjects can eat infinitely often only if every hungry subject eats (Lemma 1), and eating is finite (Lemma 6). Additionally, witnesses are throttled from eating an unbounded number of times between subjects' consecutive eating sessions. This follows from the observation that some witness is always thinking (Lemma 9) and eventually some subject is always eating (Lemma 8), and hence each witness will have to wait until one of the subjects exits eating before it can eat. Note that unlike the subjects, witnesses take turns eating (Lemmas 10 and 12). We now demonstrate correctness through a formal, rigorous proof.

Formal Proof: Consider a run of an asynchronous system augmented with a solution to $\mathcal{WF}$-$\Diamond\mathcal{WX}$. Consider two processes p and q in this run where p is correct. Let p monitor q in this run. Process p has two witness threads $p.w_0$ and $p.w_1$ which execute the action system in Alg. 1, and process q has two subject threads $q.s_0$ and $q.s_1$ which execute the action system in Alg. 2. Let $\mathcal{DX}_0$ be the $\mathcal{WF}$-$\Diamond\mathcal{WX}$ instance shared between $p.w_0$ and $q.s_0$, and let $\mathcal{DX}_1$ be the $\mathcal{WF}$-$\Diamond\mathcal{WX}$ instance shared between $p.w_1$ and $q.s_1$.

We use two terms in the proof that merit definition here:
Eating session: Each eating session of a thread is the duration between a time t_e when the process starts eating and the first time $t_x > t_e$ when the thread exits.
Hungry-Eating session: Each hungry-eating session of a thread is the duration between a time t_h when the thread becomes hungry and the first time $t_x > t_h$ when the thread exits.

LEMMA 1. *If process q is correct, and a subject $q.s_i$ becomes hungry in $\mathcal{DX}_i$, then $q.s_i$ eventually eats in $\mathcal{DX}_i$.*

PROOF. Note that $q.s_i$ and $p.w_i$ are neighbors in the dining instance $\mathcal{DX}_i$. From Action $\mathcal{W}_p$ we know that if $p.w_i$ is scheduled to eat in $\mathcal{DX}_i$, then $p.w_i$ will exit in finite time; that is, $p.w_i$ eats only for finite time. From the wait-freedom property of $\mathcal{DX}_i$ and the finite eating sessions of $p.w_i$, it follows that if q is correct and $q.s_i$ becomes hungry in $\mathcal{DX}_i$, then eventually $q.s_i$ eats in $\mathcal{DX}_i$. □

LEMMA 2. *The following invariant holds:* $(s_i.\text{state} \neq \text{eating}) \Rightarrow (\text{ping}_i = true)$.

PROOF. Initially, $s_i.\text{state} = \text{thinking}$ and $\text{ping}_i = true$. Therefore, the invariant holds initially. The only actions that change the value of ping_i are Actions $\mathcal{S}_p$ and $\mathcal{S}_x$. Action $\mathcal{S}_p$ sets ping_i to $false$ while $s_i.\text{state} = \text{eating}$. Therefore, the invariant holds true after s_i executes $\mathcal{S}_p$. Action $\mathcal{S}_x$ sets ping_i to *true* before s_i exits eating. Hence, the invariant holds true after $\mathcal{S}_x$ is executed. □

LEMMA 3. *When* $(s_i.\mathsf{state} \neq \mathsf{eating}) \vee (\mathsf{ping}_i = true)$ *there are no messages in transit between* $q.s_i$ *and* $p.w_i$.

PROOF. For the purpose of contradiction, we assume that in some run there is a message in transit between $q.s_i$ and $p.w_i$ when $(s_i.\mathsf{state} \neq \mathsf{eating}) \vee (\mathsf{ping}_i = true)$. Consider the earliest time t_f when $(s_i.\mathsf{state} \neq \mathsf{eating}) \vee (\mathsf{ping}_i = true)$ and some message is in transit between $q.s_i$ and $p.w_i$.

Since $q.s_i$ sends only pings and $p.w_i$ sends only acks, the message in transit at time t_f is either a ping or an ack.

If a ping is in transit at time t_f, then the ping must have been sent by $q.s_i$ at time $t_p < t_f$. Alternatively, if an ack is in transit at time t_f, then from Action $\mathcal{W}_x$ we know that the ack must have been sent in response to a ping sent by $q.s_i$ at time $t_p < t_f$. From Action $\mathcal{S}_p$, we know that at time t_p, $s_i.\mathsf{state} = \mathsf{eating}$ and $\mathsf{ping}_i = true$.

Let $q.s_i$ be in its k^{th} hungry-eating session at time t_p. Let this hungry-eating session begin at time $t_h < t_p$. From Action $\mathcal{S}_h$ we know that $\mathsf{trigger} = i$ at time t_h. By hypothesis, we know that in the k^{th} hungry-eating session there is no ack in transit to $q.s_i$ until time t_p, because $t_p < t_f$ and until t_p, $\mathsf{ping}_i = true$. Therefore, until time t_p, $q.s_i$ does not execute Action $\mathcal{S}_a$. In other words, at time t_p, $q.s_i$ is eating and $\mathsf{trigger} = i$.

However, at time t_f, either $q.s_i$ is not eating, or $\mathsf{ping}_i = true$. Therefore, at some time between t_p and t_f, ping_i becomes true or $q.s_i$ exits eating. In fact, ping_i is set to true only when $q.s_i$ exits eating. Therefore, $q.s_i$ exits eating between time t_p and t_f. Let such time be t_x.

Note that $q.s_i$ can exit eating only when $\mathsf{trigger} = 1 - i$, but we know that $\mathsf{trigger} = i$ at time t_p. Therefore, between time t_p and t_x, $\mathsf{trigger}$ is set to $1 - i$; that is, in the interval $[t_p, t_x]$, $q.s_i$ receives an ack. This ack could not be for the ping sent at time t_p because, at time $t_f > t_x$, either the ping sent at t_p is still in transit, or the ack generated for the ping sent at time t_p is still in transit. Therefore, the ack received in the interval $[t_p, t_x]$ must have been for a ping sent in $q.s_i$'s (say) j^{th} eating session where $j < k$. This is possible only if either the ping sent in the j^{th} eating session, or the ack generated for that ping was in transit between $q.s_i$'s j^{th} eating session and k^{th} eating session. This contradicts our earlier assumption that t_f was the earliest time at which a message is in transit between $q.s_i$ and $p.w_i$ when $(s_i.\mathsf{state} \neq \mathsf{eating}) \vee (\mathsf{ping}_i = true)$. □

LEMMA 4. *The following invariant holds:* $(s_i.\mathsf{state} = \mathsf{hungry}) \Rightarrow (\mathsf{trigger} = i)$.

PROOF. Subject $q.s_i$ becomes hungry by executing Action $\mathcal{S}_h$ which is enabled only when $\mathsf{trigger} = i$. The only action that changes the value of $\mathsf{trigger}$ to $1 - i$ is Action $\mathcal{S}_a$. But we know from Lemma 3 that there are no messages in transit to $q.s_i$ when $s_i.\mathsf{state} \neq \mathsf{eating}$. Therefore, $q.s_i$ does not execute Action $\mathcal{S}_a$ while $s_i.\mathsf{state} = \mathsf{hungry}$. Hence $(s_i.\mathsf{state} = \mathsf{hungry}) \Rightarrow (\mathsf{trigger} = i)$. □

LEMMA 5. *If* p *and* q *are correct, then during every eating session of subject* $q.s_i$ *(a)* $q.s_i$ *sends exactly one ping to witness* $p.w_i$*, and (b)* $q.s_i$ *receives exactly one ack from* $p.w_i$ after $q.s_i$ *sends the ping to* $p.w_i$.

PROOF. From Lemmas 2 and 4 we know that when $q.s_i$ is hungry $\mathsf{ping}_i = true$ and $\mathsf{trigger} = i$. Therefore, when $q.s_i$ transits to eating, and before it executes any action while it is eating, $\mathsf{ping}_i = true$ and $\mathsf{trigger} = i$.

From Lemma 3 we know that there are no messages in transit between $q.s_i$ and $p.w_i$ while $q.s_i$ is hungry. Note that $p.w_i$ sends an ack to $q.s_i$ only upon receiving a ping from $q.s_i$. Therefore, when $q.s_i$ transits to eating, there is no ack in transit to $q.s_i$ until after $q.s_i$ sends a ping to $p.w_i$.

Consequently, when $q.s_i$ transits to eating, either Action $\mathcal{S}_p$ is enabled at $q.s_i$, or Action $\mathcal{S}_p$ is NOT enabled at $q.s_i$ (but no other action is enabled):

- *Case 1.* If Action $\mathcal{S}_p$ is enabled when $q.s_i$ transits to eating, then Action $\mathcal{S}_p$ is executed. Executing Action $\mathcal{S}_p$ disables the guard for the action, so it cannot be executed again until $\mathsf{ping}_i = true$, but ping_i becomes *true* only when $q.s_i$ is exiting. Hence, $q.s_i$ sends exactly one ping to witness $p.w_i$.

- *Case 2.* If Action $\mathcal{S}_p$ is disabled when $q.s_i$ transits to eating, then $s_{1-i}.\mathsf{state} = \mathsf{eating}$ (because $\mathsf{ping}_i = true$, $s_i.\mathsf{state} = \mathsf{eating}$, and Action $\mathcal{S}_p$ is disabled). If $s_{1-i}.\mathsf{state} = \mathsf{eating}$, $\mathsf{trigger} = i$, and $s_i.\mathsf{state} = \mathsf{eating}$, then Action $\mathcal{S}_x$ is enabled at $q.s_{1-i}$. When $q.s_{1-i}$ executes Action $\mathcal{S}_x$, $q.s_{1-i}$ exits eating, and this enables Action $\mathcal{S}_p$ at $q.s_i$. This now reduces to *Case 1.*

Therefore, when $q.s_i$ transits to eating, it sends exactly one ping to witness $p.w_i$.

Note that $q.s_i$ does not exit eating until $\mathsf{trigger} = 1 - i$. The only action that sets $\mathsf{trigger}$ to $1 - i$ is Action $\mathcal{S}_a$. Therefore, $q.s_i$ does not exit eating until it executes Action $\mathcal{S}_a$ at least once. However, since $q.s_i$ sends exactly one ping to $p.w_i$, and this ping arrives at $p.w_i$ in finite time, $p.w_i$ executes Action $\mathcal{W}_p$ exactly once, and sends exactly one ack to $q.s_i$. This ack arrives at $q.s_i$ in finite time. Since $q.s_i$ does not exit until it has executed Action $\mathcal{S}_a$ at least once, when $q.s_i$ receives the ack by executing Action $\mathcal{S}_a$, $q.s_i$ is still in its current eating session. □

LEMMA 6. *If* p *and* q *are correct, then* $q.s_i$*'s eating session is always finite.*

PROOF. From Lemma 4 we know that when $q.s_i$ is hungry, $\mathsf{trigger} = i$. By inspecting the action system in Alg. 2 we know that $\mathsf{trigger}$ is set to $1 - i$ only in Action $\mathcal{S}_a$ (in which $q.s_i$ receives an ack). Therefore, it follows that $\mathsf{trigger} = i$ from the time $q.s_i$ transits to becoming hungry to the time that $q.s_i$ receives an ack from $p.w_i$ in the eating session that immediately follows.

However, from Lemma 5, we know that during each eating session, subject $q.s_i$ sends exactly one ping to $p.w_i$ (by executing Action $\mathcal{S}_p$) and *subsequently* receives exactly one ack from $p.w_i$. Therefore, when $q.s_i$ executes Action $\mathcal{S}_p$ (in its eating session), $\mathsf{trigger} = i$. By applying Lemma 4 to $q.s_{1-i}$, we know that when $q.s_i$ executes Action $\mathcal{S}_p$, $s_{1-i}.\mathsf{state} \neq hungry$.

Inspecting the guard for Action $\mathcal{S}_p$ reveals that when $q.s_i$ executes Action $\mathcal{S}_p$, $s_{1-i}.\mathsf{state} \neq \mathsf{eating}$. Therefore, when $q.s_i$ executes Action $\mathcal{S}_p$, $q.s_{1-i}$ is either exiting or thinking.

After $q.s_i$ receives the (only) ack in its eating session[3], $\mathsf{trigger} = 1 - i$ (from Action $\mathcal{S}_a$). Since $q.s_{1-i}$ cannot become hungry until $\mathsf{trigger} = 1 - i$, it follows that $q.s_{1-i}$ is still either exiting or thinking when $q.s_i$ executes Action $\mathcal{S}_a$.

[3]Note that if p crashes during $q.s_i$'s eating session, then $q.s_i$ may eat forever. This, however, does not affect the correctness of the algorithm. A more detailed discussion follows in Section 8.

Since exiting is finite in $\mathcal{DX}_{1-i}$, eventually $q.s_{1-i}$ is thinking. Since $q.s_i$ cannot exit until $q.s_{1-i}$ is eating (from Action $\mathcal{S}_x$), eventually $q.s_{1-i}$ is thinking and $\mathsf{trigger} = 1 - i$ in $q.s_i$'s current eating session.

However, when $q.s_{1-i}$ is thinking and $\mathsf{trigger} = 1 - i$, Action $\mathcal{S}_h$ is enabled at $q.s_{1-i}$. Therefore, $q.s_{1-i}$ eventually becomes hungry in $q.s_i$'s current eating session. From Lemma 1 we know that $q.s_{1-i}$ eventually eats. When $q.s_{1-i}$ transits to eating, Action $\mathcal{S}_x$ is enabled at $q.s_i$. Therefore, $q.s_i$ eventually exits eating. □

LEMMA 7. *If p and q are correct, then $q.s_0$ and $q.s_1$ eat infinitely often.*

PROOF. Let k be the number of times a subject $q.s_i$ eats in a given execution. We prove the above lemma by induction on k.

Base Case: Subjects $q.s_0$ and $q.s_1$ eat at least once (here $k = 1$).

Initially, subjects $q.s_0$ and $q.s_1$ are both thinking, and $\mathsf{trigger} = 0$. Therefore, initially Action $\mathcal{S}_h$ is enabled at $q.s_0$. Eventually $q.s_0$ executes Action $\mathcal{S}_h$ and $q.s_0$ becomes hungry. From Lemma 1, $q.s_0$ eventually eats.

From Lemma 6 we know that $q.s_0$ eventually exits. However, from Action $\mathcal{S}_x$ we know that when $q.s_0$ exits, $q.s_1$ is eating. Therefore, eventually $q.s_1$ eats.

Inductive hypothesis: Let $q.s_i$ eat k times.

We now show that if $q.s_i$ eats k times, then $q.s_i$ eats $k+1$ times. Let $q.s_i$ be in its k^{th} eating session. From Lemma 6 we know that $q.s_i$ eventually exits its k^{th} eating session. However from Action $\mathcal{S}_x$ we know that when $q.s_i$ exits eating, $q.s_{1-i}$ is eating.

From Lemma 6 we know that $q.s_{1-i}$ exits its eating session. But from Action $\mathcal{S}_x$ we know that when $q.s_{1-i}$ exits eating, $q.s_i$ is eating. Since we already know that $q.s_i$ exits its k^{th} during $q.s_{1-i}$'s current eating session, it follows that $q.s_i$ must be in its $k + 1^{st}$ eating session when $q.s_{1-i}$ exits its current eating session.

Thus, by induction, it follows that subjects $q.s_0$ and $q.s_1$ eat infinitely often □

LEMMA 8. *If q is correct, then eventually, at any given time, some subject at q is eating. In other words, the following suffix invariant holds:* $(s_1.\mathsf{state} = \mathsf{eating}) \vee (s_0.\mathsf{state} = \mathsf{eating})$.

PROOF. Let $E \equiv (s_i.\mathsf{state} = \mathsf{eating}) \vee (s_i.\mathsf{state} = \mathsf{eating})$. From Lemma 7 we know that subjects $q.s_0$ and $q.s_1$ eat infinitely often. Therefore, E must be true infinitely often.

Let E be true at some time t_E. We now show that no action in Alg. 2 falsifies E.

Action $\mathcal{S}_h$. If Action $\mathcal{S}_h$ is enabled at $q.s_i$ at time t_E, then $s_{1-i}.\mathsf{state} = \mathsf{eating}$ at t_E (because E is true at time t_E). Executing $\mathcal{S}_h$ at $q.s_i$ does not change the state of $q.s_{1-i}$, hence E is true after $q.s_i$ executes Action $\mathcal{S}_h$.

Actions $\mathcal{S}_p$ and $\mathcal{S}_a$. These actions do not change the states of $q.s_0$ or $q.s_1$. Therefore, if E is true before either action is executed, then it is true after that action is executed is execute as well.

Action $\mathcal{S}_x$. If Action $\mathcal{S}_x$ is enabled at $q.s_i$, then $q.s_{1-i}$ is eating at time t_E. Action $\mathcal{S}_x$ at $q.s_i$ does not change the state of $q.s_{1-i}$. Therefore, after Action $\mathcal{S}_x$ is executed at $q.s_i$, $s_{1-i}.\mathsf{state} = \mathsf{eating}$. Hence, E is true after executing Action $\mathcal{S}_x$.

Thus it is shown that eventually E becomes true, and after it becomes true it is never falsified. □

LEMMA 9. *At any given time t, some witness at p is thinking. In other words, the following invariant must hold:* $(w_0.\mathsf{state} = \mathsf{thinking}) \vee (w_1.\mathsf{state} = \mathsf{thinking})$.

PROOF. At time $t = 0$, both witnesses are thinking. Therefore, $((w_0.\mathsf{state} = \mathsf{thinking}) \vee (w_1.\mathsf{state} = \mathsf{thinking}))$ is *true*, and the invariant holds at time $t = 0$. Let $((w_0.\mathsf{state} = \mathsf{thinking}) \vee (w_1.\mathsf{state} = \mathsf{thinking}))$ be *true* at some time $t' \geq t$. We then show that a witness $p.w_i$ executing any action in Alg. 1 maintains the above invariant.

Action $\mathcal{W}_h$. For Action $\mathcal{W}_h$ to be enabled, $p.w_{1-i}$ must be thinking, and Action $\mathcal{W}_h$ does not change the value of $w_{1-i}.\mathsf{state}$. Hence, the invariant holds.

Action $\mathcal{W}_x$. If $p.w_i$ executes Action $\mathcal{W}_x$ at time t', then $w_i.\mathsf{state} = \mathsf{eating}$ (guard at Action $\mathcal{W}_x$) at t'. Since the invariant is assumed to hold at t', this implies $w_{1-i}.\mathsf{state} = \mathsf{thinking}$. Since executing $\mathcal{W}_x$ does not change the value of $w_{1-i}.\mathsf{state}$, the invariant holds.

Action $\mathcal{W}_p$. Executing Action $\mathcal{W}_p$ does not change the values of $w_0.\mathsf{state}$ and $w_i.\mathsf{state}$. Since the invariant holds at t' before executing Action $\mathcal{W}_p$, the invariant holds after Action $\mathcal{W}_p$ as well.

Thus shown that the invariant $((w_0.\mathsf{state} = \mathsf{thinking}) \vee (w_1.\mathsf{state} = \mathsf{thinking}))$ holds. □

LEMMA 10. *If p is correct, and a witness thread $p.w_i$ in p eats at time t_{ie}, then the other witness thread $p.w_{1-i}$ eats at time $t_{(1-i)e} > t_{ie}$.*

PROOF. Let witness $p.w_i$ eat at time t_{ie}. This enables Action $\mathcal{W}_x$ at $p.w_i$, and $p.w_i$ eventually executes $\mathcal{W}_x$. Action $\mathcal{W}_x$ this sets switch to $1 - i$, and $w_i.\mathsf{state}$ to $\mathsf{exiting}$. From Lemma 9, it follows that when w_i executes Action $\mathcal{W}_x$, $w_{1-i}.\mathsf{state} = \mathsf{thinking}$.

Exiting is finite in $\mathcal{DX}_i$, so eventually $w_i.\mathsf{state} = \mathsf{thinking}$. This enables Action $\mathcal{W}_h$ at witness w_{1-i} (because $\mathsf{switch} = 1 - i$ and $w_{1-i}.\mathsf{state} = \mathsf{thinking}$). Let w_{1-i} execute Action $\mathcal{W}_h$, and $w_{1-i}.\mathsf{state} = \mathit{hungry}$. From Lemma 6 we know that if the peer subject $q.s_{1-i}$ in $\mathcal{DX}_{1-i}$ is correct, then it eats for finite time, otherwise it crashes in finite time. Therefore from the wait-freedom property of $\mathcal{DX}_{1-i}$, we know that w_{1-i} eventually eats. Let such time be $t_{(1-i)e} > t_{ie}$. □

LEMMA 11. *If p is correct, then witnesses $p.w_{i\in\{0,1\}}$ eat infinitely often.*

PROOF. Initially $\mathsf{switch} = 0$, $w_0.\mathsf{state} = \mathsf{thinking}$, and $w_1.\mathsf{state} = \mathsf{thinking}$ in Alg. 1. This enables Action $\mathcal{W}_h$ at witness $p.w_0$. Upon executing Action $\mathcal{W}_h$, and $p.w_0.\mathsf{state} = \mathsf{hungry}$. From Lemma 6, we know that the peer subject $q.s_0$ in $\mathcal{DX}_0$ eats only for finite time. Therefore, by the wait-freedom property of $\mathcal{DX}_0$, $p.w_0$ eventually eats at time (say) t_1.

Applying Lemma 10 to $p.w_0$, we know that $p.w_1$ eats at some time $t_2 > t_1$. Applying Lemma 10 to $p.w_1$, we know that $p.w_0$ eats at some time $t_3 > t_2 > t_1$, and so on.

Thus successive invocations of Lemma 10 show that $p.w_0$ and $p.w_1$ eat infinitely often. □

LEMMA 12. *If p is correct, then between every consecutive pair of witness $p.w_i$'s eating sessions, witness $p.w_{1-i}$ eats exactly once.*

PROOF. From Lemma 11, we know that $p.w_i$ eats infinitely often. Consider $p.w_i$'s two consecutive hungry-eating sessions during the intervals $[t_{h1}, t_{x1}]$ and $[t_{h2}, t_{x2}]$. From

Lemma 9, we know that during $p.w_i$'s hungry-eating session, $p.w_{1-i}$ is thinking. From Action $\mathcal{W}_h$, we know that $\mathsf{switch} = i$ at t_{h1} and t_{h2}. However, from Action $\mathcal{W}_x$, we know that $p.w_i$ sets switch to $1-i$ at time t_{x1}. Therefore, switch is set to i during the interval (t_{x1}, t_{h2}). But the only action that sets switch to i is Action $\mathcal{S}_x$ at $p.w_{1-i}$ which is enabled only when $p.w_{1-i}$ is eating. In other words, $p.w_{1-i}$ eats at least once between $p.w_i$'s eating sessions.

By applying the above argument for both witnesses, we see that between every consecutive pair of witness $p.w_i$'s eating sessions, witness $p.w_{1-i}$ eats exactly once. □

Strong Completeness.

THEOREM 1. *The action systems in Alg. 1 and 2 satisfy strong completeness.*

PROOF. Consider a run of the action systems in Alg. 1 and 2 where a correct process p monitors a faulty process q. Since q crashes in finite time, q's subject threads send only finitely many ping messages to p. After q crashes, by wait-freedom of $\mathcal{DX}_i$, we know that p's witness threads eat infinitely often. From Action $\mathcal{W}_x$ in Alg. 1 we know that every time $p.w_i$ eats, it sets $\mathsf{havePing}_i$ to $false$. Therefore, when $p.w_i$ is scheduled for the first time after all ping messages from $q.s_i$ have been received, it sets $\mathsf{havePing}_i$ to $false$. In all subsequent eating sessions of $p.w_i$, $\mathsf{havePing}_i$ is always $false$ (because $p.w_i$ does not receive any more ping messages from $q.s_i$). Hence, $p.w_i$ permanently suspects q in Action $\mathcal{W}_x$ in Alg. 1. In other words, every crash is eventually and permanently suspected by all correct processes. □

Eventual Strong Accuracy. In order to prove eventual strong accuracy, we need to show that if p and q are correct, then eventually p never suspects q. This can happen only if eventually every time any witness $p.w_i$ executes Action $\mathcal{W}_x$, $\mathsf{havePing}_i$ is $true$. We demonstrate that as follows:

THEOREM 2. *The action systems in Alg. 1 and 2 satisfy eventual strong accuracy.*

PROOF. Consider a run of the action systems in Alg. 1 and 2 where a correct process p monitors a correct process q. Let the dining instances $\mathcal{DX}_0$ and $\mathcal{DX}_1$ stop scheduling live neighbors to eat concurrently after time t_{wx}. Let t_{stable} be the time after which the suffix invariant $(s_i.\mathsf{state} = \mathsf{eating}) \vee (s_i.\mathsf{state} = \mathsf{eating})$ from Lemma 8 holds. Consider the time $t_{start} = max(t_{wx}, t_{stable})$.

From Lemma 11, we know that the witness threads $p.w_0$ and $p.w_1$ eat infinitely often. Consider a time $t_{wi.eat} > t_{start}$ such that $p.w_0$ and $p.w_1$ have completed at least one eating session each since t_{start}, and some witness thread (say) $p.w_i$ is in its z^{th} eating session. By construction, $p.w_i$'s $z-1^{st}$ eating session started after t_{start}.

From Lemma 12 we know that $p.w_{1-i}$ eats exactly once between the $z-1^{st}$ and z^{th} eating sessions of $p.w_i$. Let that be the y^{th} eating session of $p.w_{1-i}$. By construction (and Lemma 12) we know that $w_i.\mathsf{state} = \mathsf{thinking}$ during the y^{th} eating session of $p.w_{1-i}$, and from Lemma 8 (and the $\diamond\mathcal{WX}$ of $\mathcal{DX}_i$ and $\mathcal{DX}_{1-i}$), we know $s_i.\mathsf{state} = \mathsf{eating}$. However, from Lemmas 3 and 5 we know that during $q.s_i$'s current eating session, $p.w_i$ executes Action $\mathcal{W}_p$ exactly once. So, $\mathsf{havePing}_i = true$ when $q.s_i$ exits its current eating session (because while $q.s_i$ is eating, $p.w_i$ cannot start eating, and hence cannot set $\mathsf{havePing}$ to $false$). Therefore, when $p.w_i$ starts its z^{th} eating session, $\mathsf{havePing} = true$; hence, when $p.w_i$ executes Action $\mathcal{W}_x$ in its z^{th} session, it trusts q.

The above argument applies to any i and z after t_{start}, it follows that eventually, when either witness $p.w_0$ or $p.w_1$ is scheduled to eat, it will trust q. From Lemma 11 we know that $p.w_0$ and $p.w_1$ eat infinitely often. Therefore, p eventually and permanently stops suspecting q. □

8. DISCUSSION

Weakest failure detector for wait-free dining. We have shown that $\diamond\mathcal{P}$ is necessary to solve $\mathcal{WF}$-$\diamond\mathcal{WX}$. In conjunction with earlier results demonstrating the sufficiency of $\diamond\mathcal{P}$ [12, 13], our result shows that the synchronism and temporal system properties encapsulated by $\mathcal{WF}$-$\diamond\mathcal{WX}$ are equivalent to those encapsulated by $\diamond\mathcal{P}$.

Potentially infinite eating sessions. In Alg. 2, when a correct subject $q.s_i$ eats, it does not exit until receiving an ack from its peer witness $p.w_i$. However, if p is crashed (so is $p.w_i$), then $q.s_i$ never exits, because no ack will be received. Note that the underlying $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining layer guarantees wait-freedom and eventual weak exclusion only if correct diners eat for finite time. In the above instance, even though the subject is correct, it eats for infinite time. Therefore, the underlying $\mathcal{WF}$-$\diamond\mathcal{WX}$ dining layer need not satisfy its specifications in this instance.

So does this affect the correctness of the above solution? Actually *no*, it does not. Note that the crash-fault detection is performed exclusively by the witness threads. Therefore, the behavior of the subject threads is material only when they are being 'observed' by their peer witness threads. If the witness threads crash, then the subject threads are no longer being observed, and hence their behavior does not affect the algorithm's correctness.

Wait-free eventually k-fair dining. A recent result in [13] demonstrates that $\diamond\mathcal{P}$ is sufficient to solve wait-free dining for $\diamond\mathcal{WX}$ with *eventual k-fairness*. Eventual k-fairness guarantees that every run has an infinite suffix where no correct hungry process is overtaken by any live neighbor more than k times. Therefore, in conjunction with the result in [13], our result implies that wait-freedom and eventual weak exclusion encapsulate sufficient synchronism to guarantee eventual k-fairness in scheduling the eating sessions of hungry processes. It also implies the existence of an asynchronous transformation which can convert a wait-free eventually exclusive dining solution into a wait-free, eventually exclusive, and eventually k-fair dining solution.

9. ON PERPETUAL WEAK EXCLUSION

A 2005 paper by Delporte-Gallet et al [4] proves that a more powerful composition of the *Trusting* [4] and *Strong* [3] failure detectors ($\mathcal{T} + \mathcal{S}$) is sufficient to solve the problem of Fault-Tolerant Mutual Exclusion, which guarantees wait-freedom for *perpetual* weak exclusion ($\Box\mathcal{WX}$); that is, live neighbors *never* eat simultaneously. The trusting detector $\mathcal{T}$ satisfies the following properties: (1) *strong completeness*: $\mathcal{T}$ eventually and permanently suspects all crashed processes, and (2) *trusting accuracy*: (a) $\mathcal{T}$ eventually and permanently trusts every correct process, and (b) at all times, if $\mathcal{T}$ stops trusting any process q, then that process q must be crashed. The strong detector $\mathcal{S}$ satisfies: (1) *strong completeness:* (just like $\mathcal{T}$ above), and (2) *perpetual weak accuracy*: some correct process is never suspected by any live process.

The results in [4] prove that $\mathcal{T}$ is necessary and $\mathcal{T}+\mathcal{S}$ is sufficient to solve Fault-Tolerant Mutual Exclusion (FTME) in environments where arbitrarily many process may crash. It remains unknown, however, if $\mathcal{T}$ is sufficient or if $\mathcal{T}+\mathcal{S}$ is necessary for FTME. Our work has two points of relevance. First, our reduction can be applied to the case of perpetual weak exclusion — $\Box\mathcal{WX}$ — to produce an alternative proof of the necessity of $\mathcal{T}$ for FTME. Second, a slight modification of our reduction extracts an oracle more powerful than $\mathcal{T}$, thereby indicating that $\mathcal{T}$ alone is *not* sufficient for FTME. We sketch both results in the final paragraphs below.

Applying our reduction to $\Box\mathcal{WX}$ actually implements the trusting oracle $\mathcal{T}$ instead of $\Diamond\mathcal{P}$. Consider the following proof sketch. Let the $\mathcal{DX}_0$ and $\mathcal{DX}_1$ black-box solutions satisfy FTME (wait-freedom with *perpetual weak exclusion*). Initially, witnesses w_0 and w_1 can take turns eating while subjects s_0 and s_1 are still thinking or hungry. During this finite prefix, the subjects can (and will) be suspected by the witnesses continuously. By wait-freedom, however, subject s_0 will eventually eat. Once s_0 begins eating, witness w_0 cannot eat concurrently (due to perpetual weak exclusion). After s_0 eats for the first time, the subjects hand-off their eating sessions, thereby ensuring that when the witnesses transit to eating, the $\mathsf{havePing}_i$ variables are always true. Thus, the witnesses start trusting any live process q after the subject $q.s_0$ transits to eating the first time. If q is correct, then q will be trusted permanently. If q crashes, however, then by wait-freedom the witnesses will eat infinitely often and permanently suspect q. Thus, our reduction implements $\mathcal{T}$ if the black-box subroutine implements FTME.

Implementing $\mathcal{S}|_c$ using FTME requires only a simple modification of our reduction. Initially, each process q trusts all processes, and, for each neighbor p, process q initially schedules only its subject thread $q.s_0$ to become hungry (which occurs in FTME instance $\mathcal{DX}_0$). Process q does not schedule any witness threads to become hungry until *after* all of its $q.s_0$ subject threads have been scheduled to eat (that is, q has one subject eating per neighbor p).

This modification preserves strong completeness for $\mathcal{S}|_c$, because the wait-freedom property of FTME guarantees that (1) all correct hungry subjects will eat in finite time, and so (2) all correct witnesses will eventually be scheduled, thereby reducing to the completeness of the unmodified reduction. For the accuracy property of $\mathcal{S}|_c$, let q denote the first correct process to schedule any witness thread to become hungry. Suppose this event occurs at time t. No correct process can suspect q prior to time t, because all processes initially trust q, and no other correct process has scheduled any witnesses. Furthermore, no correct process can suspect q after time t. By hypothesis, each subject $q.s_0$ has already been scheduled to eat in its respective FTME instance $\mathcal{DX}_0$. The hand-off mechanism ensures that each $q.s_0$ continues eating until (1) its ping has been acknowledged, and (2) the corresponding subject $q.s_1$ begins eating as well. Between each hand-off, the $\Box\mathcal{WX}$ of FTME prevents neighboring witnesses from eating more than once, within which time a subsequent ping from q is always received. Consequently, q is never suspected by any correct process, which thereby ensures the accuracy property of $\mathcal{S}|_c$.

Applying the foregoing reductions independently, we can extract $(\mathcal{T}+\mathcal{S}|_c)$ from FTME. It remains to show that $\mathcal{T}$ alone cannot implement $\mathcal{S}|_c$. If not, then we can conclude that $\mathcal{T}$ alone is *not* sufficient to solve FTME.

Proof. Suppose there is a deterministic, asynchronous reduction to extract $\mathcal{S}|_c$ from $\mathcal{T}$. Consider three runs of $\mathcal{S}|_c$, where p is correct in run R_p (but q crashes initially), q is correct in run R_q (but p crashes initially), and both processes are correct in run R_c. In run R_p, let $\mathcal{T}$ at p always trust p and always suspect q. By strong completeness, $\mathcal{S}|_c$ at p permanently suspects q after some time t_p. In run R_q, let $\mathcal{T}$ at q always trust q and always suspect p. By strong completeness, $\mathcal{S}|_c$ at q permanently suspects p after some time t_q. Let time $t = max(t_p, t_q)$, and suppose all messages between p and q are delayed until after time t in run R_c. $\mathcal{T}$ can suspect correct processes for any finite prefix, so up through time t let ($\mathcal{T}$ at p in R_c) = ($\mathcal{T}$ at p in R_p), and let ($\mathcal{T}$ at q in R_c) = ($\mathcal{T}$ at q in R_q). For p and q, run R_c is indistinguishable up to time t from runs R_p and R_q, resp. Thus, at time t in R_c, $\mathcal{S}|_c$ at p suspects q and $\mathcal{S}|_c$ at q suspects p, which contradicts the accuracy property of $\mathcal{S}|_c$ that some correct process is never suspected by any correct process. We conclude that $\mathcal{T}$ cannot implement $\mathcal{S}|_c$, and so $\mathcal{T}$ cannot be the weakest failure detector for FTME. □

10. REFERENCES

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *15th Int'l Conf. on Distributed Computing (DISC)*, pp. 108–122. Springer, 2001.

[2] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[3] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput.*, 65(4):492–505, 2005.

[5] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, Oct 1971.

[6] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[7] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[8] Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, April 2008.

[9] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[10] Nancy A. Lynch. Fast allocation of nearby resources in a distributed system. In *12th ACM Symp. on Theory of Computing (STOC)*, pp. 70–81, 1980.

[11] Scott M. Pike and Paolo A.G. Sivilotti. Dining philosophers with crash locality 1. In *24th IEEE Int'l Conf. on Dist. Comp. Sys. (ICDCS)*, pp. 22–29, 2004.

[12] Scott M. Pike, Yantao Song, and Srikanth Sastry. Wait-free dining under eventual weak exclusion. In *9th Int'l Conf. on Distributed Computing and Networking (ICDCN)*, pp. 135–146. Springer, 2008.

[13] Yantao Song and Scott M. Pike. Eventually k-bounded wait-free distributed daemons. In *37th IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN)*, pp. 645–655, 2007.