Wait-Free Stabilizing Dining Using Regular Registers[*]

Srikanth Sastry[1**], Jennifer Welch[2***], and Josef Widder[3 †]

[1] CSAIL, MIT
Cambridge, MA - 02139, USA
[2] Texas A&M University
College Station, TX - 77843, USA
[3] Technische Universität Wien
Vienna, Austria

Abstract. Dining philosophers is a scheduling paradigm that determines when processes in a distributed system should execute certain sections of their code so that processes do not execute 'conflicting' code sections concurrently, for some application-dependent notion of a 'conflict'. Designing a stabilizing dining algorithm for shared-memory systems subject to process crashes presents an interesting challenge: classic stabilization relies on *all* processes continuing to execute actions forever, an assumption which is violated when crash failures are considered. We present a dining algorithm that is both wait-free (tolerates any number of crashes) and is pseudo-stabilizing. Our algorithm works in an asynchronous system in which processes communicate via shared regular registers and have access to the eventually perfect failure detector $\Diamond\mathcal{P}$. Furthermore, with a stronger failure detector, the solution becomes wait-free and self-stabilizing. To our knowledge, this is the first such algorithm. Prior results show that $\Diamond\mathcal{P}$ is necessary for wait-freedom.

1 Introduction

In shared-memory distributed systems, the code for a distributed application at each process is a sequence of actions, certain sections of which — called *critical sections* — are designated as needing to be executed indivisibly with respect to the critical sections of application modules at certain other processes. Actions at different processes might conflict, for instance, because they access a shared resource or because the relative order of their execution results in a race condition. Synchronizing such actions of distributed applications, that need to

[*] We would like to thank the reviewers for their suggestions in improving the paper.

[**] This work is supported in part by NSF Award Numbers CCF-0726514, CCF-0937274, and CNS-1035199, and AFOSR Award Number FA9550-08-1-0159. This work is also partially supported by Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

[***] Supported in part by NSF grant 0964696.

[†] Supported in part by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF), and by the Vienna Science and Technology Fund (WWTF) grant PROSEED.

occur without interference, is often delegated to scheduler that is implemented as a solution to the dining philosophers problem (or *dining*, for short).

Dining is a scheduling paradigm in which critical-section actions at each process may be in conflict with a static subset of processes in the system, and a solution to the dining philosophers problem ensures that whenever a process is executing its critical-section actions, no other conflicting process is executing its respective critical-section actions.

Solutions to dining are well understood in 'fault-free' systems, in which all the components behave according to their respective specification. However, in the presence of faults — deviations of any component from its specification — designing dining algorithms becomes challenging. In this paper, we focus on solving dining in the presence of two distinct fault classes: transient faults and crash faults. A *transient fault* occurs when the state of the system is corrupted to an arbitrary state, and a *crash fault* occurs when a process ceases execution without warning and never recovers.

Often, recovery from transient faults is achieved through *stabilization* [13], which is a property that guarantees that from any arbitrary state, the system is guaranteed to converge to a 'safe' state and operate henceforth in accordance with its specification. A classic assumption for correctness in stabilizing systems is that all processes continue executing actions to recover from an arbitrary state. However, a crashed process ceases executing actions. This makes the intersection of the two fault classes an interesting avenue for research.

Contribution. While existing solutions to dining are either stabilizing [1, 2, 5, 23] or wait-free (tolerate an arbitrary number of crashes) [28, 30, 9], to our knowledge, there are none that are both. In this paper, we propose a distributed dining algorithm that is pseudo-stabilizing and wait-free. The algorithm assumes the existence of multi-reader single-writer regular registers and the eventually perfect failure detector $\Diamond\mathcal{P}$. We remark that the above assumptions are modest or even necessary: wait-free multi-reader single-writer regular register can be constructed from wait-free dual-reader single-writer safe bits [17], and $\Diamond\mathcal{P}$ is necessary for wait-free dining [29] in asynchronous systems. We see that if the algorithm accesses the perfect failure detector, then it becomes wait-free and self-stabilizing.

Organization. Background and related work is presented in Sect. 2. Section 3 describes the system model. Section 4 specifies two problems: mutual exclusion and dining philosophers. Section 5 describes a wait-free pseudo-stabilizing mutual exclusion algorithm, which is then used in the wait-free pseudo-stabilizing dining algorithm described in Sect. 6. We conclude in Sect. 7.

2 Background and Related Work

Designing dining algorithms that are stabilizing or crash-tolerant, especially wait-free, or both, has been an active area of research.

Crash tolerant and stabilizing dining. There has been some prior investigations on constructing stabilizing dining algorithms that tolerate process crashes. For instance, [24] provides a stabilizing dining algorithm that tolerates (malicious) crashes, and in [26], the stabilizing dining algorithm tolerates byzantine processes. However, in these algorithms, some correct processes could stall even if one process crashes. In contrast, we focus on the stronger property of wait-freedom which guarantees that all correct processes continue taking steps despite arbitrary process crashes.

Self-stabilizing dining. There has been a significant body of work [23, 1, 5, 25] which investigates self-stabilizing dining; however, these algorithms are not wait-free. The algorithm presented in [2], in addition to being self-stabilizing, ensures k-fairness[4]. However, the aforementioned algorithms assume that the low-level shared-memory registers in the system satisfy read/write atomicity. In contrast, we assume that we have regular registers, which are weaker than atomic registers. Furthermore, in all of the above algorithms, if some process crashes, other processes are not guaranteed to continue taking actions; in other words, these implementations are not fault-tolerant.

Crash-tolerant dining. There has been a lot of work in designing crash-tolerant dining algorithms. However, in asynchronous systems, it is impossible to design dining algorithms in which neighbors and neighbors' neighbors of a crashed process do not stall [7]; in other words, asynchronous systems cannot guarantee a crash locality of less than 2. Achieving a smaller crash locality requires some recourse to crash detection. Subsequently, [27] showed that with access to sufficient crash detection ability, the crash locality of dining algorithms can be reduced to 1. The results in [27] employed the eventually perfect failure detector ($\Diamond\mathcal{P}$)[5] [6].

Later $\Diamond\mathcal{P}$ is shown in [28, 30] to be sufficient to achieve wait-freedom at the expense of a weaker exclusion guarantee: $\Diamond\mathcal{P}$-based dining algorithms may schedule conflicting actions concurrently, but only finitely many times; that is, exclusion is only *eventual.* The results in [29] demonstrate the necessity of the above trade-off.

Failure detectors. Intuitively, failure detectors [6] are oracles that may provide hints about process crashes. Since, our goal is to implement wait-free and stabilizing schedulers, the failure detectors that we employ must also be stabilizing. Self-stabilizing implementations of failure detectors have been an area of active research. In message passing systems, one of the first self-stabilizing implementations are for the perfect failure detector proposed in [21, 20], but only when at most one process crashes. Later, self-stabilizing algorithms for failure detectors in the presence of multiple process crashes was proposed in [3] for systems that have local clocks and have bounds on the relative messages delays. Subsequently, [18] proposed time-free self-stabilizing implementations of failure detectors in a similar system as [3], but where processes do not have

[4] A k-fair scheduler guarantees the following. For any process i, between any two consecutive accesses to its respective critical section by i, no other process enters its own respective critical section more than k times.

[5] Briefly, the eventually perfect failure detector, or $\Diamond\mathcal{P}$ for short, may provide arbitrary information to processes for some arbitrary, but finite, duration; however, eventually (after some potentially unknown time), it provides perfect information about process crashes.

local clocks. More recently, in [8], Delporte-Gallet et al. propose a self-stabilizing implementation of the Ω failure detector[6].

For shared memory systems, Dolev et al. [15] propose a self-stabilizing implementation of the Ω failure detector in systems with unknown bounds on relative process speeds. In [16], Fischer et al. implement and use a variant Ω called 'Ω?', which eventually determines whether not there is a leader, to solve self-stabilizing leader election in anonymous systems.

3 System Model

The system consists of a set of processes and a set of shared single-writer multi-reader read/write regular [19] registers.

Processes. The system contains the set Π of n processes where each process is a state machine. Each process has a unique incorruptible ID from the set $\{0, \ldots, n-1\}$ and is known to all the processes in the system. Since a process ID uniquely determines a process, in the remainder of this paper, we refer to a process and its ID interchangeably.

States and private variables. The state of the system (or, *system state*) is determined by the state of each process and the state of each shared variable in the system. In turn, the state of each process is determined by the set of *private variables* at that process. A private variable at a process i accessible only to process i and no other process may read or write to that variable. The states of shared variables are discussed later.

Steps. The read and write operations of shared variables and the state changes of processes are modeled by *steps*, which are of four types: invocation, response, transition, and crash. The invocation and response steps are discussed when describing shared variables. Transition steps are discussed when describing actions. Crash steps are discussed when describing faults.

Shared variables. The system contains a set of *shared variables* that are *regular* single-writer multi-reader registers. Each shared variable is a state machine that interacts with the processes through read and write operations. Each operation consists of two steps: *invocation* by the process and *response* by the shared variable. Each shared variable may be read by any processes in the system and is *owned* by some unique fixed process. Only the owner of a shared variable may write to it.

Actions. The state change at each process is determined by a set of *actions.* Actions are guarded commands [12] which are of the form "$\{guard\} \rightarrow$ *command*"; *guard* is a predicate on the state of the process, and *command* is a representation of at most one shared memory operation followed by the new state of the process.

Precisely, each command is an ordered tuple that consists of either a read or write operation and a transition step, or just a transition step. Recall that a read (or write) operation consists of an invocation step followed by a response step. In a *transition step*, a process may change its local state (by modifying

[6] Briefly, the Ω failure detector outputs a process ID at each process infinitely often. Eventually, after some potentially unknown time, and forever thereafter, Ω is guaranteed to output the ID of some unique correct processes.

the value of private variables) based on the current state and value returned by shared-memory operation in that action (if applicable).

Given an action $A \equiv \{guard\} \rightarrow command$ at a process i, in every state s of the system in which *guard* is *true*, the action a is said to be *enabled* in s at i. The algorithm that a process follows is described as a set of actions called *program actions*.

Tasks are a partitioning of program actions at each process; a task t is simply a set of program actions at a process. Each program action belongs to exactly one task. The notion of tasks is useful in describing fair executions.

Faults. Processes are prone to *crash faults*. When a crash fault occurs at process i, the process ceases taking steps without warning and never recovers. In effect, when a crash fault occurs at a process i, it disables all the program actions at i. A crash fault at each process i is modeled as an explicit action that consists of a single *crash* step, but is assumed to not be a program action, and therefore, it is not in any task. The crash fault action for each process i is continuously enabled until it is executed; however, the action (for each process i) occurs at most once. It is admissible for a process to never crash; that is, it is admissible for a crash fault action to never occur.

Eventually perfect failure detector. We assume that the system is augmented with a self-stabilizing implementation of the *eventually perfect failure detector*, or $\Diamond\mathcal{P}$, for short. Informally, $\Diamond\mathcal{P}$ provides information about crash faults to all the processes in the system in the form of a *suspect list* which contains a set of processes; eventually, $\Diamond\mathcal{P}$ never suspects correct processes and always suspects crashed processes. More precisely, we assume that each process i has a private variable $\Diamond\mathcal{P}_i$ that contains a set of process IDs. $\Diamond\mathcal{P}$ is specified by a set of actions, one action a_i for each process i, where a_i writes a value to $\Diamond\mathcal{P}_i$ continually such that, eventually and permanently, $\Diamond\mathcal{P}_i$ contains exactly the set of IDs of crashed processes.

Although we do not provide an explicit self-stabilizing implementation of $\Diamond\mathcal{P}$ here, we remark that there are many existing self-stabilizing $\Diamond\mathcal{P}$ and Ω implementations (discussed in Sect. 2) that may be modified appropriately and employed here. The choice of the implementation depends on the partial synchrony satisfied by the underlying distributed system and is beyond the scope of this paper.

Executions. An execution describes the state evolution of a system as a (potentially infinite) sequence $\alpha = S_0, a_1, S_1, a_2, \ldots$ of alternating system states and steps such that the following properties are satisfied. An execution may start from any system state.

1. For each process i, the subsequence α_i of α that consists of all the steps at process i can be partitioned into a sequence of actions $A_1, A_2, A_3, \ldots$. Let $s_{i.0}$ be the state of process i in S_0. There exists a sequence $s_{i.1}, s_{i.2}, \ldots$ of states of process i such that for every positive natural number x, action A_x is enabled in state $s_{i.x-1}$ and applying A_x causes i to transition to state $s_{i.x}$.
2. In α, for each process i, no step at i follows a crash step at i.
3. For every shared register r, the sequence of invocation and response steps in α for r satisfies the regularity property [19]. That is, every read operation returns either the value written by latest preceding write or the value being written by an overlapping write (if applicable).

4. For each correct process i, the sequence of values of $\Diamond\mathcal{P}_i$ during the execution satisfies the properties of $\Diamond\mathcal{P}$: eventually and permanently, $\Diamond\mathcal{P}_i$ contains exactly the set of IDs of faulty processes.

An execution α is said to be a *fair execution* if it satisfies the following properties. (1) If α is a finite execution, then no program action is enabled in the final state of the system after executing α. (2) If α is an infinite execution, then for each task t, either some enabled program action in t occurs infinitely often in α, or in infinitely many states of α, no program action in t is enabled. Note that crash actions are not in any task and, therefore, need not occur 'fairly'.

In any execution α, the set of processes at which a crash fault occurs is said to be *faulty* in α, and all the other processes are said to be *correct* in α. Each process is said to be *live* until it crashes.

4 Dining and Mutual Exclusion

In this section, we describe the two problems which are the primary focus of this article: *dining philosophers* and *mutual exclusion*.

Dining. The *dining philosophers problem* [22] is a scheduling problem that is represented by an undirected graph $G = (\Pi, E)$, called a *conflict graph*, where the set Π of processes (called *diners*) denotes the set of vertices of the conflict graph. The neighbors of each process i in G are denoted $N(i)$. Each process is assumed to know the conflict graph G. The state space of each process is partitioned into four sets: *thinking*, *hungry*, *eating*, and *exiting*. A solution to dining philosophers determines when a diner can transition from a hungry state to an eating state, and from an exiting state to a thinking state. In order to specify the dining philosopher's problem, we assume that the diners satisfy a set of 'well-formedness' properties defined next.

A diner is said to be *well-formed* iff (1) a diner becomes hungry only when thinking, but may remain thinking forever, (2) a diner remains hungry until it starts eating, and (3) a correct eating diner eventually exits.

An execution of the system which satisfies the well-formedness conditions is said to be a *well-formed execution*.

A solution to dining philosophers is an algorithm $\mathcal{A}$ that satisfies the following properties. (1) There exists a non-empty set of states, called *start states* in which all the diners are thinking. (2) Furthermore, in every fair well-formed execution that starts from a start state, the following properties are satisfied.

(a) Mutual exclusion: For each diner i, while i is live and eating, no live process in $N(i)$ is eating.
(b) Eventual Exit: If a correct diner is exiting, then eventually that diner is thinking.
(c) Fairness: If some correct diner is hungry, then eventually that diner is eating.

Fix such an algorithm $\mathcal{A}$. Let $\mathcal{E}_s$ be the set of all fair well-formed executions that start from some start state. Let Q_{safe} be the set of states that occur in any execution in $\mathcal{E}_s$. The set Q_{safe} is said to be the set of *safe states* of $\mathcal{A}$.

A dining philosophers algorithm $\mathcal{A}$ is said to be pseudo-stabilizing [4] and wait-free if it guarantees that for any fair well-formed execution α (starting from an arbitrary state), there exists a suffix of α in which mutual exclusion, eventual exit, and fairness are satisfied regardless of the number of process crashes. $\mathcal{A}$ is said to be self-stabilizing [11] and wait-free, if, in addition to being pseudo-stabilizing and wait-free, it guarantees that every well-formed execution α that starts from a safe state is the suffix of some execution in $\mathcal{E}_s$.

Mutual exclusion. Mutual exclusion [10] is a degenerate case of the dining philosophers problem in which the conflict graph is a complete graph. In the mutual exclusion parlance, a thinking state is called a *remainder* state, a hungry state is called a *trying* state, an eating state is called a *critical (section)* state, and an exiting state is called an *exiting* state.

5 Wait-Free Pseudo-Stabilizing Mutual Exclusion

Our proposed wait-free pseudo-stabilizing dining algorithm is constructed in three parts. In the first part, we construct a wait-free pseudo-stabilizing ring of processes. In the second part, we deploy a modified version of Dijkstra's mutual-exclusion algorithm on the ring from part one. Finally, in the third part, we construct a wait-free pseudo-stabilizing dining algorithm using a collection of overlapping mutual exclusion instances. This section describes the first two parts

Wait-free pseudo-stabilizing ring. We use the eventually perfect failure detector $\Diamond\mathcal{P}$ to construct a wait-free pseudo-stabilizing ring over an arbitrary set $\Pi_r \subseteq \Pi$ of processes. The algorithm is straightforward. Each process $i \in \Pi_r$ (locally) determines its predecessor j in the ring as follows. The predecessor of i is a process j such that j is the largest ID smaller than i modulo n such that j is in Π_r and not suspected by $\Diamond\mathcal{P}$. Precisely, the predecessor of i is determined by the function *pred* as follows: $pred(i) = j$, where $j \in \Pi_r \setminus \Diamond\mathcal{P}_i$ and $\forall k \in \mathbb{N}^+ : 0 < k < i - j \pmod{n} : i - k \pmod{n} \notin \Pi_r \setminus \Diamond\mathcal{P}_i$.

The correctness of the above algorithm is also straightforward. Eventually, $\Diamond\mathcal{P}$ suspects exactly the crashed processes, and provides the same output to all the processes. Therefore, eventually all the live processes in Π_r have a consistent and accurate view of the processes that are live in Π_r, and they converge to a unique ring encompassing all the live processes in Π_r.

Wait-free pseudo-stabilizing mutual exclusion. We construct a wait-free pseudo-stabilizing mutual exclusion algorithm for an arbitrary set $\Pi_r \subseteq \Pi$ of processes as follows. The algorithm in [14] modifies Dijkstra's self-stabilizing mutual exclusion algorithm [11] for rings with read/write regular registers. Note that Dijkstra's algorithm (in [14] and [11]) requires some process be the 'distinguished' process whose actions are different from other processes. In our case, we require this 'distinguished' process to be correct, and we require no other process to be 'distinguished'. Each process determines whether or not it is distinguished by computing the following local function $leader : \Pi_r \to \{true, false\}$. For each process i, $leader(i)$ is $true$ iff $i = \min(\Pi_r \setminus \Diamond\mathcal{P}_i)$.

Eventually, $leader(i)$ is $true$ only for the process i with the lowest ID among the correct processes in Π_l, and for all other processes i', $leader(i')$ is $false$. Thus, the function $leader$ eventually determines a unique distinguished process in Π_r. Now we simply deploy the algorithm from [14] in the previously described wait-free pseudo-stabilizing ring over Π_r with multi-reader single-writer regular registers with the modification that a process i behaves as the distinguished process when $leader(i)$ is $true$, and it behaves as a non-distinguished process when $leader(i)$ is $false$. Thus, we obtain a wait-free pseudo-stabilizing mutual exclusion algorithm over the set Π_r of processes.

The correctness and stability of the algorithm is straightforward and has been omitted from this version of the paper.

We use multiple instances of the above algorithm in solving dining. For disambiguation, we adopt the following convention: an instance of the above algorithm over a set Π_x of processes is denoted $\mathcal{MX}_x$.

Note that $\mathcal{MX}_r$ interacts with clients at each process in Π_r. We assume that for each process $i \in \Pi_r$, $\mathcal{MX}_r$ contains a variable $mutex_i$. The variable $mutex_i$, at each process i, can have one of four values: *remainder*, *trying*, *critical*, and *exiting*. If $mutex_i$ is *remainder*, then process i is in its *remainder* section; if $mutex_i$ is *trying*, then i is *trying*; if $mutex_i$ is *critical*, then i is in its *critical section*; and if $mutex_i$ is *exiting*, then i has finished accessing its critical section and *exiting* to the remainder section.

We denote the set of safe states for this algorithm as *mutex safe states*[7]. We will use this notion of mutex safe states when arguing for the correctness of the dining algorithm that is described next.

6 Wait-Free Pseudo-Stabilizing Dining

In this section, we use multiple instances of the mutual exclusion algorithm $\mathcal{MX}$ from Sect. 5 to construct a pseudo-stabilizing wait-free dining algorithm. The algorithm is inspired by the HRA algorithm from [22].

6.1 Algorithm Description

Let $G = (\Pi, E)$ be the conflict graph. Let $\mathcal{R}$ be the set of maximal cliques in G. Let $|\mathcal{R}|$ be k. For convenience, let $\mathcal{R} = \{R_x | x \in \mathbb{N}^+ \wedge 0 < x \leq k\}$. We assume a total order on the cliques such that R_x is ordered before R_y iff $x < y$. For each clique R_x, let Π_x denote the set of processes (diners) in R_x. Each clique $R_x \in \mathcal{R}$ represents a subset of resources to be accessed in isolation by diners in Π_x. Consequently, for each clique R_x, we associate an instance of the wait-free pseudo-stabilizing mutual-exclusion algorithm $\mathcal{MX}_x$, and the participants in $\mathcal{MX}_x$ constitute the set Π_x.

For each diner i, let C_i denote the set of all cliques R_x such that $i \in \Pi_x$; that is, diner i contends for exclusive access to the all the resources associated with cliques in C_i.

Variables. For disambiguation, the variable $mutex_i$ in the mutual exclusion instance $\mathcal{MX}_x$ will be referred to as $\mathcal{MX}_x.mutex_i$. Each diner has access to the private variables $\mathcal{MX}_x.mutex_i$, where $R_x \in C_i$. Finally, we introduce a new private variable $diningState_i$ for each diner i in the system. The variable may contain one of the four values: *thinking*, *hungry*, *eating*, and *exiting*. If

[7] Note that we do not explicitly specify the set of safe states here; it is specified and described in [14]. Here we merely assert its existence, which is sufficient for our purposes.

$diningState_i$ is *thinking*, then i is thinking, if $diningState_i$ is *hungry*, then i is hungry, and so on.

Three functions. For each diner i, apart from the actions of algorithm $\mathcal{MX}_x$ for each $R_x \in C_i$, we introduce three additional actions denoted $D.1$–$D.3$ which constitute a new task. Before describing the actions, we introduce three functions $csPrefix$, $currentMutex$, and $badSuffix$ which are used in specifying the guards for the three actions.

Sequence $\mathcal{C}_i$. Let $\mathcal{C}_i$ denote the sequence over all the cliques from C_i such that a clique R_x precedes a clique R_y in $\mathcal{C}_i$ iff $x < y$.

Functions csPrefix and currentMutex. The function $csPrefix(\mathcal{C}_i)$ returns the longest prefix of $\mathcal{C}_i$ such that, for each R_x in $csPrefix(\mathcal{C}_i)$, $\mathcal{MX}_x.mutex_i = critical$ (i is in the critical section of $\mathcal{MX}_x$). The function $currentMutex(\mathcal{C}_i)$ returns the first clique following $csPrefix(\mathcal{C}_i)$ in $\mathcal{C}_i$, if such a clique exists; otherwise, it returns $\perp$.

Function badSuffix. The function $badSuffix(\mathcal{C}_i)$ is *true* iff there exists some R_x in the suffix of $\mathcal{C}_i$ following $currentMutex(\mathcal{C}_i)$ such that $\mathcal{MX}_x.mutex_i$ is either *trying* or *critical* (i is either trying or in the critical section of $\mathcal{MX}_x$).

An informal motivation for the foregoing functions follows. Upon becoming hungry, each diner i starts trying in $\mathcal{MX}_c = currentMutex(\mathcal{C}_i)$, and when i enters the critical section of $\mathcal{MX}_c$, $\mathcal{MX}_c$ becomes a part of $csPrefix(\mathcal{C}_i)$. Subsequently, i starts trying in $\mathcal{MX}_{c'} = currentMutex(\mathcal{C}_i)$ which follows $\mathcal{MX}_c$ in $\mathcal{C}_i$, and so on, until i is in the critical section of all $\mathcal{MX}$ instances in $\mathcal{C}_i$. In the absence of faults, while a diner i is hungry, i is in the critical section of all $\mathcal{MX}$ instances in $csPrefix(\mathcal{C}_i)$ and in the remainder or exiting section of all the $\mathcal{MX}$ instances in the suffix following $currentMutex(\mathcal{C}_i)$ in $\mathcal{C}_i$. However, due to a transient fault, it is possible for a diner i to be in a state in which i is either trying or in the critical section of some $\mathcal{MX}$ instance in the suffix; when this occurs, we say that the suffix is “bad”. This is captured by the predicate $badSuffix(\mathcal{C}_i)$.

Actions. We introduce three new actions (that constitute a single new task) for each process (diner) i in our proposed wait-free self-stabilizing dining algorithm. The pseudocode is given in Fig. 1.1 and described next.

The first action, *Action D.1*, is enabled when the diner (say) i is either thinking or exiting, or $badSuffix(\mathcal{C}_i)$ is *true*. When Action D.1 is executed, it sets each *mutex* variable that is not *remainder* to *exiting* and sets $diningState_i$ to *thinking*. That is, diner i is either in the remainder section or in the exit section in all the mutual-exclusion instances. Note that this transition need not satisfy the well-formedness condition of mutual-exclusion clients. For stabilization, it is important to ensure that such well-formedness violations occur only finitely many times in any execution.

The second action, *Action D.2*, is enabled when the diner i is hungry, i is not in the critical section of all the associated mutual-exclusion instances ($csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i$), and $badSuffix(\mathcal{C}_i)$ is $false$. When Action D.2 is executed, i starts trying in the mutual-exclusion instance of $currentMutex(\mathcal{C}_i)$.

The third action, *Action D.3*, is enabled when the diner i is hungry, and i is in the critical section of all the associated $\mathcal{MX}$ instances ($csPrefix(\mathcal{C}_i) = \mathcal{C}_i$). When Action D.3 is executed, diner i starts eating.

Note that the client of the dining service at each process is responsible for transitioning from thinking to hungry and from eating to thinking.

```
private variable $diningState_i$
foreach $R_x \in C_i$:
  variable $\mathcal{MX}_x.mutex_i$        Variable $mutex_i$ in instance $\mathcal{MX}_x$ described in Sect. 5
/* Note that the client at process i sets $diningState_i$ to "hungry" upon becoming hungry
and to "exiting" upon finishing eating. Client actions are not shown.
Also, the client is assumed to be "well-formed". */
The three actions below constitute a single task
------------------------------------------------------------------------------------------
{$(diningState_i = thinking) \vee (diningState_i = exiting)$
  $\vee (badSuffix(\mathcal{C}_i))$} $\longrightarrow$                              Action D.1
      foreach $R_x \in C_i$:
         if ($\mathcal{MX}_x.mutex_i \neq remainder$)
            $\mathcal{MX}_x.mutex_i \leftarrow exiting$     // If eating, then exit; if hungry, then abort.
      $diningState_i \leftarrow thinking$   // Exit in all mutex instances, and start thinking
------------------------------------------------------------------------------------------
{$(diningState_i = hungry) \wedge (csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i)$
  $\wedge(\neg badSuffix(\mathcal{C}_i))$} $\longrightarrow$                           Action D.2
      $R_x \leftarrow currentMutex(\mathcal{C}_i)$ // If hungry and in the critical section of a prefix of
      if ($\mathcal{MX}_x.mutex_i = remainder$)     // cliques, then start trying in the mutex
         $\mathcal{MX}_x.mutex_i \leftarrow trying$                // associated with the next clique
------------------------------------------------------------------------------------------
{$(diningState_i = hungry) \wedge (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$} $\longrightarrow$            Action D.3
      $diningState_i \leftarrow eating$                          // Transit from hungry to eating
```

Figure 1.1. Self-Stabilizing Wait-Free Dining. Action system at process i.

6.2 Pseudo-Stabilization

In order to establish the pseudo-stabilization property, we have to define a set of safe states for each process in the system. *The system is said to be in a safe state iff every mutual-exclusion instance is in a mutex safe state, and every live diner is in a "diner-safe state", as defined next.*

A diner i is said to be in a *diner-safe state* if (1) $badSuffix(\mathcal{C}_i)$ is false, (2) if i is eating, then every $\mathcal{MX}$ instance at i is in its critical section, and (3) if i is thinking, then every $\mathcal{MX}$ instance at i is either in its remainder section or is exiting. Precisely, $\neg badSuffix(\mathcal{C}_i) \wedge ((diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)) \wedge ((diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting)))$ is *true*. The system is said to be in a *diner-safe state* iff each live diner is in a diner-safe state.

Closure. We prove closure with respect to diner states in Lemma 4 using three helper lemmas. Let s be an arbitrary state of the system executing the action system from Fig. 1.1, where each mutual-exclusion instance is an instance of $\mathcal{MX}$.

Lemma 1. *For any process i, if $s.badSuffix(\mathcal{C}_i)$ is $false$, then for each successor s' of s, $s'.badSuffix(\mathcal{C}_i)$ is $false$.*

Lemma 2. *For any process i, if $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is true in state s, then for any successor s' of s, $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i))$ remains true.*

Lemma 3. *For any process i, if $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ is true in state s, then for any successor s' of s, $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ remains true.*

The proofs of the above three lemmas are straightforward. They consider each possible successor s' of s by considering each enabled action. By case analysis for each such action, we confirm that the lemmas are true.

Closure, with respect to diner-safe states, follows from Lemmas 1, 2, and 3. We have the following lemma.

Lemma 4. *Every fair execution of the action system from Fig. 1.1, where each mutual-exclusion instance is an instance of $\mathcal{MX}$, satisfies closure (with respect to diner-safe states): if the system eventually reaches a diner-safe state s, then the suffix of the execution following s contains only diner-safe states.*

Convergence. We prove convergence, with respect to diner safe states, in three parts. First, we prove that if the system is in a state where for some correct diner i, $badSuffix(\mathcal{C}_i)$ is $true$, then eventually, $badSuffix(\mathcal{C}_i)$ becomes $false$. Next, we prove that if the system is in a state where, for some diner i, i is eating, but $csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i$, then eventually we reach a state where, if i is eating, then $csPrefix(\mathcal{C}_i) = \mathcal{C}_i$. Finally, we show that the following. If the system is in a state where, if some diner i is thinking, but in some $\mathcal{MX}$ instance i is neither exiting nor in the remainder section, then we eventually reach a state in which the following is true. If i is thinking, then in all $\mathcal{MX}$ instances (for i), i is either exiting or in the remainder section.

For the following lemmas, fix α to be an arbitrary fair execution of the action system from Fig. 1.1, where each mutual-exclusion instance is an instance of $\mathcal{MX}$. For any given pair of states s and s' in the system-state sequence associated α, any if s' occurs after s, then s' is said to be a *descendant* of s.

Lemma 5. *In α, for each correct diner i, if $s.badSuffix(\mathcal{C}_i)$ is true for some state s, then there exists a descendant s' of s such that $s'.badSuffix(\mathcal{C}_i)$ is $false$.*

Proof sketch. Note that Action D.1 at i is enabled in state s and remains enabled until executed. Upon executing Action D.1 at i, $badSuffix(\mathcal{C}_i)$ becomes false.

Lemma 6. *In α, for each correct diner i, if the system is in a state s where $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is $false$, then, there exists a descendant s' of s in α, such that $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is true, in s'.*

Proof sketch. Note that i eats for finite durations in α. Since i is eating in state s, there exists a descendant s' of s in which i is not eating, and $(diningState_i = eating) \rightarrow (csPrefix(\mathcal{C}_i) = \mathcal{C}_i)$ is $true$, in s'. □

Lemma 7. *In α, for each correct diner i, if the system is in a state s where $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ is $false$, then there exists a descendant s' of s in α, such that $(diningState_i = thinking) \rightarrow (\forall R_x \in C_i :: (\mathcal{MX}_x.mutex_i = remainder) \vee (\mathcal{MX}_x.mutex_i = exiting))$ is true in s'.*

Proof sketch. If i becomes hungry, then the lemma is satisfied. Otherwise, note that Action D.1 at i is enabled in state s and remains enabled until executed. Upon executing Action D.1 at i, we see that for each R_x in C_i Action D.1 sets $\mathcal{MX}_x.mutex_i$ to either *remainder* or *exiting*. □

Lemma 8. *The action system from Fig. 1.1, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance, satisfies convergence (with respect to diner-safe states): from any arbitrary state, upon executing enabled program actions from Fig. 1.1 and all the $\mathcal{MX}$ instances, the system eventually reaches a diner-safe state.*

The proof follows from Lemmas 5, 6, and 7.

Thus, we have shown pseudo-stabilization with respect to diner-safe states. In order to complete the proof for pseudo-stabilization, we have to prove that the action system in Fig. 1.1 in eventually well-formed. Note that Action D.3 does not change the value of *mutex* variables, and Action D.2 changes a *mutex* variable to *trying* from *remainder*, which satisfies mutual exclusion well-formedness conditions. However, Action D.1 could set the *mutex* variables to *trying* from *exiting* and violate the well-formedness condition. Therefore, it remains to show that Action D.1 violates the well-formedness conditions only finitely many times.

Lemma 9. *In any fair execution of the action system from Fig. 1.1, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance, at each correct process i, only finitely many occurrences of Action D.1 violate mutual exclusion well-formedness conditions.*

Proof sketch. From the pseudocode, we see that Action D.1 violates mutual exclusion well-formedness conditions if line 4 is executed at a correct diner i when $\mathcal{MX}_x.mutex_i$ is *trying* for some $r_x \in C_i$. From Lemmas 8 and 4, we know that in any fair execution, eventually forever $badSuffix(\mathcal{C}_i)$ is $false$. Therefore, eventually forever, when i executes Action D.1 $diningState_i$ is either *thinking* or *exiting*. However, from Lemmas 3 and 7, we know that when $diningState_i$ is thinking, $\mathcal{MX}_x.mutex_i$ is not *trying*. Finally, if i executes Action D.1 when $diningState_i$ is *exiting*, then recall that while i was eating (just prior to exiting) $csPrefix(\mathcal{C}_i)$ is *true*. Consequently, when $diningState_i$ is *exiting*, $csPrefix(\mathcal{C}_i)$ is *true*; that is, $\mathcal{MX}_x.mutex_i$ is not *trying*. Therefore, at each correct process i, only finitely many occurrences of Action D.1 violates mutual exclusion well-formedness conditions. □

Therefore, from Lemmas 4, 8, and 9, we establish pseudo-stabilization.

6.3 Correctness

We demonstrate safety and progress starting from a safe state after $\Diamond\mathcal{P}$ stops falsely suspecting correct processes. The safety condition for wait-free dining is that no two neighboring diners are live and eating concurrently. The progress condition is that every correct hungry diner eventually eats.

Lemma 10. *In any fair execution starting from a safe state, the action system from Fig. 1.1, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance satisfies safety: no two neighboring diners are live and eating concurrently.*

Proof sketch. Let α be a fair execution starting from a safe state. Let α' be a suffix of α in which $\Diamond\mathcal{P}$ does not suspect live processes. Let i and j be two live neighbors in some safe state s in α' in which i is eating. Since i and j are neighbors there exists clique R_y such that $R_y \in C_i \cap C_j$. Since i is eating, we know that $\mathcal{MX}_y.mutex_i = critical$ in s. From the mutual exclusion property, we know that $\mathcal{MX}_y.mutex_j \neq critical$ in s. In other words, $csPrefix(\mathcal{C}_j) \neq \mathcal{C}_j$ in s. Therefore, j is not eating concurrently with i. □

Lemma 11. *In any fair well-formed execution starting from a safe state, the action system from Fig. 1.1, where each mutual-exclusion instance comprises an $\mathcal{MX}$ instance, satisfies progress: every correct hungry diner eventually eats.*

Proof sketch. For contradiction, we assume that in some fair well-formed execution α of the system starting from a safe state, some hungry correct diner never eats. If i is such a diner, then i must be trying for ever in some $\mathcal{MX}$ instance Since $\mathcal{MX}_x$ is wait-free, this implies that some correct neighbor j of i is in the critical section of $\mathcal{MX}_x$ forever. From Fig. 1.1, this means that for some other clique R_y, $y > x$, j must be trying in $\mathcal{MX}_y$ forever. Consequently, j is also hungry forever. By the total order in which processes start trying in the $\mathcal{MX}$ instances, x cannot be in the critical section of $\mathcal{MX}_y$. Therefore, some other process k must be in the critical section of $\mathcal{MX}_y$ forever, and consequently, that process k must be trying forever in some $\mathcal{MX}_z$, where $z > y$, (and therefore, k must be hungry forever), and so on. Since there are only finitely many process and finitely many $\mathcal{MX}$ instances, there must exist some process $\hat{i}$ that is (a) hungry forever, (b) in the critical section of some $\mathcal{MX}_{\hat{z}}$, and (c) not trying in any $\mathcal{MX}_{z'}$, where $z' > \hat{z}$. However, from Fig. 1.1, we see that this is impossible. Thus, we have a contradiction. □

From Lemmas 10 and 11, we establish correctness.

Theorem 1. *The action system in Fig. 1.1, where each mutual-exclusion instance is an $\mathcal{MX}$ instance, solves wait-free pseudo-stabilizing dining.*

7 Discussion

Wait-free self-stabilizing regular registers. Our algorithm assumes that the system contains wait-free self-stabilizing regular registers. We remark that existing results from [17] may be used to construct wait-free self-stabilizing regular registers from wait-free self-stabilizing safe registers. Although results in [17] construct atomic registers from regular registers, this construction is expensive and uses unbounded counters.

Achieving self-stabilization. Our proposed algorithm is wait-free and pseudo-stabilizing, and not self-stabilizing, because starting from a safe state, if $\Diamond\mathcal{P}$ falsely suspects a correct process, it could result in the system transitioning to an unsafe state. Since $\Diamond\mathcal{P}$ eventually ceases such false suspicions, we are guaranteed pseudo-stabilization. If we view a false suspicion by $\Diamond\mathcal{P}$ as a transient fault in the system, then we see that our algorithm is, in fact, self-stabilizing as well. Alternatively, if we replace $\Diamond\mathcal{P}$ with the perfect failure detector $\mathcal{P}$ [6] which never suspects live processes and eventually and permanently suspects crashed processes, our algorithm becomes self-stabilizing (in addition to being wait-free).

On assuming $\Diamond\mathcal{P}$. Recall that wait-free dining is unsolvable in asynchronous systems. Consequently, we resort to using $\Diamond\mathcal{P}$ to achieve wait-freedom. In fact, $\Diamond\mathcal{P}$ is the weakest failure detector to solve wait-free dining under eventual exclusion [29]. Therefore, assuming $\Diamond\mathcal{P}$ is necessary to solve wait-free self-stabilizing dining in any (partially synchronous) shared-memory system.

Future work. There are several ways in which our result can be extended. For instance, if we can implement self-stabilizing $\Diamond\mathcal{P}$ in a partially synchronous system with safe registers, then we can adapt our algorithm to solve wait-free self-stabilizing dining with safe registers instead of regular registers. Another avenue for improvement is in fairness. Our algorithm is weakly fair — every hungry diner eventually eats, but could be overtaken unboundedly many times by hungry neighbors. However, we know that $\Diamond\mathcal{P}$ is sufficient to solve eventually bounded-fair dining [30]. It remains to be seen if we can solve wait-free stabilizing dining with bounded fairness using $\Diamond\mathcal{P}$.

References

1. Antonoiu, G., Srimani, P.K.: Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In: Proceedings of the 5th International Euro-Par Conference on Parallel Processing. vol. 1685, pp. 823–830 (1999)
2. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilizing local mutual exclusion and daemon refinement. Chicago Journal of Theortical Computer Science 2002(1) (2002)
3. 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(11), 1177–1187 (1997)
4. Burns, J.E., Gouda, M.G., Miller, R.E.: Stabilization and pseudo-stabilization. Distributed Computing 7(1), 35–42 (1993)
5. Cantarell, S., Datta, A.K., Petit, F.: Self-stabilizing atomicity refinement allowing neighborhood concurrency. In: Proceedings of the 6th International Symposium Self-Stabilizing Systems. pp. 102–112 (2003)
6. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
7. Choy, M., Singh, A.K.: Localizing failures in distributed synchronization. IEEE Transactions on Parallel and Distributed Systems 7(7), 705–716 (1996)
8. Delporte-Gallet, C., Devismes, S., Fauconnier, H.: Robust stabilizing leader election. In: Proceedings of the 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems. pp. 219–233 (2007)
9. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing 65(4), 492–505 (2005)

10. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM 8(9), 569 (1965)
11. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11), 643–644 (1974)
12. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
13. Dolev, S.: Self-Stabilization. MIT Press (2000)
14. Dolev, S., Herman, T.: Dijkstra's self-stabilizing algorithm in unsupportive environments. In: Workshop on Self-Stabilizing Systems. pp. 67–81 (2001)
15. Dolev, S., Kat, R., Schiller, E.: When consensus meets self-stabilization. In: Proceedings of the tenth International Conference on the Principles of Distributed Systems. pp. 45–63 (2006)
16. Fischer, M., Jiang, H.: Self-stabilizing leader election in networks of finite-state anonymous agents. In: Proceedings of the 10th International Conference on the Principles of Distributed Systems. pp. 395–409 (2006), 10.1007/11945529_28
17. Hoepman, J.H., Papatriantafilou, M., Tsigas, P.: Self-stabilization of wait-free shared memory objects. Journal of Parallel and Distributed Computing 62(5), 818–842 (2002)
18. Hutle, M., Widder, J.: On the possibility and the impossibility of message-driven self-stabilizing failure detection. In: Proceeding of the 7th International Symposium on Self Stabilizing Systems. pp. 153–170. Springer Berlin / Heidelberg (2005)
19. Lamport, L.: On interprocess communication. part II: Algorithms. Distributed Computing 1(2), 86–101 (1986)
20. Line, J.C., Ghosh, S.: A methodology for constructing a stabilizing crash-tolerant application. In: Proceedings of the 13th Symposium on Reliable Distributed Systems. pp. 12 –21 (1994)
21. Line, J.C., Ghosh, S.: Stabilizing algorithms for diagnosing crash failures. In: Proceedings of the 13th Annual ACM symposium on Principles of Distributed Computing. pp. 376–376 (1994)
22. Lynch, N.A.: Upper bounds for static resource allocation in a distributed system. Journal of Computer and System Sciences 23(2), 254–278 (1981)
23. Mizuno, M., Nesterenko, M.: A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. Inf. Process. Lett. 66(6), 285–290 (June 1998)
24. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: Proceedings of the 22nd International Conference on Distributed Computing Systems. pp. 172–179 (2002)
25. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. Journal of Parallel and Distributed Computing 62(5), 766–791 (May 2002)
26. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems. pp. 22 – 29 (2002)
27. Pike, S.M., Sivilotti, P.A.: Dining philosophers with crash locality 1. In: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems. pp. 22–29 (2004)
28. Pike, S.M., Song, Y., Sastry, S.: Wait-free dining under eventual weak exclusion. In: Proceedings of the 9th International Conference on Distributed Computing and Networking. pp. 135–146 (2008)
29. Sastry, S., Pike, S.M., Welch, J.L.: The weakest failure detector for wait-free dining under eventual weak exclusion. In: Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures. pp. 111–120 (2009)
30. Song, Y., Pike, S.M.: Eventually k-bounded wait-free distributed daemons. In: IEEE International Conference on Dependable Systems and Networks. pp. 645–655 (2007)