Wait-Free Dining Under Eventual Weak Exclusion

Scott M. Pike[*], Yantao Song, and Srikanth Sastry

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

Abstract. We present a wait-free solution to the generalized dining philosophers problem under eventual weak exclusion in environments subject to crash faults. Wait-free dining guarantees that every correct hungry process eventually eats, regardless of process crashes. Eventual weak exclusion ($\diamond\mathcal{WX}$) actually allows scheduling mistakes, whereby mutual exclusion may be violated finitely-many times; for each run, however, there must exist a convergence point after which *live* neighbors never eat simultaneously. Wait-free dining under $\diamond\mathcal{WX}$ is particularly useful for synchronization tasks where eventual safety is sufficient for correctness (e.g., duty-cycle scheduling, self-stabilizing daemons, and contention managers). Unfortunately, wait-free dining is unsolvable in asynchronous systems. As such, we characterize sufficient conditions for solvability under partial synchrony by presenting a wait-free dining algorithm for $\diamond\mathcal{WX}$ using a local refinement of the eventually perfect failure detector $\diamond\mathcal{P}_1$.

Keywords: Dining Philosophers, Failure Detectors, Wait-Freedom.

1 Introduction

The dining philosophers problem (or *dining* for short) is a fundamental scheduling paradigm in which processes (called *diners*) periodically require exclusive access to a fixed subset of shared resources [1,2]. Each diner is either *thinking*, *hungry*, or *eating*. These states correspond to three basic phases of computation: executing independently, requesting resources, and utilizing shared resources in a critical section, respectively. Potential scheduling conflicts are modeled by a conflict graph in which diners with overlapping resource requirements are connected as neighbors. As such, dining is a generalization of the mutual exclusion problem, which corresponds to the special case where the conflict graph forms a clique.

Wait-free dining guarantees that every correct hungry process eventually eats, even if other processes fault by crashing. The solvability of wait-free dining depends on two primary factors: (1) the degree to which concurrency is restricted among eating diners, and (2) the degree to which crash faults can be detected reliably. The former depends on the applicable safety specification for local exclusion, while the latter depends on the degree of synchrony in the system.

[*] This work was supported by the Advanced Research Program of the Texas Higher Education Coordinating Board under Project Number 000512-0007-2006.

Safety specifications restrict concurrency among eating diners. For example, *strong exclusion* prohibits any pair of conflicting neighbors from eating simultaneously, even if one of them has crashed. This safety property models resources that can be permanently corrupted by process crashes. Unfortunately, wait-free dining under strong exclusion is vacuously unsolvable. To see why, consider any diner that crashes while eating. Wait-freedom guarantees that every correct hungry neighbor will eventually eat, but strong exclusion prohibits the same. Moreover, this result is independent of whether crashes can be detected reliably.

A less restrictive model called *weak exclusion* prohibits only *live* neighbors from eating simultaneously. This safety property models resources that are recoverable or eventually stateless after crash faults. For example, consider a wireless network where diners broadcast messages over a subset of shared frequencies. If some diner crashes while eating, then the current transmission terminates. As such, the frequency allocated to the crashed diner becomes available for subsequent use by neighboring diners.

Wait-free dining for weak exclusion is actually solvable, but it requires substantial timing assumptions, or, alternatively, access to sufficiently powerful failure detectors. A failure detector can be viewed as a distributed system service that can be queried like an oracle for information about process crashes [3]. Oracle-based algorithms are decoupled from the underlying timing assumptions about partial or even full synchrony necessary to implement such fault-detection capabilities in practice. Recent results on fault-tolerant mutual exclusion indicate that wait-free dining under weak exclusion is solvable in systems augmented with Trusting failure detectors — a relatively powerful class of oracles that can reliably detect certain crashes [4].

Unfortunately, trusting oracles require significant assumptions about network timing parameters to be implemented in practice. By contrast, less powerful oracles that are implementable in more practical models of partial synchrony — such as those for solving fault-tolerant consensus — are too weak to solve wait-free dining under weak exclusion. This problem remains unsolvable even for oracles of intermediate strength. For example, the eventually perfect failure detector $\diamond\mathcal{P}$ always suspects crashed processes, and eventually stops suspecting correct processes [3]. This oracle, which can make finitely many false-positive mistakes in any run, is more than sufficient to solve fault-tolerance consensus. Still, no $\diamond\mathcal{P}$-based algorithm can solve wait-free dining for weak exclusion; neighbors of any crashed diner will always be able to starve [5].

Our contribution examines a practical model of exclusion for wait-free dining which is solvable under modest assumptions of partial synchrony. In particular, we explore dining under *eventual weak exclusion*, and show that it is solvable using the aforementioned oracle $\diamond\mathcal{P}$. Eventual weak exclusion (abbreviated $\diamond\mathcal{WX}$ hereafter) permits finitely-many scheduling mistakes whereby conflicting diners eat together. *For each run, however, there exists a time after which no two live neighbors eat simultaneously.*

The time to convergence may be unknown, and it may also vary from run to run. Nevertheless, $\diamond\mathcal{WX}$ is sufficiently powerful to serve as a useful scheduling abstraction. For example, $\diamond\mathcal{WX}$ models recoverable resources where sharing violations precipitate at worst repairable (transient) faults. $\diamond\mathcal{WX}$ has received considerable attention recently in the context of shared-memory contention management [6], conflict managers for self-stabilizing systems [7], as well as wait-free eventually fair distributed daemons[8].

2 Background and Technical Framework

Although originally proposed by Dijkstra for a ring topology [1], dining philosophers was later generalized by Lynch for overlapping local exclusion problems on arbitrary graphs [2]. 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 resource conflicts between neighbors p and q.

Each diner is either *thinking*, *hungry*, or *eating*, but initially all diners are thinking. Diners may think forever, but they can also become hungry at any time. By contrast, eating is always finite (but not necessarily bounded). Hungry neighbors are said to be in *conflict*, because they compete for shared but 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:

Safety: Every run has an infinite suffix where no two live neighbors eat simultaneously.

Progress: Every correct hungry diner eventually eats, regardless of process crash faults.

Progress ensures fairness among hungry diners. In particular, dining solutions are not permitted to *starve* hungry processes by never scheduling them to eat. In the presence of crash faults, a dining algorithm that satisfies progress is called *wait-free* [9]. The safety requirement of eventual weak exclusion permits finitely many scheduling mistakes. A mistake occurs when two live neighbors are scheduled to eat simultaneously.

Computational Model. We consider asynchronous environments where message delay, clock drift, and relative process speeds are unbounded. A system is modeled by a set of n distributed processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ which communicate only by asynchronous message passing. We assume that the dining conflict graph is a subgraph of the communication graph, so that each pair of neighboring diners is connected by reliable FIFO channels.

Fault Patterns. Processes may fault only by crashing. A crash fault occurs when a process ceases execution (without warning) and never recovers [10]. 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. We say that *p is faulty in F* if $p \in F(t)$ for some time t; otherwise, we say that *p is correct in F*. Additionally, a process p is *live at time t* if p has not crashed by time t. That is, $p \notin F(t)$. Thus, correct processes are always live, but 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 crashes in Π. Each process has access to its own local detector module that outputs the set of processes currently suspected of having crashed. Unreliable failure detectors are characterized by the kinds of *mistakes* they can make. Mistakes can include false-negatives (*i.e.*, not suspecting a crashed process), as well as false-positives (*i.e.*, wrongfully suspecting a correct process). In Chandra and Toueg's original definition [3], each oracle class is defined by two properties: *completeness* and *accuracy*. Completeness restricts false negatives, while accuracy restricts false positives. More precisely, each oracle class is a function (defined by the intersection of a completeness property and an accuracy property), which maps each possible fault pattern to a set of admissible histories.

Our wait-free dining algorithm is based on the eventually perfect failure detector $\diamond\mathcal{P}$ from the original Chandra-Toueg hierarchy [3]. Informally, $\diamond\mathcal{P}$ is a convergent oracle that always suspects crashed processes and eventually stops suspecting correct processes. As such, $\diamond\mathcal{P}$ may commit finitely-many false positive mistakes during any run before converging to an infinite suffix during which the oracle provides reliable information about process crashes. Unfortunately, the time to convergence is not known and it may vary from run to run.

As originally defined, the scope of $\diamond\mathcal{P}$ is global, insofar as it provides information about all processes. One drawback of global oracles is that communication overhead can limit their practicality for large-scale networks. Accordingly, scope-restricted oracles have been proposed that provide information only about subsets of processes [11,12,13]. Our dining solution uses a variant of $\diamond\mathcal{P}$ defined in [14,15] for which suspect information is only provided about immediate neighbors. This local refinement, called $\diamond\mathcal{P}_1$, satisfies the following completeness and accuracy properties:

Local Strong Completeness — Every crashed process is eventually and permanently suspected by all correct neighbors.

Local Eventual Strong Accuracy — For every run, there exists a time after which no correct process is suspected by any correct neighbor.

It is worth noting that $\diamond\mathcal{P}$ cannot be implemented in purely asynchronous systems. Implementations typically use adaptive time-outs based on modest assumptions about partial synchrony. A simple technique assumes that upper bounds on message delay and relative process speed exist, but are unknown. Such bounds can be adaptively estimated by ping-ack protocols which increase a time-out threshold after each false positive. After finitely-many mistakes, the current time-out will exceed the unknown round-trip message time, after which false positives desist.

There are known implementations of $\diamond\mathcal{P}$ in several other models partial synchrony as well [3,16,17,18]. The common advantage is that $\diamond\mathcal{P}$-based algorithms are decoupled from explicit commitments to underlying detection mechanisms and/or specific timing parameters. Additionally, the local refinement $\diamond\mathcal{P}_1$ can also be implemented efficiently in sparse, large-scale, and even partitionable networks [15].

3 A Wait-Free Dining Algorithm for $\diamond\mathcal{WX}$

Our solution is based on the classic hygienic dining algorithm [19]. In hygienic dining, a unique fork is associated with each edge in the conflict graph. A hungry process must collect and hold all shared forks to eat. This provides a simple basis for safety, since at most one diner can hold a given fork at any time. Fork conflicts are resolved according to a dynamic partial ordering on process priority. After eating, diners reduce priority below all neighbors; this ensures progress by yielding to previously lower-priority diners.

It is easy to see why hygienic dining is not wait-free. Without fault detection, hungry processes starve whenever missing forks are lost to crashed neighbors. The result is actually much worse: if no process thinks forever, then the crash of any eating diner will eventually precipitate global starvation among all processes (not just neighbors).

In our solution, suspicion by $\diamond\mathcal{P}_1$ serves as a proxy for permanently missing forks. The completeness property guarantees that every crashed process will be eventually and permanently suspected by all correct neighbors. As such, hungry neighbors of crashed diners can avoid starvation by using suspicion as a proxy for permanently missing forks. Specifically, a hungry diner i can eat if, for every neighbor j, either i holds the fork shared with j, or the $\diamond\mathcal{P}_1$ oracle at i suspects j.

Unfortunately, suspicion by $\diamond\mathcal{P}_1$ is an unreliable proxy for missing forks, because the eventual accuracy property also allows false-positive mistakes. For example, if live neighbors falsely suspect each other, they may proceed to eat simultaneously, regardless of the fork. Ideally, scheduling violations should be limited by the finite number of false-positive mistakes per run. It remains to show, however, that $\diamond\mathcal{WX}$ will still be satisfied after $\diamond\mathcal{P}_1$ converges.

A deeper subtlety is the impact of oracular mistakes on maintaining a consistent ordering of process priorities. In hygienic dining, relative process priorities are typically encoded directly in the fork variables. As such, it becomes trivial for diners to reduce their priority below all neighbors after eating, because (1) diners must hold every shared fork while eating, so (2) the current priority of every neighbor is actually known.

The same technique does not work with $\diamond\mathcal{P}_1$, because false-positive mistakes may enable diners to eat despite missing critical forks. In the worst case, *two neighbors can eat simultaneously even if neither holds the fork*. This can occur if the fork is in transit, but both diners begin eating as the result of mutual suspicion. If the fork is still in transit when both diners complete eating, then neither diner knows the actual priority ordering. Unlike hygienic dining, it is impossible for both diners to reduce their own priority below all neighbors; either one diner will not lower its priority sufficiently, or both priorities will match (which could lead to symmetries resulting in deadlock).

To circumvent this difficulty, we store process priorities explicitly at each diner, and assume unique identifiers to break symmetries. Additionally, we establish wait-free progress even though priorities are reduced by arbitrary values after eating.

3.1 Algorithm Variables

Our algorithm guarantees safety using forks plus the eventual strong accuracy of $\diamond\mathcal{P}_1$. It guarantees wait-free progress using a dynamic ordering on process priorities, plus the strong completeness of $\diamond\mathcal{P}_1$. In addition to the local oracle module, each process has the following local variables. A trivalent variable state_i denotes the current dining phase: thinking, hungry, or eating. Each process also has a local integer-valued variable height_i (which can grow negatively without bound), and a unique process identifier id_i. Taken together as an ordered pair, $(\mathsf{height}_i, \mathsf{id}_i)$ determines the $\mathsf{priority}_i$ of process i. Since process identifiers are unique, every pair of priorities, x and y, can be totally ordered lexicographically as follows:

$$x < y \ \stackrel{\text{def}}{=} (x.\mathsf{height} < y.\mathsf{height}) \ \vee ((x.\mathsf{height} = y.\mathsf{height}) \wedge (x.\mathsf{id} < y.\mathsf{id}))$$

To implement the forks, we introduce two local variables for each pair of neighbors. For process i, we associate a boolean variable fork_{ij} for each neighbor j. Symmetrically, each process j has a boolean variable fork_{ji} corresponding to neighbor i. We interpret these variables as follows: fork_{ij} is true iff process i holds the unique fork that it shares with neighbor j. Alternatively, fork_{ji} is true iff j holds the fork. When the fork is in transit from one neighbor to the other, both local variables are false. Since the fork is unique and exclusive, it is never the case that both variables are true.

In addition to the forks, we also introduce a request token between each pair of neighbors. In general, if process i holds a request token, but needs the corresponding fork from j, then i can request the missing fork by sending the request token to j. Request tokens are implemented and interpreted the same as forks. For process i, we associate a unique boolean variable token_{ij} for each neighbor j. Symmetrically, each process j has a boolean variable token_{ji} corresponding to neighbor i.

3.2 Algorithm Actions

A thinking process can become hungry at any time by executing Action 1 and selecting the corresponding alternative. Action 2 is always enabled while hungry. When executed, it requests every missing fork for which no previous request is currently pending. This is achieved by sending the request token to the corresponding neighbor, including the current priority of the requesting process. As a result, the local token variable becomes false to indicate that a request has been sent.

Action 3 handles fork requests. The requested fork must be sent immediately if the recipient is thinking, but also if the recipient is hungry but has lower priority than the requestor. Otherwise, the fork request is deferred until after eating. Deferred requests are represented by holding both the shared fork and the request token. Note that if a hungry process loses a requested fork to a higher-priority neighbor in Action 3, the relinquished fork will be re-requested by subsequently executing Action 2, which is always enabled while hungry.

Action 4 simply receives forks, and Action 5 determines when a hungry process can begin eating. A hungry process i can begin eating if, for each neighbor j, process i either holds the shared fork, or currently suspects j. This is the only action that utilizes the local oracle $\diamond\mathcal{P}_1$ and it is central to the wait-freedom of the algorithm.

Action 6 exits eating and transits back to thinking. This action reduces the priority of the diner, and sends forks for any requests that were previously deferred while hungry or eating. To reduce priority, Action 6 invokes a local procedure called $Lower$ which reduces only the $height$ component of the diner's priority by some positive integer. The magnitude of the reduction is up to the algorithm designer, and can be either statically fixed or dynamically chosen at runtime.

Action 6 isolates several subtleties. In hygienic dining, a process must reduce its priority below that of all neighbors after eating. This absolute reduction forms the basis for progress, because it forces high-priority diners to yield to lower-priority neighbors. In our algorithm, oracular mistakes may enable some diners to eat without knowing the priorities of all live neighbors. As such, hygienic reductions cannot be guaranteed. Our proof of progress shows that reducing priority by an arbitrary amount is sufficient, because it still reduces the number of times any diner can overtake its live neighbors.

```
Code for process $i$, with unique identifier $\mathsf{id}_i$ and local set of neighbors $N(i)$

var  $\mathsf{state}_i$    : $\{\mathsf{thinking},\ \mathsf{hungry},\ \mathsf{eating}\}$      init, $\mathsf{state}_i = \mathsf{thinking}$
     $\mathsf{height}_i$   : integer                                init, $\mathsf{height}_i = 0$
     $\mathsf{priority}_i$ : $(\mathsf{height}_i \times \text{process-id})$              init, $\mathsf{priority}_i = (0,\ \mathsf{id}_i)$
     $\mathsf{fork}_{ij}$    : boolean, for each $j \in N(i)$           init, $\mathsf{fork}_{ij} = (i > j)$
     $\mathsf{token}_{ij}$   : boolean, for each $j \in N(i)$           init, $\mathsf{token}_{ij} = (i < j)$
     $\Diamond\mathcal{P}_1$       : local eventually perfect detector      init, $\Diamond\mathcal{P}_1 \subseteq N(i)$

 1 : $\{\mathsf{state}_i = \mathsf{thinking}\} \longrightarrow$                                    Action 1
 2 :     $\mathsf{state}_i :=$ (thinking or hungry)                              Become Hungry

 3 : $\{\mathsf{state}_i = \mathsf{hungry}\} \longrightarrow$                                      Action 2
 4 :     $\forall j \in N(i)$ where $(\mathsf{token}_{ij} \ \wedge\ \neg\mathsf{fork}_{ij})$ do                Request Missing Forks
 5 :         send-request $\langle \mathsf{priority}_i \rangle$ to $j$
 6 :         $\mathsf{token}_{ij} := \mathsf{false}$

 7 : $\{$receive-request $\langle \mathsf{priority}_j \rangle$ from $j \in N(i)\} \longrightarrow$                Action 3
 8 :     $\mathsf{token}_{ij} := \mathsf{true}$                                                Send Fork or
 9 :     if $(\mathsf{state}_i = \mathsf{thinking} \vee (\mathsf{state}_i = \mathsf{hungry} \wedge (\mathsf{priority}_i < \mathsf{priority}_j)))$   Defer
10 :     then send-fork$\langle i \rangle$ to $j$
11 :         $\mathsf{fork}_{ij} := \mathsf{false}$

12 : $\{$receive-fork $\langle j \rangle$ from $j \in N(i)\} \longrightarrow$                          Action 4
13 :     $\mathsf{fork}_{ij} := \mathsf{true}$                                          Obtain Shared Fork

14 : $\{\mathsf{state}_i = \mathsf{hungry} \wedge (\forall j \in N(i) :: (\mathsf{fork}_{ij} \vee j \in \Diamond\mathcal{P}_1))\} \longrightarrow$     Action 5
15 :     $\mathsf{state}_i := \mathsf{eating}$                                      Enter Critical Section

16 : $\{\mathsf{state}_i = \mathsf{eating}\} \longrightarrow$                                       Action 6
17 :     $Lower(\mathsf{priority}_i)$                                          Exit Critical Section
18 :     $\mathsf{state}_i := \mathsf{thinking}$                                    Send Deferred Forks
19 :     $\forall j \in N(i)$ where $(\mathsf{token}_{ij} \wedge \mathsf{fork}_{ij})$ do
20 :         send-fork$\langle i \rangle$ to $j$
21 :         $\mathsf{fork}_{ij} := \mathsf{false}$

22 : procedure $Lower$ (p : priority)                                       Reduce Priority
23 :   ensures $\mathsf{p}' := Lower\ (\mathsf{p})$ where                            Process ID Unchanged
24 :         $(\mathsf{p}'.\mathsf{id} = \mathsf{p}.\mathsf{id})$ and $(\mathsf{p}'.\mathsf{height} < \mathsf{p}.\mathsf{height})$       Integer Height Lowered
```

Algorithm 1. 1. Wait-Free Dining under Eventual Weak Exclusion

4 Proof of Correctness

Lost tokens or forks can compromise progress, while duplicated tokens or forks can compromise safety. First we prove some basic lemmas which assert that each pair of live neighbors share a unique fork and a unique request token.

Lemma 1. *There exists exactly one token between each pair of live neighbors.*

Proof. For each pair of neighbors, the initialization code creates a unique token at the lower-priority process. Since communication channels are reliable, this token is neither lost nor duplicated while in transit. Only Actions 2 and 3 can modify the token variables. No token is lost, because every token received is locally stored (Action 3), and no token is locally removed unless it is sent (Action 2). No token is duplicated, because every token sent is locally removed, and no absent token is ever sent (Action 2). Thus, token uniqueness is preserved. □

Lemma 2.1. *There exists exactly one fork between each pair of live neighbors.*

Proof. For each pair of neighbors, the initialization code creates a unique fork at the higher-priority process. Since communication channels are reliable, this fork is neither lost nor duplicated while in transit. Only Actions 3, 4, and 6 modify the fork variables. No fork is lost, because every fork received is locally stored (Action 4), and no fork is locally removed unless it is sent (Actions 3 & 6). No fork is duplicated, because every fork sent is locally removed, *and no absent fork is ever sent** *(Action 3 & 6)*. Thus, fork uniqueness is preserved. □

*Action 3 can send forks (Line 11) without verifying their local presence. If such forks are absent, then this action could compromise $\Diamond\mathcal{WX}$ by duplicating forks. As it turns out, Action 3 is never enabled unless the requested fork is actually present. This result may not be obvious from the program text, because it depends explicitly on the assumption of FIFO channels. Consequently, we prove this assertion separately below.

Lemma 2.2. *Action 3 is never enabled unless the requested fork is present.*

Proof. Suppose for contradiction that Action 3 is enabled at some process i at time t_2, but that the requested fork is absent. This action can only be enabled by i receiving a request token from some neighbor j that executed Action 2 at an earlier time $t_1 < t_2$. The condition in Line 4 asserts that j held the token but not the shared fork at time t_1. Consequently, the fork was already at i or it was in transit at time t_1.

1. Suppose the fork was in transit from j to i. By FIFO channels, the fork had to arrive at i before the request token which enabled Action 3 at time t_2. Only Actions 3 and 6 send forks, but both require the fork and token to be co-located. Thus, the fork remains at i until Action 3 became enabled at time t_2.
2. Suppose the fork was in transit from i to j. Then i must have sent the fork by executing Action 3 or 6 at some earlier time $t_0 < t_1$. As mentioned above, the token must have been co-located with the fork at time t_0. Again, by FIFO channels, j could not execute Action 2 at time t_1, because the token could not have overtaken the fork which was still in transit. □

Theorem 1. *Algorithm 1 satisfies eventual weak exclusion $\Diamond\mathcal{WX}$. That is, for every execution there exists a time after which no two live neighbors eat simultaneously.*

Proof. The safety proof is by direct construction and uses the local eventually strong accuracy property of $\diamond\mathcal{P}_1$. This property guarantees that for each run there exists a time t after which no correct process is suspected by any correct neighbor.

We observe that faulty processes cannot prevent $\diamond\mathcal{WX}$ from being established. Since faulty processes are live for only a finite prefix before crashing, they can eat simultaneously with live neighbors only finitely many times in any run. Consequently, we can restrict our focus to correct processes only.

Consider any execution α of Algorithm 1. Let t denote the time in α after which $\diamond\mathcal{P}_1$ never suspects correct neighbors. Let i be any correct process that *begins* eating after time t. By Action 5, process i can only transit from hungry to eating if, for each neighbor j, either i holds the shared fork or i suspects j. Since $\diamond\mathcal{P}_1$ never suspects correct neighbors after time t in execution α, process i must hold every fork it shares with its correct neighbors in order to begin eating.

So long as i remains eating, Actions 3 and 6 guarantee that i will defer all fork requests. As such, p will not relinquish any forks while eating. From Lemma 2.1, we know that forks cannot be duplicated either. Furthermore, $\diamond\mathcal{P}_1$ has already converged in α, so no correct neighbor can suspect p. Thus, Action 5 remains disabled for every correct hungry neighbor of i until after i transits back to thinking. We conclude that no pair of correct neighbors can *begin*[1] overlapping eating sessions after time t. □

Next we introduce some definitions to construct a metric function for the progress proof. First, we measure the priority *distance* between any two processes i and j as:

$$dist(i,j) = \begin{cases} 0, & \text{if } (\mathsf{priority}_i < \mathsf{priority}_j) \\ \mathsf{height}_i - \mathsf{height}_j, & \text{if } (\mathsf{priority}_i > \mathsf{priority}_j) \wedge (\mathsf{id}_i < \mathsf{id}_j) \\ \mathsf{height}_i - \mathsf{height}_j + 1, & \text{if } (\mathsf{priority}_i > \mathsf{priority}_j) \wedge (\mathsf{id}_i > \mathsf{id}_j) \end{cases}$$

Suppose for any pair of processes i and j that $dist(i,j) = d$ in some configuration where j is hungry. While j remains hungry, $\mathsf{priority}_j$ remains unchanged. Also, recall from Action 6 that each process reduces the height component of its priority after eating. Consequently, d is an upper bound on the maximum number of times that process i can overtake process j before either j gets scheduled to eat or $\mathsf{priority}_i < \mathsf{priority}_j$.

Now we define a metric function $M : \Pi \rightarrow \mathbb{N}$ for each diner $j \in \Pi$ as follows:

$$M(j) = \sum_{i \neq j} dist(i,j)$$

First, we observe that M is bounded below by 0, and that $M(j) = 0$ iff j currently has the highest priority value among all processes in Π. In general, the value of $M(j)$ depends only on processes that are currently higher-priority than j. This is because

[1] As a technical point, diners might forestall $\diamond\mathcal{WX}$ by eating with neighbors that began eating *before* $\diamond\mathcal{P}_1$ converged. For example, consider neighbors i and j, where i holds the shared fork, but j began eating by falsely suspecting i before $\diamond\mathcal{P}_1$ converged. Since j is already eating, but i holds the shared fork, i might violate exclusion by eating with j even after the oracle has converged. This can happen multiple times, in fact, so long as j continues to eat. The phenomenon is temporary, however, because j is either faulty and crashes, or j is correct and must exit eating within finite time. Thereafter, i and j never eat simultaneously again.

$dist(i, j) = 0$ for any process i with $\mathsf{priority}_i < \mathsf{priority}_j$. If $M(j) = b$, then b is an upper bound on how many times *any* higher-priority process can eat before either j gets scheduled to eat or $\mathsf{priority}_j$ becomes globally maximal.

We also note that the metric value of each process in a given configuration is unique: $(i \neq j) \Rightarrow M(i) \neq M(j)$. Moreover, $M(i) < M(j) \Leftrightarrow (\mathsf{priority}_i > \mathsf{priority}_j)$. These properties follow from the fact that priorities are totally ordered.

Finally,the metric value $M(j)$ never increases while process j is thinking or hungry. $M(j)$ can only increase by reducing the height component of $\mathsf{priority}_j$ in Action 6 after eating. Importantly, *this change in relative priority actually causes the metric values of all other processes to decrease*.

We are now prepared to state and prove the following helper lemma for progress:

Lemma 3. Let C be a configuration where some correct process is hungry, and let H denote the set of all hungry processes in C. The correct process $j \in H$ with minimal metric eventually eats, or some correct process i with $M(i) < M(j)$ becomes hungry.

Proof. Let j be the unique correct hungry process with minimal metric value in H. In other words, j is the highest-priority correct hungry process in configuration C. Lemma 3 holds trivially if j eats or if any correct process i with $M(i) < M(j)$ becomes hungry. Otherwise, j remains the highest-priority correct hungry process forever. We will show that this latter case leads to a contradiction.

By definition, every faulty neighbor of j will crash within finite time. By the local strong completeness of $\diamond\mathcal{P}_1$, process j will permanently suspect such processes by some unknown time t. Thereafter, j must collect forks only from its correct neighbors.

First, j will not lose any such forks. By hypothesis, j is hungry and higher priority than any correct neighbor, so any fork request received by j in Action 3 will be deferred.

Second, j will eventually acquire every fork shared with its correct neighbors. By Lemma 1, j shares a unique request token with each such neighbor. For any missing fork, Action 2 guarantees that j will eventually send the corresponding token. Since j is higher priority than any correct neighbor, these fork requests must be honored unless the recipient is currently eating. In the latter case, the requested fork will be sent when the correct neighbor exits eating in Action 6.

We conclude that if j remains hungry indefinitely, then j eventually suspects each faulty neighbor and eventually holds the shared fork with each correct neighbor. By Line 14, the guard on Action 5 is enabled. So j eats and Lemma 3 is established. □

Theorem 2: *Algorithm 1 satisfies wait-free progress. That is, every correct hungry process eventually eats.*

Proof: We prove wait-freedom by complete (strong) induction on metric values.

Base Case: Let j be a correct hungry process with $M(j) = 0$.

By definition, the metric value $M(j)$ is minimal, so Lemma 3 applies to j. There are only two outcomes: either j eats, or some process i with $M(i) < M(j)$ becomes hungry. Since metric values are unique and bounded below by 0, no such process i exists. Consequently, j eventually eats. □

Inductive Hypothesis: Suppose for $k > 0$ that every correct hungry process i with $M(i) < k$ eventually eats. It remains to show that every correct hungry process j with $M(j) = k$ eventually eats as well.

Let C be a configuration, and let j be a correct hungry process in C with $M(j) = k$. Suppose that k is the minimal metric value among all correct hungry processes in C. Then Lemma 3 applies to j, so we conclude that j eventually eats, or some correct process i with $M(i) < M(j)$ becomes hungry. Alternatively, suppose that k is *not* the minimal metric value among all correct hungry processes in C. Then some correct hungry process i with $M(i) < k$ already exists.

Either way, we conclude that j eventually eats or the inductive hypothesis applies to some correct hungry process i with $M(i) < k$. In the latter case, process i eats. As a correct diner, i eventually stops eating by executing Action 6, which thereby lowers the height component of $\mathsf{priority}_i$ and decreases $dist(i, j)$ by at least 1. Recall that while j remains hungry, $M(j)$ does not increase. Thus, any decrease in $dist(i, j)$ will cause the metric value of $M(j)$ becomes less than k. Since j is now a correct hungry process with $M(j) < k$, the inductive hypothesis applies directly to j. We conclude that j eventually eats, and that Algorithm 1 satisfies wait-free progress by complete induction. □

5 Contributions

We have examined the dining philosophers problem under eventual weak exclusion in environments subject to permanent crash faults. Eventual weak exclusion ($\Diamond\mathcal{WX}$) permits conflicting diners to eat concurrently only finitely many times, but requires that, for each run, there exists a (potentially unknown) time after which *live* neighbors never eat simultaneously. This safety property models systems where resources are recoverable or where sharing violations precipitate only transient (repairable) faults. Applications of $\Diamond\mathcal{WX}$ include shared-memory contention management [6], conflict managers for self-stabilizing systems [7], and wait-free eventually fair daemons [8].

Dining under $\Diamond\mathcal{WX}$ is unsolvable in asynchronous environments, where crash faults can precipitate permanent starvation among live diners. The contribution of our work is a wait-free dining algorithm for $\Diamond\mathcal{WX}$ in partially synchronous environments which guarantees that every correct hungry process eventually eats, even in the presence of arbitrarily many crash faults. Our oracle-based solution uses a local refinement of the eventually perfect failure detector $\Diamond\mathcal{P}_1$. This oracle always suspects crashed neighbors, and eventually stops suspecting correct neighbors. $\Diamond\mathcal{P}_1$ provides information only about immediate neighbors, and, as such, it is fundamental to the scalability of our approach, since it is implementable in partially synchronous environments with sparse communication graphs that are partitionable by crash faults.

Our work demonstrates that $\Diamond\mathcal{P}_1$ is *sufficient* for wait-free dining under $\Diamond\mathcal{WX}$. It is an open question, however, whether this oracle is actually *necessary*. This question goes to the minimality of our assumptions and the portability of our solutions to weaker models of partial synchrony. On the one hand, wait-free dining under $\Diamond\mathcal{WX}$ is a harder problem than fault-tolerant consensus; the eventually strong oracle $\Diamond\mathcal{S}$ — which is sufficient for consensus [3] — is not sufficient for wait-free dining [20]. Thus, the search for a weakest failure detector is bounded above by $\Diamond\mathcal{P}_1$ and below by $\Diamond\mathcal{S}$.

References

1. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica 1, 115–138 (1971) Reprinted in Operating Systems Techniques, Hoare, C.A.R., Perrot, R.H. (eds.), Academic Press, pp. 72–93 (1972) (An earlier version appeared as EWD310)
2. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: STOC. Proceedings of the 12th ACM Symposium on Theory of Computing, pp. 70–81 (1980)
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43, 225–267 (1996)
4. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. J. Parallel Distrib. Comput. 65, 492–505 (2005)
5. Pike, S.M., Sivilotti, P.A.G.: Dining philosophers with crash locality 1. In: ICDCS. Proceedings of the 24th IEEE International Conference on Distributed Computing Systems, pp. 22–29. IEEE, Los Alamitos (2004)
6. Guerraoui, R., Kapałka, M., Kouznetsov, P.: The weakest failure detectors to boost obstruction-freedom. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 399–412. Springer, Heidelberg (2006)
7. Gradinariu, M., Tixeuil, S.: Conflict managers for self-stabilization without fairness assumption. In: ICDCS. 27th International Conference on Distributed Computing Systems, pp. 46–53. IEEE, Los Alamitos (2007)
8. Song, Y., Pike, S.M.: Eventually k-bounded wait-free distributed daemons. In: DSN. 37th International Conference on Dependable Systems and Networks, pp. 645–655. IEEE, Los Alamitos (2007)
9. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. (TOPLAS) 13, 124–149 (1991)
10. Cristian, F.: Understanding fault-tolerant distributed systems. Comm. ACM 34, 56–78 (1991)
11. Anceaume, E., Fernández, A., Mostéfaoui, A., Neiger, G., Raynal, M.: A necessary and sufficient condition for transforming limited accuracy failure detectors. J. Comput. Syst. Sci. 68, 123–133 (2004)
12. Guerraoui, R., Schiper, A.: Γ–accurate failure detectors. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 269–286. Springer, Heidelberg (1996)
13. Raynal, M., Tronel, F.: Restricted failure detectors: Definition and reduction protocols. Information Processing Letters 72, 91–97 (1999)
14. Beauquier, J., Kekkonen-Moneta, S.: Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing failure detectors. International Journal of Systems Science 28, 1177–1187 (1997)
15. Hutle, M., Widder, J.: Self-stabilizing failure detector algorithms. In: Fahringer, T., Hamza, M.H. (eds.) PDCN. Parallel and Distributed Computing and Networks, IASTED/ACTA Press, pp. 485–490 (2005)
16. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM 35, 288–323 (1988)
17. Fetzer, C., Schmid, U., Süsskraut, M.: On the possibility of consensus in asynchronous systems with finite average response times. In: ICDCS. 25th International Conference on Distributed Computing System, pp. 271–280. IEEE, Los Alamitos (2005)
18. Sastry, S., Pike, S.M.: Eventually perfect failure detectors using ADD channels. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 483–496. Springer, Heidelberg (2007)
19. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 6, 632–646 (1984)
20. Pike, S.M.: Distributed Resource Allocation with Scalable Crash Containment. PhD thesis, The Ohio State University, Department of Computer Science & Engineering (2004)