THE WEAKEST FAILURE DETECTOR FOR SOLVING WAIT-FREE, EVENTUALLY BOUNDED-FAIR DINING PHILOSOPHERS

A Dissertation

by

YANTAO SONG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2008

Major Subject: Computer Science

THE WEAKEST FAILURE DETECTOR FOR SOLVING WAIT-FREE, EVENTUALLY BOUNDED-FAIR DINING PHILOSOPHERS

A Dissertation

by

YANTAO SONG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Scott M. Pike
Committee Members,	Jennifer L. Welch
	Jianer Chen
	Scott L. Miller
Head of Department,	Valerie E. Taylor

December 2008

Major Subject: Computer Science

ABSTRACT

The Weakest Failure Detector for Solving Wait-Free,

Eventually Bounded-Fair Dining Philosophers. (December 2008)

Yantao Song, B.S., Beijing Institute of Technology;

M.S., Chinese Academy of Sciences

Chair of Advisory Committee: Dr. Scott M. Pike

This dissertation explores the necessary and sufficient conditions to solve a variant of the dining philosophers problem. This dining variant is defined by three properties: *wait-freedom, eventual weak exclusion*, and *eventual bounded fairness.* Wait-freedom guarantees that every correct hungry process eventually enters its critical section, regardless of process crashes. Eventual weak exclusion guarantees that every execution has an infinite suffix during which no two live neighbors execute overlapping critical sections. Eventual bounded fairness guarantees that there exists a fairness bound k such that every execution has an infinite suffix during which no correct hungry process is overtaken more than k times by any neighbor. This dining variant (*WF-EBF dining* for short) is important for synchronization tasks where eventual safety (i.e., eventual weak exclusion) is sufficient for correctness (e.g., duty-cycle scheduling, self-stabilizing daemons, and contention managers).

Unfortunately, it is known that wait-free dining is unsolvable in asynchronous message-passing systems subject to crash faults. To circumvent this impossibility result, it is necessary to assume the existence of bounds on timing properties, such as relative process speeds and message delivery time. As such, it is of interest to characterize the necessary and sufficient timing assumptions to solve WF-EBF dining.

We focus on implicit timing assumptions, which can be encapsulated by failure detectors. Failure detectors can be viewed as distributed oracles that can be queried for potentially unreliable information about crash faults. The weakest detector $\mathcal{D}$ for WF-EBF dining means that $\mathcal{D}$ is both *necessary* and *sufficient*. Necessity means that every failure detector that solves WF-EBF dining is at least as strong as $\mathcal{D}$. Sufficiency means that there exists at least one algorithm that solves WF-EBF dining using $\mathcal{D}$. As such, our research goal is to characterize the weakest failure detector to solve WF-EBF dining.

We prove that the eventually perfect failure detector $\diamond\mathcal{P}$ is the weakest failure detector for solving WF-EBF dining. $\diamond\mathcal{P}$ eventually suspects crashed processes permanently, but may make mistakes by wrongfully suspecting correct processes finitely many times during any execution. As such, $\diamond\mathcal{P}$ eventually stops suspecting correct processes.

To my parents, my wife, and my daughter

ACKNOWLEDGMENTS

First I would like to thank my adviser Dr. Scott M. Pike for his intelligent guidance and constant encouragement. I have been fortunate to work with Dr. Pike not only for his deep insight in distributed computing, but also for his continuous effort to improve my English proficiency. I will never forget the days we worked together.

I also want to express my sincere gratitude to my committee members, Dr. Jianer Chen, Dr. Jennifer L. Welch, and Dr. Scott L. Miller, for their time and assistance in helping me complete the Ph.D. program. Thanks are due to Dr. Chen for his valuable suggestions and encouragement regarding my research; and I really enjoyed his classes. Thanks are due also to Dr. Welch for her help on the preparation of my dissertation. Her kindness is most appreciated. Thanks also are extended to Dr. Miller for his help in my Ph.D. study, and whose classes I also thoroughly enjoyed.

Reaching back to my masters and undergraduate study, I would like to thank my former advisers, Tiecheng Yu and Yuhe Zhang, for their guidance and supervision. The experience of working with them established a solid background for my Ph.D. study.

I also appreciate all the help from my friends and colleagues in Texas A&M University. Friendships established here are a very important part of my life at Texas A&M University. I would especially like to thank Srikanth Sastry and Kaustav Ghoshal for our intriguing discussions, which initiated many interesting ideas. I wish all the best of luck in completing their graduate study and excelling in their professional careers.

Finally, I wish to thank my parents, my wife, and my family. I could not have finished the Ph.D. program without your support and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

This chapter defines the wait-free, eventually bounded-fair dining philosophers problem (WF-EBF dining for short) and provides an overview of this dissertation, including the motivation, background, research goal, methodology, and outline.

1.1. Motivation and Problem Statement

This section will motivate and define the WF-EBF dining problem. In short, the WF-EBF dining problem is motivated to implement wait-free, eventually bounded-fair daemons, which are necessary for self-stabilizing protocols in environments where daemons are subject to crash faults. The WF-EBF dining problem is a dining philosophers variant that satisfies three properties: wait-freedom, eventual weak exclusion, and eventual bounded fairness.

1.1.1. Self-Stabilization

Self-stabilization is a technique for developing fault-tolerant systems [1, 2]. Starting from any configuration, a self-stabilizing algorithm always converges to a closed set of safe configurations from which correct behaviors follow. Transient faults, such as memory corruptions, are repairable, but can drive systems into arbitrary configurations. Self-stabilizing algorithms can tolerate finitely many transient faults, since the algorithms are able to recover the system autonomously from any configuration

The journal model is *IEEE Transactions on Computers*.

resulting from transient faults.

Self-stabilizing systems are specified by two properties: *closure* and *convergence* [3]. Every self-stabilizing system defines a set of safe configurations. Closure guarantees that, starting from any safe configuration, every action in fault-free executions results in a safe configuration. Transient faults can drive a system into an unsafe configuration. However, convergence guarantees that starting from an arbitrary (unsafe) configuration, every fault-free execution eventually yields a safe configuration.

Self-stabilizing systems are guaranteed to tolerate *finitely* many transient faults. In an infinite execution with finitely many transient faults, there exists an infinite fault-free suffix during which the system is guaranteed to converge to a safe configuration. Self-stabilization, however, is not guaranteed to tolerate *infinitely* many transient faults, since a system recovering from a transient fault may always be interrupted by another transient fault.

A fundamental assumption for self-stabilization is that every live process executes sufficiently many steps (for future references, we name this assumption as the fairness assumption). This assumption is necessary to establish the convergence property. By contradiction, suppose that a live process p executes only finitely many steps. Thereafter, a transient fault occurs at process p. If p is unable to execute subsequent steps that are necessary for detecting and correcting the transient fault, then the system might not recover from the transient fault. Hence, to tolerate transient faults, if live processes need to take steps, these processes must be scheduled to take sufficiently many subsequent steps eventually.

1.1.2. Process Scheduling

In distributed systems, process scheduling is often subject to some constraints, which are usually imposed by actions that cannot be executed concurrently. These actions are said to be *in conflict with* each other. The conflicts are usually caused by resources that are exclusively shared by processes. Two examples of conflicting actions are given below.

The first example is an algorithm in which processes communicate via shared read/write registers. If two processes execute *write* operations simultaneously to a shared register, then the *write* operations may produce an invalid value in the shared register. As such, the *write* operations are in conflict with each other, and the conflict is caused by the exclusively shared resource, the shared register.

The second example is an algorithm in which processes communicate via message passing on a shared wireless channel. If two or more processes *send* messages simultaneously via the same channel, then the messages could be corrupted or lost due to channel jamming. As such, the *send* operations are in conflict with each other, and the conflict is caused by the exclusively shared resource, the communication channel.

Violations of scheduling constraints are considered scheduling mistakes. In the above examples, the scheduling mistakes are the simultaneous *write* operations and the simultaneous *send* operations. These mistakes introduce faults (e.g., invalid register values and corrupted messages, which may be viewed as transient faults) into systems and might prevent systems from behaving correctly. Therefore, concurrency control is necessary to avoid violations of scheduling constraints imposed by conflicting actions.

Concurrency control is often coordinated by a *distributed daemon*, which continually selects a non-empty subset of processes to execute a set of non-conflicting actions concurrently [2, 4]. In other words, a distributed daemon should be able to schedule process actions so that conflicting actions cannot be executed simultaneously. Also, the fairness assumption for self-stabilization becomes a requirement for daemons to support self-stabilizing protocols: if live processes need to take steps, daemons must be able to schedule these processes to take all subsequent steps eventually.

1.1.3. Self-Stabilizing Daemons

If daemons are also subject to transient faults, then the daemons must be self-stabilizing as well to support self-stabilizing protocols. This is necessary for the following reasons.

First, transient faults to variables in a non-stabilizing daemon could result in a deadlock, by which some processes may be prevented from taking steps in the protocol being scheduled. If processes cannot take steps when they need to, they may not be able to take necessary steps to converge after transient faults.

Second, a non-stabilizing daemon may stay forever in unsafe configurations after a transient fault. In unsafe configurations, the daemon may make infinitely many scheduling mistakes, which may prevent the protocols from self-stabilizing. Although a scheduling mistake in some cases may be viewed as a transient fault to self-stabilizing protocols, the protocols are not guaranteed to tolerate infinitely many transient faults, which may be caused by infinitely many scheduling mistakes. Hence, self-stabilizing daemons are necessary to support self-stabilizing protocols.

1.1.4. Wait-Free Daemons

Crash faults may occur to daemons as well. A crash fault occurs when a process ceases execution without warning and never recovers [5]. Unlike transient faults, crash faults are permanent. We say that a process p is *correct* if p never crashes. Otherwise, p is *faulty* and eventually crashes. Additionally, a faulty process is *live* only prior to its crash, but a correct process is *live* forever.

Unfortunately, many daemons [4, 6, 7, 8, 9] are designed to tolerate transient faults only, and most of them do not address the pragmatic possibility of crash faults.

When a process crashes, these daemons may prevent correct processes from taking steps and cannot establish the fairness assumption for self-stabilization. As a result, self-stabilization cannot be guaranteed for the protocols being scheduled.

When a distributed daemon is subject to crash faults, the daemon is required to be *wait-free* to guarantee self-stabilization of the protocols being scheduled. Wait-free daemons guarantee that when a correct process needs to take steps, the process will eventually be scheduled to take such steps [10]. As such, wait-freedom establishes the fairness assumption for self-stabilization, and hence, wait-freedom is necessary for daemons to support self-stabilizing protocols.

Daemons are usually implemented by solutions to dining philosophers. Therefore, we will introduce dining philosophers in the next section.

1.1.5. Dining Philosophers

Dining philosophers [11, 12] (or *dining* for short) is a classic model of static resource-allocation scenarios, in which diners (processes) potentially need to access a fixed subset of shared resources periodically. The term "*static*" means that every time a diner needs to utilize shared resources, the diner always requests the same subset of shared resources. If two processes p and q have overlapping resource needs, then we say that there is *a potential conflict* between p and q. There is no restriction on when diners need to access shared resources. In some executions, some diners do not even need to access shared resources at all.

Each dining instance can be modeled as a *conflict graph* $G = (\Pi, E)$, where each vertex $p \in \Pi$ represents a diner (process), and each edge $(p, q) \in E$ indicates a potential conflict between p and q. Hence, no two neighbors in a conflict graph can utilize shared resources simultaneously. Also, the topology of conflict graphs can be arbitrary. Thereafter by default, *neighbors* stand for *neighbors in conflict graphs*.

For a diner, its *critical section* is a segment of code that should be executed atomically and exclusively. A critical section usually includes operations on some exclusively shared resources. In the examples from Section 1.1.2, the code that updates shared registers and the code that sends messages via shared wireless channels are critical sections. As such, no two neighbors execute their critical sections simultaneously.

At any time, each diner is either *thinking, hungry, eating*, or *exiting*. These four states correspond to four basic computation phases of a diner: executing independently, requesting access to a critical section, executing a critical section, and exiting a critical section, respectively.

Diners cycle through these four states periodically as shown in Figure 1. Initially, every process is thinking. Processes can think forever, but they can become hungry at any time. Hungry processes are scheduled to eat by dining algorithms. Correct processes can eat only for a finite period of time. As such, correct eating processes eventually finish eating and exit their critical sections. After they successfully exit their critical sections, they transit back to thinking.

State transitions of diners can be further classified as *input actions* and *output actions* [13] as shown in Figure 1. Input actions are activated by diners themselves and include the state transitions from *thinking* to *hungry* and from *eating* to *exiting.* The transition from *thinking* to *hungry* indicates that the diner wants to enter a critical section, and the transition from *eating* to *exiting* indicates that the diner wants to exit a critical section. By contrast, dining solutions decide when an output action can be executed. The code to execute output actions is a part of dining solutions. There are two output actions: the state transitions from *hungry* to *eating* and from *exiting* to *thinking.* When a dining solution decides that a hungry diner can enter its critical section, the diner transits from *hungry* to *eating.* When a dining solution decides that an exiting diner can transit back to thinking, the diner transits from *exiting* to *thinking*.

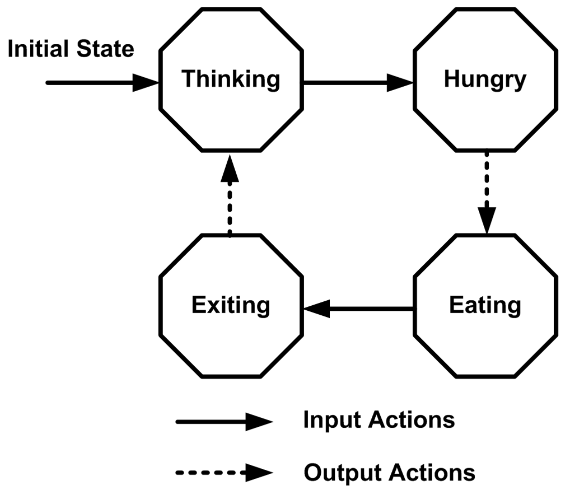


Fig. 1. State Transitions of Diners

Dining philosophers can also be viewed as a model of process scheduling, and hence, dining algorithms can be used to implement distributed daemons. Each process in a distributed system is represented as a diner in the dining problem. Local algorithms to be scheduled at each process are modeled as a set of actions. Conflicting actions can be represented as critical sections. To implement distributed daemons, a dining solution usually guarantees two properties: (1) every hungry process eventually eats, and (2) no two neighbors eat simultaneously. The first property guarantees that if a process needs to execute a conflicting action, the process will be eventually scheduled to do so. The second property guarantees that no conflicting actions can be scheduled concurrently.

1.1.6. Wait-Free, Eventually Bounded-Fair Dining Philosophers

Our research is motivated by implementing *wait-free, eventually bounded-fair daemons* (WF-EBF daemons) [14] in distributed message-passing systems. We assume that WF-EBF daemons are subject to crash faults only for the following reasons. First, most daemons [4, 6, 7, 8, 9] were designed to tolerate transient faults, not crash faults. As discussed before, it is necessary to address the pragmatic possibility of crash faults. Second, it would be ideal to tolerate both transient faults and crash faults. However, this is not necessary in some applications. For example, in wireless sensor networks, nodes are guaranteed to crash because of limited power supply, but the probability of transient faults may be very low. In this case, it is necessary to deal with crash faults; otherwise, a crashed node may prevent other nodes behaving correctly. However, it is unnecessary to deal with transient faults because it is highly unlikely that transient faults will occur.

In the context of dining philosophers, WF-EBF daemons are defined by three properties as follows.

- **Eventual Weak Exclusion:** *For each execution, there exists an unknown time after which no two live neighbors eat simultaneously.*

- **Wait-Freedom:** *Every correct hungry process eventually eats, regardless of how many processes crash.*

- **Eventual k-bounded Waiting:** *There exists a natural number k, such that for each execution, there exists a time t after which no live process can be overtaken more than k times by any neighbor.* A hungry process p is *overtaken* by a neighbor q each time that q gets scheduled to eat while p remains continuously hungry [15, 16].

Eventual Weak Exclusion($\Diamond\mathcal{WX}$) [17, 18] allows live neighbors to eat simultaneously finitely many times. Intuitively, $\Diamond\mathcal{WX}$ defines a class of unreliable daemons that can make finitely many mistakes during any execution by wrongfully scheduling live neighbors to eat simultaneously. However, $\Diamond\mathcal{WX}$ eventually stops making mistakes. As such, each execution under $\Diamond\mathcal{WX}$ eventually converges, after which no scheduling mistake can be made. The convergence time is unknown and may vary from execution to execution.

Although $\Diamond\mathcal{WX}$ seems too permissive to be useful, $\Diamond\mathcal{WX}$ is actually a meaningful abstraction for process scheduling. We argue that $\Diamond\mathcal{WX}$ is still sufficient for underlying daemons to support self-stabilizing protocols.

First, self-stabilizing daemons already support $\Diamond\mathcal{WX}$ implicitly. Transient faults can drive self-stabilizing daemons into unsafe configurations in which the daemons may make scheduling mistakes. For example, consider a ring-based mutual exclusion algorithm [1] in which a process can eat only when it holds a unique token that circulates in the ring. Mutual exclusion is a special case of dining philosophers on a completely connected conflict graph. If a transient fault duplicates the token, then two live neighbors might eat simultaneously, and hence scheduling mistakes occur. However, self-stabilizing daemons recover from unsafe configurations in finitely many steps. Given that transient faults are finitely many, self-stabilizing daemons can make, at most, finitely many scheduling mistakes. Hence, self-stabilizing daemons already imply $\Diamond\mathcal{WX}$.

Second, if a daemon wrongfully schedules two processes to utilize shared resources simultaneously, then such a scheduling mistake might be viewed as a transient fault to the self-stabilizing protocols being scheduled. The two examples in Section 1.1.2 (i.e., two processes simultaneously execute write operations to a shared register and communicate via a shared wireless channel) already illustrate the potential impact of scheduling mistakes when a daemon wrongfully schedules two processes to utilize exclusive resources simultaneously. Given that there can be only finitely many such scheduling mistakes, there can be only finitely many transient faults on the protocols. The protocols can correct these transient faults. Thus, for systems in which scheduling mistakes can be considered transient faults, $\Diamond\mathcal{WX}$ is still sufficient for underlying daemons to support self-stabilizing protocols.

Wait-Freedom [10] is designed to tolerate crash faults. In the context of dining philosophers, wait-freedom guarantees that every correct hungry process eventually eats, regardless of how many processes crash. In the worst case, where $n-1$ processes crash in a system of n processes, the remaining correct process can still make progress to eat. However, wait-freedom does not impose an upper bound on how many steps are needed for a correct hungry process to eat.

As discussed before, for a daemon subject to crash faults, wait-freedom is necessary for the daemon to support self-stabilizing protocols. Without wait-freedom, the daemon may never schedule correct processes to take steps in the presence of crash faults. When a transient fault occurs in protocol layers, correct processes may not be able to take necessary steps to tolerate the transient fault.

Eventual k-Bounded Waiting($\Diamond k$-$\mathcal{BW}$) is a form of bounded fairness. We say that a hungry process p is *overtaken* by a neighbor q, if q *goes to eat* while p remains continuously hungry [15, 16]. Note that "q goes to eat" differs from "q is eating" and "q eats". The expression "q goes to eat" emphasizes the moment when q *enters* its critical section, while "q is eating" and "q eats" emphasize the time periods during which q executes its critical section. Eventual k-bounded waiting($\Diamond k$-$\mathcal{BW}$) guarantees that for each execution, there exists a time t after which no live process goes to eat more than k times, while any live neighbor is waiting to eat. The fairness bound k is a natural number for all executions, and k may or may not hold during finite prefixes of each execution. However, each execution eventually converges at some time after which the fairness bound k holds. The convergence time t is unknown and may vary from execution to execution.

Introducing $\Diamond k$-$\mathcal{BW}$ into daemons is more about practical concerns to improve the convergence speed of stabilizing protocols. The convergence speed of a transient fault is usually measured by *asynchronous rounds* necessary to recover from the transient fault. In each asynchronous round, every process takes at least one step [2]. When a transient fault occurs, some processes should take actions to correct the transient fault. Wait-freedom only guarantees that these correcting actions will eventually be scheduled, but $\Diamond k$-$\mathcal{BW}$ can further guarantee that during an infinite suffix, while a process is waiting to execute a correcting action, no other neighbor can execute more than k actions. Hence, during this infinite suffix, when the system is recovering from transient faults, the number of steps in each asynchronous round is reduced in the worst case.

Non-correcting actions may propagate the effects of previous transient faults to other parts of the underlying systems. With $\Diamond k$-$\mathcal{BW}$, correcting actions may be scheduled earlier and even before non-correcting actions propagate the effects of previous transient faults. Hence, the effect of transient faults may not be propagated as fast as it would have otherwise. Thus, the convergence speed of transient faults in the worst case can also be improved from this perspective.

A natural question is: why not introduce perpetual k-bounded waiting ($\Box k$-$\mathcal{BW}$)? $\Box k$-$\mathcal{BW}$ guarantees that for all executions, no live process can be overtaken more than k times by any neighbor ever. We argue that it is useless to ensure $\Box k$-$\mathcal{BW}$ under $\Diamond\mathcal{WX}$. $\Diamond\mathcal{WX}$ can make scheduling mistakes during finite prefixes. As such, $\Diamond\mathcal{WX}$ may wrongfully schedule actions during finite prefixes of any execution; thus, guaranteeing fairness during these finite prefixes is meaningless. Also, implementing $\Box k$-$\mathcal{BW}$

requires stronger assumptions on underlying systems than $\Diamond k$-$\mathcal{BW}$.

1.2. Background

To establish our research goal, this section will briefly review the necessary background about solvability of wait-free dining. Solvability of wait-free dining depends on the synchronism of underlying systems. In short, wait-free dining cannot be solved in asynchronous systems, may or may not be solved in partially synchronous systems, and can be solved in synchronous systems.

Asynchronous message-passing systems assume nothing about timing properties [19], such as message delivery time and relative process speeds. In such a system, messages can be arbitrarily delayed, and processes can be arbitrarily slow. As such, algorithms designed for asynchronous systems (i.e., asynchronous algorithms) cannot utilize any timing assumptions.

It is known that wait-free dining cannot be solved deterministically in asynchronous message-passing systems subject to crash faults [20]. This impossibility result is based on the intrinsic difficulty of reliable fault detection in asynchronous systems. Assume that there exists a wait-free asynchronous dining algorithm. For a system of two neighbors p and q, where q is correct, let us consider a configuration C in which p is eating and q is hungry. In one future extension from C, p crashes immediately. According to wait-freedom, q eventually eats at a time t. In another extension, p is also correct and never crashes. By asynchrony, messages can be arbitrarily delayed. Hence, it is possible that q receives no messages before time t in both extensions. As such, from the perspective of process q, both extensions are indistinguishable before time t, and q does not know whether p is crashed or still live. Because the algorithm is deterministic, q has to behave the same in both extensions, and hence has to eat at time t in both extensions. If p is still eating at time t in the later extension, then the mutual exclusion property is violated. Therefore, wait-freedom is impossible for asynchronous dining algorithms because processes cannot reliably detect crash faults in asynchronous systems.

Wait-free dining can be solved in *synchronous* message-passing systems, which have the strongest timing assumptions. Synchrony assumes that there exist a *known* upper bound Δ on message delivery time and a *known* upper bound Φ on relative process speeds [21, 22]. As such, we can design a timeout mechanism with known timeout durations, and during each duration, each correct process p is guaranteed to receive a message from each correct process. By this mechanism, processes can reliably detect and consequently react to crash faults, and wait-free dining can be solved. However, synchrony is overkill for wait-free dining; completely reliable fault detection is sufficient but unnecessary.

Most real systems fall into the category of *partial synchrony* between pure synchrony and pure asynchrony. Partially synchronous systems still satisfy some timing assumptions, but those assumptions are not as strong as those of pure synchrony. For example, a partially synchronous system $\mathcal{M}_1$ may assume that the upper bounds on both message delivery time and relative process speeds are *unknown*.

Wait-free dining may or may not be solvable in a partially synchronous system. The solvability of wait-free dining depends on the fault detection implementable in the underlying system. A partially synchronous system usually assumes limited knowledge on timing assumptions (i.e., the bounds on timing properties are unknown and/or hold after some unknown time). The limited knowledge prevents us from implementing completely reliable fault detection. For a partially synchronous system, its timing assumptions decide the strength of implementable fault detection. For example, the system $\mathcal{M}_1$ can be used to implement eventually reliable fault detection, which guarantees that every crash fault will be detected by all correct processes eventually and permanently, and each execution has an infinite suffix during which no correct process is suspected by any correct process. As such, solvability of wait-free dining eventually depends on the underlying timing assumptions. We will explore the necessary and sufficient timing assumptions to solve WF-EBF dining.

1.3. Research Goal

Our research goal is to explore the necessary and sufficient timing assumptions to solve WF-EBF dining in distributed message-passing systems. Specifically, eventually reliable fault detection encapsulates the minimal synchronism required to solve WF-EBF dining. That is, WF-EBF dining can be solved if and only if eventually reliable fault detection can be implemented in the underlying system.

We also explore limitations of eventually reliable fault detection on the solvability of wait-free dining. First, eventually reliable fault detection cannot deterministically solve wait-free, *perpetually* bounded-fair dining under eventual weak exclusion. Perpetual bounded fairness guarantees that there exists a fairness bound k such that during every execution, no live process goes to eat more than k times, while any of its neighbors waits to eat. Second, there does not exist a *thinking-quiescent* algorithm that uses eventually reliable fault detection to deterministically solve WF-EBF dining. Thinking quiescence guarantees that if a process remains thinking permanently after some time, then this process eventually stops sending and receiving messages. This impossibility result implies that to achieve bounded fairness, an intrinsic performance penalty is unavoidable, so that if a correct process eats infinitely often, then its correct neighbors must send and receive infinitely many messages.

1.4. Methodology

Explicit timing assumptions are not suitable for exploring our research goal. Recall that explicit timing assumptions ensure the existence of bounds on timing properties. Assume that we encode an explicit timing assumption (e.g., bounds on message delivery times) into an algorithm to solve WF-EBF dining. This explicit timing assumption is sufficient to solve WF-EBF dining, but is not necessary. We may also be able to use another explicit timing assumption (e.g., bounds on relative process speeds) to solve WF-EBF dining. The solvability of WF-EBF dining indeed depends on the failure detection that can be implemented in the underlying system. Explicit timing assumptions tend to obscure the fundamental properties of fault detection necessary or sufficient to solve a given problem. Also, an algorithm that explicitly uses a timing assumption (e.g., bounds on message delivery times) may not be applied to the systems with another timing assumption (e.g., bounds on relative process speeds). As such, we will not use explicit timing assumptions to explore our research goal.

We explore our research goal by augmenting asynchronous systems with failure detectors, which can be used to encapsulate *implicit* timing assumptions. Informally, an unreliable failure detector can be viewed as a distributed oracle that can be queried for (potentially unreliable) information about process crashes [23]. Most failure detectors cannot be implemented in asynchronous systems. In general, their implementations rely on timing assumptions of underlying systems. Therefore, failure detectors can be viewed as an encapsulation of *implicit* timing assumptions necessary to implement abstract detection properties. Instead of dealing with the actual operational features of partial synchrony (e.g., bounds on timing properties), algorithms using failure detectors need only consider the axiomatic features of the failure detectors. As such, failure detectors provide an essential separation of concerns between fault detection properties and their underlying implementation mechanisms.

We consider the local eventually perfect failure detector $\diamond\mathcal{P}_1$ ("diamond P sub 1") as the detector that encapsulates the necessary and sufficient timing assumptions to solve WF-EBF dining. $\diamond\mathcal{P}_1$ is defined with respect to the underlying communication graph in which each node represents a process, and each edge (p, q) represents a bidirectional communication channel between processes p and q. Also, communication neighbors represent neighbors in the communication graph. The failure detector $\diamond\mathcal{P}_1$ is specified as follows [24, 25]:

- **Local Perpetual Strong Completeness:** *Every crashed process is eventually and permanently suspected by all of its correct communication neighbors.*

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

Informally, $\diamond\mathcal{P}_1$ suspects crashed communication neighbors eventually and permanently, but may make mistakes by wrongfully suspecting each correct communication neighbor finitely many times during any execution. However, $\diamond\mathcal{P}_1$ *eventually* stops suspecting correct communication neighbors. The time for $\diamond\mathcal{P}_1$ to converge (i.e., $\diamond\mathcal{P}$ stops wrongfully suspecting correct communication neighbors) is unknown and may vary from execution to execution.

$\diamond\mathcal{P}_1$ is a locally scope-restricted refinement of the eventually perfect failure detector $\diamond\mathcal{P}$. The failure detector $\diamond\mathcal{P}$ suspects crashed processes eventually and permanently, but may make mistakes by wrongfully suspecting correct processes finitely many times during any execution. $\diamond\mathcal{P}_1$ satisfies the properties of $\diamond\mathcal{P}$, but only with respect to immediate neighbors in the communication graph. The difference between $\diamond\mathcal{P}_1$ and $\diamond\mathcal{P}$ will be discussed further in Chapter III.

Our ultimate research goal then becomes to prove that the failure detector $\diamond\mathcal{P}_1$ is the weakest failure detector for solving WF-EBF dining. That is, $\diamond\mathcal{P}_1$ is both *sufficient* and *necessary* for solving WF-EBF dining. This ultimate goal is achieved by two steps. First, to prove that $\diamond\mathcal{P}_1$ is sufficient to solve WF-EBF dining, we will construct an algorithm that solves WF-EBF dining using $\diamond\mathcal{P}_1$. Second, to prove that $\diamond\mathcal{P}_1$ is also necessary, we will show that every failure detector that solves WF-EBF dining is at least as strong as $\diamond\mathcal{P}_1$. That is, $\diamond\mathcal{P}_1$ can be implemented in every system in which WF-EBF dining can be solved.

To prove that a failure detector $\mathcal{D}$ is *necessary* to solve a problem $\mathcal{B}$, a commonly-used technique is to construct a reduction of $\mathcal{D}$ to the problem $\mathcal{B}$ itself. Such a reduction algorithm implements $\mathcal{D}$ using any solution to $\mathcal{B}$. The proof technique is shown in Figure 2. Consider a failure detector $\mathcal{D}'$ and a black-box solution $\mathcal{A}'$ that uses $\mathcal{D}'$ to solve problem $\mathcal{B}$. Next, suppose there exists a reduction algorithm T that implements $\mathcal{D}$ using any solution to $\mathcal{B}$. As shown in Figure 2, the failure detector $\mathcal{D}'$ can implement the solution $\mathcal{A}'$, and $\mathcal{A}'$ can, in turn, implement $\mathcal{D}$. By transitivity, $\mathcal{D}'$ can be used to implement $\mathcal{D}$. Therefore, $\mathcal{D}'$ is at least as strong as $\mathcal{D}$. Thus, $\mathcal{D}$ is, in fact, necessary to solve the problem $\mathcal{B}$.

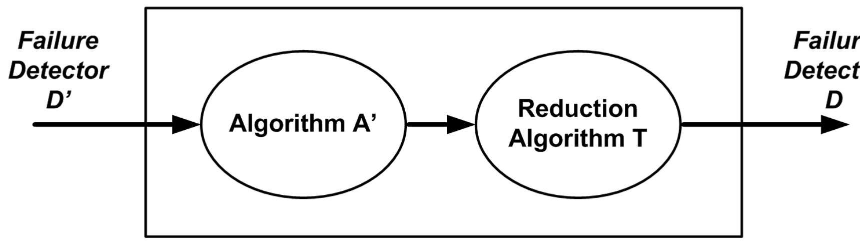


Fig. 2. The Diagram to Prove Necessity by Reduction

In our case, we will construct a reduction algorithm that uses any WF-EBF dining solution to implement $\diamond\mathcal{P}_1$. WF-EBF dining solutions are simply used as a black box, and the reduction algorithm does not know or utilize any internal feature of such solutions. The reduction only utilizes the properties (i.e., wait-freedom, eventual weak exclusion, and eventual k-bounded waiting) of the black-box solutions.

1.5. Outline

The remaining chapters are organized as follows:

Chapter II describes the underlying system from three aspects: the distributed system model, fault model, and timing assumptions. This chapter also defines many system-related notations, such as configurations, actions, and fault patterns.

Chapter III formally defines failure detectors and many related notations, such as solvability, reducibility, schedules, and executions. This chapter also introduces many important failure detectors, such as the Chandra-Toueg failure detectors. Additionally, we discuss implementations of failure detectors in this chapter.

Chapter IV introduces dining variants with different safety and progress properties. This chapter also reviews solvability of wait-free dining with respect to different safety properties and systems. Finally, we briefly introduce other classic paradigms of resource allocation problems.

Chapter V presents a wait-free dining algorithm under $\Diamond\mathcal{WX}$ using $\Diamond\mathcal{P}_1$. This relatively simple algorithm gives some insight into how to design wait-free dining algorithms using failure detectors and how to prove correctness.

Chapter VI formally defines bounded-fairness properties and proves that $\Diamond\mathcal{P}$ cannot deterministically solve wait-free, perpetual k-bounded waiting dining under eventual weak exclusion. As such, we are forced to consider a weaker fairness property, eventual k-bounded waiting, in the next chapter.

Chapter VII presents a WF-EBF algorithm using $\Diamond\mathcal{P}_1$. Hence, this chapter demonstrates that $\diamond\mathcal{P}_1$ is sufficient to solve WF-EBF dining. This algorithm is based on the asynchronous doorway algorithm of Choy and Singh [26].

Chapter VIII proves that there does not exist a thinking-quiescent algorithm that uses $\diamond\mathcal{P}$ to deterministically solve WF-EBF dining. This impossibility result demonstrates a limitation of $\diamond\mathcal{P}$ on solving wait-free dining, and implies an intrinsic performance penalty on achieving bounded fairness such that correct processes might need to send and receive infinitely many messages.

Chapter IX constructs a reduction algorithm that can use any WF-EBF solution to implement $\diamond\mathcal{P}_1$. Therefore, $\diamond\mathcal{P}_1$ is necessary for solving WF-EBF dining, and $\diamond\mathcal{P}_1$ is the weakest failure detector for solving WF-EBF dining.

Finally, Chapter X summarizes our research contributions and presents several open problems.

CHAPTER II

UNDERLYING SYSTEMS

This chapter describes the underlying systems upon which the algorithms and proofs in this dissertation are built. The underlying systems can be modeled from three aspects: the distributed system model, fault model, and timing assumptions. We begin by presenting the distributed system model, which is a distributed message-passing system with reliable first-in-first-out(FIFO) communication channels. Next, we introduce the underlying fault model, by which only crash faults can occur. Finally, we assume that timing assumptions in the underlying system are sufficient to implement the failure detector $\diamond\mathcal{P}_1$. However, these timing assumptions are not necessarily required in all layers of the system.

2.1. Distributed System Model

The underlying system is modeled as a distributed message-passing system composed of a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that communicate only by message passing. A process is modeled as a finite set of local variables, together with a finite set of actions that may read or write these variables. All variables are local; that is, no process can access or modify variables at any other process.

We assume that processes communicate through reliable bidirectional FIFO channels only. As such, messages are neither lost, duplicated, nor corrupted. Also, every message sent to a correct process p via a channel is eventually received by p in the order the message was sent. We use a *communication graph* to model inter-process connections created by communication channels. In a communication graph, each node represents a process, and each edge (p, q) represents a bidirectional FIFO channel between processes p and q. We also assume that for a dining instance, its conflict graph is a subgraph of the underlying communication graph. In other words, each pair of neighbors in the conflict graph must be connected by at least one FIFO channel.

Configurations of the underlying system are a combination of the local state of all processes, and the state of all communication channels [19]. The local state of a process is determined by the values of its local variables. The state of a communication channel is expressed as the set of messages which is in transit in this channel.

An *action* ϕ transforms a system from one configuration $\mathcal{C}$ to another configuration $\mathcal{C}'$, where configurations $\mathcal{C}$ and $\mathcal{C}'$ are allowed to be the same. We use $\phi(\mathcal{C})$ to denote the configuration resulting from the action ϕ, and hence, $\mathcal{C}' = \phi(\mathcal{C})$. An action is performed by a single process p and can be expressed as a guarded command,

$$\langle guard \rangle \rightarrow \langle command \rangle$$

where $\langle guard \rangle$ is a predicate on the local state of process p, and $\langle command \rangle$ is a finite sequence of executable statements [27]. We say that an action is *enabled in* (or *applicable to*) a configuration if and only if its $\langle guard \rangle$ evaluates to true in that configuration. Only enabled actions can be scheduled to execute their $\langle command \rangle$. We also assume that the underlying system satisfies weak fairness so that if an action is continuously enabled, then this action will eventually be scheduled. Furthermore, we assume that all actions are *atomic*; that is, action commands must either be finished completely or not be executed at all.

To simplify our presentation, we posit a discrete global clock, a conceptual device which cannot be accessed by processes in Π. The range of clock ticks is the set of natural numbers, denoted as $\mathbb{T}$.

2.2. Fault Model

We assume that only crash faults can occur in the underlying system. Recall that a crash fault occurs when a process ceases execution without warning and never recovers [5]. Also, processes cannot crash at will and have no prior knowledge about where and when crash faults occur in executions.

Faults can be modeled by augmenting the set of program actions with a supplemental set of fault actions [28]. In our case, crash faults can be modeled by adding a local boolean variable up_p and a fault action to each process p, where up_p is initially true. The fault action is shown as follows:

$$\mathsf{true} \rightarrow up_p := \mathsf{false}$$

This fault action occurs if and only if process p crashes. Unlike program actions, fault actions are not required to satisfy weak fairness; otherwise, every process must crash eventually. Because processes cannot crash at will, up_p cannot be modified by program actions. [1] Because crashed processes cannot execute any action, program actions are enabled only when up is true, and they can be remodeled as

$$\langle guard \rangle \wedge up \rightarrow \langle command \rangle$$

We use *fault patterns* to model the occurrence of crash faults in a given run. Formally, a fault pattern F is defined as a function from the range $\mathbb{T}$ of the global clock to the powerset of processes 2^{Π} (i.e., the set of all subsets of Π). For example, $F(t)$ represents the subset of processes that has crashed by time t in fault pattern F. Since crash faults are permanent, $F(t)$ is monotonically non-decreasing; that is, $\forall t \in \mathbb{T} : F(t) \subseteq F(t+1)$.

[1]The variable up is introduced here only for modeling crash faults. It will not be used in the following chapters.

In a given run, a process is either *faulty* or *correct*. Formally, a process p is faulty in a fault pattern F if and only if p crashes at some time t; that is, $\exists t \in \mathbb{T} : p \in F(t)$. The set of faulty processes is denoted $faulty(F)$, where $faulty(F) = \cup_{t \in \mathbb{T}} F(t)$. By contrast, a process p is correct in F if and only if p never crashes; that is, $\forall t \in \mathbb{T} : p \notin F(t)$. We denote the set of correct processes as $correct(F)$, where $correct(F) = \Pi - faulty(F)$. Additionally, we say that p is live at time t (i.e., $p \in live(t)$) if and only if p has not crashed by time t (i.e., $p \notin F(t)$). Thus, correct processes are always live, and faulty processes are live only prior to crashing.

2.3. Timing Assumptions

We consider the underlying system as a partially synchronous system in which the failure detector $\diamond\mathcal{P}_1$ can be implemented. The failure detector $\diamond\mathcal{P}_1$ cannot be implemented in purely asynchronous systems, in which messages can be arbitrarily delayed and process speeds can be arbitrarily slow. As such, using $\diamond\mathcal{P}_1$ implies that the underlying system must satisfy some timing assumptions (e.g., bounded message delays or bounded relative process speeds).

The underlying system can also be viewed as an asynchronous system augmented with $\diamond\mathcal{P}_1$. As such, no explicit timing assumptions can be directly used in our algorithms and proofs. Hence, our research can focus on axiomatic features of failure detectors, instead of operational features of timing assumptions.

Using $\diamond\mathcal{P}_1$ does not necessarily mean that the timing assumptions must hold on all layers of the underlying system. We argue that some layers in systems augmented with $\diamond\mathcal{P}_1$ may still be purely asynchronous. For example, while the layer to implement $\diamond\mathcal{P}_1$ must be partially synchronous, application layers may still be asynchronous. This argument can be clarified from two aspects: message delivery time and relative process speeds.

Failure-detector messages (i.e., messages used to detect crash faults) and application messages can be treated differently by the communication medium. For example, failure-detector messages are essentially control messages for implementing a system service for crash-fault detection. As such, failure-detector messages may be routed with higher priorities to satisfy prerequisite assumptions on partial synchrony. By contrast, application messages may be routed with lower priorities and subject to arbitrary delays arising from such factors as lower bandwidth, load-balanced routing, queuing delays, and unfavorable quality-of-service guarantees. Therefore, application messages are allowed to be arbitrarily delayed in asynchronous systems augmented with $\Diamond\mathcal{P}_1$.

Similarly, fault-detection tasks and applications can be treated differently by processes as well. Processes may give higher priorities to fault detection tasks and lower priorities to applications. Applications may experience arbitrary step delays due to such factors as cache misses, page faults, server loads, and preemptive priority-based CPU scheduling. Applications, therefore, may be executed arbitrarily slowly.

CHAPTER III

FAILURE DETECTORS

This chapter introduces failure detectors, a concept originally defined by Chandra and Toueg [23]. Recall that we already gave an intuitive definition of failure detectors in Chapter I. An unreliable failure detector can be viewed as a distributed oracle that can be queried for possibly incorrect information about crash faults. This chapter will give the formal definition and introduce related concepts, including algorithms, schedules, executions, solvability, reducibility, and weakest failure detectors. This chapter will also introduce the hierarchy of Chandra-Toueg failure detectors [23], the local eventually perfect failure detector $\diamond\mathcal{P}_1$ [14, 18], and the trusting failure detector $\mathcal{T}$ [29]. Finally, we will also discuss implementations of failure detectors.

3.1. Formal Definition

For each failure detector, there is a range $\mathcal{R}$ of values output by the failure detector. In this dissertation, failure detectors output a set of processes currently suspected of having crashed. Hence, $\mathcal{R}$ is the powerset 2^{Π}. [1]

A *failure detector history* H with range $\mathcal{R}$ is a function that maps $\Pi \times \mathbb{T}$ to the range $\mathcal{R}$. For example, $H(p,t)$ denotes the output of the failure detector module at process p at time t, and $q \in H(p,t)$ means that process p suspects process q at time

[1]Not all failure detectors output a set of suspected processes. Some failure detectors output a set of trusted processes, such as the failure detector Ω in [30] that outputs only one trusted process. Some failure detectors do not even provide binary information about crashes (i.e., suspected or trusted), such as the accrual failure detector [31] that provides a suspicion level on a continuous scale for processes.

t in H. The failure detector modules at two different processes need not output the same value. Hence, for two processes, p and q, $H(p, t)$ may or may not be equal to $H(q, t)$.

Informally, a *failure detector* provides (possibly incorrect) information about the fault pattern in a given execution. Formally, a failure detector $\mathcal{D}$ is a function that maps each fault pattern F to a set of failure detector histories $\mathcal{D}(F)$ with range $\mathcal{R}_{\mathcal{D}}$, where $\mathcal{D}(F)$ is the set of all possible failure detector histories permitted by $\mathcal{D}$ for the fault pattern F, and $\mathcal{R}_{\mathcal{D}}$ is the range of values output by failure detector $\mathcal{D}$. During each execution, $\mathcal{D}$ outputs only one failure detector history. During two or more executions with the same fault pattern, $\mathcal{D}$ may be permitted to output different failure detector histories from the set $\mathcal{D}(F)$. Also, a *failure detector class* can be specified by a set of properties that every failure detector in this class must satisfy.

3.2. Algorithms, Schedules, and Executions

An *algorithm* $\mathcal{A}$ is defined by a set of automata, one for each process in Π. For algorithms, computation proceeds in steps. Each step is an action in which a process p may (1) receive a message to p, and/or (2) get a value from its failure detector module, and/or (3) send a message, and/or (4) change its variable values. We assume that each step is executed atomically.

An algorithm is *deterministic* if and only if for each process, its state transition is determined totally by its current state and received messages. For a deterministic algorithm, its behaviors are completely determined by the input and the current system configuration. Given a particular input and a particular configuration, deterministic algorithms always produce the same output.

A *schedule* is a finite or infinite sequence of actions (steps). A schedule S is denoted $S = S_0, S_1, S_2...$, where $S_0, S_1, S_2...$ are actions. If the first action S_0 is enabled in a configuration C, and each action S_{i+1} is also enabled in the configuration resulting from the previous action S_i, then we say that the schedule S is applicable to the configuration C. For deterministic algorithms, applying a finite schedule S to a configuration C uniquely determines the final configuration, denoted $S(C)$.

An *execution* (or run) of an algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ is expressed as a 5-tuple $R = \langle F, H_{\mathcal{D}}, I, S, T \rangle$, where F is the fault pattern of this execution, $H_{\mathcal{D}}$ is the failure detector history output by $\mathcal{D}$ in this execution, I is the initial configuration, S is an infinite schedule which is applicable to I, and T is an infinite sequence of increasing time values denoting when each action in S occurs.

An *execution segment* of an algorithm $\mathcal{A}$ using a failure detector $\mathcal{D}$ is also expressed as a 5-tuple $R = \langle F, H_{\mathcal{D}}, C, S, T \rangle$, where C is the starting configuration of this segment (and may or may not be the initial configuration of this execution), S is a finite schedule which is applicable to the configuration C, T is a finite sequence of increasing time values denoting when each action in S occurs, and $|S| = |T|$. If C is the initial configuration of this run, then R is called a *partial run* or *a finite prefix* of this run. For deterministic algorithms, applying a finite schedule S to a configuration C uniquely determines an execution segment, denoted $exec(C, S) = \langle F, H_{\mathcal{D}}, C, S, T \rangle$.

3.3. Solvability

A problem $\mathcal{B}$ is defined by a set of properties that every run must satisfy. An algorithm $\mathcal{A}$ solves problem $\mathcal{B}$ using a failure detector $\mathcal{D}$ if and only if all runs of $\mathcal{A}$ using $\mathcal{D}$ satisfy the properties that define $\mathcal{B}$, and we say that $\mathcal{D}$ is sufficient to solve the problem $\mathcal{B}$. Consequently, $\mathcal{B}$ can be solved in every system in which $\mathcal{D}$ can be implemented.

We say that a class of failure detectors $\mathcal{C}$ is sufficient to solve problem $\mathcal{B}$ (or $\mathcal{B}$ can be solved using $\mathcal{C}$) if, for every failure detector $\mathcal{D} \in \mathcal{C}$ (i.e., $\mathcal{D}$ is in the class $\mathcal{C}$), there exists an algorithm that solves $\mathcal{B}$ using $\mathcal{D}$. However, there may or may not exist an algorithm $\mathcal{A}$ that uses every failure detector $\mathcal{D} \in \mathcal{C}$ to solve $\mathcal{B}$. If such an algorithm $\mathcal{A}$ exists, then we say that algorithm $\mathcal{A}$ solves problem $\mathcal{B}$ using $\mathcal{C}$.

In some cases, whether a failure detector is sufficient to solve a problem depends on the underlying *environment*, which is defined by a set of possible fault patterns [30]. Some publications [23, 29] explore solvability with respect to a specific environment, and hence, they say that a failure detector class $\mathcal{C}$ is sufficient for solving a problem $\mathcal{B}$ *in an environment* $\mathcal{E}$. This dissertation, however, explores solvability of wait-free dining problems in any environment. To simplify the presentation, we just say "$\mathcal{C}$ is sufficient for solving problem $\mathcal{B}$" rather than "$\mathcal{C}$ is sufficient for solving problem $\mathcal{B}$ in an environment $\mathcal{E}$".

3.4. Reducibility

Given two failure detectors $\mathcal{D}$ and $\mathcal{D}'$, if there exists an algorithm $T_{\mathcal{D}' \to \mathcal{D}}$ that transforms $\mathcal{D}'$ to $\mathcal{D}$, then we write $\mathcal{D}' \succeq \mathcal{D}$ and say that $\mathcal{D}$ is *weaker than* or *reducible to* $\mathcal{D}'$, or $\mathcal{D}'$ is *stronger than* $\mathcal{D}$. The algorithm $T_{\mathcal{D}' \to \mathcal{D}}$ is called a reduction algorithm and uses $\mathcal{D}'$ to maintain a local variable $output_p$ at each process p. The variable $output_p$ emulates the output of $\mathcal{D}$ at p. Consequently, every problem that can be solved using $\mathcal{D}$ can also be solved using $\mathcal{D}'$.

Based on reducibility, the following relationships exist between two failure detectors $\mathcal{D}$ and $\mathcal{D}'$:

If $\mathcal{D}' \succeq \mathcal{D}$, then we say that $\mathcal{D}'$ is stronger than $\mathcal{D}$, or $\mathcal{D}$ is weaker than $\mathcal{D}'$.

If $\mathcal{D}' \succeq \mathcal{D}$ and $\mathcal{D} \not\succeq \mathcal{D}'$, then we write $\mathcal{D}' \succ \mathcal{D}$ and say that $\mathcal{D}'$ is *strictly* stronger than $\mathcal{D}$, or $\mathcal{D}$ is *strictly* weaker than $\mathcal{D}'$.

If $\mathcal{D}' \succeq \mathcal{D}$ and $\mathcal{D} \succeq \mathcal{D}'$, then we write $\mathcal{D}' \cong \mathcal{D}$ and say that $\mathcal{D}'$ and $\mathcal{D}$ are *equivalent.*

If $\mathcal{D}' \not\succeq \mathcal{D}$ and $\mathcal{D} \not\succeq \mathcal{D}'$, then we say that $\mathcal{D}$ and $\mathcal{D}'$ are *incomparable.*

The relational operators $\succeq$ and $\succ$ are transitive. For example, given three failure detectors $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$, if $\mathcal{D}_1 \succeq \mathcal{D}_2$ and $\mathcal{D}_2 \succeq \mathcal{D}_3$, then $\mathcal{D}_1 \succeq \mathcal{D}_3$. If $\mathcal{D}_1 \succ \mathcal{D}_2$ and $\mathcal{D}_2 \succ \mathcal{D}_3$, then $\mathcal{D}_1 \succ \mathcal{D}_3$.

For two classes of failure detectors $\mathcal{C}$ and $\mathcal{C}'$, if, for every failure detector $\mathcal{D}' \in \mathcal{C}'$, there exists a failure detector $\mathcal{D} \in \mathcal{C}$ such that $\mathcal{D}' \succeq \mathcal{D}$, then we write $\mathcal{C}' \succeq \mathcal{C}$ and say that $\mathcal{C}'$ is stronger than $\mathcal{C}$, or $\mathcal{C}$ is weaker than $\mathcal{C}'$. Consequently, every problem that can be solved using $\mathcal{C}$ can also be solved using $\mathcal{C}'$. The following relationships exist between two failure detector classes $\mathcal{C}$ and $\mathcal{C}'$:

If $\mathcal{C}' \succeq \mathcal{C}$, then we say that $\mathcal{C}'$ is stronger than $\mathcal{C}$, or $\mathcal{C}$ is weaker than $\mathcal{C}'$.

If $\mathcal{C}' \succeq \mathcal{C}$ and $\mathcal{C} \not\succeq \mathcal{C}'$, then we write $\mathcal{C}' \succ \mathcal{C}$ and say that $\mathcal{C}'$ is strictly stronger than $\mathcal{C}$, or $\mathcal{C}$ is strictly weaker than $\mathcal{C}'$.

If $\mathcal{C}' \succeq \mathcal{C}$ and $\mathcal{C} \succeq \mathcal{C}'$, then we write $\mathcal{C}' \cong \mathcal{C}$ and say that $\mathcal{C}'$ and $\mathcal{C}$ are equivalent.

If $\mathcal{C}' \not\succeq \mathcal{C}$ and $\mathcal{C} \not\succeq \mathcal{C}'$, then we say that $\mathcal{C}$ and $\mathcal{C}'$ are incomparable.

The relational operators $\succeq$ and $\succ$ are also transitive for failure detector classes.

3.5. Weakest Failure Detectors

Formally, a failure detector $\mathcal{D}$ is the weakest failure detector for solving a problem $\mathcal{B}$ if and only if $\mathcal{D}$ is necessary and sufficient for solving $\mathcal{B}$ [30].

- **Necessity:** *Every failure detector that is sufficient to solve $\mathcal{B}$ is stronger than $\mathcal{D}$.*

- **Sufficiency:** *There exists at least one algorithm that solves $\mathcal{B}$ using $\mathcal{D}$.*

The weakest failure detector $\mathcal{D}$ for a problem $\mathcal{B}$ encapsulates the necessary and sufficient timing assumptions to solve $\mathcal{B}$ [30]. That is, $\mathcal{D}$ can be implemented in all distributed systems in which $\mathcal{B}$ can be solved, and $\mathcal{B}$ can be solved in all systems in which $\mathcal{D}$ can be implemented. In other words, $\mathcal{D}$ represents the minimal requirements on synchronism needed to solve the problem $\mathcal{B}$.

There are many important weakest failure detector results [29, 30, 32, 33]. These results will be introduced later when the corresponding failure detectors are introduced.

3.6. The Chandra-Toueg Hierarchy

When Chandra and Toueg introduced the concept of failure detectors in [23], they also defined a hierarchy of eight failure detectors. These failure detectors were introduced to circumvent the impossibility result of consensus in faulty environments [19, 23]. To do so, several algorithms using Chandra-Toueg failure detectors were developed to solve consensus in faulty environments. Also, the Chandra-Toueg failure detector $\diamond\mathcal{W}$ was proved to be the weakest oracle for solving consensus with a majority of correct processes [30]. We introduce the Chandra-Toueg failure detectors here not only because they are the most frequently-used failure detectors, but also because they characterize fundamental properties of crash fault detection.

Chandra-Toueg failure detectors are characterized by the kinds of mistakes they can make. Mistakes include false negatives and false positives, where the former define the mistakes by which crashed processes are not suspected, and the latter define the mistakes by which correct processes are wrongfully suspected. As such, failure detector classes are usually defined by two properties: *completeness* which restricts false negatives, and *accuracy* which restricts false positives.

We will first define two completeness properties and four accuracy properties. Next, we will define eight classes of Chandra-Toueg failure detectors by combining these completeness and accuracy properties. The relationships among these failure detectors are also discussed.

3.6.1. Completeness

Completeness is either strong or weak. These properties are defined as follows.

Strong Completeness: Every faulty process is eventually and permanently suspected by all correct processes. Formally, if a failure detector $\mathcal{D}$ satisfies strong completeness, then

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{T}, \forall p \in faulty(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q, t').$$

Strong completeness prohibits false-negative mistakes. If a process crashes, then it will be suspected by every correct process eventually and permanently. However, strong completeness does not specify when a faulty process will be suspected. Faulty processes might be suspected even before they actually crash.

Weak Completeness: Each faulty process is eventually suspected by *some* correct process permanently. Formally, if a failure detector $\mathcal{D}$ satisfies weak completeness, then

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{T}, \forall p \in faulty(F), \exists q \in correct(F), \forall t' \geq t : p \in H(q, t').$$

Weak completeness permits (some) false-negative mistakes. However, for each faulty process p, there exists *some* correct process q that eventually suspects p permanently. Similarly, weak completeness does not specify when process p will be suspected by q.

3.6.1.1. Transformation from Weak Completeness to Strong Completeness

In environments where the underlying communication graph cannot be partitioned by crash faults, a failure detector with weak completeness can be transformed into a failure detector with strong completeness. This transformation algorithm [23] is called the *gossip* protocol and works as follows. Every process maintains two set variables, where *suspect* is the output of the failure detector with weak completeness, and *output* emulates the output of the failure detector with strong completeness. Every live process p periodically sends its suspect list $suspect_p$ to all other processes. When a live process q receives the list $suspect_p$ from p, q adds all processes of $suspect_p$ into $output_q$, and q removes p from $output_q$. The emulated output *output* satisfies strong completeness, and this transformation preserves the perpetual and eventual accuracy properties defined in the next section [23].

However, this transformation does not work when the underlying communication graph can be partitioned by crash faults. In such an environment, messages are not guaranteed to arrive at their destinations, and hence the transformation does not work correctly.

3.6.2. Accuracy

Accuracy is either *perpetual* or *eventual*, and is either *strong* or *weak*. Hence, there are four accuracy properties: *perpetual strong accuracy*, *perpetual weak accuracy*, *eventual strong accuracy*, *eventual weak accuracy*.

Perpetual Strong Accuracy: No live process is suspected by any live process. Formally, if a failure detector $\mathcal{D}$ satisfies perpetual strong accuracy, then

$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathbb{T}, \forall p, q \in live(t) : p \notin H(q,t).$

Perpetual strong accuracy prohibits false-positive mistakes. Correct processes cannot be suspected at any time, and faulty processes cannot be suspected before they actually crash. If a process is suspected, then this process must have already crashed.

Perpetual Weak Accuracy: There exists *some* correct process that is never suspected by any live process. Formally, if a failure detector $\mathcal{D}$ satisfies weak accuracy, then

$\forall F, \forall H \in \mathcal{D}(F), \exists p \in \mathit{correct}(F), \forall t \in \mathbb{T}, \forall q \in \mathit{live}(t) : p \notin H(q,t).$

Perpetual weak accuracy permits (some) false-positive mistakes. However, at least one correct process cannot be suspected by any other process at any time.

Eventual Strong Accuracy: For each execution, there exists an unknown time t after which no correct process is suspected by any correct process. Formally, if a failure detector $\mathcal{D}$ satisfies eventual strong accuracy, then

$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{T}, \forall t' \geq t, \forall p, q \in \mathit{correct}(F) : p \notin H(q,t')$

Eventual strong accuracy guarantees that false-positive mistakes can occur at most finitely many times in any given run. Correct processes can wrongfully suspect other correct processes finitely many times before time t, but correct processes eventually stop suspecting other correct processes. For a faulty process p, while p is still live, p may wrongfully suspect any other live process, including correct and faulty processes. However, faulty processes eventually crash, and thereafter they cannot make false-positive mistakes.

Eventual Weak Accuracy: For each execution, there exists an unknown time t after which *some* correct process is not suspected by any correct process. Formally, if a failure detector $\mathcal{D}$ satisfies eventual weak accuracy, then

$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{T}, \exists p \in \mathit{correct}(F), \forall t' \geq t, \forall q \in \mathit{correct}(F) : p \notin H(q,t')$

Eventual weak accuracy guarantees that correct processes cannot wrongfully suspect some correct process p infinitely many times. After time t, p must be trusted by all correct processes.

3.6.3. The Chandra-Toueg Detectors

As shown in Table I, Chandra and Toueg [23] defined eight classes of failure detectors by combining the two completeness properties and the four accuracy properties defined in the previous section. These detectors can be partially ordered based on their reducibility as shown in Figure 3. For two neighboring detector classes in Figure 3, the higher detector class is stronger than the lower detector class.

Table I. The Chandra-Toueg Failure Detectors

	Accuracy			
Completeness	perpetual strong	perpetual weak	eventual strong	eventual weak
strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
weak	*Quasi-Perfect* $\mathcal{Q}$	*Weak* $\mathcal{W}$	*Eventually Quasi-Perfect* $\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

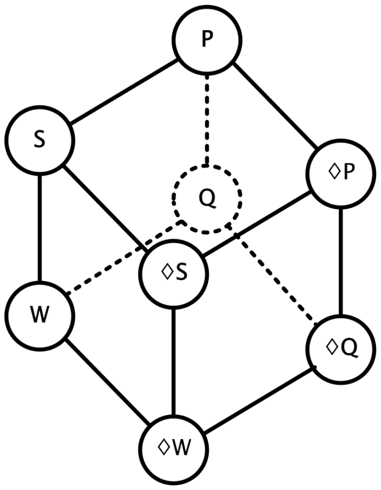


Fig. 3. The Chandra-Toueg Hierarchy

The perfect failure detector $\mathcal{P}$ is defined by perpetual strong completeness and perpetual strong accuracy. $\mathcal{P}$ does not make any false-positive or false-negative mistakes. As such, every faulty process will be suspected eventually and permanently, and suspected processes must have already crashed. $\mathcal{P}$ is the strongest failure detector among all Chandra-Toueg detectors.

The eventually perfect failure detector $\Diamond\mathcal{P}$ is defined by perpetual strong completeness and eventual strong accuracy. $\Diamond\mathcal{P}$ cannot make any false-negative mistakes, but is permitted to make only *finitely* many false-positive mistakes by wrongfully suspecting correct processes. However, $\Diamond\mathcal{P}$ eventually converges to an infinite suffix during which $\Diamond\mathcal{P}$ does not make any false-positive mistakes. The convergence time is unknown and may vary from execution to execution.

The failure detector $\Diamond\mathcal{P}$ is sufficiently powerful to solve many crash-tolerant distributed problems, such as consensus [23], stable leader election [34], and failure-locality-1 dining philosophers [17, 35]. The failure detector $\Diamond\mathcal{P}$ is also the weakest failure detector with *bounded outputs* to solve quiescent reliable communication [32]. However, $\Diamond\mathcal{P}$ may not be the weakest failure detector *in the universe* to solve quiescent reliable communication. Also, this problem can be solved in asynchronous systems using the heartbeat failure detector with unbounded outputs.

The failure detector $\Diamond\mathcal{P}$ is also the weakest failure detector for solving wait-free contention managers in shared-memory systems [33]. Wait-free contention managers ensure that every process has "enough time" in isolation to complete its operation, even in the presence of high contention and crash faults. In [33], wait-free contention managers are used to boost any obstruction-free algorithm to a wait-free algorithm. Obstruction freedom guarantees progress for every process only if every process executes its own operation in isolation for sufficiently long time. Therefore, any obstruction-free algorithm can be converted into a wait-free algorithm by a wait-free contention manager. Also, contention management provides an excellent application of eventual weak exclusion, insofar as the manager essentially regulates access to trying (rather than critical) sections. While eventual weak exclusion can schedule two processes to eat simultaneously (i.e., both are inside critical sections) finitely many times, a contention manager may schedule two processes to be hungry simultaneously (i.e., both are inside trying sections) finitely many times.

The strong failure detector $\mathcal{S}$ is defined by perpetual strong completeness and perpetual weak accuracy. In each execution, $\mathcal{S}$ can make *infinitely* many false-positive mistakes. However, $\mathcal{S}$ cannot wrongfully suspect all correct processes. There exists some correct process that is never suspected by $\mathcal{S}$. By definition, $\mathcal{S}$ is strictly weaker than $\mathcal{P}$, but incomparable with $\diamond\mathcal{P}$ [23].

The eventually strong failure detector $\diamond\mathcal{S}$ is defined by perpetual strong completeness and eventual weak accuracy. $\diamond\mathcal{S}$ can wrongfully suspect every correct process, but eventually, some correct process will never be suspected by $\diamond\mathcal{S}$. By definition, $\diamond\mathcal{S}$ is strictly weaker than $\diamond\mathcal{P}$, $\mathcal{S}$, and $\mathcal{P}$.

The failure detectors $\mathcal{Q}$, $\mathcal{W}$, $\diamond\mathcal{Q}$, and $\diamond\mathcal{W}$ are the Chandra-Toueg failure detectors with weak completeness. As shown in Table I, each failure detector with weak completeness has a corresponding failure detector with strong completeness, and they only differ on completeness. For example, $\mathcal{Q}$ corresponds to $\mathcal{P}$. By the gossip protocol discussed previously, $\mathcal{Q}$, $\mathcal{W}$, $\diamond\mathcal{Q}$, and $\diamond\mathcal{W}$ can be transformed into $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$, respectively, in networks that are not partitionable, and the accuracy properties can be preserved in the transformation. Figure 3 illustrates the relationships between any pair of corresponding failure detectors: $\mathcal{P} \succeq \mathcal{Q}$, $\mathcal{S} \succeq \mathcal{W}$, $\diamond\mathcal{P} \succeq \diamond\mathcal{Q}$, and $\diamond\mathcal{S} \succeq \diamond\mathcal{W}$. We also know $\mathcal{Q} \succeq \mathcal{W}$, $\mathcal{Q} \succeq \diamond\mathcal{Q}$, $\mathcal{W} \succeq \diamond\mathcal{W}$, and $\diamond\mathcal{Q} \succeq \diamond\mathcal{W}$. Therefore, $\diamond\mathcal{W}$ is the weakest failure detector among all Chandra-Toueg failure detectors.

It is worth pointing out that the failure detector $\diamond\mathcal{W}$ is the weakest detector for solving consensus with a majority of correct processes in asynchronous message-passing systems [30]. [2] This was the first weakest-failure-detector result. However, when a majority of processes is faulty, $\Diamond\mathcal{W}$ is still necessary but insufficient.

3.7. The Local Eventually Perfect Failure Detector $\Diamond\mathcal{P}_1$

The failure detector $\Diamond\mathcal{P}_1$ is specified by local strong completeness and local eventual strong accuracy [24, 25] as follows, where communication neighbors are neighbors in communication graphs, and $CN(p)$ denotes the set of communication neighbors of process p.

- **Local Perpetual Strong Completeness:** *Every faulty process is eventually and permanently suspected by all of its correct communication neighbors.* Formally,

  $\forall F, \forall H \in \Diamond\mathcal{P}_1(F), \exists t \in \mathbb{T}, \forall p \in faulty(F), \forall q \in (CN(p) \cap correct(F)), \forall t' \geq t : p \in H(q, t')$.

- **Local Eventual Strong Accuracy:** *For every execution, there exists an unknown time after which no correct process is suspected by any of its correct communication neighbors.* Formally,

  $\forall F, \forall H \in \Diamond\mathcal{P}_1(F), \exists t \in \mathbb{T}, \forall t' \geq t, \forall p \in correct(F), \forall q \in (CN(p) \cap correct(F)) : p \notin H(q, t')$

The detector $\Diamond\mathcal{P}_1$ is allowed to suspect correct communication neighbors finitely many times in any execution. However, $\Diamond\mathcal{P}_1$ must converge at some point after which the detector provides reliable information about neighbor crashes. Unfortunately, the

[2] In this paper [30], the eventually weak failure detector $\Diamond\mathcal{W}$ is denoted $\mathcal{W}$.

convergence time is unknown and may vary from run to run. Also, $\Diamond\mathcal{P}_1$ is defined with respect to communication neighbors; for other processes, $\Diamond\mathcal{P}_1$ is permitted to behave arbitrarily.

Recall that for a dining instance, its conflict graph is a subgraph of its communication graph. Hence, for a process p, its conflicting neighbors (i.e., its neighbors in the conflict graph) are also its communication neighbors. Let $N(p)$ be the set of conflicting neighbors of p; then we can conclude $N(p) \subseteq CN(p)$. As such, every faulty process is eventually and permanently suspected by all correct *conflicting* neighbors; for every execution, there exists an unknown time after which no correct process is suspected by any correct *conflicting* neighbor. Thus, local strong completeness and local eventual strong accuracy also hold for conflicting neighbors.

The detector $\Diamond\mathcal{P}_1$ can be transformed to its global version $\Diamond\mathcal{P}$ if the underlying communication graph cannot be partitioned by crash faults. In such an environment, any two live processes can communicate at any time directly or indirectly via message relay. As such, for a process p, its communication neighbors can broadcast the information about p (i.e., p is alive or has crashed) to all other processes, and the information will be delivered eventually and correctly via reliable channels. Therefore, $\Diamond\mathcal{P}_1$ can be transformed to $\Diamond\mathcal{P}$ in unpartitionable communication graphs.

However, this transformation cannot be applied to environments where the communication graph is partitionable. $\Diamond\mathcal{P}_1$ may be implemented in such environments, but $\Diamond\mathcal{P}$ cannot. Figure 4 shows a communication graph that is partitioned by process q crashing. As a result, processes p and r cannot communicate with each other, and hence, p and r cannot monitor the status of each other. Meanwhile, it is still possible for processes p and r to detect the status of q. This example illustrates that implementations of $\Diamond\mathcal{P}_1$ require fewer assumptions on underlying systems than those of $\Diamond\mathcal{P}$. Thus, $\Diamond\mathcal{P}_1$ is weaker than $\Diamond\mathcal{P}$.

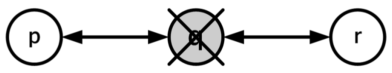


Fig. 4. A Partitionable Communication Graph

3.8. The Trusting Failure Detector

The trusting failure detector $\mathcal{T}$ was defined by Delporte-Gallet et al. [29] and is characterized by three properties: strong completeness, eventual strong accuracy, and trusting accuracy. Because $\mathcal{T}$ satisfies strong completeness and eventual strong accuracy, $\mathcal{T}$ eventually suspects crashed processes permanently, and eventually stops suspecting correct processes. The trusting accuracy property requires that if a previously trusted process is suspected, then that process must be crashed. Delporte-Gallet et al. [29] also proved that $\mathcal{T}$ is strictly stronger than $\diamond\mathcal{P}$, but strictly weaker than the perfect failure detector $\mathcal{P}$.

Delporte-Gallet et al. [29] showed that $\mathcal{T}$ is the weakest failure detector to solve fault-tolerant mutual exclusion (i.e., wait-free mutual exclusion) under perpetual weak exclusion ($\Box\mathcal{WX}$), given a majority of correct processes. $\Box\mathcal{WX}$ requires that no two live neighbors eat simultaneously ever. Mutual exclusion is a special case of dining philosophers on a completely connected conflict graph. The detector $\mathcal{T}$ is both necessary and sufficient to solve wait-free mutual exclusion under $\Box\mathcal{WX}$ with a majority of correct processes. Hence, $\mathcal{T}$ is the weakest. However, if a majority of processes are faulty, then $\mathcal{T}$ is necessary but may be insufficient.

To solve wait-free mutual exclusion under $\Box\mathcal{WX}$ in any environment, a more powerful composition of the failure detectors $\mathcal{T} + \mathcal{S}$ is introduced [29]. Recall that the strong failure detector $\mathcal{S}$ [23] is defined by strong completeness and perpetual weak accuracy. For strong completeness, $\mathcal{S}$ eventually suspects crashed processes permanently. For perpetual weak accuracy, there exists some correct process that is never suspected by $\mathcal{S}$. The composition of $\mathcal{T} + \mathcal{S}$ outputs two independent sets of suspected processes. $\mathcal{T} + \mathcal{S}$ is sufficient to solve wait-free mutual exclusion under $\Box\mathcal{WX}$ in any environment.

3.9. Implementations of Failure Detectors

There are three commonly-used methods to implement failure detectors: timeout-based, lease-based, and ping-ack implementations. In practice, most implementations adopt some sort of timeout mechanism to utilize timing assumptions in the underlying system. We will introduce these three implementation techniques and review the system models in which $\Diamond\mathcal{P}$ can be implemented.

3.9.1. Timeout-Based Implementations

Timeout-based implementations (or heartbeat-based implementations) are the most commonly-used method to implement failure detectors [23, 36, 37, 38]. [3] Given two processes p and q, without loss of generality, let p monitor q. The monitored process q periodically sends heartbeat messages to all other processes, including p. When such a message arrives at process p, p trusts q, resets its corresponding timer, and starts to count down on q again. If p times out on q, then p suspects q and adds q into its suspect list. If p later receives a heartbeat message from q, then p realizes that it made a mistake. To fix this mistake, p removes q from the suspect list. Also, p attempts to prevent such mistakes in the future by increasing the timeout duration.

This method usually assumes the existence of bounds on timing properties and guarantees that there exists an infinite suffix, during which the timeout durations are larger than inter-arrival durations of two consecutive heartbeat messages. In this

[3]In some publications [37, 38], this method is also called the *push model.*

suffix, if q and p are correct, heartbeat messages from q always arrive at p before the timer times out, and hence, q will not be suspected. After a process crashes, no message can be sent, and hence the timer eventually times out. Therefore, crashed processes will be eventually and permanently suspected.

3.9.2. Lease-Based Implementations

Lease-based implementations are another way to detect crash faults [39]. For process p to monitor process q, process q (the lessee) submits a lease to p (the lessor). Process q should renew the lease before the lease expires. If the lease expires without being renewed, p suspects q and sends q a notification. If q is not crashed, then q receives the notification. As a response to the notification, process q resubmits a new lease with a longer lease term to prevent such mistakes from occurring in the future. Upon receiving the new lease, p realizes that it made a mistake, and p removes q from its suspect list to fix the mistake. If q actually crashes, then p eventually stops receiving messages from q, and hence, suspects q permanently.

There are many commonalities between lease-based implementations and timeout-based implementations. For example, in both implementations, the monitoring processes (i.e., process p) are passively waiting for messages, and p holds a timer for the monitored process (i.e., process q).

Lease-based implementations differ from timeout-based implementations on who controls the timeout durations. In lease-based implementations, the monitored process (i.e., process q) controls the timeout durations (i.e., the length of lease terms). In timeout-based implementations, the monitoring process (i.e., process p) controls the timeout durations.

3.9.3. Ping-Ack Implementations

In ping-ack implementations [36], processes actively send messages to detect crash faults. [4] For process p to monitor process q, when p wants to know the status of q, process p actively sends a ping message to q. Meanwhile, p sets the corresponding timer and starts to count down on q. When q receives this ping message, q replies by sending an ack message to p. Process p expects to receive this ack message before the timer times out. If p times out on q before this message arrives, then p suspects q. If p later receives the ack message, then p removes q from the suspect list and attempts to prevent such mistakes in the future by increasing the length of timeout durations.

3.9.4. Models to Implement $\diamond\mathcal{P}$

The ability of a model to implement $\diamond\mathcal{P}$ usually depends on two factors. One factor is timing assumptions. As we discussed before, $\diamond\mathcal{P}$ cannot be implemented in asynchronous systems. Hence, all models sufficient to implement $\diamond\mathcal{P}$ must assume some known or unknown, eventual or perpetual bounds on message delays and/or relative process speeds. Another important factor is the reliability of communication channels. Although most models assume reliable communication channels, there do exist some models [40] in which messages can be lost.

Many system models [23, 34, 36, 40, 41, 42, 43, 44] are sufficient to implement $\diamond\mathcal{P}$. This section will introduce several important models, including $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$ [22, 23], the finite average-response-time model [36], and the average delayed/dropped model [40].

[4] In some publications [37, 38], this method is also called the *pull model*.

3.9.4.1. Model $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$

Among all the models, $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ are the first and most important models used to implement $\diamond\mathcal{P}$ [23]. These three classic models of partial synchrony assume reliable communication channels and are defined as follows.

Model $\mathcal{M}_1$ guarantees that for each execution, there exist *unknown* bounds on message delays and relative process speeds [22]. These bounds are *unknown* and *perpetual.* Systems modeled by $\mathcal{M}_1$ behave synchronously, but since the actual bounds are not known, they cannot be used to advantage for reliable failure detection.

Model $\mathcal{M}_2$ guarantees that for each execution, there exist *known* bounds on message delays and relative process speeds, but these timing bounds hold only after a finite and unknown prefix of the execution [22]. These bounds are *known* but *eventual.* Systems modeled by $\mathcal{M}_2$ eventually conform to known bounds, but they may exhibit highly asynchronous behaviors during any finite prefix of each run.

Model $\mathcal{M}_3$ guarantees that for each execution, there exist *unknown* bounds on message delays and relative process speeds, and these bounds hold only after a finite and unknown period of time [23]. The bounds are *unknown* and *eventual.* $\mathcal{M}_3$ can be considered a weak union of $\mathcal{M}_1$ and $\mathcal{M}_2$, where the bounds exist, but are unknown and only hold after some unknown stabilization time.

The general approach to implementing $\diamond\mathcal{P}$ in such models is to use an adaptive timeout mechanism that increases the estimated timeout threshold after each false positive. For model $\mathcal{M}_1$, the estimated threshold will eventually surpass the de facto bounds after a finite number of false positives. For model $\mathcal{M}_2$, the initial threshold can be calculated based on the known bounds, and the system will eventually converge to the known bounds, after which false positives cease. For model $\mathcal{M}_3$, the estimated threshold will eventually surpass the eventual bound after a finite number of false positives.

3.9.4.2. Finite Average-Response-Time Model

Another important model for implementing $\diamond\mathcal{P}$ is presented by Fetzer et al. [36], called the finite average-response-time model (FAR model for short), which does not assume any upper bound on message delays or relative process speeds. Instead of using reliable channels, the FAR model adopts acknowledged stubborn channels: if a correct process p sends a message m to a correct process q, and p delays sending other messages until p receives an acknowledgment (*ack*) from q, then the message m and the ack message are eventually delivered to q and p, respectively. The FAR model assumes that there exists a lower bound on time to increment an integer variable. Response times are time durations from the time p sends a message to the time the ack message arrives. The FAR model also assumes that the average response time is finite but unbounded. A ping-ack protocol is used to implement $\diamond\mathcal{P}$ in the FAR model such that the estimated timeout period is eventually larger than the next response time. However, the implementation is impractical, because computation over an unbounded number of messages is required.

3.9.4.3. Average Delayed/Dropped Model

Recent research [40] discovered that $\diamond\mathcal{P}$ can be implemented in systems in which infinitely many messages can be lost, and there is only limited knowledge about the pattern of message losses. Sastry and Pike [40] introduced Average Delayed/Dropped channels (ADD channels). To our knowledge, the ADD model is weaker than other models sufficient for implementing $\diamond\mathcal{P}$ using bounded memory. On an ADD channel, every message is either privileged or non-privileged, but it is unknown whether any specific message is privileged. There is no guarantee about non-privileged messages;

they can be arbitrarily delayed or dropped. As such, infinitely many messages can be dropped, and ADD channels are unreliable. For each run, however, there exists an unknown time window $w \in \mathbb{N}$ and two unknown bounds d and r, where d is an upper bound on the average delay of the last w privileged messages, and r is the upper bound on the ratio of non-privileged messages to privilege message in per window w. These assumptions guarantee that each sufficiently long window contains at least one privileged message, which has an average upper bound on message delays. Hence, there exists an upper bound on inter-arrival times of messages. Therefore, an adaptive timeout method can be used to implement $\diamond\mathcal{P}$.

CHAPTER IV

DINING PHILOSOPHERS

Dining philosophers is usually characterized by the two properties: *progress* and *safety*. A progress property guarantees that some "good thing" eventually happens; for every partial execution, there always exists an infinite extension such that the "good thing" eventually occurs in this extended execution [45]. A safety property guarantees that some "bad thing" never happens; if this safety property is violated in an execution, then there exists a finite prefix during which the "bad thing" can be recognized at some point [45]. In decreasing order of strength, this chapter will introduce three progress properties (i.e., wait-freedom, lockout freedom, and deadlock freedom) and three safety properties (i.e., perpetual strong exclusion, perpetual weak exclusion, and eventual weak exclusion).

This chapter also reviews solvability of wait-free dining. Solvability of wait-free dining depends on both synchronism of underlying systems and dining specifications. Therefore, we discuss solvability of wait-free dining with respect to different safety properties and different systems. In particular, solvability of wait-free dining under perpetual weak exclusion will be discussed in detail.

Dining philosophers is only one of many paradigms for resource allocation problems. Other classic paradigms include mutual exclusion, drinking philosophers, and job scheduling. We will also give a brief introduction to these paradigms and discuss the relationships among these paradigms.

4.1. Progress Properties

In this section, three progress properties are defined in decreasing order of strength: wait-freedom, lockout freedom, and deadlock freedom. These properties mainly differ on two aspects: whether progress is global or local, and whether crash faults are considered. Their definitions are shown as follows.

- **Wait-Freedom**: Regardless of how many processes crash, every correct hungry process eventually eats [10].
- **Lockout Freedom**: In every fault-free execution, every correct hungry process eventually eats [15, 46].
- **Deadlock Freedom**: In every fault-free execution, if some correct process is hungry, then some correct process eventually eats [15, 46].

Wait-freedom guarantees progress *locally* for individual processes and considers crash faults. Originally, wait-freedom guarantees that every correct process finishes any operation within finitely many steps, regardless of execution speeds of other processes [10]. In the context of dining, wait-freedom guarantees that every correct hungry process goes to eat within finitely many steps, regardless of process crashes. As such, *every* correct hungry process can make progress to eat. Hence, progress is ensured locally for individual processes.

By contrast, lockout freedom guarantees progress *locally* for individual processes, but only in fault-free environments. In the absence of crash faults, every process is correct, and lockout freedom is equivalent to wait-freedom. However, in the presence of crash faults, lockout freedom does not guarantee that every correct hungry process eventually eats. In the worst case, all correct hungry processes may starve.

Deadlock freedom only guarantees progress *globally* in fault-free environments. If a set of processes is hungry, then at least some process in this set will eventually eat. However, there is no guarantee that every hungry process eventually eats. As such, deadlock freedom only excludes the possibility of global starvation (i.e., all hungry processes starve), and hence, it is a global progress property. Similar to lockout freedom, deadlock freedom is defined with respect to fault-free environments. In the presence of crash faults, all hungry processes may starve in the worst case.

Because this dissertation mainly explores the dining problem in the presence of crash faults, the following sections will focus on wait-free dining only.

4.2. Safety Properties

In this section, three safety properties are defined in decreasing order of strength: perpetual strong exclusion, perpetual weak exclusion, and eventual weak exclusion. Since eventual weak exclusion has already been discussed in Chapter I, this section will mainly focus on perpetual strong exclusion and perpetual weak exclusion.

- **Perpetual Strong Exclusion**: No two neighbors eat simultaneously.
- **Perpetual Weak Exclusion**: No two live neighbors eat simultaneously.
- **Eventual Weak Exclusion**: For each execution, there exists a time after which no two live neighbors eat simultaneously.

4.2.1. Perpetual Strong Exclusion

Perpetual strong exclusion is defined with respect to all processes, including live and crashed processes. Therefore, processes are prohibited to eat simultaneously with any neighbors, even crashed neighbors.

Perpetual strong exclusion is used to model unrepairable shared resources that can be corrupted permanently by process crashes. For example, every hard disk has a partition table to store partition records. If a process p crashes while p is updating this table, then it is possible that the table is permanently corrupted. As a result, data in the hard disk may be permanently lost. Hence, data stored in hard disks can be considered as an unrepairable shared resource (assume that there is no backup disk for the data).

However, perpetual strong exclusion is needlessly restrictive for practical applications in which shared resources are recoverable. For example, transient corruptions on shared memory are usually recoverable. Consequently, it is necessary to consider weaker exclusion models from a practical perspective, such as perpetual weak exclusion and eventual weak exclusion.

4.2.2. Perpetual Weak Exclusion

Perpetual weak exclusion ($\Box\mathcal{WX}$) guarantees that no two *live* neighbors eat simultaneously. $\Box\mathcal{WX}$ is defined with respect to live processes. As such, a live process p is prohibited from eating simultaneously with any of its live neighbors, but p is not prohibited from eating simultaneously with its crashed neighbors.

Perpetual weak exclusion is used to model repairable shared resources. For example, wireless channels are such a repairable shared resource. At any time, only one process can send messages via a wireless channel. If a process p crashes while p is communicating via a channel, then the channel may be left in an unusable state for a short period of time. However, this channel eventually recovers from the crash fault and becomes available again for other processes.

Perpetual weak exclusion is weaker than perpetual strong exclusion. In fault-free environments, all processes are correct, and hence, $\Box\mathcal{WX}$ is equivalent to perpetual strong exclusion. In the presence of crash faults, $\Box\mathcal{WX}$ allows live processes to eat concurrently with crashed neighbors, but perpetual strong exclusion does not. Hence, perpetual strong exclusion impose more constraints on dining algorithms than $\Box\mathcal{WX}$.

4.2.3. Eventual Weak Exclusion

Eventual weak exclusion ($\Diamond\mathcal{WX}$) is even weaker than $\Box\mathcal{WX}$. Recall that $\Diamond\mathcal{WX}$ guarantees that for each execution, there exists a time after which no two live neighbors eat simultaneously. Similar to $\Box\mathcal{WX}$, $\Diamond\mathcal{WX}$ does not prohibit a live process from eating simultaneously with its crashed neighbors as well. However, unlike $\Box\mathcal{WX}$, $\Diamond\mathcal{WX}$ does allow live processes to eat with their live neighbors simultaneously finitely many times.

4.3. Solvability of Wait-Free Dining

This section discusses solvability of wait-free dining under perpetual strong exclusion and perpetual weak exclusion. Solvability of wait-free dining depends not only on synchronism of underlying systems, but also on dining specifications (e.g., safety properties). This section reviews several solvability or unsolvability results with respect to both safety properties and synchronism of underlying systems.

4.3.1. Perpetual Strong Exclusion

Wait-free dining is unsolvable under perpetual strong exclusion simply by its definition. Perpetual strong exclusion guarantees that no two neighbors eat simultaneously. Therefore, if a process crashes while it is eating, then no hungry neighbor can be scheduled to eat. Thus, wait-free dining under perpetual strong exclusion is unsolvable in all distributed systems.

4.3.2. Perpetual Weak Exclusion

In short, wait-free dining under $\Box\mathcal{WX}$ is unsolvable in asynchronous systems, may or may not be solvable in partially synchronous systems, and is solvable in synchronous systems.

4.3.2.1. Asynchronous Systems

Wait-free dining is unsolvable in asynchronous message-passing systems. This impossibility result is based on the intrinsic difficulty of reliable fault detection in asynchronous systems. Choy and Singh [20] proved this impossibility result by showing that failure locality 1 is impossible for asynchronous dining algorithms under $\Box\mathcal{WX}$. Failure locality ($\mathcal{FL}$) is defined as the maximal geographic radius (in conflict graphs) of processes which could be affected by a single crash fault [20]. $\mathcal{FL}1$ means that in the worst case, a crash fault can only cause the immediate neighbors of the crashed process to starve. Wait-freedom implies failure locality 0. Hence, wait-free dining under $\Box\mathcal{WX}$ is impossible in asynchronous systems.

With respect to crash tolerance, failure locality 2 ($\mathcal{FL}2$) is the best result achievable by asynchronous dining solutions under $\Box\mathcal{WX}$ [20, 47, 48, 49, 50]. $\mathcal{FL}2$ means that in the worst case, a crash fault can starve processes within at most two-hop neighborhood of the crashed process in conflict graphs.

4.3.2.2. Partially Synchronous Systems

Partial synchrony can be encapsulated by failure detectors. Therefore, we study solvability of wait-free dining under $\Box\mathcal{WX}$ with respect to failure detectors. Solvability of wait-free dining under $\Box\mathcal{WX}$ depends on the strength of failure detectors.

The eventually perfect failure detector $\Diamond\mathcal{P}$ is insufficient to solve wait-free dining under $\Box\mathcal{WX}$. Pike and Sivilotti [35] proved this impossibility result by showing that no dining algorithm can achieve $\mathcal{FL}0$ using $\Diamond\mathcal{P}$. Clearly, any failure detector weaker than $\Diamond\mathcal{P}$ is also insufficient to solve wait-free dining under $\Box\mathcal{WX}$. The same paper [35] also presents three dining algorithms that achieve $\mathcal{FL}1$ under $\Box\mathcal{WX}$ using $\Diamond\mathcal{P}$. Therefore, the optimal failure locality is 1 for dining under $\Box\mathcal{WX}$ using $\Diamond\mathcal{P}$.

As discussed in Chapter III, the trusting failure detector $\mathcal{T}$ has been used to explore solvability of wait-free dining under $\Box\mathcal{WX}$ in the context of mutual exclusion [29]. The detector $\mathcal{T}$ is both necessary and sufficient to solve wait-free mutual exclusion under $\Box\mathcal{WX}$ in environments with a majority of correct processes. Hence, $\mathcal{T}$ is the weakest in such environments. The composition of $\mathcal{T} + \mathcal{S}$ is sufficient to solve wait-free mutual exclusion under $\Box\mathcal{WX}$ in any environment, but it is not known whether or not $\mathcal{T} + \mathcal{S}$ is necessary.

This dissertation can be viewed as an extension of the work by Pike and Sivilotti [35]. To circumvent the unsolvability result in [35], we could either (1) use stronger failure detectors, or (2) explore weaker exclusion models and still use $\Diamond\mathcal{P}$. Delporte-Gallet et al. [29] explored the first option by using the stronger failure detector $\mathcal{T}$. This dissertation pursues the second option by exploring a weaker exclusion model, eventual weak exclusion ($\Diamond\mathcal{WX}$).

4.3.2.3. Synchronous Systems

Wait-free dining under $\Box\mathcal{WX}$ is solvable in synchronous systems. In such a system, reliable fault detection can be implemented. Hence, crashed processes will be suspected eventually and permanently, and suspected processes must have already been crashed. Once a crash fault is reliably detected, correct processes stops waiting on the crashed process, and the resources being occupied by the crashed process can be reallocated. As such, correct processes cannot be permanently blocked from eating by crashed processes.

4.4. Hierarchy of Resource Allocation Problems

Dining philosophers is related to three other resource allocation problems: mutual exclusion, drinking philosophers, and job scheduling. Including dining, these four problems are related in a way such that the problems are generalized in the following order: *mutual exclusion, dining philosophers, drinking philosophers*, and *job scheduling*. Mutual exclusion is a special case of dining, dining is a special case of drinking, and drinking is a special case of job scheduling. Their solutions are correlated as well. Solutions to a problem can be directly applied to solve its special cases, and solutions to a special case are often used as a basic block (or subroutine) to build solutions to the generalized problem.

4.4.1. Mutual Exclusion

Mutual exclusion is a model of *static* resource-allocation scenarios, where every pair of processes has overlapping resource needs [51, 52, 53]. The term *"static"* means that each process always requires a *fixed* (i.e., statically determined) set of resources every time the process becomes hungry. Each mutual exclusion instance can be modeled as a conflict graph of a clique in which every pair of processes is neighboring to each other. As such, concurrency is minimized such that at any time, at most one process can utilize shared resources.

Many algorithms have been proposed to solve mutual exclusion [29, 52, 54, 55, 56, 57, 58]. The first mutual exclusion algorithm was given by Dijkstra and only uses read/write registers [52]. Lamport [55] presented the bakery algorithm, which considers processes that wish to enter critical sections as customers in a bakery. Customers are served according to the time they arrive to the bakery. If two or more customers arrive the bakery at the same time, then the customer with the lowest id is served first (process ids are static and unique). Burns and Lynch [54] also gave a lower bound on the number of read/write registers to solve deadlock-free mutual exclusion. Since our dissertation mainly focuses on dining philosophers, we only mentioned a few important mutual exclusion algorithms. For readers who are interested in this topic, the book by Michel Raynel [53] is a good coverage until 1986, and the paper by James et al. [51] provides a recent survey of this topic in shared-memory systems.

4.4.2. Dining Philosophers

Dining philosophers is a generalization of mutual exclusion. Dining differs from mutual exclusion on the topology of conflict graphs. Dining conflict graphs can be arbitrary topologies, but conflict graphs for mutual exclusion must be a clique. Hence, mutual exclusion is a special case of dining philosophers where the conflict graph is always a clique.

Many algorithms have been developed for the dining philosophers problem [12, 26, 35, 59, 60, 61, 62, 63, 64, 65]. These algorithms are developed for different purposes; they may focus on improving failure locality [20, 26, 35, 47, 48, 50], self-stabilization [4, 47, 66], response time [26, 59, 63, 65] (i.e., the time delay between a process becoming hungry and the process going to eat), or message complexity [26, 63] (i.e., in the worst case, the number of messages sent or received by a process during any response times). Some algorithms are developed under shared-memory systems [4, 62, 65], while others are developed under message-passing systems [14, 26, 35, 63].

Dining solutions often utilize several techniques, including doorways [14, 26, 35], dynamic partial orders [18, 35, 60], distributed queues [12, 59, 63], randomization [67, 68], and modular constructions [48, 69]. Chapter V and VII will give a detailed description on dynamic partial orders and doorways.

It is worth to point out that dining can be solved by a modular construction using mutual exclusion solutions as a black-box subroutine [69]. In this algorithm, each shared resource r is associated with a mutual exclusion instance, which is composed of proxies of all processes sharing r. All shared resources are statically and totally ordered. A hungry process p requests its shared resources one by one according to the total ordering of resources. Only after p holds a shared resource, p can request the next one. Hungry processes do not relinquish shared resources until after they finish eating. Competition for a shared resource is solved by the associated mutual exclusion instance.

4.4.3. Drinking Philosophers

Drinking philosophers [70] is a *dynamic* resource allocation problem. For the drinking problem, the topology of conflict graphs can be arbitrary. Each pair of neighbors in conflict graphs shares a number of distinct bottles (resources). Each drinker (analogous to diners) cycles through four states: *tranquil, thirsty, drinking*, and *exiting*, which are analogous to four diner states: *thinking, hungry, eating*, and *exiting*, respectively. Each time a tranquil process becomes thirsty, it requests a *dynamically determined* subset of bottles shared with its neighbors. Therefore, the drinking problem allows neighbors to drink simultaneously when they require disjoint subsets of bottles. Comparing with dining, drinking philosophers increases potential concurrency and resource utilization. Dining is a special case of drinking where each drinker always requests all of its shared bottles.

The drinking problem can be solved by modular construction using dining solutions. Chandy and Misra [70] constructed a drinking solution directly based on the hygienic dining algorithm. Welch and Lynch [13] also gave a modular drinking algorithm in which any dining solution can be used as a black-box subroutine. Welch and Lynch's algorithm uses the black-box dining subroutine to guarantee that a thirsty drinker p eventually gets the priority to utilize *all* of its shared bottles. Because p has the priority, p eventually holds all *necessary* bottles for this drinking session and goes to drink. After that, p gives up the priority to release unneeded bottles. This action potentially increases concurrency and resource utilization.

4.4.4. Job Scheduling

Job scheduling [59] further generalizes the drinking philosophers problem. In job scheduling, jobs (analogous to critical sections) are created dynamically at processes. Each job requires a set of resources. If two jobs have overlapping resource requirements, then these two jobs are incompatible. Since jobs are dynamically created, the conflict graph of jobs is dynamic. As such, job scheduling extends drinking and dining by allowing philosophers to be added into and removed from conflict graphs dynamically. Drinking philosophers is actually a special case of job scheduling where the conflict graph is static. Job scheduling algorithms guarantee that every job is eventually executed, and no incompatible jobs can be executed concurrently.

Awerbuch and Saks [59] presented a modular job scheduling algorithm that uses any dining algorithm as a black-box subroutine. This algorithm maintains a distributed priority queue in which each slot contains one or more compatible jobs. Only jobs at the head of the priority queue can be scheduled to execute. New jobs are added to the tail of the queue, and jobs move toward the head by using the black-box dining algorithm to resolve conflicts among these jobs.

4.5. Summary

This chapter outlines dining variants and other paradigms of resource allocation problems. These dining variants are defined with respect to three progress properties (wait-freedom, lockout freedom, and deadlock freedom) and three safety properties (perpetual strong exclusion, perpetual weak exclusion, and eventual weak exclusion). The paradigms include mutual exclusion, dining philosophers, drinking philosophers, and job scheduling.

We also discussed solvability of wait-free dining under perpetual strong exclusion and perpetual weak exclusion ($\Box\mathcal{WX}$). Wait-free dining is unsolvable under perpetual strong exclusion. For $\Box\mathcal{WX}$, we introduced solvability of wait-free dining using two failure detectors $\Diamond\mathcal{P}$ and $\mathcal{T}$. In particular, $\Diamond\mathcal{P}$ is insufficient to solve wait-free dining under $\Box\mathcal{WX}$. This intrigues one question: whether $\Diamond\mathcal{P}$ can solve wait-free dining under a weaker exclusion model, eventual weak exclusion? The next chapter will answer this question by giving a wait-free dining algorithm under $\Diamond\mathcal{WX}$ using $\Diamond\mathcal{P}$.

CHAPTER V

A WAIT-FREE DINING SOLUTION USING $\Diamond\mathcal{P}_1$

This chapter presents a wait-free dining algorithm under eventual weak exclusion ($\Diamond\mathcal{WX}$) using $\Diamond\mathcal{P}_1$ [18]. This algorithm is based on the classic hygienic dining algorithm by Chandy and Misra [60, 70]. Therefore, we first describe the original hygienic dining solution briefly. Next, we describe our algorithm in detail. Finally, we prove that our algorithm satisfies eventual weak exclusion and wait-freedom.

5.1. Original Hygienic Dining Algorithm

The original hygienic dining algorithm was presented by Chandy and Misra [60, 70] in asynchronous fault-free systems. This dining algorithm satisfies lockout freedom and perpetual weak exclusion ($\Box\mathcal{WX}$). Two key techniques are used in this algorithm: forks for safety (i.e., $\Box\mathcal{WX}$) and a dynamic partial ordering for progress (i.e., lockout freedom). Note that in the original hygienic algorithm, a diner has only three states: *thinking, hungry*, and *eating*.

A unique fork [1] is associated with each edge in the conflict graph. Neighbors connected by a conflict edge share the corresponding fork, which is used to resolve conflicts over the overlapping set of resources needed by both neighbors. In order to

Part of this chapter is reprinted from *Proceedings of the 9th International Conference on Distributed Computing and Networking (ICDCN)*, S. Pike, Y. Song, and S. Sastry, Wait-Free Dining Under Eventual Weak Exclusion, pp. 135-146, Copyright (2008), with permission from Springer.

[1]A popular misconception is that forks are resources. We emphasize that forks are *not* resources. Note that forks are not even mentioned in the problem specification; they are simply used as a technique to guarantee safety.

eat, a hungry process must collect and hold all of its shared forks. This provides a simple basis for safety, since at most one neighbor can hold a given fork at any time.

A fork is either *clean* or *dirty*. The forks held by eating processes are dirty. Dirty forks remain dirty until they are cleaned. Dirty forks are cleaned only when they are sent to other processes. [2] As such, processes always receive clean forks. A process cannot relinquish clean forks. Clean forks are held only by hungry processes, and they remain clean until they are used to eat. Initially, all forks are dirty.

A request token is associated with each fork. A hungry process p can request a missing fork only when p holds the corresponding token. p requests a shared fork by sending the token to the neighbor. A fork and the corresponding token are initially located at different processes.

Processes can be partially ordered. This partial ordering can be represented as a precedence graph, which is constructed by assigning directions to each edge in the conflict graph. In a precedence graph, each edge (p,q) is directed from process p to q if and only if (1) p holds the fork shared with q, and the fork is clean, or (2) q holds the shared fork, and the fork is dirty, or (3) the fork is in transit from q to p. The direction from p to q indicates that p has priority (or precedence in [70]) over q. We say that process p has a higher priority, and q has a lower priority. When two hungry processes are competing a shared fork, the conflict is always resolved in favor of the higher-priority process. Also, the direction of an edge (p, q) can be changed only when the higher-priority process p starts to eat. Initially, forks are dirty and are assigned in a way such that the precedence graph is acyclic. When a process p starts eating, all of its shared forks become dirty, and hence, all edges incident on p must be directed toward p. By this way, the precedence graph is always acyclic.

[2]That is the reason why this algorithm is called *hygienic*.

The hygienic dining algorithm works as follows. Initially, every process is thinking and may become hungry at any time. A hungry process p requests a missing fork shared with its neighbor q only when p holds the corresponding token. When q receives the request token, q grants the request only if q is not eating, and the fork shared with p is dirty. Otherwise, q defers the request until after q eats. Therefore, requests for a dirty fork are always granted, but requests for a clean fork are always deferred. After p collects all of its shared forks, p goes to eat, and all of its shared forks become dirty. This action makes all edges incident on p toward p, and hence, maintains acylicity of the precedence graph. After eating, the deferred forks will be sent out.

This algorithm satisfies perpetual weak exclusion. Each pair of neighbors shares a unique fork, and hence, at most one neighbor can hold the fork at any time. Since eating processes must hold all of their shared forks, no two neighbors eat simultaneously.

This algorithm satisfies lockout freedom. The proof is based on strong induction on the depth of the partial ordering. If a process has no higher-priority neighbors, then its depth is 0. For other process q, its depth is the maximum number of edges along a path to q from some process with depth 0. The base case guarantees that processes with depth 0 eventually eat, because such a process has priority over all of its neighbors. The inductive step assumes that all hungry processes with depth less than d eventually eat, and guarantees that processes with depth d eventually eat. Given such a process q with depth d, q can be blocked from eating only by its hungry higher-priority neighbors r. However, the depth of r must be less than d, and by inductive hypothesis, r eventually eats and loses priority over q. Therefore, q eventually collects all of its shared forks and goes to eat.

5.2. Wait-Free Dining Algorithm Using $\Diamond\mathcal{P}_1$

In the original hygienic solution [60], if a process crashes while holding a fork, then the corresponding neighbor will never get the fork. If this neighbor is hungry, then it starves. Thus, the original hygienic solution is not wait-free.

To achieve wait-freedom, we use suspicion from $\Diamond\mathcal{P}_1$ as a proxy for permanently missing forks [18]. That is, a process p is permitted to eat if and only if for each neighbor q, p is either holding the fork shared with q or suspecting q. Although processes that crash while holding a fork will never send that fork, the corresponding neighbor is still able to eat by using suspicion in place of the missing fork. Since the local strong completeness property of $\Diamond\mathcal{P}_1$ guarantees that every crashed process will be eventually and permanently suspected by all correct neighbors, no process starves because its neighbor crashes.

Unfortunately, the local eventual strong accuracy property allows $\Diamond\mathcal{P}_1$ to make mistakes by wrongfully suspecting live neighbors finitely many times during any run. Therefore, two live neighbors which suspect each other may both proceed to eating, regardless of which neighbor actually holds the shared fork. Such violations of mutual exclusion are caused by false-positive mistakes, which can occur only finitely many times during the finite prefix. In the infinite suffix, because no two correct neighbors can suspect each other, and because at most one of them can hold the shared fork, eventually no correct neighbors eat simultaneously. Hence, eventual weak exclusion is satisfied.

The local eventual strong accuracy property also has an impact on progress. In the original hygienic algorithm, an eating process must reduce its priority below that of all neighbors. This absolute reduction forms the basis for progress, because processes cannot remain a higher priority after eating to overtake their hungry neighbors.

More importantly, cycles cannot be formed in the precedence graph. This absolute reduction is based on the fact that in the hygienic solution, eating processes know the relative priority of all neighbors by holding all shared forks. However, the absolute reduction does not hold when we use suspicion from $\diamond\mathcal{P}_1$ as a proxy for missing forks. False-positive mistakes of $\diamond\mathcal{P}_1$ may enable a process p to eat without holding the fork p should hold. In the worst case, two neighbors may suspect each other, and then go to eat simultaneously even if neither holds the fork. If the fork is still in transit when both neighbors complete eating, then neither neighbor knows the actual priority ordering. As a result, it is impossible for both processes to reduce their own priority below all neighbors, and even worse, cycles may be formed in the precedence graph. Deadlock may be created. Hence, wait-freedom cannot be guaranteed.

To ensure wait-freedom, we store process priorities explicitly at each process. For each process p, its priority $priority_p$ can be expressed as an ordered pair $(height_p, id_p)$, where $height_p$ is a local integer-valued variable, and id_p is a globally unique process identifier. For two processes p and q, we say that process p has a higher priority than q (i.e., $priority_p > priority_q$) if and only if:

$$\begin{aligned} & (\mathsf{priority}_p.\mathsf{height} > \mathsf{priority}_q.\mathsf{height}) \\ \mathsf{or} \quad & (\mathsf{priority}_p.\mathsf{height} = \mathsf{priority}_q.\mathsf{height}) \wedge (\mathsf{priority}_p.\mathsf{id} > \mathsf{priority}_q.\mathsf{id}) \end{aligned}$$

Because process identifiers are unique, no two processes have the same priority at any time, and processes can be totally ordered based on their priorities. As a result, no deadlock can be formed. After eating, processes simply reduce their height by an *arbitrary* integer number. As such, processes do not need to know the priority of any neighbor to reduce its priority. After finitely many eating sessions, processes cannot remain a higher priority to overtake their hungry neighbors. Therefore, Wait-freedom is guaranteed.

The algorithmic pseudocode is presented as an action system of guarded commands shown in Algorithm 1.

5.2.1. Algorithm Variables

In addition to the failure detector module, each process has four types of local variables: *state*, *priority*, *fork*, and *token*. Moreover, we use $N(p)$ to denote the set of process p's neighbors and $\Diamond\mathcal{P}_1$ to denote the set of processes being suspected by p.

The variable $state_p$ simply denotes the current dining phase of process p; it is either *thinking, hungry, eating*, or *exiting*. Initially, every process is thinking.

The $priority_p$ variable for each process p consists of two parts: a local integer-valued variable $height_p$ and a globally unique process identifier id_p, where process identifiers are static and can be totally ordered. Taken together as an ordered pair, $(height_p, id_p)$ determines the priority of process p. When two hungry neighbors are competing for a shared fork, the conflict is resolved in favor of the higher-priority process. Since process identifiers are unique and static, no two processes have the same priority at any time. Therefore, processes can be *totally* ordered lexicographically by their priorities.

The local variable *fork* is introduced to implement forks. Each process p has a Boolean variable $fork_{pq}$ associated with each neighbor q. Symmetrically, q also has a Boolean variable $fork_{qp}$ associated with p. The variable $fork_{pq}$ is true exactly when process p holds the unique fork shared with q. When the fork is in transit from one neighbor to the other, both local variables $fork_{pq}$ and $fork_{qp}$ are false. Since the fork is unique and exclusive, it is impossible for both variables to be true.

In addition to the forks, a request token is also introduced for each pair of neighbors. Tokens are implemented the same as forks. For a process p, we associate a Boolean variable $token_{pq}$ with each neighbor q. Symmetrically, q also has a Boolean

```
Code for process $p$ with its unique identifier $\mathsf{id}_p$ and with the set of its neighbors $N(p)$

var $\mathsf{state}_p$    : {thinking, hungry, eating, exiting}     initially, $\mathsf{state}_p = \mathsf{thinking}$
    $\mathsf{priority}_p$ : $(\mathsf{height}_p, \mathsf{id}_p)$                          initially, $\mathsf{priority}_p = (0, \mathsf{id}_p)$
    $\mathsf{fork}_{pq}$     : boolean, for each neighbor $q$             initially, $\mathsf{fork}_{pq} = (\mathsf{id}_p > \mathsf{id}_q)$
    $\mathsf{token}_{pq}$    : boolean, for each neighbor $q$             initially, $\mathsf{token}_{pq} = (\mathsf{id}_p < \mathsf{id}_q)$
    $\diamond\mathcal{P}_1$       : local eventually perfect detector

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

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

 7 : {receive-request $\langle \mathsf{priority}_q \rangle$ from $q \in N(p)$} $\longrightarrow$                 Action 3
 8 :       $\mathsf{token}_{pq} :=$ true
 9 :       if $(\mathsf{state}_p = \mathsf{thinking} \vee (\mathsf{state}_p = \mathsf{hungry} \wedge (\mathsf{priority}_p < \mathsf{priority}_q)))$
10 :       then send-fork$\langle p \rangle$ to $q$                                     Send Forks or
11 :             $\mathsf{fork}_{pq} :=$ false                                         Defer Requests

12 : {receive-fork $\langle q \rangle$ from $q \in N(p)$} $\longrightarrow$                            Action 4
13 :       $\mathsf{fork}_{pq} :=$ true                                         Obtain Shared Forks

14 : {$\mathsf{state}_p = \mathsf{hungry} \wedge (\forall q \in N(p) :: (\mathsf{fork}_{pq} \vee q \in \diamond\mathcal{P}_1))$} $\longrightarrow$          Action 5
15 :       $\mathsf{state}_p :=$ eating                                      Enter Critical Section

16 : {$\mathsf{state}_p = \mathsf{eating}$} $\longrightarrow$                                         Action 6
17 :       $\mathsf{state}_p :=$ exiting                                      Exit Critical Section

18 : {$\mathsf{state}_p = \mathsf{exiting}$} $\longrightarrow$                                        Action 7
19 :       Lower($\mathsf{priority}_p$)
20 :       $\mathsf{state}_p :=$ thinking                                  Transit Back to Thinking
21 :       $\forall q \in N(p)$ where $(\mathsf{token}_{pq} \wedge \mathsf{fork}_{pq})$ do
22 :           send-fork$\langle p \rangle$ to $q$                                  Send Deferred Forks
23 :           $\mathsf{fork}_{pq} :=$ false

24 : procedure Lower (x : priority)                                      Reduce Priority
25 :       x.height := x.height $- m$
```

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

variable $token_{qp}$ corresponding to p. A process *cannot* request a missing fork *without* holding the corresponding token, and a process requests a missing fork by sending the token to the corresponding neighbor. Similarly, the token is unique and exclusive, and hence, it is never the case that both token variables $token_{pq}$ and $token_{qp}$ are true. Initially, between each pair of neighbors, the fork is at the neighbor with the higher identifier, and the token is at the neighbor with the lower identifier.

5.2.2. Algorithm Actions

Action 1 states that a thinking process can become hungry at any time or remain thinking forever. Action 1 is actually not an internal action of the dining algorithm and is formalized only for completeness of process behaviors.

Action 2 requests missing forks. This action is always enabled at hungry processes. When executed, the hungry process p requests every missing fork for which no previous request is currently pending. This is achieved by sending the request token to the corresponding neighbor. Also, p encodes its current priority into the token message. As a result, the token variable is set to false to indicate that a request has been sent.

Action 3 decides whether or not to grant a fork request. Via Action 3, processes receive a fork request. If the recipient is thinking, or hungry but has a priority lower than the request sender, then the fork must be sent immediately. Otherwise, the request is deferred until after the recipient exits eating. Eating and exiting processes always defer requests, as do higher-priority hungry processes. Deferred requests are represented by holding both the shared fork and the request token. Note that if a hungry process loses a 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.

Action 5 determines when a hungry process can proceed to eat. A hungry process p can go to eat, if for each neighbor q, process p either holds the shared fork or currently suspects q. Action 5 is the only action that utilizes the failure detector $\diamond\mathcal{P}_1$. In this action, suspicion serves as a proxy for missing forks.

Action 6 transits an eating process to exiting. This action is not an internal action of the dining solution and formalized only for completeness of process behaviors.

Action 7 transits a process from exiting back to thinking. Via this action, an exiting process reduces its priority, transits back to thinking, and sends the forks that were previously deferred while hungry, eating, or exiting. This action invokes a local procedure *Lower* that reduces the *height* component of priority by some positive integer m. The magnitude of the reduction m is up to the algorithm designer, and can be either statically fixed or dynamically chosen at runtime.

Action 7 masks a deeper subtlety that arises from an intrinsic limitation on local knowledge. After eating, a process p does not need to reduce its priority lower than that of all the neighbors. p only needs to reduce its height by an arbitrary integer number. Hence, p does not need to know the priority of its neighbors. Also, reducing height by an arbitrary integer number is still sufficient for wait-freedom.

5.3. Correctness Proof

This section proves that Algorithm 1 satisfies the safety and progress properties.

5.3.1. Safety

Safety ($\diamond\mathcal{WX}$) is guaranteed by uniqueness of forks and local eventual strong accuracy of $\diamond\mathcal{P}_1$. We will first prove Lemma 1; that is, there exists exactly one fork for each pair of live neighbors. Next, we prove that Algorithm 1 satisfies eventual weak exclusion by direct construction in Theorem 1.

Lemma 1 *In Algorithm 1, there exists exactly one fork for each pair of live neighbors.*

Proof. Initially, there exists exactly one fork for each pair of neighbors. Reliable communication channels also guarantee that forks cannot be lost, duplicated, and corrupted in transit. Therefore, we only need to concentrate on Action 3, 4, and 7, in which fork variables are modified. By Action 3 and 7, no absent fork can be sent. Hence, no fork can be duplicated. By reliable communication channels and Action 4, forks in transit will be eventually received and stored at a local variable. Hence, no fork can be lost. Thus, Lemma 1 holds. $\square$

Theorem 1 (Safety): *Algorithm 1 satisfies eventual weak exclusion. That is, for every run, there exists a time after which no two live neighbors eat simultaneously.*

Proof. The safety proof is by direct construction and depends on Lemma 1 and the local eventual strong accuracy property. Local eventual strong accuracy guarantees that for each run, there exists a time t after which no correct process is suspected by any correct neighbor, where t is unknown and may vary from run to run.

First, faulty processes cannot prevent safety 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. Hence, we will restrict our focus to just correct processes.

Second, correct neighbors cannot *begin* overlapping eating sessions after $\diamond\mathcal{P}_1$ converges. For each execution α, let t be the time after which $\diamond\mathcal{P}_1$ never suspects correct neighbors. Let p be any correct process that *begins* eating after time t. By Action 5, process p can transit from hungry to eating if for each neighbor q, p either holds the shared fork or suspects q. Since $\diamond\mathcal{P}_1$ never suspects correct neighbors after time t, process p must hold all the forks shared with its correct neighbors in order to begin eating.

So long as p remains eating, Action 3 guarantees that p will defer all fork requests. Thus, p will not relinquish any fork while eating. Furthermore, $\diamond\mathcal{P}_1$ has already converged after time t, so no correct neighbor can suspect p either. Because of Lemma 1, there exists exactly one fork for each pair of live neighbors. Consequently, Action 5 remains disabled for every correct hungry neighbor of p until after p transits back to thinking. Thus, we conclude that no pair of correct neighbors can *begin* overlapping eating sessions after time t.

Third, let us consider the eating sessions that begin before time t. After time t, correct neighbors may still eat simultaneously because their current eating sessions may begin before time t. For example, if two correct neighbors p and q start their overlapping eating sessions before time t, then it is allowed that both of them remain eating through time t. However, correct processes can eat only for a finite period of time. There exists a time t' after which all eating sessions that begin before time t end. Since no pair of correct neighbors can *begin* overlapping eating sessions after time t, no *correct* neighbors eat simultaneously after $max(t', t)$.

Suppose that all faulty processes crash by time t'' in α. After time t'', every live process is correct. Thus, after the time $max(t, t', t'')$, no *live* neighbors eat simultaneously. Theorem 1 holds. $\square$

5.3.2. Progress

The progress (i.e., wait-freedom) proof is based on local strong completeness of $\diamond\mathcal{P}_1$ and a dynamic total ordering on process priorities. Local strong completeness serves as a proxy so that a process does not need to wait for a fork shared with a crashed neighbor. The dynamic total ordering guarantees that no deadlock can be formed, and also guarantees that no correct hungry process can be overtaken infinitely many times by its neighbors. Hence, every correct hungry process eventually eats. We will first define process depth in the ordering, and then prove the progress property by complete induction on process depth.

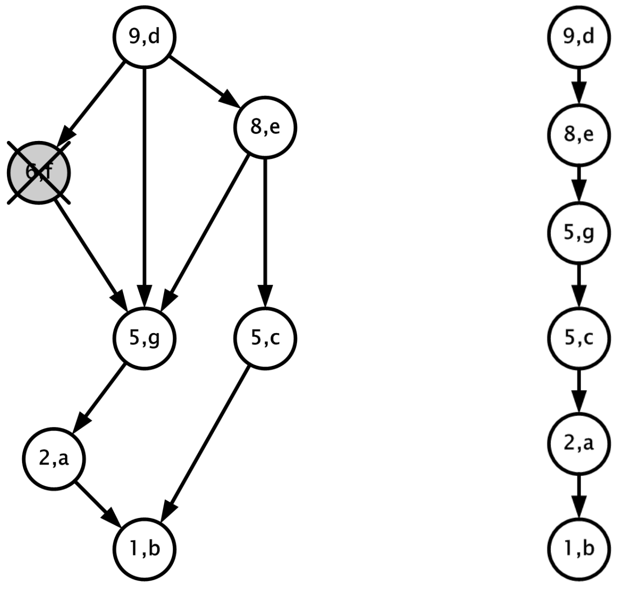
Process Depth. The depth of a process p at time t is defined as the number of processes in $live(t)$ that have a priority greater than the priority of p, where $live(t)$ is the set of live processes at time t. Therefore, $depth(p,t)$ is formally defined as follows:

$$depth(p,t) = |\{\forall q \in live(t) : priority_q > priority_p : q\}| \tag{5.1}$$

One example of the depth function is shown in Figure 5. Subfigure 5(a) shows a conflict graph with priorities for each process. For example, process a has a priority $(2, a)$, in which 2 is its height, and a is its *id*. Process *id*s are represented by lowercase letters, and process *id*s are ordered lexicographically as $a < b < c < d \cdots$. At the time t, process f is crashed, and all other processes are live.

Since no two processes have the same priority, the live processes in Subfigure 5(a) can be totally ordered as shown in Subfigure 5(b) based on priority. Note that the crashed process f is not included. Process d has the highest global priority, so $depth(d,t) = 0$. Although process c has a *relative* depth 2 in the conflict graph, its *absolute* depth is 3 in the global depth ordering. Process c is lower than process g (i.e., $depth(c,t) > depth(g,t)$), because although both g and c have equal heights, process g has the higher process *id* than c (i.e., $g > c$).

The depth of a process can increase only when the process exits eating and lowers its priority. While a process is hungry, its depth cannot increase, but may decrease when other higher-priority processes lower their priorities.



(a) Conflict Graph (b) Depth Ordering

Fig. 5. Absolute Depth of Live Processes in A Conflict Graph

Theorem 2 (Progress): *Algorithm 1 is wait-free so that every correct hungry process eventually eats.*

Proof. The proof is constructed *after* $\Diamond\mathcal{P}_1$ converges. A hungry process may or may not be scheduled to eat before $\Diamond\mathcal{P}_1$ converges. However, we only need to prove that after $\Diamond\mathcal{P}_1$ converges, every correct hungry process is guaranteed to eat eventually.

Wait-freedom is proved by complete induction on process depth. The base case shows that every correct hungry process with depth 0 eventually eats. The inductive step assumes that every correct hungry process with $depth < d$ eventually eats, and then proves that correct hungry processes with depth d also eventually eat.

Base Case: $depth = 0$.

The base case states that for a correct hungry process p, if $depth(p,t) = 0$ at time t, then p eats at a later time. To prove the base case, we must show that for each neighbor, p eventually holds the shared fork continuously, or suspects that neighbor, or both (Action 5). We partition the neighbors of p into two disjoint sets: *correct* and *faulty*.

First, let us consider the *faulty* neighbors. Every faulty neighbor eventually crashes. By the local strong completeness property of $\diamond\mathcal{P}_1$, every crashed neighbor is eventually and permanently suspected by p. Thus, p will eventually and permanently suspect all faulty neighbors.

Note that process p may go to eat not necessarily by suspecting its faulty neighbor q. Process p may already hold the shared fork when p becomes hungry, or p may get the shared fork before the faulty neighbor q crashes. Thus, process p may proceed to eat before p suspects q. However, p is not guaranteed to get the shared forks from its faulty neighbors.

Second, let us consider the *correct* neighbors. We will show that process p eventually holds every fork shared with its correct neighbors continuously until after p eats. After becoming hungry, p sends requests for all of its missing forks (Action 2). Since channels are reliable, each fork request sent to a correct neighbor q will eventually be received, thereby causing q to execute Action 3. There are two possible outcomes: (1) if q is thinking or hungry, then q sends the shared fork immediately (recall $priority_p > priority_q$), or (2) if q is eating or exiting, then the fork request is temporarily deferred. As a correct process, however, q will exit eating after a finite period of time (Action 6). As a result, q immediately or eventually sends the requested fork back to p (Action 7).

Process p cannot lose its forks while hungry. The reason is that hungry processes relinquish forks only to their higher-priority hungry neighbors (Action 3), and process p has no higher-priority neighbors ($depth(p, t) = 0$).

At this point, we can conclude that p is guaranteed to collect every shared fork from each correct neighbor, and that p either collects the shared fork or permanently suspects each faulty neighbor. Thus, p is guaranteed to eat by Action 5.

The base case also proves that a correct hungry process at *any* depth will not wait indefinitely on its lower-priority neighbors. In the special case of depth 0, this is enough to establish progress, because all neighbors have lower priority. This result will be applied again in the inductive step.

Inductive Step: $depth = d$.

The inductive step states that if every correct hungry process with $depth < d$ is guaranteed to eat eventually, then every correct hungry process with depth d is also guaranteed to eat eventually. Given such a correct hungry process p which depth is d at time t, we will show that for each neighbor, p eventually holds the shared fork continuously, or suspects that neighbor, or both.

Because no process reduces its priority while hungry, and no process increases its priority at any time, the depth of p will never become deeper than d so long as p remains hungry. Priority changes only when an eating process executes Action 7 to transit back to thinking. This change is achieved by invoking the local procedure $Lower(priority_p)$, which reduces the *height* component of $priority_p$ by an arbitrary positive integer. Therefore, while p remains hungry, its lower-priority neighbors always stay lower than p in the total ordering, but its higher-priority neighbor may not always stay higher than p.

For each lower-priority neighbor, p eventually holds the shared fork continuously, or suspects that neighbor, or both. This claim can be proved by applying the same argument in the base case, because while p remains hungry, p always has a higher priority than these neighbors.

Higher-priority neighbors are analyzed by two cases: (1) some higher-priority live process is faulty, or (2) all higher-priority processes are correct.

Case 1: *There exists a faulty process r and $depth(r, t) < d$ at time t.*

Faulty processes eventually crash. Hence, there is a later time t' at which r crashes, where $t' \geq t$. Crashed processes are irrelevant to depth, so the crash of r actually causes the depth of p to rise, even though the priority of p remains unchanged. Consequently, we have $depth(p, t') < d$ at time t', and so by applying the inductive hypothesis, we conclude that p eventually eats.

Case 2: *Every process that has a higher priority than p at time t is correct.*

All future extensions after time t can be divided into two subcases: (a) futures in which every higher-priority process eventually thinks forever, and (b) futures in which some higher-priority process becomes hungry infinitely often.

Case 2(a) *There exists a time $t'' > t$ after which all correct higher-priority processes remain thinking forever.*

We will show that p eventually acquires all forks shared with these correct higher-priority neighbors. Suppose that q is a correct higher-priority neighbor that thinks forever after time t'', and suppose that p does not currently hold the fork shared with q at time t''. If the fork is already in transit from q to p, then it will arrive eventually. Otherwise, Action 2 remains enabled while p is hungry, so the fork request has or will be sent to q. Since q thinks forever after time t'', the fork request will be granted when q executes Action 3. Once the fork arrives at p, it will never be relinquished again to q because q never becomes hungry in the future. Consequently, p will collect and hold the shared fork for each of its higher-priority neighbors, and will collect the shared fork or permanently suspect each of its lower-priority neighbors. This enables Action 5, so p eventually eats.

Case 2(b) *Some correct higher-priority process r becomes hungry infinitely often.*

Recall that $depth(p, t) = d$ at time t. Since r is correct and has a higher priority than p at time t, we know that $depth(r, t) < d$. Every time that r becomes hungry with depth less than d, we can apply the inductive hypothesis to conclude that r

eventually eats. Since r is a correct process, it eventually transits from exiting back to thinking by executing Action 7. This action reduces the integer-valued height component of its priority by at least 1 every time it is executed. Because p does not change its priority while p is hungry, r can go to eat finitely many times while p is hungry before the priority of r must be lower than that of p. At this time, *depth* of p is lower than d, and by the inductive hypothesis, p eventually goes to eat.

At this point, the inductive step has been proved. Based on the base case and inductive step, we can conclude that Theorem 2 holds. $\square$

It is worth to point out that the mistakes permitted by local eventual strong accuracy may enable p to eat earlier. For example, $\diamond\mathcal{P}_1$ could initially suspect every neighbor, and thereby enable p to proceed directly to eating. Although such mistakes may have implications for safety, they cannot forestall progress.

5.4. Analysis

Algorithm 1 satisfies some useful properties. First, it requires only *bounded capacity* on channels (i.e., the number of messages on any channel at any given time is upper bounded). Algorithm 1 has two types of messages: fork requests (tokens) and forks. Because forks and tokens are unique between each pair of neighbors, there can be at most two messages in transit between each pair of neighbors at any time. Therefore, the number of messages is bounded on any channel. Note that although we do not count the messages used to implement $\diamond\mathcal{P}_1$, some implementation of failure detectors (e.g., ping-ack implementations) also requires only *bounded capacity* on channels. Second, correct thinking processes are quiescent; they eventually stop communicating with other processes. This property can be demonstrated as follows. Assume that a correct process remains thinking forever after a time t. For each neighbor q, if q also remains thinking forever after some time t', then eventually p and q do not need to communicate according to the algorithm. If q becomes hungry infinitely often, then q eventually has to hold the fork shared with p to eat, and eventually q will not return the fork back because p will never request the fork after time t. Thereafter, neither q nor p needs to request the fork, and there is no message between p and p.

Algorithm 1 also has some disadvantages. First, unbounded local memory is used. If a process eats infinitely many times, then the height component of its priority will decrease infinitely many times, and every time it is decreased by at least by 1. Therefore, *priority* variables are unbounded and need unbounded local memory. Second, in order to eat, a process may need to send or receive a finite but unbounded number of messages while hungry. The reason is that a process can be overtaken by its neighbor finitely but unboundedly many times. For a process p, every time a higher-priority neighbor becomes hungry, the neighbor can send a message to preempt the fork from p. This can occur finitely many times until the neighbor decreases its priority lower than p.

CHAPTER VI

BOUNDED FAIRNESS

This chapter focuses on bounded fairness properties of dining philosophers. First, we formally define two perpetual bounded fairness properties (i.e., perpetual bounded waiting and perpetual k-bounded waiting) and two eventual bounded fairness properties (i.e., eventual bounded waiting and eventual k-bounded waiting). Second, we explore the relationships among these fairness properties. Third, we prove that $\Diamond\mathcal{P}$ cannot deterministically solve wait-free, *perpetual k-bounded waiting* dining under $\Diamond\mathcal{WX}$. This impossibility result forces us to consider a weaker fairness property, eventual k-bounded waiting, in the next chapter. Finally, we review research works related to bounded fairness properties.

6.1. Bounded-Waiting Properties

Starting from technical terms such as hungry sections and wait functions, this section formally defines bounded-waiting properties.

6.1.1. Hungry Sections and Wait Functions

A *hungry section* is an execution segment; it starts when a process i becomes hungry, and ends when i goes to eat or crashes. Process i remains hungry during its hungry sections. Because processes may become hungry more than once in an execution, processes may have multiple hungry sections in an execution. We denote the mth hungry section of process i in an execution α as $\mathcal{HS}^{\alpha}_{i,m}$, and denote the starting and ending time of $\mathcal{HS}^{\alpha}_{i,m}$ as $ts^{\alpha}_{i,m}$ and $te^{\alpha}_{i,m}$, respectively. As such, process i

must be alive before time $te^{\alpha}_{i,m}$. For algorithms that are not wait-free, $te^{\alpha}_{i,m}$ may be infinite.

A *partial hungry section* $\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2)$ is a segment of the hungry section $\mathcal{HS}^{\alpha}_{i,m}$. The partial hungry section $\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2)$ starts from time t_1 and ends at time t_2, where $ts^{\alpha}_{i,m} \le t_1 < t_2 \le te^{\alpha}_{i,m}$. When $t_1 = ts^{\alpha}_{i,m}$ and $t_2 = te^{\alpha}_{i,m}$, $\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2)$ becomes the hungry section $\mathcal{HS}^{\alpha}_{i,m}$. Therefore, hungry sections are a special case of partial hungry sections. In particular, if $t_1 = ts^{\alpha}_{i,m}$, we say $\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2)$ is a *section prefix*. If $t_2 = te^{\alpha}_{i,m}$, we say $\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2)$ is a *section suffix*.

A *wait function* returns the number of times that a process goes to eat during a (partial) hungry section. Hence, wait functions map from HS $\times$ Π to $\mathbb{N}$, where HS denotes a set of (partial) hungry sections, Π is the set of processes in the system, and $\mathbb{N}$ is a set of natural numbers. WAIT$(\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2), j)$ returns the number of times that process j goes to eat during the partial hungry section $\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2)$. Formally, WAIT$(\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2), j) = k$ if and only if

$$\exists r \in \mathbb{N} : t_1 < te^{\alpha}_{j,r} < te^{\alpha}_{j,r+1} < te^{\alpha}_{j,r+2} \cdots < te^{\alpha}_{j,r+k-1} < t_2$$

where $te^{\alpha}_{j,r}$ represents the time when process j goes to eat in execution α for the rth time. If processes i and j are neighbors, then WAIT$(\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2), j)$ is the number of times that i is *overtaken* by j while i remains hungry from t_1 to t_2.

6.1.2. Perpetual Bounded-Waiting Properties

Intuitively, perpetual bounded-waiting properties guarantee that wait functions are upper bounded for every pair of live neighbors during each execution. If the upper bound may vary from execution to execution, then it is *perpetual bounded waiting* ($\Box\mathcal{BW}$). If there exists an upper bound k for all executions, then it is *perpetual k-bounded waiting* ($\Box k$-$\mathcal{BW}$).

Perpetual Bounded Waiting($\Box\mathcal{BW}$): for each execution, there exists a natural number r such that no live process goes to eat more than r times while any live neighbor is waiting to eat. For a dining algorithm $\mathcal{A}$, let $\mathcal{EXEC}_{\mathcal{A}}$ be the set of all possible executions of algorithm $\mathcal{A}$, and express $\mathcal{EXEC}_{\mathcal{A}}$ as $\{\alpha_1, \alpha_2, \cdots\}$. Also, let $\mathcal{HSS}_{\alpha}$ be the set of all hungry sections in an execution α, and let $N(i)$ be the set of process i's neighbors. Formally, if $\mathcal{A}$ satisfies $\Box\mathcal{BW}$, then:

$$\forall \alpha \in \mathcal{EXEC}_{\mathcal{A}}, \exists r \in \mathbb{N}, \forall i \in \Pi, \forall \mathcal{HS}^{\alpha}_{i,m} \in \mathcal{HSS}_{\alpha}, \forall j \in N(i) :$$
$$\mathsf{WAIT}(\mathcal{HS}^{\alpha}_{i,m}, j) \leq r$$

Note that bounded waiting is defined with respect to neighbors; that is, processes i and j are neighbors in the above definition. If processes i and j are not neighbors, then WAIT$(\mathcal{HS}^{\alpha}_{i,m}, j)$ may not be bounded by r. Hence, $\Box\mathcal{BW}$ guarantees that for each execution, there exists an upper bound r on the wait function *for every pair of live neighbors.*

The bound r may vary from execution to execution. Let us denote the bound in execution α_i as r_i, then for all executions in $\mathcal{EXEC}_{\mathcal{A}}$, the bounds can be expressed as a set $\mathcal{R}_{\mathcal{A}} = \{r_1, r_2, \cdots\}$. $\Box\mathcal{BW}$ does not impose limits on r_i; there may or may not exist a maximum value in set $\mathcal{R}_{\mathcal{A}}$. If such a value k does exist, then a stronger bounded-waiting property, *perpetual k-bounded waiting*, can be defined as follows.

Perpetual k-Bounded Waiting($\Box k$-$\mathcal{BW}$): there exists a natural number k for every execution such that no live process goes to eat more than k times, while any live neighbor is waiting to eat. In other words, for all executions, there exists an upper bound k on the wait function for every pair of live neighbors. Formally, if the algorithm $\mathcal{A}$ satisfies $\Box k$-$\mathcal{BW}$, then:

$$\exists k \in \mathbb{N}, \forall \alpha \in \mathcal{EXEC}_{\mathcal{A}}, \forall i \in \Pi, \forall \mathcal{HS}^{\alpha}_{i,m} \in \mathcal{HSS}_{\alpha}, \forall j \in N(i) :$$
$$\mathsf{WAIT}(\mathcal{HS}^{\alpha}_{i,m}, j) \leq k$$

6.1.3. Eventual Bounded-Waiting Properties

Intuitively, eventual bounded-waiting properties guarantee that wait functions are *eventually* upper bounded for every pair of live neighbors during each execution. For each execution α, there exists a *convergence time* t after which the bound holds. If the bound may vary from execution to execution, then it is *eventual bounded waiting* ($\Diamond\mathcal{BW}$). If there exists an upper bound k for all executions, then it is *eventual k-bounded waiting* ($\Diamond k$-$\mathcal{BW}$).

For an execution α, the convergence time t may split some hungry sections into two parts: the part before time t (i.e., a section prefix ending at time t) and the part after time t (i.e., a section suffix starting from time t). For these splitted hungry sections, bounded fairness holds only during the section suffixes that start from time t. As such, we define two disjoint sets of (partial) hungry sections based on the time t: $\mathcal{HSS}_\alpha(0,t)$ and $\mathcal{HSS}_\alpha(t,\infty)$ as follows.

The set $\mathcal{HSS}_\alpha(0,t)$ consists of (1) all hungry sections ending before time t in execution α, and (2) all section prefixes ending at time t in execution α.

The set $\mathcal{HSS}_\alpha(t,\infty)$ consists of (1) all hungry sections starting after time t in execution α, and (2) all section suffixes starting from time t in execution α.

Eventual Bounded Waiting($\Diamond\mathcal{BW}$): for each execution, there exist a natural number r and an unknown time t after which no live process goes to eat more than r times, while any live neighbor is waiting to eat. In other words, for each execution, there exists an upper bound on wait functions for every pair of live neighbors and *for all (partial) hungry sections in $\mathcal{HSS}_\alpha(t,\infty)$*. The convergence time t and the bound r may vary from execution to execution. Formally, if a dining algorithm $\mathcal{A}$ satisfies $\Diamond\mathcal{BW}$, then:

$$\forall\alpha\in\mathcal{EXEC}_\mathcal{A},\exists r\in\mathbb{N},\exists t\in\mathbb{T},\forall i\in\Pi,\forall\mathcal{HS}^\alpha_{i,m}(t_1,t_2)\in\mathcal{HSS}_\alpha(t,\infty),\forall j\in N(i):$$

$$\mathsf{WAIT}(\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2), j) \leq r.$$

Eventual k-Bounded Waiting($\Diamond k$-$\mathcal{BW}$): there exists a natural number k such that for each execution, there exists a time t after which no live process goes to eat more than k times, while any live neighbor is waiting to eat. In other words, for all executions, there exists an upper bound k on wait functions for every pair of live neighbors and *all (partial) hungry sections in* $\mathcal{HSS}_{\alpha}(t, \infty)$, where the convergence time t may vary from execution to execution. Formally, if a dining algorithm $\mathcal{A}$ satisfies $\Diamond k$-$\mathcal{BW}$, then:

$$\exists k \in \mathbb{N}, \forall \alpha \in \mathcal{EXEC}_{\mathcal{A}}, \exists t \in \mathbb{T}, \forall i \in \Pi, \forall \mathcal{HS}^{\alpha}_{i,m}(t_1, t_2) \in \mathcal{HSS}_{\alpha}(t, \infty), \forall j \in N(i) :$$
$$\mathsf{WAIT}(\mathcal{HS}^{\alpha}_{i,m}(t_1, t_2), j) \leq k.$$

6.2. The Hierarchy of Bounded-Fairness Properties

This section explores the relationships among those bounded fairness properties and establishes a simple hierarchy for those properties.

The relationship between two fairness properties is defined as follows. For two fairness properties $\mathcal{P}_1$ and $\mathcal{P}_2$, if every algorithm that satisfies $\mathcal{P}_1$ also satisfies $\mathcal{P}_2$, we say that $\mathcal{P}_1$ is *stronger than* $\mathcal{P}_2$ (i.e., $\mathcal{P}_1 \geq \mathcal{P}_2$). Meanwhile, if $\mathcal{P}_2$ is not stronger than $\mathcal{P}_1$ (i.e., $\mathcal{P}_2 \ngeq \mathcal{P}_1$), then we say that $\mathcal{P}_1$ is *strictly* stronger than $\mathcal{P}_2$ (i.e., $\mathcal{P}_1 > \mathcal{P}_2$). If $\mathcal{P}_1$ is stronger than $\mathcal{P}_2$, and $\mathcal{P}_2$ is also stronger than $\mathcal{P}_1$, then $\mathcal{P}_1$ is *equivalent* to $\mathcal{P}_2$ (i.e., $\mathcal{P}_1 = \mathcal{P}_2$).

Based on the definitions, we can directly conclude Lemma 2, 3, and 4 as follows.

Lemma 2 *perpetual k-bounded waiting is stronger than eventual k-bounded waiting, and perpetual bounded waiting is also stronger than eventual bounded waiting.*

Based on their definitions, $\Box k$-$\mathcal{BW}$ can be viewed as a special case of $\Diamond k$-$\mathcal{BW}$

where the fairness bound k "eventually" holds from the beginning of executions. Similarly, $\Box\mathcal{BW}$ can be viewed as a special case of $\Diamond\mathcal{BW}$ where the fairness bounds "eventually" hold from the beginning of executions. Hence, Lemma 2 holds. $\Box$

Lemma 3 *Eventual k-bounded waiting is stronger than eventual bounded waiting.*

Based on the definitions, $\Diamond k\text{-}\mathcal{BW}$ is a special case of $\Diamond\mathcal{BW}$ where an eventual fairness bound k exists for all executions. $\Box$

Lemma 4 *Given two natural numbers k_1 and k_2, if $k_1 < k_2$, then perpetual k_1-bounded waiting is stronger than perpetual k_2-bounded waiting, and eventual k_1-bounded waiting is stronger than eventual k_2-bounded waiting.*

Clearly, every algorithm that satisfies perpetual/eventual k_1-bounded waiting also satisfies perpetual/eventual k_2-bounded waiting. Hence, Lemma 4 holds. $\Box$

Lemma 5 *The perpetual bounded-waiting property ($\Box\mathcal{BW}$) is equivalent to the eventual bounded-waiting property ($\Diamond\mathcal{BW}$).*

Proof: Lemma 2 already shows that $\Box\mathcal{BW} \geq \Diamond\mathcal{BW}$. Hence, we only need to prove $\Diamond\mathcal{BW} \geq \Box\mathcal{BW}$. To do so, we will show that if an algorithm $\mathcal{A}$ satisfies $\Diamond\mathcal{BW}$, then for each execution α of $\mathcal{A}$, there exists a perpetual fairness bound. That is, for each execution α, there exists an upper bound on the wait function for every pair of live neighbors and all hungry sections in α (i.e., the set $\mathcal{HSS}_\alpha$).

Because algorithm $\mathcal{A}$ satisfies $\Diamond\mathcal{BW}$, for execution α, there exists a bound r_1 and a time t after which no live process can go to eat more than r_1 times while other live neighbors are waiting to eat. We partition $\mathcal{HSS}_\alpha$ into three subsets based on time t: hungry sections starting after time t, hungry sections ending before time t, and others. We will prove that for every pair of live neighbors, the wait function is bounded for hungry sections in these three subsets one by one.

First, the wait function is bounded for every pair of live neighbors and all hungry sections *starting after* time t. Based on the definition of $\Diamond\mathcal{BW}$, the wait function is bounded for every pair of live neighbors and every (partial) hungry section in $\mathcal{HSS}_\alpha(t,\infty)$. Recall that $\mathcal{HSS}_\alpha(t,\infty)$ consists of all hungry sections starting after time t and all section suffixes starting at time t. Therefore, the wait function is bounded by r_1 for every pair of live neighbors and all hungry sections starting after time t.

Second, the wait function is bounded for every pair of live neighbors and all hungry sections *ending before* time t. According to $\Diamond\mathcal{BW}$, the convergence time t must be finite, and hence $\mathcal{HSS}_\alpha(0,t)$ contains finite elements (i.e., $|\mathcal{HSS}_\alpha(0,t)| < \infty$). Consequently, there must exist an upper bound r_2 on the wait function for the set $\mathcal{HSS}_\alpha(0,t)$. Recall that $\mathcal{HSS}_\alpha(0,t)$ consists of all hungry sections ending before time t and all section prefixes ending at time t. Therefore, the wait function is also bounded by r_2 for all hungry sections ending before time t.

Finally, the wait function is bounded for every pair of live neighbors and all hungry sections that are splitted by time t. Such a hungry section is splitted into two parts by time t: the section prefix ending at time t and the section suffix starting at time t. As discussed previously, for every pair of live neighbors, the wait function is bounded by r_1 for all section suffixes starting at time t and is also bounded by r_2 for all section prefixes ending at time t. Consequently, the wait function is bounded by $r_1 + r_2$ for all such hungry sections and every pair of live neighbors.

Every hungry section in $\mathcal{HSS}_\alpha$ belongs to one of the three subsets. Hence, the wait function is bounded by $r_1 + r_2$ for every pair of live neighbors and all hungry sections in execution α. Thus, $\Diamond\mathcal{BW} \geq \Box\mathcal{BW}$, and $\Diamond\mathcal{BW} = \Box\mathcal{BW}$. □

Lemma 6 *Eventual k-bounded waiting is stronger than perpetual bounded waiting.*

Proof: Lemma 3 already shows that $\Diamond k\text{-}\mathcal{BW} \geq \Diamond\mathcal{BW}$, and Lemma 5 shows that $\Diamond\mathcal{BW} = \Box\mathcal{BW}$. Therefore, $\Diamond k\text{-}\mathcal{BW} \geq \Box\mathcal{BW}$. $\Box$

Based on the above lemmas, relationships among bounded fairness properties can be expressed as a hierarchy in Figure 6, where $\mathcal{P}1 \rightarrow \mathcal{P}2$ denotes that $\mathcal{P}1$ is stronger than $\mathcal{P}2$.

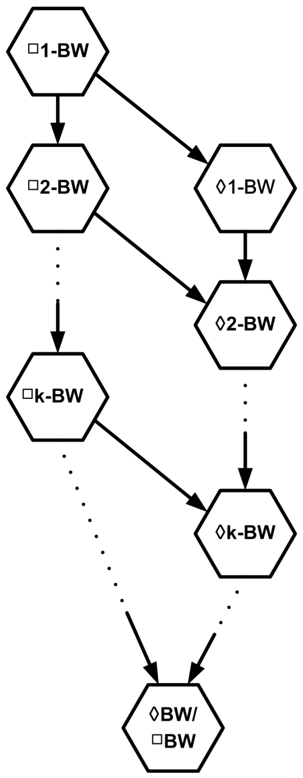


Fig. 6. Hierarchy of Bounded-Fairness Properties

6.3. Impossibility of Wait-free, Perpetually Bounded-Fair Dining Using $\Diamond\mathcal{P}_1$

Chapter V already demonstrated that $\Diamond\mathcal{P}_1$ can solve wait-free dining under $\Diamond\mathcal{WX}$. This solvability result intrigues one question: can $\Diamond\mathcal{P}_1$ be used to pursue an additional bounded fairness property for wait-free dining under $\Diamond\mathcal{WX}$? This section proves that $\Diamond\mathcal{P}_1$ cannot deterministically solve wait-free, *perpetual k-bounded* *waiting* dining under $\diamond\mathcal{WX}$. Hence, we are forced to consider a weaker bounded fairness property, *eventual k-bounded waiting*, in the next chapter.

We prove that $\diamond\mathcal{P}_1$ cannot deterministically solve wait-free, perpetual k-bounded waiting dining under $\diamond\mathcal{WX}$ by two steps. First, we show that for any two different fault patterns, $\diamond\mathcal{P}_1$ can provide the same information in finite prefixes of executions. Next, we prove the impossibility by contradiction using the fact that even in asynchronous systems augmented with $\diamond\mathcal{P}_1$, application layers may still be asynchronous.

Informally, $\diamond\mathcal{P}_1$ is not guaranteed to provide distinctly different information on different fault patterns over any finite prefix of executions. Let F_1 and F_2 be any two fault patterns on the set of processes Π. For an execution α, denote α^t as the finite prefix of α up through time t. Then we have Lemma 7 as follows.

Lemma 7 *Given an execution α_1 on the fault pattern F_1, for any time t and every live process p in Π, there exists an execution α_2 on the fault pattern F_2 such that $\diamond\mathcal{P}_1$ outputs the same information at process p in the finite prefixes α_1^t and α_2^t.*

Let us denote $\mathcal{H}_1(p,t')$, $\mathcal{H}_2(p,t')$ as the suspect lists of $\diamond\mathcal{P}_1$ at time t' at process p in the executions α_1, α_2 respectively. Lemma 7 states that at any time $t' \leq t$, $\mathcal{H}_1(p,t')$ is equal to $\mathcal{H}_2(p,t')$.

Proof. This lemma is proved by direct construction. We show that at any time $t' \leq t$, if a process q is added into or removed from $\mathcal{H}_1(p,t')$, it is eligible for $\diamond\mathcal{P}_1$ to do the same operation on $\mathcal{H}_2(p,t')$.

Case 1. Suppose that process q is *added* into $\mathcal{H}_1(p,t')$ at time t'. In the fault pattern F_2, process q is either faulty or correct. If q is faulty in F_2, clearly $\diamond\mathcal{P}_1$ can suspect q by adding q into $\mathcal{H}_2(p,t')$. If q is correct in F_2, $\diamond\mathcal{P}_1$ is still allowed to add q into $\mathcal{H}_2(p,t')$, because $\diamond\mathcal{P}_1$ can wrongfully suspect correct processes finitely many times. Although $\diamond\mathcal{P}_1$ has to remove q from $\mathcal{H}_2(p,t')$ at a later time, $\diamond\mathcal{P}_1$ can remove

q after time t.

Case 2. Suppose that process q is *removed* from $\mathcal{H}_1(p,t')$ at time t'. If q is correct in F_2, then it is clearly eligible for $\diamond\mathcal{P}_1$ to remove q from $\mathcal{H}_2(p,t')$. If q is faulty, $\diamond\mathcal{P}_1$ can still remove q from $\mathcal{H}_2(p,t')$. Because $\diamond\mathcal{P}_1$ is not forced to suspect a crashed process permanently from time t', removing q from the suspect list does not violate the strong completeness property of $\diamond\mathcal{P}_1$. Although $\diamond\mathcal{P}_1$ has to add q into $\mathcal{H}_2(p,t')$ at a later time, $\diamond\mathcal{P}_1$ can add q after time t.

Because $\diamond\mathcal{P}_1$ can perform the same operations in two finite prefixes α_1^t and α_2^t, $\mathcal{H}_1(p,t')$ and $\mathcal{H}_2(p,t')$ can be the same at any time t' before t. Lemma 7 holds. □

Theorem 3 *For asynchronous message-passing systems augmented with $\diamond\mathcal{P}_1$, there does not exist a deterministic dining algorithm that satisfies both wait-freedom and perpetual k-bounded waiting under eventual weak exclusion.*

Proof. Theorem 3 is proved by contradiction, which is formed by two indistinguishable executions. By contradiction, assume that there exists a deterministic dining algorithm $\mathcal{A}$, which is wait-free and perpetual k-bounded waiting under $\diamond\mathcal{WX}$ using $\diamond\mathcal{P}_1$. The fairness bound k is a natural number. Consider a system of two processes, p and q, and any execution prefix α resulting in a configuration C, in which both p and q are hungry. Starting from C, we construct a finite reference extension γ_r, in which p crashes and q eats $k+1$ times. Also, we force another finite alternate extension γ_a, in which p and q are correct, and perpetual k-bounded waiting is violated. The two executions are shown in Figure 7.

The finite reference extension γ_r is constructed as follows. Process q is correct, and process p crashes in the next step at time t_{pc}. Let C_1 denote the configuration at time t_{pc}. Consider an infinite schedule σ applicable to the configuration C_1. We require that the execution $exec(C_1,\sigma)$ is admissible such that process q takes infinitely

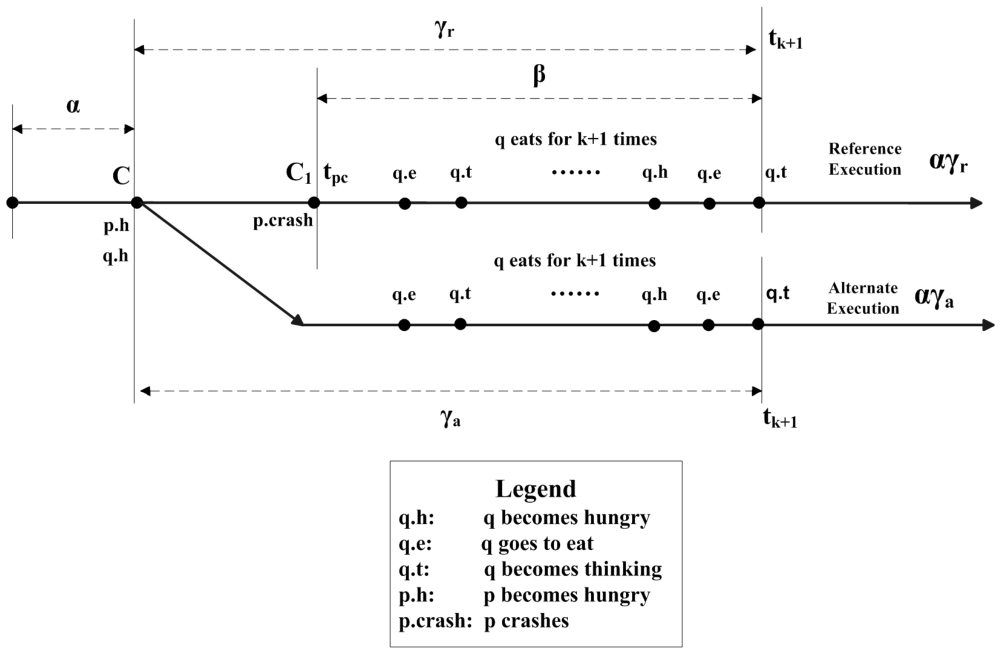


Fig. 7. Impossibility of Wait-Free, Perpetually Bounded-Fair Dining Using $\Diamond\mathcal{P}_1$

many steps and does not think permanently. Since q does not think forever, if q is thinking, then q eventually becomes hungry. Because the algorithm $\mathcal{A}$ is wait-free, q eventually eats. Because correct processes can eat only for a finite period of time, q eventually exits eating. As such, q becomes hungry infinitely often and also goes to eat infinitely often. So let t_{k+1} denote the time when q finishes its $k+1^{st}$ eating since p crashed, and let β be the finite execution segment from time t_{pc} to time t_{k+1}. Also let schedule σ_β be the finite schedule corresponding to β.

Now consider the alternate extension γ_a after α, where p and q are correct. γ_a is constructed by applying the schedule σ_β to the configuration C. As such, γ_a can be expressed as $exec(C, \sigma_\beta)$. However, we have to show that the schedule σ_β is applicable to the configuration C.

First, since the schedule σ_β only consists of actions of process q, we need to show that process p is permitted not to take steps in the finite execution γ_a. Process

p does not crash as it did in the reference execution. Also, because p is hungry in configuration C, and algorithm $\mathcal{A}$ is wait-free, p must eat at some time. Recall that in asynchronous systems augmented with $\diamond\mathcal{P}_1$, application layers may still be asynchronous (Section 2.3). Hence, p is permitted to be very slow in application layers. Additionally, any (application) message sent by process q in the execution γ_a can be subject to arbitrary delays as well. In particular, all such messages sent by q can be delayed until after time t_{k+1}. Hence, p is permitted to stay hungry and not to take any step in γ_a.

Second, we need to show that process q is still allowed to execute actions of the schedule σ_β in γ_a. By Lemma 7, $\diamond\mathcal{P}_1$ can provide the same information to process q in the finite execution prefixes $\alpha\gamma_a$ and $\alpha\gamma_r$. Meanwhile, any message sent by process p in γ_a can be delayed until after time t_{k+1}. Therefore, the execution segments γ_a and γ_r are indistinguishable from the perspective of process q. Because the algorithm $\mathcal{A}$ is deterministic, process q can still go to eat $k+1$ times in execution γ_a.

Based on the above analysis, we conclude that the schedule σ_β is applicable to configuration C. At this point, we know that during the finite execution segment γ_a, process p remains hungry, and process q goes to eat $k+1$ times. Therefore, perpetual k-bounded waiting is violated in γ_a. This contradicts the initial assumption that the algorithm $\mathcal{A}$ is perpetual k-bounded waiting. Thus, Theorem 3 holds. $\square$

Note that this proof is established before $\diamond\mathcal{P}_1$ converges and uses the fact that application layers may still be asynchronous. This implies that this impossibility result is mainly due to unreliability of the information provided by $\diamond\mathcal{P}_1$. During the finite prefixes, processes cannot distinguish a faulty process from a slow process based on the information provided by $\diamond\mathcal{P}_1$. This proof demonstrates the limitation of $\diamond\mathcal{P}_1$ on achieving perpetual bounded fairness.

6.4. Related Work

6.4.1. Bounded-Waiting Algorithms

Bounded waiting, also called bounded bypassing in the literature [46, 71], has been studied mostly in the context of mutual exclusion [16, 56, 58, 72, 73]. Therefore, most algorithms introduced here are designed for mutual exclusion.

Many bounded-waiting algorithms [16, 71, 72, 73, 74] are designed for *shared-memory systems*. Burns [72] presented a linear-waiting mutual exclusion algorithm. This algorithm uses n read/write shared variables and one test-and-set-high shared variable in a system of n processes. Burns et al. [16] presented two bounded-waiting algorithms for mutual exclusion with deadlock freedom. Burns et al. [16] also proved a lower bound on shared memory needed to achieve perpetual k-bounded waiting. If an algorithm solves mutual exclusion with both deadlock freedom and perpetual k-bounded waiting, then this algorithm must use at least n distinct shared-memory states. In the context of dining philosophers, Beauquier et al. [4] presented a bounded-waiting algorithm in shared-memory systems. Since our dissertation focuses on message-passing systems, we only mention a few algorithms under shared-memory systems and skip the description.

Under message-passing systems, Lamport [56] presented a bounded-waiting algorithm with reliable FIFO channels for mutual exclusion. This algorithm uses logical clocks to order events, and when two events have the same timestamp, process ids are used to break the tie. Hence, all logical events can be totally ordered. Meanwhile, each hungry process synchronizes its own timestamp with other processes by a ping-ack protocol. As such, after a process becomes hungry at a timestamp T_1, no other process can have three consecutive hungry events (i.e., becomes hungry) with timestamps earlier than T_1. Therefore, this algorithm is perpetual 2-bounded waiting.

Ricart and Agrawala [58] also presented a perpetual k-bounded waiting algorithm for mutual exclusion in message-passing systems, where $k = 2$ under FIFO channels, and $k = n$ under non-FIFO channels. This algorithm also maintains a total ordering of events by a similar way as in [56]. When two processes are competing for entering critical sections, the conflict is always resolved in favor of the process which hungry event is earlier in the ordering. As such, perpetual k-bounded waiting can be achieved.

6.4.2. Related Concepts

In some papers [55, 75, 76, 77], "bounded waiting" is defined with respect to a subphase of hungry sections, and the subphase is defined by a particular solution such as doorway algorithms. *A doorway* [26, 55, 75] is a piece of code such that a process p inside the doorway (i.e., already started executing the code) prevents its neighbors from entering the doorway until after p exits the doorway (i.e., finishes the execution). Processes can eat only inside the doorway. In [75], "r-bounded waiting" guarantees that while a hungry process p is inside the doorway, none of its neighbors can goes to eat more than r times. However, before entering the doorway, p can be overtaken finitely but unboundedly many times by its neighbor. Therefore, these definitions [55, 75, 76, 77] are different from our definition.

Unison [78, 79] can be viewed as another form of fairness. In a unison system, each process has a clock variable, and the clock is assigned to a value $i + 1$ if and only if the clock value of every neighboring process is either i or $i + 1$. Note that *neighbors* here are not neighbors in conflict graphs. As a result, while the clock of a process stays at a value i, its neighbors can increment their clock values at most twice (i.e., from $i - 1$ to i and from i to $i + 1$). Hence, unison can be considered a form of perpetual 2-bounded waiting. Several self-stabilizing unison algorithms [78, 79] were presented to tolerate transient faults, but not crash faults.

CHAPTER VII

A WAIT-FREE, EVENTUALLY BOUNDED-FAIR DINING SOLUTION USING $\Diamond\mathcal{P}_1$

This chapter demonstrates that $\Diamond\mathcal{P}_1$ is sufficient to solve the wait-free, eventually bounded-fair dining philosophers problem (i.e., WF-EBF dining). The previous chapter shows that $\Diamond\mathcal{P}_1$ is sufficient for solving wait-free dining under $\Diamond\mathcal{WX}$. This chapter takes the further step to show that $\Diamond\mathcal{P}_1$ can be used to achieve an additional property for wait-free dining: eventual k-bounded waiting. In particular, a wait-free, eventual 2-bounded waiting dining algorithm is constructed [14]. [1]

We present the 2-bounded waiting algorithm as follows. Because our algorithm is related to the classic asynchronous doorway dining algorithm [26], we will briefly describe the original doorway algorithm first. After that, we will describe the basic idea, variables, and actions of our algorithm, and then give the correctness proof.

7.1. The Original Asynchronous Doorway Algorithm

The asynchronous doorway algorithm was presented by Choy and Singh [26] in asynchronous message-passing systems. This doorway algorithm satisfies perpetual weak exclusion and lockout freedom. Specifically, safety (i.e., perpetual weak exclu-

Part of this chapter is reprinted from *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Y. Song and S. Pike, Eventually k-Bounded Wait-Free Distributed Daemons, pp. 645-655, Copyright (2007), with permission from IEEE.

[1]The solution presented here is different from the one published in [14], in which diners have only three states: *thinking, hungry*, and *eating*. To be consistent with this dissertation, we revised the algorithm in [14] with the additional *exiting* state.

sion) is ensured by forks, and progress (i.e., lockout freedom) is guaranteed by static process priorities, an asynchronous doorway, and the fork-collection scheme. However, this algorithm is not wait-free, and its failure locality is 3. We will first introduce forks, colors, and doorways, and then briefly describe this algorithm. Note that in the original doorway algorithm, diners have only three states: thinking, hungry, and eating. For a detailed description and the correctness proof, we refer readers to the original papers [26, 61].

Forks are used to guarantee safety as they are used in the hygienic algorithm. Forks are unique between each pair of neighbors, and processes must hold all shared forks to eat. Hence, perpetual weak exclusion can be guaranteed.

Colors represent *static* process priorities. Initially, every process is assigned a *color* such that no two neighbors have the same color. Color assignment can be done by applying node-coloring algorithms to conflict graphs. For example, a greedy algorithm can color all nodes by $\delta + 1$ colors in a conflict graph, where δ is the maximum degree of the conflict graph. Process colors are static and never change, and they are represented by integers.

For each pair of neighbors, the neighbor with a higher color has a higher priority. [2] When two neighbors are competing for the shared fork, the conflict is always resolved in favor of the higher-color neighbor. Let $color(p)$ denote the color of a process p. For two neighbors p and q, if $color(p) > color(q)$, then p has priority over q. As such, we say that p is a *high* neighbor of q, and q is a *low* neighbor of p. For a process, the forks shared with its high neighbors are called *high* forks, and the forks shared with its low neighbors are called *low* forks. It seems that a process may be overtaken by its high

[2] In the original paper [26], a *lower* color represents a *higher* priority. The presentation is changed here because it is easier for readers to associate *higher* colors with *higher* priorities.

neighbors infinitely many times and starve. However, this situation cannot happen because an asynchronous doorway is used to prevent processes from overtaking their low neighbors infinitely many times.

A doorway [26, 55, 75] is a piece of code with the following property: each process p inside the doorway attempts to prevent its neighbors from entering the doorway until after p exits the doorway. A doorway usually consists of an entry code and an exit code. A process *enters* the doorway by executing the entry code, and *exits* the doorway by executing the exit code. A process p is said to be *inside* a doorway if and only if p entered the doorway but has not exited the doorway yet; otherwise, p is *outside* the doorway. Eating processes must be inside the doorway.

To enter a doorway, a process p must get a permission from each neighbor q. However, processes inside the doorway attempt to prohibit their neighbors from entering the doorway. As such, while q is inside the doorway, q refuses to give permissions to p, regardless of their relative priorities (i.e., colors). Put otherwise, only when q is outside the doorway, q can grant permissions to p. A simple implementation for doorways is that process p checks the state of its neighbor q [26]. If p observes q is outside the doorway, then p gets the permission from q.

A doorway is *asynchronous* if processes only need *independent* permissions to enter the doorway. That is, after a process p becomes hungry, once p receives a permission from a neighbor q, then p does not need permissions from q any more before entering the doorway. By the above simple implementation, p just needs to observe that q is outside the doorway once, and thereafter, p does not need to check the state of q before entering the doorway. By contrast, to enter a *synchronous* doorway, p must observe that all neighbors are outside the doorway *simultaneously.*

The asynchronous doorway algorithm works as follows. Initially, every process is thinking and outside the doorway, and between each pair of neighbors, the higher-priority process holds the shared fork. In order to eat, a process must be inside the doorway and hold all of its shared forks. Thus, a hungry process must go through two phases to eat. At any time, a hungry process is either outside the doorway (Phase 1) or inside the doorway (Phase 2). Only processes inside the doorway are allowed to collect forks. Therefore, the goal for a process in Phase 1 is to enter the doorway. In Phase 2, processes aim to collect all missing forks. Processes stay inside the doorway until after they eat. Hence, no process can eat outside the doorway, and thinking processes are always outside the doorway. These two phases are described as follows.

Phase 1. Once a process p becomes hungry, p begins to collect permissions from all of its neighbors to enter the doorway. Its neighbor q grants a permission to p only when q is outside the doorway. After p gets a permission from each neighbor, p enters the doorway and proceeds to Phase 2.

Phase 2. Once process p enters the doorway, p executes *a fork-collection scheme* to collect all of its missing forks. Process p first requests all missing high forks. [3] For each high neighbor q, if q is outside the doorway, or q is also in Phase 2 but missing some of q's high forks, then q grants the fork request immediately. Otherwise, q defers the fork request until q either exits the doorway or loses some of q's high forks. Meanwhile, while p is still missing some high forks, p will grant any fork request from any neighbor.

Once p holds all the high forks, p begins to collect its missing low forks. For each low neighbor q, if q is outside the doorway or has not collected all of its shared forks, then q immediately grants the fork request. Otherwise, q defers the request until after q eats. Meanwhile, while p holds all high forks, p defers all fork requests from its low neighbors. However, p can be preempted by its high neighbors and must

[3]Note that we use higher colors for higher priorities. *High forks* actually are "low forks" in the original algorithm.

yield the forks requested by its high neighbors. As a result of losing high forks, p yields all previously deferred forks and re-requests missing high forks again.

After p collects all forks, p goes to eat. While eating, p defers fork requests from any neighbor. After eating, p transits back to thinking, exits the doorway, and sends all previously deferred forks.

Safety is ensured simply by uniqueness of forks between each pair of neighbors.

Lockout freedom can be demonstrated by showing that hungry processes can make progress in both Phase 1 and Phase 2. Every process inside the doorway eventually goes to eat and exits the doorway (Phase 2), and every hungry process eventually enters the doorway (Phase 1).

Hungry processes can make progress to eat in Phase 2. First, consider processes with the highest color hc and inside the doorway. Such a process p_{hc} is guaranteed to collect all forks and goes to eat, because p_{hc} has a higher priority than any of its neighbors. Second, let us consider processes with color $hc-1$ and inside the doorway. Given such a process p_{hc-1}, after processes with color hc exit the doorway, p_{hc-1} will prevent its neighbors with color hc from entering the doorway. Hence, p_{hc-1} has a higher priority than any of its neighbor inside the doorway, and p_{hc-1} eventually eats and exits the doorway. Continuing this reasoning, every process inside the doorway eventually eats. The key issue is that processes inside the doorway attempt to prevent their neighbors from entering the doorway. This prevents a lower-priority process from being overtaken infinitely many times by its higher-priority neighbors.

Hungry processes can make progress to enter the doorway in Phase 1. By Phase 2, every process inside the doorway eventually exits the doorway. Thus, every hungry process eventually gets permissions from all of its neighbors and is guaranteed to enter the doorway eventually. Because hungry processes can make progress in both Phase 1 and Phase 2, every hungry process is guaranteed to eat.

The asynchronous doorway algorithm is not wait-free. For example, if a process p crashes while eating, then none of its neighbors can get the fork shared with p, and hence, its hungry neighbors starve.

The asynchronous doorway algorithm also does not satisfy bounded waiting. Such an example is shown in Figure 8. While a hungry process q is waiting a permission from its neighbor p to enter the doorway, another neighbor r can always get permissions from q to enter the doorway, and then go to eats. This can occur finitely many times, since q will eventually enter the doorway. However, there does not exist an upper bound on how many times r can overtake q while q is hungry and outside the doorway.

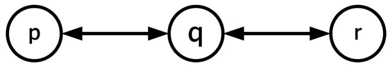


Fig. 8. An Illustrative Example of Unbounded-Waiting Properties

7.2. Wait-Free, Eventual 2-Bounded Waiting Dining Algorithm

To solve WF-EBF dining, we revise the original doorway algorithm as follows. First, to achieve wait-freedom, we adopt suspicion from $\diamond\mathcal{P}_1$ as a proxy to replace both missing forks and missing permissions from crashed neighbors. This change results in eventual weak exclusion, because in finite prefixes of any run, two neighbors can enter the doorway simultaneously and go to eat simultaneously both by wrongfully suspecting each other. Second, to achieve bounded waiting, we modify the doorway so that while a hungry process p is hungry, each neighbor q can receive at most two permissions from p. However, during finite prefixes of any run, q can wrongfully suspect p, and hence q can enter the doorway without the permission from p. As such, *eventual* 2-bounded waiting is achieved. A ping-ack protocol is designed to implement this modified asynchronous doorway. We describe this algorithm based on the two phases that a hungry process must go through to eat. The algorithmic pseudocode is presented as an action system of guarded commands as shown in Algorithm 2.

Phase 1. When a process becomes hungry, it tries to enter the doorway. In order to do so, we implement the asynchronous doorway by *a ping-ack protocol*. A hungry process p sends *ping* messages to its neighbor q to request permissions to enter the doorway. The neighbor q grants the permission to p by replying an *ack* (i.e., acknowledgment) message.

This ping-ack protocol alone cannot guarantee wait-freedom. All crashed processes stop sending messages, including ack messages. Consequently, if a process crashes, its hungry neighbors are potentially blocked outside the doorway and starve.

To achieve wait-freedom, we use suspicion from $\diamond\mathcal{P}_1$ in place of missing acks from crashed neighbors. A hungry process p enters the doorway if and only if for each neighbor q, p suspects q or received an ack from q during the current hungry section of p. The local strong completeness property of $\diamond\mathcal{P}_1$ guarantees that every crashed process will be eventually and permanently suspected by all correct neighbors. Therefore, crashed processes cannot block their hungry neighbors from entering the doorway.

To achieve eventual k-bounded waiting, we require that while a process p is hungry, it can grant *at most one* ack per neighbor q. We also require that while a ping request is still pending (i.e., sent a ping, but has not received the corresponding ack back), a process cannot send another ping message to the corresponding neighbor. This mechanism ensures that during each hungry section of p, its neighbor q can receive at most two ack messages. One ack might be sent while p was thinking, and another ack might be sent while p is hungry. Hence, while a process is hungry, its neighbor can enter the doorway by permissions at most twice.

```
$N(p)$ denotes the set of neighbors of process $p$                                  Code for process $p$

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

3 : $\{(\mathsf{state}_p = \mathsf{hungry}) \wedge \neg\mathsf{inside}_p\} \longrightarrow$      Action 2
4 :     $\forall q \in N(p)$ where $(\neg\mathsf{pinged}_{pq} \wedge \neg\mathsf{ack}_{pq})$ do      Request Acks
5 :             $\mathsf{send\text{-}ping}\langle p\rangle$ to $q$;  $\mathsf{pinged}_{pq} := \mathsf{true}$;

6 : $\{\mathsf{receive\text{-}ping}$ from $q \in N(p)\} \longrightarrow$             Action 3
7 :     if $(\mathsf{inside}_p \vee \mathsf{replied}_{pq})$                         Receive a Ping
8 :             $\mathsf{deferred}_{pq} := \mathsf{true}$;                          Defer Sending Ack
9 :     else
10 :            $\mathsf{send\text{-}ack}\langle p\rangle$ to $q$;  $\mathsf{replied}_{pq} := (\mathsf{state}_p = \mathsf{hungry})$;      Send an Ack

11 : $\{\mathsf{receive\text{-}ack}$ from $q \in N(p)\} \longrightarrow$             Action 4
12 :    $\mathsf{ack}_{pq} := ((\mathsf{state}_p = \mathsf{hungry}) \wedge \neg\mathsf{inside}_p)$ ;      Receive an Ack
13 :    $\mathsf{pinged}_{pq} := \mathsf{false}$;

14 : $\{(\mathsf{state}_p = \mathsf{hungry}) \wedge (\forall q \in N(p) :: (\mathsf{ack}_{pq} \vee (q \in \Diamond\mathcal{P}_1)))\} \longrightarrow$      Action 5
15 :    $\mathsf{inside}_p := \mathsf{true}$;                                        Enter the Doorway
16 :    $\forall q \in N(p)$ do
17 :            $\mathsf{ack}_{pq} := \mathsf{false}$;  $\mathsf{replied}_{pq} := \mathsf{false}$;

18 : $\{(\mathsf{state}_p = \mathsf{hungry}) \wedge \mathsf{inside}_p\} \longrightarrow$      Action 6
19 :    $\forall q \in N(p)$ where $(\mathsf{token}_{pq} \wedge \neg\mathsf{fork}_{pq})$ do      Request Missing forks
20 :            $\mathsf{send\text{-}request}\langle\mathsf{color}_p\rangle$ to $q$;  $\mathsf{token}_{pq} := \mathsf{false}$;

21 : $\{\mathsf{receive\text{-}request}\langle\mathsf{color}_q\rangle$ from $q \in N(p)\} \longrightarrow$      Action 7
22 :    $\mathsf{token}_{pq} := \mathsf{true}$;                                       Receive a Fork Request
23 :    if $(\neg\mathsf{inside}_p \vee ((\mathsf{state}_p = \mathsf{hungry}) \wedge (\mathsf{color}_p < \mathsf{color}_q)))$
24 :            $\mathsf{send\text{-}fork}\langle p\rangle$ to $q$;  $\mathsf{fork}_{pq} := \mathsf{false}$;

25 : $\{\mathsf{receive\text{-}fork}\langle q\rangle$ from $q \in N(p)\} \longrightarrow$      Action 8
26 :    $\mathsf{fork}_{pq} := \mathsf{true}$;                                        Receive a Fork

27 : $\{((\mathsf{state}_p = \mathsf{hungry}) \wedge \mathsf{inside}_p \wedge (\forall q \in N(p) :: (\mathsf{fork}_{pq} \vee (q \in \Diamond\mathcal{P}_1))))\} \rightarrow$      Action 9
28 :    $\mathsf{state}_p := \mathsf{eating}$;                                        Enter Critical Section

29 : $\{\mathsf{state}_p = \mathsf{eating}\} \longrightarrow$                        Action 10
30 :    $\mathsf{state}_p := \mathsf{exiting}$;                                       Exit Eating

31 : $\{\mathsf{state}_p = \mathsf{exiting}\} \longrightarrow$                       Action 11
32 :    $\mathsf{inside}_p := \mathsf{false}$;  $\mathsf{state}_p := \mathsf{thinking}$;      Exit the Doorway
33 :    $\forall q \in N(p)$ where $(\mathsf{token}_{pq} \wedge \mathsf{fork}_{pq})$ do
34 :            $\mathsf{send\text{-}fork}\langle p\rangle$ to $q$;  $\mathsf{fork}_{pq} := \mathsf{false}$;      Send Deferred Forks
35 :    $\forall q \in N(p)$ where $(\mathsf{deferred}_{pq})$ do
36 :            $\mathsf{send\text{-}ack}\langle p\rangle$ to $q$;  $\mathsf{deferred}_{pq} := \mathsf{false}$;      Send Deferred Acks
```

Algorithm 2 A Wait-Free, Eventually Bounded-Fair Dining Algorithm

Finally, the revised ping-ack protocol works as follows. After becoming hungry, a process p sends a ping message to each neighbor q if there is no pending ping request from p to q. Upon q receiving the ping, q sends the ack if (1) q is thinking, or (2) q is hungry, outside the doorway, and has *not* already sent an ack to p during the current hungry session of q. Otherwise, q defers sending the ack until after q eats and exits the doorway. A hungry process p enters the doorway if and only if for each neighbor q, p suspects q or received an ack from q during the current hungry section of p.

Note that it is possible for two neighbors to enter the doorway simultaneously at any time. If two neighbors suspect each other (before $\diamond\mathcal{P}_1$ converges), then both can enter the doorway regardless of the ack messages from each other. Alternatively, neighbors can receive acks from each other simultaneously while outside the doorway, and then enter the doorway together. The symmetry between hungry neighbors inside the doorway is resolved by the color-based priority scheme in Phase 2.

Phase 2. To achieve wait-freedom, we use suspicion from $\diamond\mathcal{P}_1$ in place of missing forks shared with crashed neighbors. A hungry process p goes to eat if and only if for each neighbor q, p holds the fork shared with q or suspects q.

Phase 2 adopts the following fork-collection scheme. After entering the doorway, a hungry process p requests each missing fork by sending the token to the corresponding neighbor q. Upon receiving this request, neighbor q grants the shared fork if (1) q is outside the doorway, or (2) q is hungry, inside the doorway, but has a priority lower than p (where process priorities are also represented by the static node colors). Otherwise, q defers the fork request until after q exits the doorway.

Figure 9 shows the diner state transitions and the corresponding actions in Algorithm 2, where the gray area indicates that the diner is inside the doorway. Notice that while a process is hungry, it may or may not be inside the doorway.

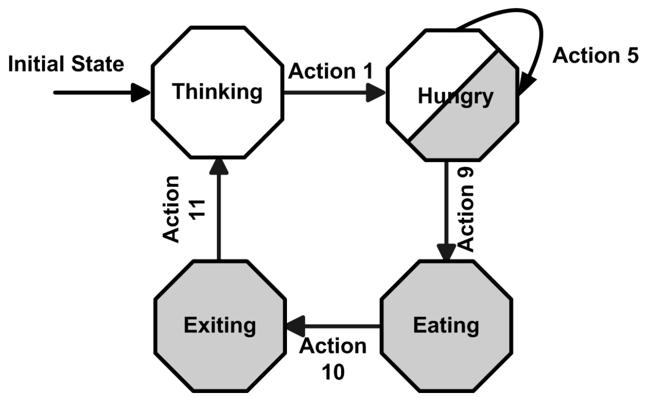


Fig. 9. Diner State Transitions in Algorithm 2

7.2.1. Algorithm Variables

In addition to the failure detector module, each process has nine types of local variables: *color*, *inside*, *state*, *pinged*, *ack*, *deferred*, *replied*, *fork*, and *token*. These local variables can be partitioned into three sets according to their usage: state variables, ping-ack variables, and fork-collection variables.

State variables are used to represent the current status of a process. Each process p has three state variables: $color_p$, $inside_p$, $state_p$.

The integer-valued variable $color_p$ represents the color of process p. Upon initialization, we assume that each *color* variable is assigned a locally unique value so that no two neighbors have the same color. Recall that color assignment can be done by node-coloring algorithms in polynomial time using only $O(\delta)$ distinct values. Color values denote process priorities and are static after initialization. For each pair of neighbors p and q, process p has a higher priority than q if and only if $color_p > color_q$.

Each process p also has two variables to describe its current status: a quadrivalent variable $state_p$ and a Boolean variable $inside_p$. Variable $state_p$ denotes the current dining phase; it is either *thinking, hungry, eating*, or *exiting*. Variable $inside_p$ indicates whether process p is currently inside the doorway. Initially, every process is outside the doorway and thinking.

Ping-ack variables are used to implement the ping-ack protocol. Each process p has four Boolean variables associated each neighbor q: pinged_{pq}, ack_{pq}, $\mathit{deferred}_{pq}$, and $\mathit{replied}_{pq}$. Initially, all of these variables are false.

The variable pinged_{pq} is true if and only if there is a pending ping request from p to q. Such a *pending ping request* covers the following three situations: a ping request is on its way from p to q, or is being deferred by q, or a replied ack is on its way to p.

The *ack* variables are used to remember acks received during the current hungry section. If a hungry process p receives an ack from its neighbor q while p is outside the doorway, the variable ack_{pq} is set to true to remember this received ack. Upon entering the doorway, processes do not need to remember received acks, and hence *ack* variables are reset to false. Therefore, variable ack_{pq} is true if and only if the hungry process p is outside the doorway and has received an ack from q during the current hungry session of p.

The variable $\mathit{deferred}_{pq}$ is used to represent whether process p is deferring a ping request from q. Variable $\mathit{deferred}_{pq}$ is true if and only if p is currently deferring a ping request from q, and it remains true until p exits the doorway.

To achieve eventual 2-bounded waiting, process p can only send one ack to each neighbor q while p is hungry. To do so, the variable $\mathit{replied}_{pq}$ is used by p to record ack messages sent to q. Variable $\mathit{replied}_{pq}$ is true if and only if process p has sent an ack to neighbor q during the current hungry session of p. Upon entering the doorway, processes do not need to remember acks sent, then *replied* variables are reset to false.

Fork-collection variables are used to implement the fork-collection scheme. Each process p has two local Boolean variables associated with each neighbor q: fork_{pq} and token_{pq}. Symmetrically, q has variables fork_{qp} and token_{qp} associated with p. The *fork* and *token* variables are implemented and interpreted the same as in the hygienic dining algorithm (Chapter V). Between each pair of neighbors, forks and tokens are unique and exclusive. Initially, the fork is held by the higher-color neighbor, and the token is held by the lower-color neighbor.

7.2.2. Algorithm Actions

There are eleven actions in Algorithm 2. These actions can be partitioned into three sets: ping-ack actions, fork-collection actions, and other actions.

Action 1 states that a thinking process can become hungry at any time or remain thinking forever. This action is not an internal action of Algorithm 2 and is formalized only for completeness of process behaviors. Upon becoming hungry, processes are still outside the doorway.

Ping-ack actions include Action 2, 3, 4, and 5. They are used to implement the ping-ack protocol.

Action 2 sends ping messages. This action is always enabled while a process p is hungry and outside the doorway. For each neighbor q, if the ack from q is missing, and no ping request to q is pending, then process p sends a ping request to q. As a result, $pinged_{pq}$ becomes true to indicate the existence of the pending ping request.

Action 3 decides whether a process p grants a ping request. The ping request can be deferred for two reasons: (1) p is inside the doorway, or (2) p is outside the doorway but has sent an ack during its current hungry session. Otherwise, p sends an ack immediately. As a result, if p is hungry and outside the doorway, the corresponding *replied* variable is set to true to remember the outgoing ack.

Action 4 simply receives an ack from a neighbor q. As a result, the variable $pinged_{pq}$ variable is set back to false to indicate that no ping request is pending to q. The variable ack_{pq} needs to be set to true only when process p is hungry and outside the doorway.

Action 5 determines when a hungry process enters the doorway. If, for each neighbor q, a hungry process p either received the ack or suspects q continuously, p eventually enters the doorway. After p enters the doorway, since p does not need to remember received acks, all of its *ack* variables are reset to false. Also, because p always defers ping requests while inside the doorway, its *replied* variables are reset to false as well.

Fork-collection actions include Action 6, 7, and 8. They are used to implement the fork-collection scheme.

Action 6 requests missing forks. This action is always enabled while hungry processes are inside the doorway. In this action, processes send tokens to request all missing forks. Process colors are encoded in request messages as a parameter.

Action 7 decides whether a process p should grant a fork request when p receives the fork request. If p is outside the doorway, or hungry but has a color lower than the requesting neighbor, then p sends the shared fork immediately. Otherwise, process p defers the fork request until after p exits the doorway.

Action 8 simply receives a fork.

Other actions are actions that cannot be classified into the above two sets.

Action 9 determines when a hungry process goes to eat. If a hungry process p is inside the doorway, and for each neighbor q, p either holds the shared fork continuously or suspects q, then p eventually eats.

Action 10 states that an eating process eventually transits to exiting. Correct processes can eat only for a finite period of time. An eating process eventually finishes eating and transits to exiting by executing Action 10. This action is also not an internal action of Algorithm 2 and is formalized only for completeness of process behaviors. Upon finishing eating, processes are still inside the doorway.

By executing Action 11, exiting processes transit back to thinking and exit the doorway. All deferred fork requests and deferred ping requests are granted.

7.3. Correctness Proof

The section proves that Algorithm 2 satisfies eventual weak exclusion (i.e., $\diamond\mathcal{WX}$), wait-freedom, and eventual 2-bounded waiting.

7.3.1. Safety

The mechanism to guarantee $\diamond\mathcal{WX}$ is the same in both Algorithm 1 and Algorithm 2. Recall that Algorithm 1 (in Chapter V) is also wait-free under $\diamond\mathcal{WX}$. In both algorithms, a hungry process p goes to eat if and only if for each neighbor q, p either holds the shared fork or suspects q (Action 9 in Algorithm 2 and Action 5 in Algorithm 1). Although Algorithm 2 requires that a hungry process must be inside the doorway to eat, this cannot prevent the same safety proof from being applied to both algorithms. The safety proof in Algorithm 1 relies on only two assumptions: $\diamond\mathcal{P}_1$ can wrongfully suspect its neighbors finitely many times (i.e., local eventual strong accuracy of $\diamond\mathcal{P}_1$), and the fork between every pair of neighbors is unique and exclusive. As such, we will focus on establishing the second assumption next. The previous $\diamond\mathcal{WX}$ proof in Theorem 1 can be directly applied to Algorithm 2.

Lemma 8 *In Algorithm 2, when a process receives a fork request, the process must hold the requested fork.*

Suppose that a process p is requested for a fork that it does not hold. If p is outside the doorway or hungry but has a lower color, p would duplicate the fork (Action 7). As a result, uniqueness of forks and the safety property are violated. This lemma shows that the above situation never happens based on FIFO channels.

Proof: This lemma is proved by direct construction in two steps. The first step shows that when a process receives a fork, that process must not be holding the corresponding token. The second step concludes the lemma based on the first step.

The first step is shown as follows. A process can send forks only in Action 7 and 11, in which the process must hold the corresponding token. After sending the fork, the process may send the token to re-request the fork (Action 6). Because of FIFO channels, the token must arrive to the recipient process after the fork arrives. Consequently, when a process p receives a fork, or a fork is in transit to p, p must not hold the corresponding token; the token must be either at the corresponding neighbor or in transit to p.

The second step is shown as follows. Based on the first step, we can further conclude that when a process p sends a token to its neighbor q, the fork cannot be in transit to p. By contradiction, suppose that the fork is in transit to p. By the conclusion in the first step, the corresponding token must be in transit to p or at q. This contradicts that p sends the token to q. Hence, when p sends a token to its neighbor q, the fork must be either at q or is intransit to q. Because of FIFO channels, the token must arrive at q after the fork arrives. Also, q cannot relinquish the fork without receiving the token. Thus, Lemma 8 holds; when a process receives a token, the process must hold the corresponding fork. $\square$

Lemma 9 *The fork is unique between each pair of neighbors.*

If two duplicated forks exist between two neighbors, both neighbors can eat simultaneously infinitely often. Therefore, uniqueness of forks is necessary for $\diamond\mathcal{WX}$.

Proof: Only when processes send a fork which they do not hold, could the fork be duplicated. However, a process p sends a fork, only because p received a fork request (Actions 7 and 10). By Lemma 8, when a process receives a fork request, the process must hold the fork. Therefore, processes cannot send a fork that they do not hold, and hence, forks cannot be duplicated. Lemma 9 holds. □

Theorem 4 *Algorithm 2 satisfies eventual weak exclusion: for each execution, there exists a time after which no two live neighbors eat simultaneously.*

The proof is the same as in Theorem 1. □

7.3.2. Progress

Theorem 5 *Algorithm 2 satisfies wait-freedom: every correct hungry process eventually eats.*

Proof: In order to eat, every hungry process must go through two phases: Phase 1 (outside the doorway) and Phase 2 (inside the doorway). Correspondingly, our progress proof consists of two parts. The first part shows progress in Phase 2: every correct hungry process inside the doorway eventually eats. The second part shows progress in Phase 1: every correct process outside the doorway eventually enters the doorway. We first prove progress in Phase 2 because progress in Phase 1 relies on progress in Phase 2. Progress in Phase 2 is proved in Lemma 10, 11, and 12. Progress in Phase 1 is proved in Lemma 13. □

Lemma 10 *Let processes p and q be correct neighbors, where p is hungry and inside the doorway, and $color_p > color_q$. If p does not suspect q, p eventually holds the fork shared with q continuously until after p eats.*

Proof: If process p has not sent a fork request to q, and the fork is missing, p will request the fork shared with q (Action 6). Upon receiving the fork request, because $color_p > color_q$, q defers the fork request only when it is eating or exiting (Action 7). Since q is correct, q eats only for a finite period of time. Thus, q eventually exits eating and sends all deferred forks, including the fork shared with p (Action 11). Thus, p eventually holds the shared fork. Next we show that p will hold the fork continuously until after p eats.

Because $color_p > color_q$, while p is inside the doorway, p defers any fork request from q (Action 7). Consequently, p will hold the fork continuously until after p eats. Thus, Lemma 10 holds. □

Lemma 11 *Between each pair of neighbors p and q, there exists at most one pending ping request initiated by process p at any time.*

Proof: While there is a pending ping request from p to q, the variable $pinged_{pq}$ remains true until after p receives an ack from q. While $pinged_{pq}$ remains true, process p cannot send another ping message to q (Action 2). Lemma 11 holds. □

Lemma 12 *(Progress in Phase 2): Every correct hungry process inside the doorway eventually eats.*

Proof: This lemma is proved by complete induction on the ordering of process colors. The base case shows that every correct hungry process inside the doorway with the highest color *hc* eventually eats. The inductive step assumes that every correct hungry process inside the doorway with a color higher than a color d eventually eats, and proves that every correct hungry process inside the doorway with color d eventually eats.

We start our proof after $\diamond\mathcal{P}_1$ converges. Processes inside the doorway may or may not eat before $\diamond\mathcal{P}_1$ converges, but every correct hungry process inside the doorway is *guaranteed* to eat eventually after $\diamond\mathcal{P}_1$ converges.

Base Case: process $color = hc$.

Let p be a correct hungry process inside the doorway, and $color_p = hc$. Because no two neighboring processes have the same color, $color_p$ is higher than the color of all neighbors. We partition all neighbors of p into two sets: *correct* and *faulty*.

All faulty neighbors eventually crash. By local strong completeness of $\diamond\mathcal{P}_1$, p eventually and permanently suspects all faulty neighbors.

By contrast, p cannot suspect correct neighbors after $\diamond\mathcal{P}_1$ converges. Since $color_p$ is higher than the color of any neighbor, by Lemma 10, p will hold the forks shared with its correct neighbors continuously until after p eats. Thus, eventually for each neighbor q, p either suspects q permanently or holds the fork shared with q continuously. As such, Action 9 is enabled continuously at p, and p eventually eats.

Inductive Step: process $color = d$.

Assume that every correct hungry process inside the doorway with a $color > d$ eventually eats. The inductive step will prove that every correct hungry process inside the doorway with color d eventually eats.

Consider a correct hungry process p inside the doorway with color d. We partition all neighbors of p into three sets: faulty neighbors, Low_p (correct neighbors with a color lower than d) and $High_p$ (correct neighbors with a color higher than d). Because no two neighbors have the same color, every correct neighbor belongs to either Low_p or $High_p$.

By the same analysis in the base case, process p suspects all faulty neighbors eventually and permanently, and p also eventually holds all forks shared with neighbors in Low_p.

For each neighbor q in $High_p$, process p will eventually hold the fork shared with q. If p is not holding the fork and has not requested the fork from q, p sends a fork request to q (Action 6). Upon receiving the fork request, q defers it only when q is inside the doorway (either hungry, eating, or exiting). By the inductive hypothesis (recall $color_q > d$), if q is hungry, q eventually eats. Because q is correct, q eventually finishes eating and sends all deferred forks, including the fork shared with p.

While hungry and inside the doorway, process p may still lose forks to its neighbor in $High_p$. This can happen only when the $High_p$ neighbors are also hungry and inside the doorway. Hence, we need to show that p eventually holds the forks shared with $High_p$ neighbors *continuously* until after p eats. In particular, after $\diamond\mathcal{P}_1$ converges, p can lose the fork to each $High_p$ neighbor at most once before p eats.

After $\diamond\mathcal{P}_1$ converges, p cannot be suspected by any $High_p$ neighbor q. Thus, in order to enter the doorway, q needs to collect acks from all of its correct neighbors, including process p. It is possible that q receives an ack from p, which was sent before p entered the doorway. However, by Lemma 11, there exists at most one pending ping request from p to q at any time. Also, while p is inside the doorway, p defers any ping request. Thus, while p is inside the doorway, q could receive at most one ack from p, which was sent before p entered the doorway. Consequently, q can enter the doorway at most once while p is inside the doorway. Hence, p may lose the fork to q at most once before p goes to eat. By the inductive hypothesis, q eventually eats and exits the doorway. After that, q is blocked outside the doorway until after p exits the doorway. While q is blocked outside the doorway, process p cannot lose the fork to q. Thus, p holds the fork shared with q continuously until after p eats.

For each neighbor q, p either suspects q permanently or holds the fork continuously. Therefore, Action 9 is enabled continuously at p, and p eventually eats. The inductive step is proved.

By the base case and the inductive step, Lemma 12 holds. □

Lemma 13 *(Progress in Phase 1): Every correct hungry process outside the doorway eventually enters the doorway.*

This lemma is proved based on a set of processes $H(t)$, where t denotes a time. We say that a process p belongs to the set $H(t)$ if and only if at time t, p is correct, hungry, outside the doorway, and none of its correct hungry neighbors has been outside the doorway longer than p. We denote t_{ph} as the time when process p started its current hungry session. Hence, process p became hungry at time t_{ph} and remains hungry through time t. Because neighbors can become hungry simultaneously, set $H(t)$ may include neighboring processes.

Proof: We only need to prove that every process in $H(t)$ eventually enters the doorway. For a correct hungry process q that is not in the set $H(t)$, if q remains outside the doorway for a sufficiently long time, q eventually joins in a set $H(t')$ at a later time t' and then enters the doorway. To show that every process p in $H(t)$ eventually enters the doorway, we need to prove that for each neighbor q, p either suspects q permanently or eventually receives an ack from q.

We also start our proof after $\diamond\mathcal{P}_1$ converges. The progress in Phase 1 is guaranteed after $\diamond\mathcal{P}_1$ converges.

Every faulty neighbor eventually crashes. By strong completeness of $\diamond\mathcal{P}_1$, process p suspects all faulty neighbors eventually and permanently.

For each correct neighbor q, we will show that p eventually receives an ack from q. After process p becomes hungry, p starts to collect acks from all neighbors (Action 2). By Lemma 11, after executing Action 2, for each neighbor q, if the ack from q is missing, then there exists exactly one pending ping request initiated by p. Note that this pending ping request could be sent during the current hungry session, or perhaps a previous hungry session. Suppose that neighbor q receives the ping message at time t_{qr}. Neighbor q will grant the ping request immediately except for two reasons as shown next (Action 3, Line 7). However, we claim that q will send the deferred ack eventually.

(1) $\mathit{inside}_q = \mathit{true}$. This condition indicates that q is inside the doorway (either hungry or eating). If q is hungry, by Lemma 12, q eventually eats. Because correct processes can eat only for a finite period of time, q eventually exits the doorway and sends the deferred ack (Action 11).

(2) $\mathit{replied}_{qp} = \mathit{true}$. This condition indicates q must be hungry and outside the doorway at the time t_{qr}. Suppose that q sets its variable *replied*$_{qp}$ as true at a time t_{qs}. Consequently, time t_{qs} must be earlier than t_{qr} (i.e., $t_{qs} < t_{qr}$), *replied*$_{qp}$ remained true from t_{qs} through t_{qr}, and q must be hungry and outside the doorway from time t_{qs} through t_{qr}.

First, we claim that time t_{qs} must be earlier than t_{ph} (i.e., $t_{qs} < t_{ph}$). By contradiction, suppose that $t_{ph} \leq t_{qs}$, then $t_{ph} \leq t_{qs} < t_{qr}$. That means that while p remains hungry from time t_{ph} to time t_{qr}, q received two ping messages at time t_{qs} and t_{qr} from p, respectively. When a process sets its *replied* variable to true in Action 3 (line 10), an ack must be sent to the corresponding neighbor. In this case, q must send an ack to p at time t_{qs} because *replied*$_{qp}$ was set to true at time t_{qs}. Consequently, p should receive this ack message after time t_{ph}. Because p only needs one ack to enter the doorway, p never sends the second ping which is received by q at time t_{qr}. This forms a contradiction with the assumption that q receives a ping from p at time t_{qr}. Hence, time t_{qs} must be earlier than t_{ph}.

Next, we claim that process q cannot remain hungry and outside the doorway from time t_{qs} to t. By contradiction, suppose that q remains hungry and outside the doorway from time t_{qs} to t. By the above conclusion, time t_{qs} must be earlier than t_{ph}. Hence, q would be hungry and stay outside the doorway longer than p at time t. That contradicts to our assumption that p is in $H(t)$. Therefore, q must enter the doorway once between time t_{qs} and time t. By Lemma 12, q will eventually eat and thereafter send the deferred ack to p.

Thus, for each neighbor q, p either suspects q permanently or eventually receives an ack from q. Hence, Action 5 is enabled continuously, and process p enters doorway eventually. Lemma 13 holds. □

7.3.3. Eventual 2-Bounded Waiting

Theorem 6 *Algorithm 2 satisfies eventual 2-bounded waiting: for each execution, there exists a time after which no live process p goes to eat more than twice while any live neighbor is hungry.*

Proof: This theorem is proved by direct construction. We first conclude the convergence time for each execution, and then prove that the 2-bounded waiting property holds after the convergence time.

Suppose that $\diamond\mathcal{P}_1$ converges at time t_1. Assume that there exists a set of correct hungry processes at time t_1, denoted as $Hungry(t_1)$. By Theorem 5, every correct hungry process eventually eats. Therefore, there exists a time t_2 after which all correct processes in $Hungry(t_1)$ eat, where $t_1 \leq t_2$. Thus, after time t_2, no hungry session of correct processes starts before $\diamond\mathcal{P}_1$ converges.

Faulty processes eventually crash. Thus, there exists a time t_3 after which every live process must be correct. Let time t_c be $max(t_2, t_3)$. We claim time t_c is the convergence time after which no live process p goes to eat more than twice while any live neighbor q is hungry.

After t_c, since $\diamond\mathcal{P}_1$ has already converged, no correct process wrongfully suspects any correct neighbor. Thus, to enter the doorway, process p must receive an ack from each correct neighbor. If process p goes to eat more than twice, then its live neighbor q must send at least three acks to p. While q is hungry and outside the doorway, after q sends the first ack message, $replied_{pq}$ is set to true. When q receives the second ping message, q will defer the ping message (Action 3). Thus, while q is hungry, q can send at most one ack message to p.

Although q can send at most one ack message to p while q is hungry, p still can receive two acks from q. It is possible that q sent an ack to p just before q became hungry. When q became hungry, the ack message was still in transit to p. Consequently, p could enter the doorway at most twice while q is hungry. Thus, after time t_c, p can go to eat at most twice while q is hungry. Theorem 3 holds. $\square$

7.4. Analysis

Algorithm 2 needs only *bounded* local memory at each process. Each process p has nine types of local variables. The variables $state_p$ and $inside_p$ need a fixed size of local memory, and variable $color_p$ needs $log_2(\delta)$ bits of local memory, where δ refers to the maximal degree of the conflict graph. Process p also has six Boolean variables associated with each neighbor q: $fork_{pq}$, $token_{pq}$, $pinged_{pq}$, ack_{pq}, $replied_{pq}$, and $deferred_{pq}$. Putting them together, each process needs $log_2(\delta) + 6\delta + c_1$ bits of memory, where c_1 is a constant value. In the worst case where $\delta = n$ (i.e, the conflict graph is clique), each process needs $O(n)$ bits of local memory.

Algorithm 2 requires only *bounded capacity* on communication channels. This property means that both the size of messages and the number of messages at any time are upper bounded. First, consider the size of messages. We need to encode process *id* into fork messages and process *color* into token messages. Both process *id* and *color* need at most $log_2(n)$ bits. So each message is upper bounded by $O(log_2(n))$ bits. Second, consider the number of messages at any time. Algorithm 2 has four types of messages: *ping, ack, fork*, and *token*. Because forks and tokens are unique, at most one fork and one token are in transit simultaneously between each pair of neighbors. By Lemma 11, at any time, at most one ping or ack message initiated by each process is in transit. Because both neighbors may initiate a ping/ack message, at most two ping/ack messages are in transit between each pair of neighbors at any time. Thus, at most four messages are in transit between each pair of neighbors at any time. The number of messages in transit is also upper bounded at any time.

Algorithm 2 is not thinking quiescent. Thinking quiescence guarantees that if a correct process remains thinking after some time, then this process eventually stops sending and receiving messages. Consider a correct thinking process p. In Algorithm 2, whenever any neighbor q becomes hungry, p needs to reply acks to permit q to enter the doorway. Hence, Algorithm 2 is not thinking quiescent. Actually, thinking quiescence is impossible for any WF-EBF dining algorithm using $\diamond\mathcal{P}$. The impossibility of thinking quiescence is a tradeoff to achieve bounded fairness for wait-free dining under $\diamond\mathcal{WX}$ using $\diamond\mathcal{P}$. We will prove this impossibility result in the next chapter.

CHAPTER VIII

IMPOSSIBILITY OF THINKING QUIESCENCE IN WAIT-FREE, EVENTUALLY BOUNDED-FAIR DINING USING $\diamond\mathcal{P}_1$

This chapter proves the impossibility of a potential performance improvement for WF-EBF dining: $\diamond\mathcal{P}_1$ cannot deterministically solve *thinking-quiescent* WF-EBF dining. A *thinking-quiescent* dining algorithm guarantees that if a correct process remains thinking forever from some time, then this process eventually stops sending and receiving messages [32]. [1] The impossibility of thinking quiescence reveals an intrinsic performance penalty on achieving bounded fairness. That is, to achieve bounded fairness, if a correct process eats infinitely often, then its correct neighbors must send and receive infinitely many messages, regardless of the dining state of the neighbors.

We prove the impossibility of thinking quiescence in Theorem 7 by using the same method as that in Theorem 3. Recall that Theorem 3 proves that $\diamond\mathcal{P}$ cannot deterministically solve wait-free, perpetually bounded-fair dining. Theorem 7 is also proved by contradiction formed by two indistinguishable executions, and also uses the fact that in asynchronous systems augmented with $\diamond\mathcal{P}_1$, application layers can still be asynchronous.

Theorem 7 *For asynchronous message-passing systems augmented with $\diamond\mathcal{P}_1$, there does not exist a deterministic thinking-quiescent algorithm that solves wait-free, even-*

[1] A similar concept, *non-cooperation*, is studied for mutual exclusion in shared-memory systems [80]. An algorithm is *non-cooperative* if thinking processes do not need to write into shared objects to communicate with other processes [52, 55, 57].

tually bounded-fair dining.

Proof: This theorem is proved by contradiction, which is formed by two indistinguishable executions. By contradiction, assume that there exists a deterministic thinking-quiescent algorithm $\mathcal{A}$ that solves wait-free, eventual k-bounded waiting dining under $\diamond\mathcal{WX}$ using $\diamond\mathcal{P}_1$. The fairness bound k is a natural number. Consider a system of two *correct* processes p and q. We construct a finite reference execution γ_r, in which p remains thinking forever, and q eats $k+1$ times. Next, we force another finite alternate execution γ_a, in which p becomes hungry at least once, and eventual k-bounded waiting is violated. The two executions are shown in Figure 10.

The reference execution γ_r is constructed as follows. Consider any execution prefix resulting in a configuration C_1 at time t_1, in which $\diamond\mathcal{P}_1$ and eventual k-bounded waiting have already converged, and q are thinking. Also, assume that process p remains thinking forever after time t_1. Because the algorithm $\mathcal{A}$ is thinking quiescent, then there exists a time t_2 after which p stops communicating with q. The configuration at time t_2 is denoted as C_2. Consider an infinite schedule σ applicable to C_2. We also require that $exec(C_2, \sigma)$ is admissible such that process q takes infinitely many steps and cannot think permanently. Since q does not think forever, if q is thinking, then q eventually becomes hungry. Because the algorithm $\mathcal{A}$ is wait-free, q eventually eats. Because correct processes can eat only for a finite period of time, q eventually exits eating. As such, q becomes hungry infinitely often and also goes to eat infinitely often. Let time t_3 be the time when q finishes its $k+1^{st}$ eating since time t_2. The finite execution γ_r ends at time t_3. Let β be the finite execution segment from time t_2 to t_3, and let σ_β be the finite schedule corresponding to β. As such, the schedule σ_β only consists of actions of process q.

Now consider the alternate execution γ_a, in which p does not think forever. γ_a

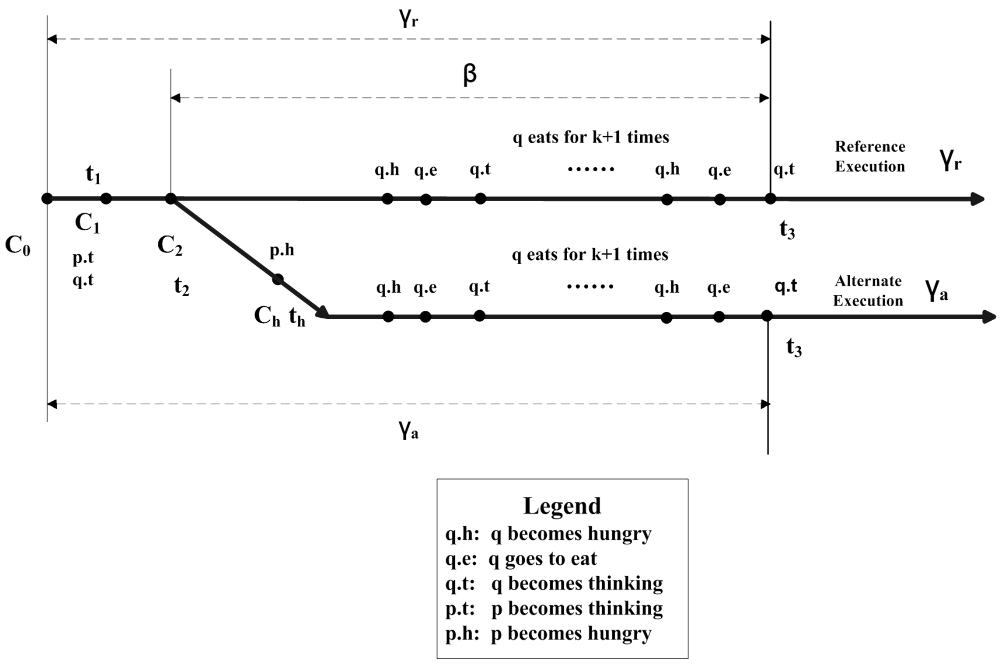


Fig. 10. Impossibility of Thinking-Quiescent Algorithms For Wait-Free, Eventually Bounded-Fair Dining

is constructed by extending from the configuration C_2. At the next step immediately after time t_2, process p becomes hungry at time t_h. Let C_h be the configuration at t_h. After that, the finite schedule σ_β is applied to the configuration C_h. We need to show that σ_β is applicable to C_h.

First, process p is permitted not to take actions in the finite execution γ_a. Because p is correct and hungry at time t_h, p must go to eat at a later time. Recall that in asynchronous systems augmented with $\diamond\mathcal{P}_1$, application layers may still be asynchronous (Section 2.3). Hence, p is permitted to be very slow in application layers. Additionally, any (application) message sent to process q after time t_2 can be subject to arbitrary delays as well. In particular, all such messages can be delayed until after time t_3. Hence, p is permitted to stay hungry and not to take steps until time t_3.

Second, we need to show that process q is still allowed to execute actions of the schedule σ_β in γ_a. Because the fault patterns are the same in both executions γ_r and γ_a, clearly $\diamond\mathcal{P}_1$ can provide the same information to process q in γ_a and γ_r. Meanwhile, any message sent by process p after time t_2 can be delayed until after time t_3. Therefore, the finite executions γ_a and γ_r are indistinguishable from the perspective of process q. Because the algorithm $\mathcal{A}$ is deterministic, process q can still go to eat $k+1$ times in execution γ_a.

Based on the above analysis, we conclude that the schedule σ_β is applicable to configuration C_h. At this point, we know that after time t_2 in γ_a, process q goes to eat $k+1$ times while process p is hungry. Recall that $\diamond k\text{-}\mathcal{BW}$ has converged before t_2. Therefore, eventual k-bounded waiting is violated in γ_a. This contradicts the initial assumption that the algorithm $\mathcal{A}$ satisfies eventual k-bounded waiting. Thus, Theorem 7 holds. $\square$

Theorem 7 does not utilize the fact that $\diamond\mathcal{P}_1$ can make finitely many false-positive mistakes by wrongfully suspecting correct neighbors. Instead, the proof is established after $\diamond\mathcal{P}_1$ converges. That suggests that stronger failure detectors, such as the perfect failure detector $\mathcal{P}$, still cannot solve thinking-quiescent WF-EBF dining. Hence, using failure detectors, the performance penalty is unavoidable to achieve bounded fairness for wait-free dining; correct processes might need to send and receive infinitely many messages.

Chapter VII already shows that $\diamond\mathcal{P}_1$ is sufficient to solve WF-EBF dining. This chapter suggests a limitation of $\diamond\mathcal{P}_1$ on solving WF-EBF dining. Hence, it is of interest to explore whether $\diamond\mathcal{P}_1$ is also necessary for solving WF-EBF dining. The next chapter answers this question positively by showing that $\diamond\mathcal{P}_1$ is also necessary.

CHAPTER IX

THE WEAKEST FAILURE DETECTOR FOR SOLVING WAIT-FREE, EVENTUALLY BOUNDED-FAIR DINING PHILOSOPHERS

This chapter demonstrates that $\diamond\mathcal{P}_1$ is necessary for solving wait-free, eventually bounded-fair dining. Chapter VII has already shown that $\diamond\mathcal{P}_1$ is sufficient to solve WF-EBF dining. Therefore, $\diamond\mathcal{P}_1$ is the weakest (and optimal) failure detector for solving WF-EBF dining.

To prove that $\diamond\mathcal{P}_1$ is necessary, we construct a reduction algorithm that can use any WF-EBF dining solution to implement $\diamond\mathcal{P}_1$ [81]. This reduction algorithm converts the properties of WF-EBF dining algorithms to an eventually reliable timeout-based mechanism to detect crash faults. In particular, wait-freedom is used to establish local strong completeness, and both $\diamond k$-$\mathcal{BW}$ and $\diamond\mathcal{WX}$ are used to establish local eventual strong accuracy. Presumably, we cannot utilize any explicit timing assumption. Therefore, we assume that there does not exist any bound on timing properties. We also assume that the fairness bound k is *unknown.* Because $\diamond\mathcal{P}_1$ is defined with respect to communication graphs, *neighbors* in this chapter stand for *communication neighbors*, instead of *conflicting neighbors.*

This chapter will first design the timeout-based mechanism by stepwise refinement. Thereafter, we describe the reduction algorithm in detail. Finally, we prove that the implemented failure detector satisfies both local strong completeness and local eventual strong accuracy.

9.1. Basic Idea

This section designs the timeout-based mechanism by stepwise refinement based on an example, in which a correct process p monitors the status of another process q (i.e., whether q has cashed). We will start from a simple timeout-based mechanism that guarantees local strong completeness only. Thereafter, we will refine this timeout-based mechanism step by step until the fault detection is eventually reliable.

Step 1. Let us consider a simple timeout-based mechanism that guarantees local strong completeness only. This mechanism constructs a dining instance $\mathcal{M}'$ that consists of p and q only, and requires that correct processes cannot remain thinking permanently. Process p has a counter and a timeout duration associated with q. Every time q becomes hungry, q sends a *renew* message to p. When p receives the message, p resets the counter to 0. Every time p eats, p increases the counter by 1. When the counter exceeds the duration, p suspects q.

This simple mechanism guarantees local strong completeness simply by wait-freedom. Assume that p is correct and q crashes at some time. As a result, p will eventually receive no *renew* messages from q, and hence the counter will eventually be monotonically non-decreasing. Because correct processes cannot remain thinking forever, if p is thinking, then p eventually becomes hungry. Because of wait-freedom, p eventually eats. Hence, p will eat infinitely often, and the counter eventually exceeds the timeout duration. As such, p suspects q.

Unfortunately, this simple mechanism cannot guarantee local eventual strong accuracy. Assume that both p and q are correct. There must exist an infinite number of execution segments during which q remains thinking. While q is thinking, p can eat finitely but unboundedly many times. As such, p can eat unboundedly many times during each renew interval (i.e., the execution segments between the times that

p receives two consecutive renew messages). Hence, there is no upper bound on the counter, and the counter might exceed the timeout duration infinitely many times. As a result, p may suspect q infinitely many times.

Step 2. To establish local eventual strong accuracy, we will revise the timeout-based mechanism so that the $\diamond k\text{-}\mathcal{BW}$ and $\diamond\mathcal{WX}$ properties can be utilized to establish an eventual upper bound on the counter. That is, there exists a time after which p can eat boundedly many times during each renew interval. In order to do so, we first reconstruct the dining instance as shown in Figure 11.

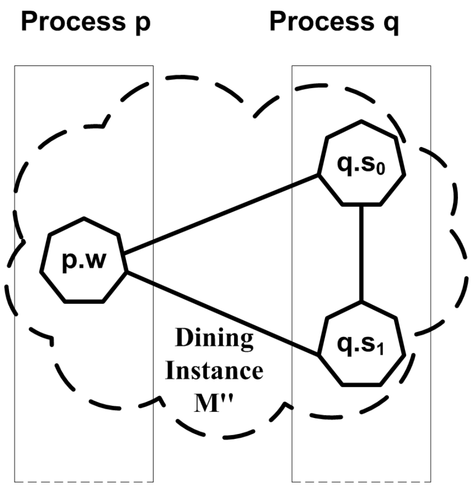


Fig. 11. Reconstructed Dining Instance

The instance $\mathcal{M}''$ comprises three threads: a witness thread $p.w$ at process p and two subject threads, $q.s_0$ and $q.s_1$, at process q. Each thread is modeled as a set of actions which are executed by the underlying physical process. For example, $p.w$ is executed by p, and $q.s_0$ and $q.s_1$ are executed by q. Hence, their failure semantics are correlated. Whenever a process crashes, all threads at this process crash as well. We say a thread is correct, faulty, or live, if and only if the underlying process is correct, faulty, or live, respectively.

This reconstructed instance $\mathcal{M}''$ is used to guarantee that for each execution, there *do not* exist an infinite number of execution segments during which both $q.s_0$ and $q.s_1$ are thinking. The subject threads $q.s_0$ and $q.s_1$ behave like q in the instance $\mathcal{M}'$ such that subjects send a *renew* message each time they become hungry. The witness thread $p.w$ also maintains a counter and a timeout duration associated with q, and $p.w$ behaves like p in $\mathcal{M}'$. Also, no correct thread can remain thinking permanently. The key part is that a subject thread can exit eating only when another subject thread is hungry or eating. As such, each execution has an infinite suffix during which at least one subject thread is in its hungry-eating session at any time (i.e., hungry or eating at any time). A hungry-eating session is an execution segment that starts when a diner becomes hungry and lasts until the diner finishes eating.

This reconstruction alone still cannot guarantee that the counter is eventually upper bounded. Since we assume no bounds on any timing property, *renew* messages may be arbitrarily slow. As a result, each renew interval may encompass finitely but unboundedly many hungry-eating sessions of each subject thread. Therefore, $\diamond k$-$\mathcal{BW}$ still cannot be used to establish an upper bound on the counter.

Step 3. Finally, we introduce a *renew-ack* protocol that can utilize $\diamond k$-$\mathcal{BW}$ and $\diamond\mathcal{WX}$ to establish an eventual upper bound on the counter. This protocol coordinates the behaviors of subject threads so that each renew interval is encompassed by one hungry-eating session of some subject thread $q.s_i$, where $i \in \{0, 1\}$ (For simplicity, we use $q.s_i$ to represent "$q.s_0$ or $q.s_1$", where $i \in \{0, 1\}$.). As such, during each renew interval, the subject $q.s_i$ remains continuously hungry and then continuously eating. After both $\diamond k$-$\mathcal{BW}$ and $\diamond\mathcal{WX}$ converge, $p.w$ can eat at most k times during each renew interval. Therefore, the counter is eventually upper bounded.

The renew-ack protocol works as follows. As shown in Figure 12, every time the witness $p.w$ receives a renew message from a subject $q.s_i$ (where $i \in \{0, 1\}$), $p.w$

replies an ack message. Since $q.s_i$ receives this ack message, $q.s_i$ is *being watched* by $p.w$ until $q.s_i$ finishes eating. The underlying dining algorithm alone decides when a hungry subject goes to eat. However, a subject can exit its eating session only when both $q.s_0$ and $q.s_1$ are *being watched* by $p.w$. As such, a subject exits eating only when the ongoing (or current) renew interval has already been covered by the current hungry-eating session of the sibling subject. Therefore, each renew interval is encompassed by one hungry-eating session of some subject thread. By $\diamond k$-$\mathcal{BW}$ and $\diamond\mathcal{WX}$, the counter is eventually upper bounded by an integer number related to the fairness bound k.

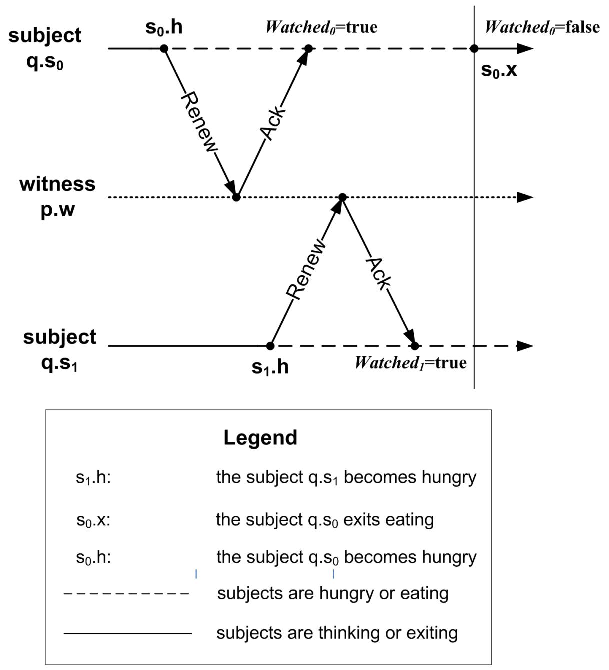

subject q.s0
s0.h
Watched0=true
Watched0=false
s0.x
Renew
Ack
witness p.w
Renew
Ack
subject q.s1
s1.h
Watched1=true
Legend
s1.h: the subject q.s1 becomes hungry
s0.x: the subject q.s0 exits eating
s0.h: the subject q.s0 becomes hungry
subjects are hungry or eating
subjects are thinking or exiting

Fig. 12. The Renew-Ack Protocol

Because the fairness bound k is unknown, we cannot set the initial timeout duration above the eventual bound on the counter. As such, we assign any positive integer to the initial timeout duration. If p wrongfully suspects q, p eventually receives a *renew* message from q and knows it made a mistake. As a response to this false-positive mistake, p increases the timeout duration. Eventually, the counter cannot exceed the duration. Consequently, local eventual strong accuracy can be established.

9.2. Algorithm Description

9.2.1. Structure

Each pair of neighbors p and q is associated with two symmetric WF-EBF dining instances as shown in Figure 13. The instance $\mathcal{M}_{pq}$ is used for p to monitor the status of q (i.e., whether q has crashed), and the instance $\mathcal{M}_{qp}$ is used for q to monitor the status of p. Therefore, each process is associated with at most $2 \times \delta$ dining instances, where δ is the maximum degree of the conflict graph. Note that in the symmetric instance $\mathcal{M}_{qp}$, the witness versus subject roles of p and q would be reversed, as in $\mathcal{M}_{pq}$. Since the two instances are symmetric, we restrict our presentation only to the instance $\mathcal{M}_{pq}$ in which process p monitors process q. Actions for subject threads and witness threades are presented in Algorithm 3a and 3b, respectively.

Although logically distinct with respect to $\mathcal{M}_{pq}$, the subject threads $q.s_0$ and $q.s_1$ are implemented as a single stream of execution. More specifically, each subject thread is a distinct set of actions, the union of which is executed under interleaving semantics by process q.

We also require that no correct thread can remain thinking permanently. This requirement is critical for local strong completeness. If $p.w$ remains thinking permanently after some time t, then the counter will never increase after time t. As such, the counter may never exceed the timeout duration after time t. If q crashes after t, p will never suspect q.

The above requirement is also critical for local eventual strong accuracy. Assume

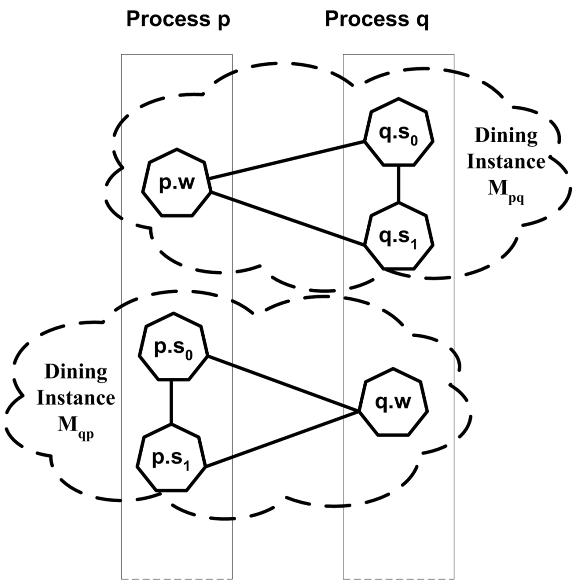


Fig. 13. Symmetric Failure Detection Between Processes p and q

that either $q.s_0$ or $q.s_1$ remains thinking permanently after some time. Consequently, there exist an infinite number of execution segments during which both $q.s_0$ and $q.s_1$ are thinking. As discussed previously, $p.w$ can eat finitely but unboundedly many times while both $q.s_0$ and $q.s_1$ are thinking. Hence, there is no eventual upper bound on the counter, and $p.w$ may wrongfully suspect q infinitely many times.

Fortunately, thinking diners are permitted to become hungry at any time. Recall that becoming hungry is an input action and is activated only by diners themselves. As such, our reduction algorithm is able to control correct threads such that they cannot remain thinking permanently, and the underlying dining solutions are still used as a black-box subroutine.

9.2.2. Timeout-Based Fault Detection

Process p maintains a timer ω_{pq} for each neighbor q. This timer is associated with the instance $\mathcal{M}_{pq}$. In particular, ω_{pq} uses *exiting events* of the witness (i.e., $p.w$

transits from eating to exiting) as an abstract time measurement. In other words, this timer measures logical time based on a special type of logical events, exiting events at $p.w$. We use *exiting events* to measure the logical time simply because exiting events are input actions. As such, we do not need to modify the underlying dining algorithm. Another state transition related to eating sessions, from hungry to eating, is an output actions. To utilize output actions, we need to revise the underlying dining algorithm. Thus, exiting events at $p.w$ are used to measure logical time.

The fault-detection scheme works as follows. To implement the timer ω_{pq}, process p has two integer-valued variables: $term_{pq}$ and $counter_{pq}$, where $term_{pq}$ represents the current timeout duration, and $counter_{pq}$ counts the number of exiting events that have occurred at $p.w$ since the last time that the timer was reset. Every time a subject $q.s_i$ becomes hungry, the subject $q.s_i$ sends a *renew* message to the witness $p.w$ to reset the timer ω_{pq}. Upon receiving this message, $p.w$ resets the variable $counter_{pq}$ to 0. Every time $p.w$ exits eating, $counter_{pq}$ gets incremented by one. If $counter_{pq}$ ever exceeds $term_{pq}$ (the timeout duration), then p times out on q, and $p.w$ starts to suspect q. If $p.w$ receives a renew message from $q.s$ at a later time, then $p.w$ knows that it wrongfully suspected q. Consequently, $p.w$ stops suspecting q and increases the timeout duration $term_{pq}$ to respond the false-positive mistake.

Timers are self-adaptive. The timeout duration $term_{pq}$ is actually an estimate of the eventual upper bound on the variable $counter_{pq}$. Every time p wrongfully suspects q (i.e., $counter_{pq} > term_{pq}$), the bound is reestimated by increasing the value of the variable $term_{pq}$.

9.2.3. Renew-Ack Protocol

The renew-ack protocol coordinates subject behaviors so that each renew interval is encompassed by one hungry-eating session of some subject thread. To implement this protocol, a local Boolean variable called *watched* is introduced to each subject. Initially, *watched* is false. The renew-ack protocol has been shown in Figure 12.

The protocol works as follows. When the witness $p.w$ receives a *renew* message sent by a hungry subject $q.s_i$, $p.w$ resets the variable $counter_{pq}$ to 0 as usual, but it also replies an *ack* message back to the subject $q.s_i$. This ack message serves as confirmation that the timer has been reseted. Upon receiving this ack message, the subject $q.s_i$ sets its $watched_i$ variable to true. The hungry subject $q.s_i$ may eventually eat, but $q.s_i$ cannot exit eating until the *watched* variables of both subjects are true.[1] Upon exiting, the subject $q.s_i$ sets its own $watched_i$ variable back to false, thereby disabling the exit guard for the sibling subject. This simple synchronization scheme guarantees that no eating subject can exit its critical section until *both* subject threads have set their *watched* variables to true.

The renew-ack protocol establishes local eventual strong accuracy by implementing an elastic clock that eventually synchronizes to regulate the relative progress of the witness and subject threads. For each run, consider an infinite suffix during which (1) $\diamond k\text{-}\mathcal{BW}$ has already converged, so that the fairness bound k holds, (2) the timeout duration *term* is greater than the eventual upper bound on *counter*, and (3) $\diamond\mathcal{WX}$ has already converged so that no live threads eat simultaneously. During such a suffix, correct subjects always reset the timer before it times out, and hence, no correct process is suspected by its correct neighbors.

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

```
Actions of a subject thread $q.s_i$ at process $q$ in dining instance $\mathcal{M}_{pq}$
where $i \in \{0,1\}$ and $q.s_j$ denotes the sibling subject of $q.s_i$

1 : $\{\mathsf{state}_i = \mathsf{thinking}\} \longrightarrow$                          Action $\mathcal{S}_h$
2 :     $\mathsf{state}_i := \mathsf{hungry}$;                                        Becomes Hungry
3 :     send $\langle\mathsf{renew}\rangle$ to witness $p.w$;

4 : $\{$upon receiving $\langle\mathsf{ack}\rangle$ from witness $p.w\} \longrightarrow$   Action $\mathcal{S}_a$
5 :     $\mathsf{watched}_i := \mathsf{true}$;                                        Receives acks

6 : $\{(\mathsf{state}_i = \mathsf{eating}) \wedge \mathsf{watched}_i \wedge \mathsf{watched}_j\} \longrightarrow$   Action $\mathcal{S}_x$
7 :     $\mathsf{watched}_i := \mathsf{false}$;                                       Exits eating
8 :     $\mathsf{state}_i := \mathsf{exiting}$;
```

Algorithm 3a The Reduction Algorithm at Subjects

```
Actions of the witness thread $p.w$ at process $p$ in dining instance $\mathcal{M}_{pq}$

1 : $\{\mathsf{state}_w = \mathsf{thinking}\} \longrightarrow$                          Action $\mathcal{W}_h$
2 :     $\mathsf{state}_w := \mathsf{hungry}$;                                        Becomes Hungry

3 : $\{\mathsf{state}_w = \mathsf{eating}\} \longrightarrow$                            Action $\mathcal{W}_x$
4 :     if $(q \notin \mathsf{suspect}(p))$                                           Exit Eating
5 :             $\mathsf{counter}_{pq} := \mathsf{counter}_{pq} + 1$;                  Increments Counter by 1
6 :     $\mathsf{state}_w := \mathsf{exiting}$;

7 : $\{$upon receiving $\langle\mathsf{renew}\rangle$ from subject $q.s_i\} \longrightarrow$   Action $\mathcal{W}_r$
8 :     $\mathsf{counter}_{pq} := 0$;                                                 Resets the timer
9 :     $\mathsf{suspect}(p) := \mathsf{suspect}(p) - \{q\}$;                         removes $q$ from $\mathsf{suspect}(p)$
10 :    send $\langle\mathsf{ack}\rangle$ to subject $q.s_i$;

11 : $\{\mathsf{counter}_{pq} > \mathsf{term}_{pq}\} \longrightarrow$                  Action $\mathcal{W}_s$
12 :    $\mathsf{suspect}_p := \mathsf{suspect}_p \cup \{q\}$;                         Timer times out
13 :    $\mathsf{term}_{pq} := \mathsf{counter}_{pq}$;                                Increases the term
```

Algorithm 3b The Reduction Algorithm at Witnesses

9.2.4. Algorithm Variables

For each dining instance $\mathcal{M}_{pq}$, there are four variables for the witness thread $p.w$, and two variables for each subject thread $q.s_i$, where $i \in \{0, 1\}$.

Witness variables. The witness thread $p.w$ in $\mathcal{M}_{pq}$ has four local variables: $state_w$, $term_{pq}$, $counter_{pq}$, and $suspect_p$.

Variable $state_w$ is used to denote the current dining phase of witness $p.w$, which is either *thinking, hungry, eating* or *exiting.* Initially, $p.w$ is *thinking.*

The integer-valued variable $counter_{pq}$ measures the elapsed logical time since the last time the timer ω_{pq} was reset. Initially, $counter_{pq}$ is 0. Every time $p.w$ exits eating, the variable $counter_{pq}$ increases by 1. When $p.w$ receives a renew message, $counter_{pq}$ is reset to 0.

The integer-valued variable $term_{pq}$ represents the current timeout duration for timer ω_{pq}. Variable $term_{pq}$ is initially 1 and monotonically non-decreasing. Every time the timer ω_{pq} times out, $term_{pq}$ increases to the value of $counter_{pq}$.

Variable $suspect_p$ denotes the set of processes that are currently suspected by process p. This variable is a set variable and shared by all witnesses at process p. Initially, $suspect_p$ can be any subset of $\Pi - \{p\}$.

Subject variables. Each subject thread $q.s_i$ in $\mathcal{M}_{qp}$ has two local variables: $state_i$ and $watched_i$, where $i \in \{0, 1\}$.

Local variable $state_i$ denotes the current dining state of subject $q.s_i$, which is either *thinking, hungry, eating* or *exiting.* Initially, subject $q.s_i$ is thinking.

Each subject $q.s_i$ has a Boolean local variable $watched_i$, which is used to implement the *renew-ack* protocol. Initially set to false, variable $watched_i$ is set to true only when the subject $q.s_i$ receives an *ack* message. This ack message serves as a confirmation that witness $p.w$ has received the last renew message sent by $q.s_i$ and has already reseted the timer ω_{pq}. Thereafter, variable $watched_i$ remains true until $q.s_i$ exits eating. A subject can exit eating only when both $watched_0$ and $watched_1$ variables are true. Thus, hungry-eating sessions of two subjects always overlap each other, and every renew interval is encompassed by some hungry-eating session of some subject.

The true value of a *watched* variable indicates that the corresponding subject has already reseted the timer ω_{pq} during its current hungry-eating session. While subject $q.s_i$ is thinking or exiting, variable $watched_i$ must be false; but not vice versa. While variable $watched_i$ is true, the subject $q.s_i$ must be either hungry or eating; but not vice versa. Subjects can have both *false-valued watched* variables only during finite prefixes of an execution. Once a *watched* variable becomes true, at least one *watched* variable is true thereafter at any time.

9.2.5. Algorithm Actions

Algorithm 3a has three actions for subject threads.

Action $\mathcal{S}_h$ states that a correct thinking subject eventually becomes hungry. As such, no correct subject remains thinking permanently. Immediately after becoming hungry, the subject sends a *renew* message to $p.w$.

Action $\mathcal{S}_a$ simply receives an *ack* message. As a result, the subject $q.s_i$ sets its $watched_i$ variable to true. This means that $q.s_i$ is *being watched* by the witness $p.w$.

Action $\mathcal{S}_x$ coordinates the hungry-eating sessions of subjects $q.s_0$ and $q.s_1$. Action $\mathcal{S}_x$ is enabled at an eating subject $q.s_i$ only when both *watched* variables are true. In Action $\mathcal{S}_x$, subject $q.s_i$ sets its own $watched_i$ variable to false before $q.s_i$ exits eating. Therefore, after $q.s_i$ exits eating, this exiting action is disabled at the sibling subject $q.s_j$, and $q.s_j$ must stay in its current hungry-eating session until the variable $watched_i$ becomes true again.

Algorithm 3b has four actions for witness threads.

Action $\mathcal{W}_h$ simply states that a correct thinking witness eventually becomes hungry. As such, no correct witness remains thinking permanently.

Action $\mathcal{W}_x$ indicates that a correct eating witness eventually exits eating. This action is actually the exiting event that is used as an abstract time measurement for timers. Every time witness $p.w$ exits eating, if p does not suspect q, then the variable $counter_{pq}$ increases by 1.

Action $\mathcal{W}_r$ is activated when witness $p.w$ receives a *renew* message. Upon receiving this message, $p.w$ resets the timer ω_{pq} by resetting the variable $counter_{pq}$ to 0, and $p.w$ sends an *ack* message to the subject who sent the *renew* message. Also, q is removed from the suspect list $suspect_p$.

Action $\mathcal{W}_s$ is enabled when the timer ω_{pq} times out. As a result, p suspects q by adding q into the suspect list $suspect_p$. The timeout duration $term_{pq}$ increases to the current value of $counter_{pq}$.

9.3. Correctness Proof

This section formally proves that the implemented failure detection satisfies both local strong completeness and local eventual strong accuracy.

9.3.1. Local Strong Completeness

Theorem 8 *Every crashed process is eventually and permanently suspected by all correct neighbors.*

Without loss of generality, we consider the timer ω_{pq} and the associated instance $\mathcal{M}_{pq}$, in which process p monitors process q. We need to show that for every execution α in which p is correct and q is faulty, p suspects q eventually and permanently.

Theorem 8 is proved by direct construction. Process q is faulty and eventually crashes. Therefore, there exists a time t after which p never receives a renew message from q. Consequently, after time t, Action $\mathcal{W}_r$ is disabled at witness $p.w$, and the timer ω_{pq} is never reseted. As a result, ω_{pq} eventually times out, and p eventually and permanently suspects q.

Proof. Since process q is faulty, it eventually crashes. After q crashes, both subject $q.s_0$ and $q.s_1$ never send renew messages. Consequently, process q sends a finite number of renew messages to p, and p can receive only a finite number of renew messages from q. Thus, there exists a time t after which process p never receives a renew message from q.

After time t, because witness $p.w$ cannot receive *renew* messages, variable $counter_{pq}$ will never be reset to 0 (i.e., Action $\mathcal{W}_r$ is permanently disabled at $p.w$). Therefore, variable $counter_{pq}$ never decreases after time t. If p suspects q after time t, then p will suspect process q permanently. Next, we will prove that if p does not suspect q, then p will eventually suspect q.

By Action $\mathcal{W}_h$, correct thinking witnesses eventually become hungry. Wait-freedom guarantees that correct hungry witnesses eventually eat. Correct eating witnesses eventually exit eating by executing Action $\mathcal{W}_x$. Thus, witness $p.w$ becomes hungry infinitely often and exits eating infinitely often. Every time $p.w$ exits eating, if p is not suspecting q, then $counter_{pq}$ increases by 1. Consequently, $counter_{pq}$ eventually exceeds $term_{pq}$, timer ω_{pq} times out, and p suspects q (Action $\mathcal{W}_s$). Thus, if p does not suspect q, p will eventually suspect q.

If p does not suspect q after time t, p will eventually suspect q. If p suspects q after time t, p will suspect q permanently. Thus, Theorem 8 holds. □

9.3.2. Local Eventual Strong Accuracy

We also consider the timer ω_{pq} and the instance $\mathcal{M}_{pq}$. To prove local eventual strong accuracy, we need to show that for every execution α in which both p and q are correct, there exists a time after which p does not suspect q.

The proof is organized as follows. We will first introduce some notations that will be used. Next, we will prove that every renew interval is encompassed by some hungry-eating session of some subject (in Lemma 14). Finally, we prove local eventual strong accuracy using Lemma 14.

An *event* is an execution of some action. We denote the n^{th} execution of Action $\mathcal{A}$ as the event $\mathcal{A}^n$. For example, $\mathcal{W}_r^n$ denotes the n^{th} execution of Action $\mathcal{W}_r$ by witness $p.w$, and $\mathcal{S}_{x,i}^n$ denotes the n^{th} execution of Action $\mathcal{S}_x$ by subject $q.s_i$, where $i \in \{0, 1\}$. Also, to simplify the presentation, we say that a timer-resetting event $\mathcal{W}_r^n$ *is activated by* a subject if and only if this event is enabled by receiving a *renew* message from the subject.

For an execution α, $\alpha[\phi_1, \phi_2]$ denotes the execution segment that starts from event ϕ_1 and ends at event ϕ_2 (including ϕ_1 and ϕ_2). Therefore, the n^{th} renew interval in execution α can be formally represented as $\alpha[\mathcal{W}_r^n, \mathcal{W}_r^{n+1}]$. Let time t_n be the n^{th} time that the timer ω_{pq} is reseted (i.e., $p.w$ executes the event $\mathcal{W}_r^n$). Clearly, $t_n < t_{n+1}$.

Lemma 14 *Every renew interval $\alpha[\mathcal{W}_r^n, \mathcal{W}_r^{n+1}]$ is encompassed by some hungry-eating session of the subject that does not activate the timer-resetting event $\mathcal{W}_r^{n+1}$.*

Proof. This lemma is proved by induction. Base case proves that the lemma holds for the first renew interval. The inductive step assumes that the lemma holds for the $n-1^{st}$ interval, and then proves that the lemma holds for the n^{th} renew interval.

Base Case: the first renew interval $\alpha[\mathcal{W}_r^1, \mathcal{W}_r^2]$.

The base case is shown in Figure 14. Without loss of generality, assume that event $\mathcal{W}_r^1$ is activated by subject $q.s_0$. We claim that the first interval $\alpha[\mathcal{W}_r^1, \mathcal{W}_r^2]$ is encompassed by the first hungry-eating session of the subject $q.s_0$. Also, the subject $q.s_0$ does not activate the event $\mathcal{W}_r^2$.

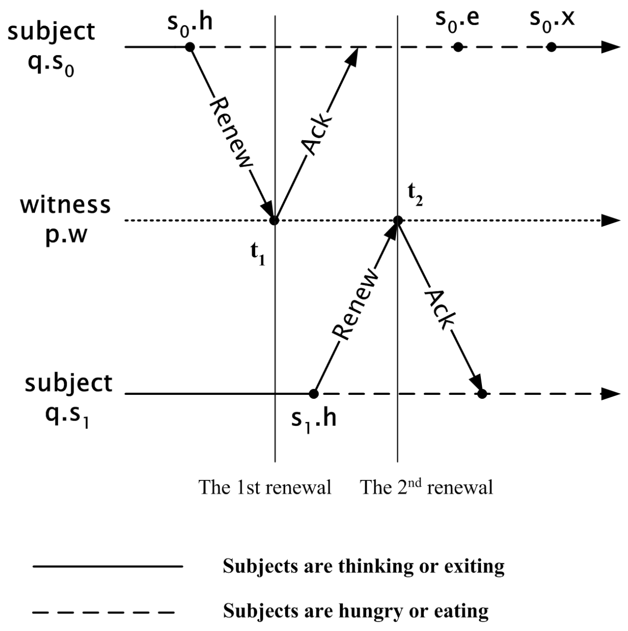


Fig. 14. Base Case in Lemma 14 Proof

When an event $\mathcal{W}_r^n$ occurs at time t_n, the subject $q.s_i$ that activates this event must be hungry or eating. Event $\mathcal{W}_r^n$ is enabled by receiving a renew message sent by the subject $q.s_i$. This renew message can be sent only when the subject $q.s_i$ is hungry (Action $\mathcal{S}_h$). Also, $q.s_i$ cannot exit eating until it receives the *ack* message sent at the event $\mathcal{W}_r^n$. Hence, $q.s_i$ must be hungry or eating at time t_n.

Since subject $q.s_0$ enables the event $\mathcal{W}_r^1$, $q.s_0$ must be hungry or eating at time t_1. Next, we will show that $q.s_0$ stays in its hungry-eating session through time t_2.

By Action $\mathcal{S}_x$, subject $q.s_0$ cannot exit eating until its sibling subject $q.s_1$ sets the variable $watched_1$ to true. The variable $watched_1$ is initially false and can be set to true only after $q.s_1$ sends a renew message and receives an *ack* message from $p.w$.

Thus, the event $\mathcal{W}_r^2$ must be activated by $q.s_1$, and subject $q.s_0$ cannot exit its eating session up through time t_2.

Since subject $q.s_0$ cannot exit its eating session up through time t_2, the first renew interval is encompassed by the first hungry-eating session of $q.s_0$, and $q.s_0$ does not activate $\mathcal{W}_r^2$. Lemma 14 holds for the base case.

Inductive Step: the n^{th} interval $\alpha[\mathcal{W}_r^n, \mathcal{W}_r^{n+1}]$.

This inductive step is shown in Figure 15. Without loss of generality, assume that subject $q.s_0$ activates the event $\mathcal{W}_r^{n+1}$. The inductive step assumes that the lemma holds for the n-1^{st} interval, and proves that the lemma also holds for the n^{th} interval $\alpha[\mathcal{W}_r^n, \mathcal{W}_r^{n+1}]$. In particular, the subject $q.s_1$ stays in its hungry-eating session from time t_n to t_{n+1}. This is proved by two steps. First, $q.s_1$ is hungry or eating at time t_n. Second, $q.s_1$ remains in its hungry-eating session from time t_n up through t_{n+1}.

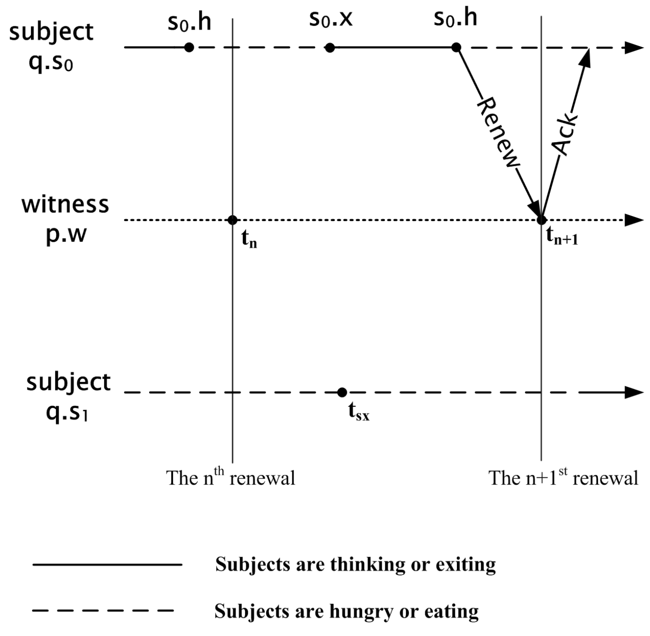


Fig. 15. Inductive Step in Lemma 14 Proof

Assume that $q.s_i$ activates the event $\mathcal{W}_r^n$, where i could be 0 or 1. By the analysis in the base case, $q.s_i$ must be hungry or eating at time t_n. Consider the sibling subject who does not activate the event $\mathcal{W}_r^n$. By inductive hypothesis, the n-1st renew interval is encompassed by a hungry-eating session of the sibling subject. Hence, this sibling subject must also be hungry or eating at time t_n. Therefore, both subjects $q.s_0$ and $q.s_1$ are hungry or eating at time t_n.

We claim that subject $q.s_1$ cannot exit eating from time t_n up through t_{n+1}. By contradiction, assume that $q.s_1$ exits exiting once at a time t_{sx}, where $t_n < t_{sx} \leq t_{n+1}$. Based on Action $\mathcal{S}_x$, variable $watched_0$ must be true at time t_{sx}, and $watched_1$ becomes false after time t_{sx}. Variable $watched_1$ can become true again only after subject $q.s_1$ resets the timer later after time t_{sx}. Because the next timer-resetting event $\mathcal{W}_r^{n+1}$ is not enabled by $q.s_1$, variable $watched_1$ stays false from time t_{sx} through time t_{n+1}.

Since variable $watched_0$ is true at time t_{sx}, subject $q.s_0$ must already send a renew message to $p.w$ during its current hungry-eating session (the hungry-eating session including time t_{sx}), and received an ack back. Hence, there is no renew or ack message in transit at time t_{sx} between subject $q.s_0$ and witness $p.w$. Meanwhile, since $watched_1$ is false during time period $[t_{sx}, t_{n+1}]$, Action $\mathcal{S}_x$ is disabled at subject $q.s_0$. Thereafter, subject $q.s_0$ stays in its hungry-eating session during time period $[t_{sx}, t_{n+1}]$. As such, subject $q.s_0$ cannot send a renew message to activate the event $\mathcal{W}_r^{n+1}$ at time t_{n+1}. This contradicts the initial assumption that subject $q.s_0$ activates the event $\mathcal{W}_r^{n+1}$. Thus, subject $q.s_1$ must stay in its hungry-eating session from t_n to t_{n+1}. The inductive step is proved.

Based on the base case and the inductive step, Lemma 14 holds. □

Theorem 9 *For each execution, there exists a time after which no correct process is suspected by any correct neighbor.*

This theorem is proved by three steps. The first step shows that if p wrongfully suspects q, then p eventually stops suspecting q. The second step shows that for each execution α, there exists a time t_1 after which the variable $counter_{pq}$ is eventually upper bounded by $k+1$. The final step shows that for execution α, there exists a time t_2 after which the variable $counter_{pq}$ never exceeds the variable $term_{pq}$. Therefore, after time t_2, the timer ω_{pq} never times out, and p never suspects q.

Proof. If p wrongfully suspects q during some finite prefix, p eventually stops suspecting q. Because q is correct, $q.s_0$ and $q.s_1$ cannot remain thinking permanently. Because of wait-freedom, correct hungry subjects eventually eat. Thus, subjects $q.s_0$ and $q.s_1$ become hungry infinitely often and send an infinite number of renew messages. Because communication channels are reliable, $p.w$ receives infinitely many renew messages. If p wrongfully suspects q, $p.w$ eventually receives a renew message, and p stops suspecting q.

There exists a time t_1 after which $counter_{pq}$ is upper bounded by $k+1$. After $\Diamond\mathcal{WX}$ and $\Diamond k$-$\mathcal{BW}$ converge in execution α, $p.w$ still receives infinitely many renew messages. Let $p.w$ receives the first renew message at time t_1 since $\Diamond\mathcal{WX}$ and $\Diamond k$-$\mathcal{BW}$ converge. As such, $counter_{pq}$ is reset to 0 at time t_1.

Because $\Diamond k$-$\mathcal{BW}$ has already converged before time t_1, witness $p.w$ goes to eat at most k times while any subject ($q.s_0$ or $q.s_1$) is hungry. Also, because $\Diamond\mathcal{WX}$ has already converged before time t_1, witness $p.w$ cannot eat while subjects are eating. Thus, after time t_1, witness $p.w$ goes to eat at most k times while any subject is in its hungry-eating session. Note that when a subject becomes hungry, the witness may already be in its eating session. Thus, after time t_1, while a subject remains in its hungry-eating session, witness $p.w$ can have at most $k+1$ (partial) eating sessions.

By Lemma 14, every renew interval is encompassed by some hungry-eating session of some subject. Thus, after time t_1, each interval contains at most $k+1$ (partial) eating sessions of witness $p.w$. Consequently, after time t_1, witness $p.w$ can execute the exiting action (Action $\mathcal{W}_x$) at most $k+1$ times in each interval. Therefore, after time t_1, variable $counter_{pq}$ is upper bounded by $k+1$.

Since time t_1 is finite, there must exist a maximum value k_m for $counter_{pq}$ before time t_1. Therefore, in execution α, there exists an upper bound $k' = max(k_m, k+1)$ for the variable $counter_{pq}$. Every time $counter_{pq}$ exceeds $term_{pq}$, $term_{pq}$ is increased to the value of $counter_{pq}$ (Action $\mathcal{W}_s$). Because $term_{pq}$ is monotonically non-decreasing, $term_{pq}$ eventually reaches to the bound k' at some time t_2, where time t_2 may or may not be earlier than t_1. After time t_2, timer ω_{pq} never times out, and p never adds q into the suspect list $suspect_p$.

Therefore, any wrongfully suspected process is eventually removed from the suspect list. After time t_2, p never adds q into the suspect list. Thus, the implemented failure detector satisfies local eventual strong accuracy, and Theorem 9 holds. $\square$

9.3.3. $\diamond\mathcal{P}_1$ Is Necessary for Solving Wait-Free, Eventually Bounded-Fair Dining

Although underlying WF-EBF dining algorithms require some timing assumptions, this reduction algorithm itself does not utilize any explicit timing assumption. This is a critical condition to prove the necessity of $\diamond\mathcal{P}_1$. If the reduction algorithm itself works only with some explicit timing assumptions that are sufficient to implement $\diamond\mathcal{P}_1$, the methodology proving the necessity of $\diamond\mathcal{P}_1$ does not work. Fortunately, our algorithm does not utilize any timing assumptions.

Theorem 10 *The local eventually perfect failure detector $\diamond\mathcal{P}_1$ is necessary for solving wait-free, eventually bounded-fair dining.*

Proof. The reduction algorithm can use any WF-EBF dining algorithm to implement a failure detector that satisfies both local strong completeness and local eventual strong accuracy. Also, this reduction algorithm does not utilize any explicit timing assumption. Thus, every failure detector that solves WF-EBF dining is stronger than $\diamond\mathcal{P}_1$. Hence, Theorem 10 holds. $\square$

9.4. $\diamond\mathcal{P}_1$ Is the Weakest Failure Detector for Solving Wait-Free, Eventually Bounded-Fair Dining

Chapter VII proves that $\diamond\mathcal{P}_1$ is sufficient for solving wait-free, eventually bounded-fair dining. This chapter proves that $\diamond\mathcal{P}_1$ is also necessary. Therefore, we can conclude the following theorem.

Theorem 11 *The local eventually perfect failure detector $\diamond\mathcal{P}_1$ is the weakest failure detector for solving wait-free, eventually bounded-fair dining.*

9.5. Analysis

This reduction algorithm needs *finite local memory* on each process. Every process p has six types of local variables, among which only variables *term* and *counter* are unbounded. Although variables *term* and *counter* are bounded by some value in each execution of each dining instance, there does not exist an upper bound for all executions of all dining instances. Thus, variables *term* and *counter* need finite but unbounded space.

This reduction algorithm needs only *bounded capacity* on communication channels. That is, both the size of messages and the number of messages in transit are upper bounded at any time. First, let us consider the size of messages. The process *id* information needs to be encoded into the message and requires $log_2(|\Pi|)$ bits. Also, the message type needs to be specified in all messages. The message type requires a constant number of bits. Thus, every message needs $log_2(|\Pi|) + c$ bits, where c is a constant. Second, let us consider the number of messages in transit between each pair of neighbors. Between a subject and a witness, at most one message can be in transit at any time. The message in transit is either a *renew* or an *ack* message. There are two dining instances for each pair of neighbors, and each instance includes two subjects and one witness. Therefore, at most four messages can be in transit at any given time between each pair of neighbors. Hence, the number of messages in transit is also bounded at any time.

CHAPTER X

SUMMARY OF CONTRIBUTIONS AND OPEN PROBLEMS

This chapter gives a retrospective review of the research contributions and elucidates some open problems for the future research.

10.1. Summary of Contributions

Our research is related to two major research lines in distributed computing: failure detectors and dining philosophers. Hence, our research contributes to these two research lines. In summary, the contributions include four parts: (1) the definition of the local failure detector $\Diamond\mathcal{P}_1$, (2) solvability of wait-free dining using $\Diamond\mathcal{P}_1$, (3) the weakest failure detector $\Diamond\mathcal{P}_1$ for solving wait-free, eventually bounded-fair dining, and (4) the formal definition of bounded-fairness properties with respect to dining.

10.1.1. Defining $\Diamond\mathcal{P}_1$

The local failure detector $\Diamond\mathcal{P}_1$ is defined intuitively in Chapter I and formally in Chapter III. Actually, $\Diamond\mathcal{P}_1$ is a local refinement of the eventually perfect failure detector $\Diamond\mathcal{P}$ on communication graphs. $\Diamond\mathcal{P}_1$ eventually and permanently suspects all faulty communication neighbors, and eventually stops suspecting correct communication neighbors.

The local failure detector $\Diamond\mathcal{P}_1$ is designed to solve the dining problem. Dining is essentially an abstraction of overlapping mutual exclusion problems and also called the local mutual exclusion problem in some paper [4]. As such, global information about crash faults is not necessary for solving wait-free dining. The local failure detection $\diamond\mathcal{P}_1$ is sufficient to solve the wait-free dining problems under $\diamond\mathcal{WX}$ as shown in Algorithm 1 and 2.

Another reason to use the local detector $\diamond\mathcal{P}_1$ is that $\diamond\mathcal{P}_1$ can be implemented in more systems than $\diamond\mathcal{P}$. Chapter III shows that $\diamond\mathcal{P}_1$ can be transformed into $\diamond\mathcal{P}$ by a gossip protocol in environments where communication graphs cannot be partitioned by crash faults. However, the cost to build a non-partitionable system is sometimes prohibitively high. In practice, many modern computer networks can be partitioned by crashes on few critical nodes (e.g., routers, switches). As such, $\diamond\mathcal{P}$ cannot be implemented in such systems. By contrast, $\diamond\mathcal{P}_1$ is defined only with respect to neighboring processes in communication graphs. Implementations of $\diamond\mathcal{P}_1$ impose fewer requirements on the underlying system. Also, dining does not require global fault detection. Consequently, $\diamond\mathcal{P}_1$ is introduced into this dissertation.

10.1.2. Solvability of Wait-Free Dining Using $\diamond\mathcal{P}_1$

Section 4.3 shows that $\diamond\mathcal{P}_1$ cannot solve wait-free dining under perpetual weak exclusion. To solve wait-free dining, we relaxed the dining safety constraint to eventual weak exclusion. Four chapters (Chapter V, VII, VI, and VIII) are devoted to demonstrate what $\diamond\mathcal{P}_1$ can and cannot solve in the context of wait-free dining. Clearly, one major contribution of this research is the solvability of wait-free dining using $\diamond\mathcal{P}_1$ under $\diamond\mathcal{WX}$.

Chapter V shows that $\diamond\mathcal{P}_1$ can solve wait-free dining under $\diamond\mathcal{WX}$. This dining algorithm (i.e., Algorithm 1) is based on the hygienic dining algorithm [60]. Essentially, we use suspicion of $\diamond\mathcal{P}_1$ as a proxy to replace missing forks shared with crashed neighbors. Safety (i.e., $\diamond\mathcal{WX}$) is guaranteed by uniqueness of forks and local eventual strong accuracy of $\diamond\mathcal{P}_1$, and progress (i.e., wait-freedom) is guaranteed by local strong completeness of $\diamond\mathcal{P}_1$. This algorithm is relatively straightforward. However, this algorithm is the preliminary step to solve wait-free, eventually bounded-fair dining, and provides valuable insight on designing and proving wait-free dining algorithms.

Chapter VI proves that $\Diamond\mathcal{P}_1$ cannot achieve *perpetual* k-bounded waiting with respect to wait-free dining under $\Diamond\mathcal{WX}$. This impossibility result is proved in the finite prefix during which $\Diamond\mathcal{P}_1$ can make finitely many false-positive mistakes, and utilizes the fact that in asynchronous systems augmented with $\Diamond\mathcal{P}_1$, some layers may still be asynchronous. Therefore, this impossibility result may be avoided using a stronger failure detector. This impossibility result also forces us to consider a weaker bounded-fairness property, *eventual* k-bounded waiting.

Chapter VII shows that $\Diamond\mathcal{P}_1$ can solve wait-free, eventually bounded-fair dining. In addition to wait-freedom and eventual weak exclusion, $\Diamond\mathcal{P}_1$ can be used to achieve an important fairness property, eventual k-bounded waiting. This dining algorithm (i.e., Algorithm 2) is based on the classic asynchronous doorway algorithm [61]. This algorithm uses not only suspicion of $\Diamond\mathcal{P}_1$ to replace missing forks and acks from crashed neighbors, but also uses a modified asynchronous doorway to guarantee the bounded-fairness property. Therefore, this algorithm proves that $\Diamond\mathcal{P}_1$ is sufficient for solving wait-free, eventually bounded-fair dining. This result is also the first step to prove the weakest failure detector for solving wait-free, eventually bounded-fair dining.

Chapter VIII proves that $\Diamond\mathcal{P}_1$ *cannot* deterministically solve thinking-quiescent WF-EBF dining. The proof actually does not use any feature of $\Diamond\mathcal{P}_1$, but uses the fact that in asynchronous systems augmented with $\Diamond\mathcal{P}_1$, some layers may still be asynchronous. Therefore, this impossibility result also applies to the perfect failure detector $\mathcal{P}$. This impossibility result shows an intrinsic performance penalty to achieve bounded fairness. That is, to achieve bounded fairness, if a correct process eats infinitely often, then its correct neighbors must send and receive infinitely many messages.

10.1.3. The Weakest Failure Detector for Solving Wait-Free, Eventually Bounded-Fair Dining

The failure detector $\diamond\mathcal{P}_1$ is the weakest failure detector for solving WF-EBF dining. This is the most important contribution in this dissertation. We actually establish a tight upper bound on synchronism needed to solve WF-EBF dining. That is, $\diamond\mathcal{P}_1$ encapsulates the minimal synchronism required to solve WF-EBF dining. Every message-passing system in which WF-EBF dining can be deterministically solved must also be sufficient to implement $\diamond\mathcal{P}_1$, and vice versa.

The weakest failure detector $\diamond\mathcal{P}_1$ is proved by two steps. First, Chapter VII constructs a doorway-based dining algorithm that uses $\diamond\mathcal{P}_1$ to solve WF-EBF dining. Hence, $\diamond\mathcal{P}_1$ is sufficient for solving WF-EBF dining. Second, Chapter IX constructs a reduction algorithm that can uses any WF-EBF solution as a black-box subroutine to implement $\diamond\mathcal{P}_1$. Hence, $\diamond\mathcal{P}_1$ is also necessary. Because $\diamond\mathcal{P}_1$ is both sufficient and necessary, $\diamond\mathcal{P}_1$ is the weakest one for solving WF-EBF dining.

10.1.4. Defining Bounded-Fairness Properties

Chapter VI formally defines two perpetual bounded-fairness properties ($\Box\mathcal{BW}$ and $\Box k$-$\mathcal{BW}$) and two eventual bounded-fairness properties ($\diamond\mathcal{BW}$ and $\diamond k$-$\mathcal{BW}$). A simple hierarchy is established for these fairness properties. A key feature of these definitions is that crash faults are considered. These formal definitions not only help us understand these fairness properties in faulty environments, but also establish a solid ground for the future research.

10.2. Open Problems and Future Work

Our research also invokes several open problems for the future study. We will outline some of these new problems and suggest the future directions of investigation.

10.2.1. Wait-Free, Self-Stabilizing Dining Algorithms

Transient faults are not considered in this dissertation. It is of interest to explore dining algorithms that can tolerate both transient faults and crash faults. Clearly, to tolerate crash faults, failure detectors are necessary; to tolerate transient faults, transient fault detection and correction are also necessary. The key for an algorithm to self-stabilize is that variable dependency cannot form a cycle. Therefore, an algorithm with a simpler variable dependency may be easier to self-stabilize. Stronger failure detectors may be needed to achieve a simple variable structure. Hence, stronger failure detectors may be needed to solve wait-free and self-stabilizing dining.

10.2.2. In Search of Weakest Failure Detectors

Our work proves that $\diamond\mathcal{P}_1$ is the weakest failure detector for solving wait-free, eventually bounded-fair dining. However, WF-EBF dining is only one variant of the dining philosophers problem. There are many other dining variants, such as wait-free dining under eventual weak exclusion and wait-free, perpetually bounded-fair dining. It is of interest to explore the weakest failure detectors for these variants.

Chapter V shows that $\diamond\mathcal{P}_1$ is sufficient to solve wait-free dining under eventual weak exclusion. However, $\diamond\mathcal{P}_1$ is also sufficient to solve a harder dining problem, wait-free, eventually bounded-fair dining. This seemingly implies that the weakest failure detector for wait-free dining under $\diamond\mathcal{WX}$ should be strictly weaker than $\diamond\mathcal{P}_1$. To weaken $\diamond\mathcal{P}_1$, we could weaken either local strong completeness or local eventual strong accuracy. However, if local strong completeness is weakened, then correct processes do not know which neighbor is really crashed. As such, correct processes may wait for crashed neighbors permanently, and wait-freedom may not be achieved. If local eventual strong accuracy is weakened, then correct processes do not know which neighbor is really correct. As such, $\Diamond\mathcal{WX}$ may not be achieved. Hence, our conjecture is that $\Diamond\mathcal{P}_1$ may be the weakest failure detector for wait-free dining under $\Diamond\mathcal{WX}$ as well.

Chapter VIII proves that $\Diamond\mathcal{P}_1$ cannot solve wait-free, perpetually bounded-fair dining. This impossibility proof is constructed on finite prefixes during which $\Diamond\mathcal{P}_1$ can make finitely many false-positive mistakes. Hence, this impossibility result might be avoided by using a stronger failure detector with perpetual accuracy properties. Using the perfect detector $\mathcal{P}$, Algorithm 2 can achieve perpetual weak exclusion, perpetual k-bounded waiting, and wait-freedom. However, $\mathcal{P}$ is too strong, and we do not know whether perpetual accuracy is necessary for solving this problem. The research on the weakest failure detector for this problem may focus on whether perpetual accuracy is necessary.

10.2.3. Hierarchy of Fairness Properties

Chapter IV constructed a simple hierarchy for bounded-fairness properties. This hierarchy, however, is not complete. For example, we do not know the relationship between eventual k_1-bounded waiting ($\Diamond k_1$-$\mathcal{BW}$) and perpetual k_2-bounded waiting ($\Box k_2$-$\mathcal{BW}$), where $k_1 < k_2$. Intuitively, $\Diamond k_1$-$\mathcal{BW}$ does not guarantee any fairness bound (either k_1 or k_2) during finite prefixes of executions. Hence, $\Diamond k_1$-$\mathcal{BW}$ is not stronger than $\Box k_2$-$\mathcal{BW}$. Intuitively, for $\Box k_2$-$\mathcal{BW}$, the fairness bound k_2 may be reached infinitely often in some executions. Hence, $\Box k_2$-$\mathcal{BW}$ cannot guarantee that each execution is eventually bounded by the fairness bound k_1, and is also not stronger than $\Diamond k_1$-$\mathcal{BW}$. Thus, our conjecture is that these two fairness properties are incomparable.

Other questions include whether eventual k-bounded waiting ($\Diamond k$-$\mathcal{BW}$) and perpetual k-bounded waiting ($\Box k$-$\mathcal{BW}$) are *strictly* stronger than eventual bounded waiting ($\Diamond\mathcal{BW}$) and perpetual bounded waiting ($\Box\mathcal{BW}$), respectively. $\Diamond k$-$\mathcal{BW}$ and $\Box k$-$\mathcal{BW}$ specify the (eventual) fairness bound k for all executions, while $\Diamond\mathcal{BW}$ and $\Box\mathcal{BW}$ only guarantee that each execution has an (eventual) fairness bound. Hence, our conjecture is that $\Diamond k$-$\mathcal{BW}$ and $\Box k$-$\mathcal{BW}$ are *strictly* stronger than $\Diamond\mathcal{BW}$ and $\Box\mathcal{BW}$, respectively.

REFERENCES

[1] E. W. Dijkstra, “Self-stabilizing systems in spite of distributed control,” *Communications of the ACM*, vol. 17, no. 11, pp. 643–644, November 1974.

[2] S. Dolev, *Self-Stabilization.* Cambridge, MA: MIT Press, 2000.

[3] A. Arora and M. G. Gouda, “Closure and convergence: A foundation of fault-tolerant computing,” *IEEE Transactions on Software Engineering (TSE)*, vol. 19, no. 11, pp. 1015–1027, November 1993.

[4] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette, “Self-stabilizing local mutual exclusion and daemon refinement,” *Chicago Journal of Theoretical Computer Science*, vol. 2002, no. 1, July 2002.

[5] F. Cristian, “Understanding fault-tolerant distributed systems,” *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, February 1991.

[6] G. Antonoiu and P. K. Srimani, “Mutual exclusion between neighboring nodes in a tree that stabilizes using read/write atomicity,” *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par)*, ser. Lecture Notes in Computer Science, vol. 1470, pp. 545–553, September 1998.

[7] S. Cantarell, A. K. Datta, and F. Petit, “Self-stabilizing atomicity refinement allowing neighborhood concurrency,” *Proceedings of the 6th International Symposium on Self-Stabilizing Systems (SSS)*, ser. Lecture Notes in Computer Science, vol. 2704, pp. 102–112, June 2003.

[8] M. G. Gouda and F. Haddix, “The alternator,” *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*, pp. 48–53, May 1999.

[9] M. Mizuno and M. Nesterenko, "A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments," *Information Processing Letters*, vol. 66, no. 6, pp. 285–290, June 1998.

[10] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 1, pp. 124–149, January 1991.

[11] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, June 1971, reprinted in *Operating Systems Techniques*, Academic Press, pp. 72-93, 1972. An earlier version appeared as EWD310.

[12] N. A. Lynch, "Fast allocation of nearby resources in a distributed system," *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 70–81, April 1980.

[13] J. L. Welch and N. A. Lynch, "A modular drinking philosophers algorithm," *Distributed Computing*, vol. 6, no. 4, pp. 233–244, July 1993.

[14] Y. Song and S. M. Pike, "Eventually k-bounded wait-free distributed daemons," *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 645–655, June 2007.

[15] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2004.

[16] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson, "Data requirements for implementation of n-process mutual exclusion using a single shared variable," *Journal of the ACM*, vol. 29, no. 1, pp. 183–205, January 1982.

[17] S. M. Pike, "Distributed resource allocation with scalable crash containment," Ph.D. dissertation, The Ohio State University, Department of Computer Science

& Engineering, August 2004.

[18] S. M. Pike, Y. Song, and S. Sastry, "Wait-free dining under eventual weak exclusion," *Proceedings of the 9th International Conference on Distributed Computing and Networking (ICDCN)*, pp. 135–146, January 2008.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[20] M. Choy and A. K. Singh, "Localizing failures in distributed synchronization," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 7, no. 7, pp. 705–716, July 1996.

[21] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM*, vol. 34, no. 1, pp. 77–97, January 1987.

[22] C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, April 1988.

[23] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, March 1996.

[24] J. Beauquier and S. Kekkonen-Moneta, "Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors," *International Journal of Systems Science*, vol. 28, no. 11, pp. 1177–1187, July 1997.

[25] M. Hutle and J. Widder, "On the possibility and the impossibility of message-driven self-stabilizing failure detection," *Proceedings of the 8th International*

Symposium on Self-Stabilizing Systems (SSS), ser. Lecture Notes in Computer Science, vol. 3764, pp. 153–170, October 2005.

[26] M. Choy and A. K. Singh, “Efficient fault-tolerant algorithms for distributed resource allocation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 3, pp. 535–559, May 1995.

[27] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, August 1975.

[28] F. Cristian, “A rigorous approach to fault-tolerant programming,” *IEEE Transactions on Software Engineering (TSE)*, vol. SE-11, no. 1, pp. 23–31, January 1985.

[29] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov, “Mutual exclusion in asynchronous systems with failure detectors,” *Journal of Parallel and Distributed Computing (JPDC)*, vol. 65, no. 4, pp. 492–505, April 2005.

[30] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, July 1996.

[31] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, “The ϕ accrual failure detector,” *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pp. 66–78. IEEE Computer Society, October 2004.

[32] M. K. Aguilera, W. Chen, and S. Toueg, “On quiescent reliable communication,” *SIAM Journal on Computing*, vol. 29, no. 6, pp. 2040–2073, 2000.

[33] R. Guerraoui, M. Kapalka, and P. Kouznetsov, “The weakest failure detectors to boost obstruction-freedom,” *Proceedings of the 20th International Symposium*

on Distributed Computing (DISC), ser. Lecture Notes in Computer Science, vol. 4167, pp. 399–412, September 2006.

[34] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Stable leader election," *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 2108, pp. 108–122, October 2001.

[35] S. M. Pike and P. A. G. Sivilotti, "Dining philosophers with crash locality 1," *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 22–29, March 2004.

[36] C. Fetzer, U. Schmid, and M. Susskraut, "On the possibility of consensus in asynchronous systems with finite average response times," *Proceedings of the 25th IEEE International Conference on Distributed Computing System (ICDCS)*, pp. 271–280, June 2005.

[37] M. Larrea, A. Fernandez, and S. Arevalo, "Optimal implementation of the weakest failure detector for solving consensus," *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 52–59, October 2000.

[38] I. Sotoma and E. R. M. Madeira, "Adaptation–algorithms to adaptive fault monitoring and their implementation on CORBA," *Proceedings of the 3rd IEEE International Symposium on Distributed Objects and Applications (DOA)*, pp. 219–228, September 2001.

[39] C. Fetzer, "Enforcing perfect failure detection," *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 350–357, April 2001.

[40] S. Sastry and S. M. Pike, "Eventually perfect failure detectors using ADD channels," *Proceedings of the 5th International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, ser. Lecture Notes in Computer Science, vol. 4742, pp. 483–496, August 2007.

[41] M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN)*, pp. 354–363, June 2002.

[42] C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol," *Proceedings of the 8th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 146–153, December 2001.

[43] M. Larrea, S. Arevalo, and A. Fernandez, "Efficient algorithms to implement unreliable failure detectors in partially synchronous systems," *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, ser. Lecture Notes in Computer Science, vol. 1693, pp. 34–49, September 1999.

[44] A. Mostefaoui, E. Mourgaya, and M. Raynal, "Asynchronous implementation of failure detectors," *Proceedings of 33rd IEEE International Conference on Dependable Systems and Networks (DSN)*, pp. 351–360, June 2003.

[45] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, October 1985.

[46] N. A. Lynch, *Distributed Algorithms.* San Francisco, CA: Morgan Kaufmann, 1996.

[47] M. Nesterenko and A. Arora, "Dining philosophers that tolerate malicious

crashes," *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pp. 191–198, July 2002.

[48] I. Rhee, "Optimal fault-tolerant resource allocation in dynamic distributed systems," *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pp. 460–467, October 1995.

[49] P. A. G. Sivilotti, S. M. Pike, and N. Sridhar, "A new distributed resource-allocation algorithm with optimal failure locality," *Proceedings of the 12th International Conference on Parallel and Distributed Computing and Systems (PDCS)*, vol. 2, pp. 524–529, August 2000.

[50] Y.-K. Tsay and R. L. Bagrodia, "An algorithm with optimal failure locality for the dining philosophers problem," *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, ser. Lecture Notes in Computer Science, vol. 857, pp. 296–310, September 1994.

[51] J. H. Anderson, Y.-J. Kim, and T. Herman, "Shared-memory mutual exclusion: major research trends since 1986," *Distributed Computing*, vol. 16, no. 2-3, pp. 75–110, September 2003.

[52] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, September 1965.

[53] M. Raynal, *Algorithms for Mutual Exclusion.* Cambridge, MA: MIT Press, 1986.

[54] J. E. Burns and N. A. Lynch, "Bounds on shared memory for mutual exclusion," *Information and Computation*, vol. 107, no. 2, pp. 171–184, December 1993.

[55] L. Lamport, “A new solution of Dijkstra’s concurrent programming problem,” *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, August 1974.

[56] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[57] G. L. Peterson, “Myths about the mutual exclusion problem,” *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, June 1981.

[58] G. Ricart and A. K. Agrawala, “An optimal algorithm for mutual exclusion in computer networks,” *Communications of the ACM*, vol. 24, no. 1, pp. 9–17, January 1981.

[59] B. Awerbuch and M. E. Saks, “A dining philosophers algorithm with polynomial response time,” *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 65–74, October 1990.

[60] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, MA: Addison-Wesley, 1988.

[61] M. Choy and A. K. Singh, “Efficient fault tolerant algorithms for resource allocation in distributed systems,” *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 593–602, May 1992.

[62] P. Keane and M. Moir, “A general resource allocation synchronization problem,” *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 557–564, April 2001.

[63] I. Page, T. Jacob, and E. Chern, “Fast algorithms for distributed resource allocation,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 4, no. 2, pp. 188–197, February 1993.

[64] I. Rhee, “A modular algorithm for resource allocation,” *Distributed Computing*, vol. 11, no. 3, pp. 157–168, August 1998.

[65] E. Styer and G. L. Peterson, “Improved algorithms for distributed resource allocation,” *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 105–116, August 1988.

[66] S.-T. Huang, “The fuzzy philosophers,” *Proceedings of the 15th IPDPS Workshop on Parallel and Distributed Processing*, ser. Lecture Notes in Computer Science, vol. 1800, pp. 130–136, May 2000.

[67] O. M. Herescu and C. Palamidessi, “On the generalized dining philosophers problem,” *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 81–89, August 2001.

[68] D. Lehmann and M. O. Rabin, “On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem,” *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 133–138, January 1981.

[69] V. K. Idimadakala, “Dining philosophers with masking tolerance to crash faults,” Master’s thesis, Texas A&M University, Department of Computer Science, December 2006.

[70] K. M. Chandy and J. Misra, “The drinking philosophers problem,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, no. 4, pp. 632–646, October 1984.

[71] S.-H. Chen and T.-L. Huang, “A fair and space-efficient mutual exclusion,” *Proceedings of the 11th International Conference on Parallel and Distributed Systems*

(ICPADS), pp. 467–473, July 2005.

[72] J. E. Burns, “Mutual exclusion with linear waiting using binary shared variables,” *The ACM Special Interest Group on Algorithms and Computation Theory (SIGACT) News*, vol. 10, no. 2, pp. 42–47, 1978.

[73] M. A. Eisenberg and M. R. McGuire, “Further comments on Dijkstra’s concurrent programming control problem,” *Communications of the ACM*, vol. 15, no. 11, p. 999, November 1972.

[74] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, “Distributed FIFO allocation of identical resources using small shared space,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 1, pp. 90–114, January 1989.

[75] L. Lamport, “The mutual exclusion problem: Part II—statement and solutions,” *Journal of the ACM*, vol. 33, no. 2, pp. 327–348, April 1986.

[76] E. A. Lycklama and V. Hadzilacos, “A first-come-first-served mutual-exclusion algorithm with small communication variables,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 558–576, October 1991.

[77] R. L. Rivest and V. R. Pratt, “The mutual exclusion problem for unreliable processes: Preliminary report,” *Proceedings of the 17th IEEE Annual Symposium on the Foundations of Computer Science (FOCS)*, pp. 1–8, October 1976.

[78] J.-M. Couvreur, N. Francez, and M. Gouda, “Asynchronous unison,” *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 486–493, June 1992.

[79] M. G. Gouda and T. Herman, "Stabilizing unison," *Information Processing Letters*, vol. 35, no. 4, pp. 171–175, August 1990.

[80] H. Attiya and R. Friedman, "Limitations of fast consistency conditions for distributed shared memories," *Information Processing Letters*, vol. 57, no. 5, pp. 243–248, March 1996.

[81] Y. Song, S. M. Pike, and S. Sastry, "The weakest failure detector for wait-free, eventually fair mutual exclusion," Technical Report TAMU-CS-TR-2007-2-2, Texas A&M University, College Station, TX USA, Feburary 2007.

VITA

Yantao Song was born in Changzhi, Shanxi Province, China. He attended high school No. 2 in Changzhi, Shanxi Province, China and received his high school diploma in July 1991. Mr. Song entered Beijing Institute of Technology in September 1991 and received his baccalaureate degree in Automatic Control in July 1995. Upon completion, Mr. Song entered the masters program at the Institute of Acoustics, Chinese Academy of Sciences and received his masters degree in Signal and Information Processing in July 1998. Then Mr. Song entered graduate school in the department of computer science, Texas A&M University in fall 2004. He passed his preliminary exams and was admitted to Ph.D. candidacy in December 2007. He received his Doctoral degree in December 2008.

Mr. Song can be reached at Department of Computer Science, Texas A&M University, 3112 TAMU, College Station, TX 77843. His email address is songyantao@gmail.com.

This dissertation was typeset in LaTeX by Yantao Song.