Dining Philosophers with Crash Locality 1

Scott M. Pike
The Ohio State University
Columbus, OH 43210, USA
pike@cis.ohio-state.edu

Paolo A. G. Sivilotti
The Ohio State University
Columbus, OH 43210, USA
paolo@cis.ohio-state.edu

Abstract

Ideally, distributed algorithms isolate the side-effects of faults within local neighborhoods of impact. Failure locality quanti£es this concept as the maximum radius of impact caused by a given fault. We present new locality results for the dining philosophers problem subject to crash failures. The optimal crash locality for dining is 0 in synchronous networks, but degrades to 2 in asynchronous networks. Using the eventually-perfect failure detector $\diamond\mathcal{P}$, we construct the £rst known dining algorithms with crash locality 1 under partial synchrony. These algorithms close the failure-locality complexity gap and improve the crash tolerance of resource allocation algorithms in practical networks. We prove the optimality of our results with two fundamental theorems. First, no dining solution using $\diamond\mathcal{P}$ achieves locality 0. Second, $\diamond\mathcal{P}$ is the weakest failure detector in the Chandra-Toueg hierarchy to realize locality 1.

1. A Complexity Gap for Crash Locality?

Masking fault tolerance guarantees that a system satis£es its speci£cation, even in the presence of certain faults. Unfortunately, masking tolerance can be prohibitively expensive or simply impossible. An alternative approach is to quarantine faults by restricting their negative side-effects to small, local neighborhoods of impact. *Failure locality* measures impact as the radius of the worst-case set of processes disrupted by a given fault [1]. The locality radius of a distributed algorithm demarcates a halo outside of which faults are masked. As such, failure locality is a metric that quanti£es masking tolerance, where classical masking corresponds to failure locality 0.

Dining philosophers is a fundamental problem in distributed resource allocation. Originally proposed by Dijkstra for a ring topology [2], dining was later generalized to arbitrary con¤ict graphs by Lynch [3]. Dining solutions are a basic building block for higher-order synchronization problems such as drinking philosophers [4], job-scheduling [5], and committee coordination [6]. Dining has also been a canonical problem for research on crash-failure locality. For asynchronous systems, it is known that 2 is a lower bound on the failure locality of dining algorithms subject to crash faults [1]. This bound is tight, as several existing algorithms achieve it [7]–[10]. By contrast, crash locality 0 is possible in synchronous systems, because standard time-out mechanisms can reliably distinguish crashed processes from ones that are merely slow.

This complexity gap merits investigation for at least three reasons. First, there is a theoretical question whether this is truly a gap in complexity, or simply a gap in our knowledge of fault localization. Second, many real systems are best modeled by intermediate degrees of *partial synchrony* [11], [12], which, although weaker than pure synchrony, nonetheless satisfy stronger timing properties than outright asynchrony. As a practical matter, can we improve fault localization in such systems? Third, since failure locality denotes a radius, the cardinality of the worst-case neighborhood of impact is *exponential* in this value. Thus, for a failure-local-*k* algorithm, a single crashed process may disrupt as many as δ^k other nodes, where δ is the maximum degree of the con¤ict graph. The upshot is that even small improvements in failure locality translate into dramatic improvements in crash tolerance.

This paper addresses the foregoing questions by proposing a general technique for constructing dining algorithms with improved crash-failure localization. Our approach is based on a concept of *skepticism* that can be implemented using unreliable failure detectors from the class $\diamond\mathcal{P}$. The eventually perfect failure detectors in this class have known implementations in several practical models of partial synchrony [12]–[14]. Our main result is a suite of dining algorithms that use skepticism to tolerate process crashes with failure locality 1. We illustrate the generality of this approach by transforming three characteristically diverse dining algorithms into augmented solutions with crash locality 1.

To our knowledge, these algorithms are the £rst to achieve failure locality 1 for dining algorithms subject to crash failures in non-synchronous networks. Our result reduces the cardinality of the worst-case neighborhood of impact from quadratic to linear in δ. We also provide two theoretical results. First, we prove that $\diamond\mathcal{P}$ is insuf£cient for tolerating crashes with failure locality 0. Consequently, our algorithms achieve a tight lower bound on failure localization with respect to $\diamond\mathcal{P}$. Second, we prove that $\diamond\mathcal{P}$ is the *weakest* class of failure detectors in the classic Chandra-Toueg hierarchy [13] capable of realizing failure locality 1. Thus, using $\diamond\mathcal{P}$ to implement skepticism was also a necessary assumption.

The remainder of this paper is organized as follows. Section 2 de£nes the background and transformational strategy of the paper. Sections 3–5 illustrate transformations of the following dining algorithms: asynchronous doorways [1]; hierarchical resource allocation [3]; hygienic [6]. Sections 6–8 prove impossibility results for crash-local-1 dining with $\diamond\mathcal{P}$.

2. Background and Transformation Strategy

This section de£nes the terms and concepts used in our transformations. We defer the formal framework required for the impossibility theorems to Section 6.

Computational Model. We consider asynchronous systems where message delay, clock drift, and relative process speeds are unbounded. A system is modeled by a set of n distributed processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ that communicate only by asynchronous message passing. Every pair of processes in the communication graph is connected by a reliable channel, such that every message sent to a correct process is eventually received by that process, and messages are neither duplicated, lost, nor corrupted. Processes may fail only by crashing. A process cannot crash at will, but only as the result of a crash fault, which occurs when a process ceases execution (without warning) and remains permanently crashed thereafter [15].

Dining Philosophers. A dining instance is modeled by an undirected con¤ict graph $DP = (\Pi, E)$, where each vertex $p \in \Pi$ represents a diner, and each edge $(p, q) \in E$ represents a potential resource con¤ict between diners p and q. Adjacent diners are called *neighbors* and are modeled by a neighbor relation $N \subseteq V \times V$ such that $N(p, q)$ iff $(p, q) \in E$. Furthermore, each diner is in one of three states: *thinking*, *hungry*, or *eating*. Diners control their transitions from thinking to hungry, and from eating back to thinking. Thinking may last inde£nitely, but eating durations must be £nite. Hungry neighbors are said to be in *con¤ict*, because both are vying for shared (but mutually exclusive) resources. A correct dining algorithm is a con¤ict-resolution strategy to schedule diner transitions from hungry to eating, subject to:

Safety: Live neighbors never eat simultaneously.
Progress: Every hungry process eventually eats.

The safety property is weaker than strict mutual exclusion. Speci£cally, a process may eat simultaneously with *crashed* neighbors [16]. This decision is motivated by considering resources that are either stateless (such as transmission frequencies) or recoverable (such as by checkpointing or atomic transactions). By contrast, strict mutual exclusion forbids neighbors from eating simultaneously whatsoever. Strict exclusion better models corruptible resources. For example, a stateful resource may be rendered unusable if left in an inconsistent state by a crashing diner. This paper does not consider strict exclusion, because (fundamentally) it is less interesting in the context of fault localization; to wit, strict exclusion makes the lower bound on crash locality 1 by *de£nition*, so differences in failure-detection capabilities become less meaningful.

Tolerance Metric. Failure locality measures the robustness of a distributed algorithm in the presence of faults, which, for this paper, we restrict to process crashes. Accordingly, we will use the terms *failure locality* and *crash locality* interchangeably. Previous work on crash locality considered only token-based algorithms for which fail-safety (i.e., mutual exclusion in the presence of crashes) is often trivial [7]–[10]. Accordingly, we restrict our attention to progress violations (i.e., processes that *starve* as the result of a crash fault). The *k-neighborhood* of a process p is the set of processes reachable by at most k hops from p in the con¤ict graph. A dining algorithm is said to have *crash locality k* if the worst-case set of processes that starve (as the result of a crashed process p) is a subset of the k-neighborhood of p. Equivalently, every process q beyond the k-neighborhood of a crashed process p will continue to satisfy its safety and progress speci£cation.

Impossibility Results. Our goal in the £rst half of this paper is to construct failure-local-1 ($\mathcal{FL}_1$) dining algorithms. This requirement re£nes the original progress property so that (even in the presence of crashes) hungry processes must eventually eat if no process in their 1-neighborhood crashes:

$\mathcal{FL}_1$-Progress: Every hungry process eventually eats or has a crashed process in its 1-neighborhood (possibly itself).

Even though hungry neighbors of a crashed process are permitted to starve, $\mathcal{FL}_1$ dining cannot be implemented in asynchronous systems subject to even a single crash fault. This impossibility result is a corollary of the fact that 2 is the optimal crash locality for dining in asynchronous systems [1]. Wait-for dependencies can form when a hungry neighbor of a crashed diner is unable to make progress, but is also uncertain whether it can safely yield priority to other hungry neighbors without precipitating its own starvation. As wait-chains extend throughout the con¤ict graph, resource utilization and concurrency degrade and widespread starvation can ensue. For intolerant algorithms, a single crash failure can potentially cascade into global starvation, a fact that underscores the importance of improved techniques for fault-localization.

Failure Detectors. To circumvent the impossibility results, we augment the asynchronous model of computation with an *eventually perfect failure detector* $\diamond\mathcal{P}$ ("diamond P"). An unreliable failure detector [13] can be viewed as a distributed oracle that can be queried for (possibly incorrect) information about crashes in Π. Each process has access to its own local detector module that outputs the set of processes currently suspected of having crashed. Unreliable failure detectors are characterized by the kinds of *mistakes* they can make. Mistakes can include false-negatives (*i.e.*, not suspecting a crashed process), as well as false-positives (*i.e.*, wrongfully suspecting a correct process). Unreliable detectors from the class $\diamond\mathcal{P}$ are characterized by the following properties [13]:

Strong Completeness: Every crashed process is eventually and permanently suspected by every correct process.

Eventual Strong Accuracy: For each run, there exists a time after which no correct process is wrongfully suspected by any other correct process.

Failure detectors in $\diamond\mathcal{P}$ may commit an arbitrary (but £nite) number of false-positive mistakes during the computational pre£x of any run. After some point, however, $\diamond\mathcal{P}$ detectors converge to being well-founded, after which they provide reliable information about crashes. Unfortunately, the time to convergence is not known, and it may vary from run to run.

Transformation Strategy: We say that a $\diamond\mathcal{P}$ detector is *well-founded* once it converges to strong accuracy. Since convergence occurs after some £nite pre£x, the detector remains well-founded for an in£nite computational suf£x thereafter. Furthermore, we say that a process p is *skeptical* if some process in p's 1-neighborhood is suspected by p's local detection module. If a process p becomes hungry, but does not proceed to eating for a suf£ciently long time, then eventually p will still be hungry and its failure detector will have become well-founded. Thereafter, if its neighbors never crash, p will never become skeptical and so must be permitted to eat, eventually. Alternatively, if some neighbor of p crashes, then (by strong completeness) p's detector module will suspect that neighbor, and p will eventually become permanently skeptical. Thereafter, p is not required to eat under $\mathcal{FL}_1$. These observations yield the following proof obligation for establishing the progress of $\mathcal{FL}_1$ dining algorithms:

Proof Obligation for $\mathcal{FL}_1$-Progress: *Every hungry process eventually eats or becomes permanently skeptical.*

Many asynchronous dining algorithms can be transformed into $\mathcal{FL}_1$ algorithms by using $\diamond\mathcal{P}$ to implement skepticism. Achieving $\mathcal{FL}_1$ requires neighbors of a crashed process to isolate other neighbors from the effects of the crash. Using skepticism as a local proxy for crashing neighbors, we get the following principle: *Skeptical philosophers should not prevent their neighbors from eating.*

The actions to execute while skeptical will, of course, depend on the algorithm being transformed. As a general guide, however, note that in most non-degenerate algorithms, a thinking process never prevents a neighbor from eating. Thus, a skeptical process could protect its neighbors by behaving like a thinking process. This general heuristic must be applied with care. Speci£cally, it is not always possible to just "cleanly release" all acquired permissions and resources. In many dining algorithms, the transition to thinking occurs only from the eating state. Thus, establishing the conditions necessary to behave like a thinking process may be predicated on having £rst established the conditions required to be eating. Since a process can (potentially) become skeptical in any state, it may not always be possible to satisfy the invariant that characterizes a thinking process. As we will show, in such cases, a weaker (but reachable) invariant may suf£ce.

The transformations in the following sections are based on the following algorithms: Section 3 transforms the Asynchronous Doorway Algorithm by Choy and Singh [1]; Section 4 transforms the Hierarchical Resource Allocation algorithm by Lynch [3]; and Section 5 transforms the Hygienic Algorithm by Chandy and Misra [6]. These candidates were chosen primarily for their diversity as doorway-based, permission-based, and token-based dining algorithms, respectively.

3. Asynchronous Doorway Algorithm

The Asynchronous Doorway Algorithm (ADA) is due to Choy and Singh, who were the £rst to consider the failure locality of dining algorithms [1]. ADA was the £rst of several algorithms constructed by Choy and Singh using various doorway mechanisms to ensure progress and improve failure locality. For an in-depth description of ADA and its proof of correctness, we refer the reader to the original papers [1], [7].

3.1. Original Algorithm (Synopsis)

In ADA, diners are assumed to have a static total ordering on process IDs. As an optimization, a partial order is suf£cient, provided each diner's ID is distinct from each of its neighbors. Standard node-coloring algorithms can compute such orderings, which, in the worst case, require as many as $\delta + 1$ distinct IDs (where δ is the maximum degree of the con¤ict graph). A process shares a fork with each neighbor and it must hold all shared forks to eat. Con¤ict resolutions are always resolved (statically) in favor of the diner with the higher ID. To prevent higher-priority processes from starving lower-priority neighbors, ADA uses an asynchronous doorway to prevent unbounded over-taking. See Fig. 1 for reference.

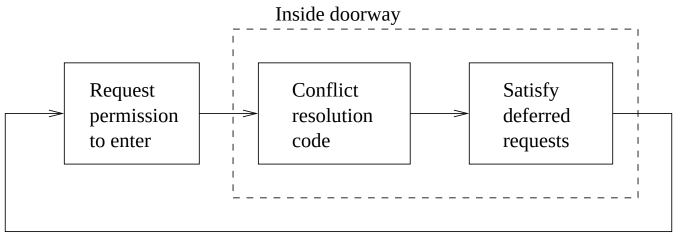


Figure 1. Asynchronous doorway algorithm.

Thinking processes are always outside the doorway. A process can only eat if it is inside the doorway *and* it holds all of its forks. Thus, upon becoming hungry, a process must eventually enter the doorway and then collect all of its forks. ADA accomplishes this in two phases.

Phase 1: Upon becoming hungry, a process £rst requests permission from all of its neighbors to enter the doorway. Any neighbor also outside the doorway grants this permission without delay. Neighbors inside the doorway, however, defer granting permission until after they proceed to (and £nish) eating. This mechanism prevents unbounded over-taking, because higher-priority neighbors cannot re-enter the doorway until lower-priority neighbors already inside the doorway have exited. Once a hungry process receives permissions from all neighbors, it proceeds inside the doorway to Phase 2.

Phase 2: Once a hungry process is inside the doorway, it requests all missing forks. Neighbors still outside the doorway (whether thinking or hungry) yield their forks without delay. Hungry neighbors inside the doorway yield their forks based on process ID; lower IDs yield the fork immediately (and re-request it), but higher IDs defer the fork request until after they proceed to (and £nish) eating. Eating neighbors also defer fork requests until they £nish eating, after which they exit the doorway and grant all deferred permissions and fork requests (thereby enabling hungry neighbors to make progress).

Correctness. Safety is guaranteed because a process eats only if it holds all of its forks, and no two neighbors can hold the shared fork simultaneously. Progress is achieved in two stages. First, hungry diners inside the doorway eventually eat. This is because higher priority neighbors eventually exit the doorway and are not permitted back in. Second, hungry diners outside the doorway eventually enter. Similarly, this is because every neighbor eventually exits the doorway, at which point all deferred permissions are granted.

3.2. Transformation

ADA has crash locality $\delta+2$ (see [1] for details). Informally, this is because permissions are granted *asynchronously*. In the worst-case, the con¤ict graph requires $\delta+1$ distinct process IDs. It is possible for $\delta+1$ hungry processes to enter the doorway simultaneously if all receive their £nal permission at once. If the highest priority process then crashes while inside the doorway, starvation may cascade down the entire chain of $\delta+1$ processes, plus an additional process waiting outside the doorway for a permission that will never arrive.

Wait-chains originate with crashed processes inside the doorway. High-priority neighbors executing Phase 2 extend such chains by deferring permissions and fork requests. By contrast, thinking processes and hungry processes outside the doorway (executing Phase 1) always grant permissions and forks without delay. Thus, we can transform ADA into an $\mathcal{FL}_1$ algorithm simply by having skeptical hungry processes behave like thinking processes. Speci£cally, such processes inside the doorway should abort Phase 2 by granting all deferred requests and exiting. Once outside the doorway, a skeptical process should continue granting permissions and forks to any requesting neighbors (just as a thinking process would). The strong completeness of $\diamond\mathcal{P}$ guarantees that every neighbor of a crashed process eventually becomes permanently skeptical, so the transformed algorithm is $\mathcal{FL}_1$ because nobody else waits on a skeptical process.

If a hungry process p becomes skeptical, its progress may be stalled by exiting the doorway. To witness progress, we consider two cases. First, if p's skepticism is well-founded, then some neighbor will or has crashed; since $\mathcal{FL}_1$ dining does not require p to eat with a crashed neighbor, it's permissible for p to remain permanently skeptical and never eat. In case two, p's skepticism may not be well-founded, but instead was the result of some false-positive suspicion committed by $\diamond\mathcal{P}$. If p ceases to be skeptical in the future, it can safely resume being hungry by starting over with Phase 1. Since $\diamond\mathcal{P}$ makes only £nitely many mistakes before converging to strong accuracy, p will either continue to Phase 2 and eventually eat, or a neighbor will crash and return us to case 1. Either way, p eventually eats or becomes permanently skeptical.

4. Hierarchical Resource Allocation

The Hierarchical Resource Allocation algorithm (HRA) by Lynch is described in [3] and [17]. HRA is so-named because every resource in the system is totally ordered according to some static, hierarchical ranking. Like the total ordering on process IDs in ADA, this condition can be relaxed to a partial order, provided that it projects a total order on the individual resource requirements of each diner.

4.1. Original Algorithm (Synopsis)

Each resource in the system has an associated FIFO queue of process IDs which is shared among all diners requiring that resource. Being the process ID at the head of a queue implies holding the "fork" for the associated resource. Upon becoming hungry, a diner requests its required resources, one at a time, in rank order from least to greatest. To accomplish this, a hungry diner enqueues its process ID onto the queue associated with the lowest-ranked resource in its required resource set. The diner waits until it gets to the head of the current queue, after which it enqueues its process ID into the queue of its next-lowest resource. A process eats only once it gets to the head of every queue in its resource set (*i.e.*, once it has obtained all of its forks). After eating, a process dequeues its ID from every queue in its resource set.

Correctness. HRA implements mutual exclusion because a process only eats if it holds all of its forks. At most one process can hold the fork for any given resource, so neighboring processes cannot eat simultaneously. Furthermore, progress is guaranteed because the policy of requesting resources in rank-order prevents request cycles from forming among the diners. For details of this proof, we refer the reader to [3] and [17].

4.2. Transformation

In ADA, a skeptical hungry process could curtail wait-chains simply by behaving like a thinking process. Following suit, what invariant de£nes the behavior of thinking processes in HRA? One property is that the ID of a thinking process is never in any queue. Unfortunately, this condition is too strong to satisfy (uniformly) by every skeptical process. Still, this approach is instructive to consider as a point of analysis.

Upon becoming skeptical, a hungry diner is at the head of zero or more queues. These queues correspond to the resources for which the process already holds forks. To break wait-chains, a skeptical diner should clearly dequeue its ID from all such queues. Beyond this, however, a skeptical hungry process is potentially in *at most* one more queue, still waiting to get to the head. If the process ever gets to the head, it can dequeue its ID to satisfy the invariant of a thinking process (stated above). Progress, however, is not guaranteed, because the skeptical process may be suspecting a neighbor that has actually crashed, and whose ID is currently *farther ahead* in this same queue. In this case, the ID of the skeptical process will be stuck in the resource queue, never to be removed.

As it turns out, this unbreakable dependency is only the appearance of a problem. Any diner waiting in the same queue as a crashed process is already an immediate neighbor of that process, so progress is not required under $\mathcal{FL}_1$ dining. Our original invariant was too strong, but a weaker (and reachable) invariant will suf£ce: *A thinking process is never at the head of any queue*. This condition can be satis£ed locally by any skeptical diner. The only wrinkle is that if a hungry diner ceases to be skeptical (due to false-positive suspicion), it must wait until its ID has been removed *from all resource queues* before restarting the hierarchical queuing scheme. Otherwise, cycles may form among the request queues.

5. Hygienic Dining Philosophers

The hygienic algorithm is by Chandy and Misra [4]. The hygienic algorithm uses process priorities to break symmetries in resolving con¤icts. The algorithm derives its name from the mechanism used to encode the relative priority between neighbors: their shared fork is either clean or dirty. In the hygienic algorithm (where only "clean" forks are exchanged), process priorities are *dynamic*, because processes lower their priority below their neighbors after eating.

5.1. Original Algorithm (Synopsis)

A fork is associated with each edge of the con¤ict graph. Upon becoming hungry, a diner requests all missing forks. Diners relinquish forks only if the request comes from a higher priority neighbor, otherwise the request is deferred. Priority is encoded in the state of the fork: clean or dirty. If a diner holds a clean fork, it has higher priority than the requesting neighbor. If a process holds a dirty fork, it has lower priority than that neighbor. Thus, requests for dirty forks are granted while requests for clean forks are deferred. After eating, all forks become dirty and all deferred requests are satis£ed.

```
Become hungry →
    **if** hold all forks, begin eating
    **else** request all missing forks
Receive fork →
    ¤ip state of fork (clean/dirty)
    **if** hold all forks, begin eating
Receive request →
    **if** fork is dirty and not eating
        send fork and, if hungry, re-request
    **else** defer request
Done eating →
    all forks become dirty
    satisfy deferred fork requests
```

Figure 2. Original Hygienic Algorithm

Correctness. Safety is ensured because eating diners must hold every shared fork. To ensure progress, priorities are initially assigned so as to form an acyclic graph, *i.e.*, a partial order. This graph is modi£ed only by a diner eating and then lowering its priority below all its neighbors. This modi£cation preserves the acyclicity of the graph. Since a diner only lowers its priority when it eats, a hungry diner either eats or rises in the partial order. Since the partial order is £nite, a hungry diner eventually eats. Proof details can be found in [4] and [6].

5.2. Transformation

The crash locality of hygienic dining is the diameter of the con¤ict graph (global). If an eating process crashes at the top of the partial order, its lower priority hungry neighbors will never receive their requested fork. These neighbors, in turn, will not relinquish (clean) forks to their lower priority neighbors. Process priorities are dynamic, so remote processes will eventually fall under the wait-chain too once their priority reduces after suf£ciently many eating sessions.

To create an $\mathcal{FL}_1$ algorithm, we observe that thinking processes always satisfy fork requests without delay, and hence, no process waits on a thinking neighbor. This property is also true of any diner whose priority is lowest among all its neighbors. Thus, a skeptical diner can break wait-chains by lowering its priority below that of all (non-skeptical) neighbors. That is, a skeptical diner relinquishes forks (whether clean or dirty) to any hungry, non-skeptical neighbor. The new algorithm (summarized in Fig. 3) is augmented with actions for transitions into and out of skepticism. The action for receiving a request also relinquishes the fork if skeptical.

```
Become hungry →
    **if** hold all forks, begin eating
    **else** request all missing forks
Receive fork →
    ¤ip state of fork (clean/dirty)
    **if** hold all forks, begin eating
Receive request →
    **if** fork is dirty and not eating
        send fork and, if hungry, re-request
    **else if** skeptical and not eating
        send fork
    **else** defer request
Become skeptical →
    **if** not eating,
        satisfy deferred fork requests
Stop being skeptical →
    **if** hungry,
        **if** hold all forks, begin eating
        **else** request all missing forks
Done eating →
    all forks become dirty
    satisfy deferred fork requests
```

Figure 3. Transformed Hygienic Algorithm

The correctness of this algorithm derives from the fact that the transformed priority scheme is still acyclic. Since every skeptical process has lower priority than any non-skeptical process, a cycle in priority (if one exists) cannot contain both kinds of processes. Among processes of the same kind, however, priority is determined by the state of the forks as in the original algorithm, which is guaranteed to be acyclic.

Unlike the original hygienic algorithm, a hungry diner can lower its priority in two ways without eating: (1) by becoming skeptical or (2) by having a neighbor become non-skeptical. After the detector becomes well-founded, a diner who becomes skeptical will eventually become permanently skeptical. Similarly, a skeptical neighbor may only become non-skeptical for a £nite period of time (and this only occurs when the detector is well-founded and suspects a process that will crash, but hasn't crashed *yet*). Either way, priority will reduce under these two conditions only a £nite number of times. In the in£nite suf£x thereafter, a hungry process will rise in the partial order and eat, or become permanently skeptical. Thus, $\mathcal{FL}_1$ progress is satis£ed.

6. Complexity Questions for $\mathcal{FL}_1$ Dining

Having shown that $\mathcal{FL}_1$ dining is possible using $\Diamond\mathcal{P}$, it is important to place the meaning of this result in context. The simplicity of the foregoing transformations might suggest that we have assumed too much by using $\Diamond\mathcal{P}$. This conjecture has two dimensions. First, is $\Diamond\mathcal{P}$ suf£ciently strong to achieve $\mathcal{FL}_0$ dining? Second, is there a strictly weaker (or simply incommensurable) failure detector that can also achieve $\mathcal{FL}_1$?

If the answer to either of these questions is "yes" then our $\mathcal{FL}_1$ transformations using $\Diamond\mathcal{P}$ have only diminished signi£cance. Sections 7 and 8 will prove two impossibility theorems to the contrary, but £rst we pause to introduce some formal machinery used in our proofs. For consistency with existing literature, we adopt the model used in Chandra and Toueg's original formalization of failure detectors [13].

Failure Patterns. We posit a discrete global clock $\mathcal{T}$, with the set of natural numbers $\mathbb{N}$ as its range of clock ticks. $\mathcal{T}$ is merely a conceptual device to simplify our presentation; since $\mathcal{T}$ is inaccessible to processes in Π, it cannot be used to advantage in any algorithm executed by the system. A *failure pattern* F models the occurrence of crash failures in a given run. Speci£cally, F is a function from the global time range $\mathcal{T}$ to the powerset of processes 2^{Π}, where $F(t)$ denotes the subset of processes that have crashed by time t. Since crashed processes never recover, F is monotonically non-decreasing: $\forall t_1 < t_2 : F(t_1) \subseteq F(t_2)$. We say that p *crashes in* F if $p \in \bigcup_{t\in\mathcal{T}} F(t)$; otherwise, we say that p *is correct in* F.

Failure Detectors. A *failure detector history* is a function H from $\Pi \times \mathcal{T}$ to 2^{Π}, where $H(p,t)$ is the set of processes suspected by p's local detector module at time t. We say that p *suspects* q *at time* t *in history* H, if process $q \in H(p,t)$. A failure detector history may output con¤icting information to different processes; thus, for $p \neq q$, it is possible that $H(p,t) \neq H(q,t)$. A failure detector $\mathcal{D}$ provides (possibly incorrect) information about the failure pattern F that occurs in a given run. The unreliability of a failure detector depends on its £delity to actual failure patterns. Formally, a *failure detector* $\mathcal{D}$ is a function that maps each failure pattern, F, to a set of failure detector histories, $\mathcal{D}(F)$. The range of $\mathcal{D}(F)$ is the set of admissible failure detector histories using failure detector $\mathcal{D}$ in algorithmic runs with failure pattern F.

Algorithms. An algorithm A is a set of n deterministic automata, one for each process in Π. A *run of algorithm* A *using a failure detector* $\mathcal{D}$ is a 5-tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$, where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is some admissible history $\mathcal{D}$ on failure pattern F, I is an initial con£guration, S is an in£nite sequence of steps (or *schedule*) of A, and T is an in£nite sequence of increasing time values denoting when each step in the schedule S occurred.

Impossibility Proofs. Although an algorithm has access to its failure detector history $H_{\mathcal{D}}$, the actual failure pattern of a given run is inaccessible. Since failure detectors can be unreliable, it is possible for distinct failure patterns to yield identical failure detector histories. A fundamental approach to proving impossibility results is to construct two runs R and R' with different failure patterns and yet identical schedules and failure detector histories along some pre£x of both runs. Since we consider only deterministic algorithms, the next step of the algorithm must be the same in both runs. This forces a contradiction if the underlying failure patterns actually require distinct behaviors in the respective futures of each run.

7. $\mathcal{FL}_1$ Dining is Optimal for $\Diamond\mathcal{P}$

In this section, we prove that $\Diamond\mathcal{P}$ is not powerful enough to implement dining algorithms with crash locality 0. In conjunction with our transformations, Theorem 1 shows that $\mathcal{FL}_1$ is a tight lower bound for dining algorithms using $\Diamond\mathcal{P}$.

Theorem 1: *No deterministic asynchronous algorithm can solve dining philosophers with crash locality 0 using* $\Diamond\mathcal{P}$.

Proof. Suppose such an $\mathcal{FL}_0$ dining algorithm, A, exists. We will force a contradiction by constructing two runs that are indistinguishable to A along some £nite pre£x that violates mutual exclusion. Consider the system $\Pi = \{p, q\}$, where p and q are neighbors, and where q is hungry while p is eating at some time t_1. We extend this partial run with two possible futures based on distinct failure patterns F and F'.

In failure pattern F, process p crashes at some future time $t_2 > t_1$ while still eating. By hypothesis, algorithm A has crash locality 0. Thus, in an extended run R based on failure pattern F, process q must eventually be scheduled to eat. Process q cannot eat before p crashes without violating mutual exclusion, so let $t_3 > t_2$ denote the time when q begins eating. By contrast, in failure pattern F' neither process crashes. Moreover, in an extended run R' based on this failure pattern, it is perfectly admissible for process p to continue eating until some time $t_4 > t_3$.

Now consider possible failure detector histories H and H' of runs R and R', respectively. In history H, p's detector never suspects anyone, but q's detector suspects p initially and permanently. That is, $H(p,t) = \emptyset$ and $H(q,t) = \{p\}$ for all $t \in \mathcal{T}$. History H' is identical to H, except that q's detector permanently stops suspecting process p after time t_4. That is, $H'(q,t) = \{p\}$ only for $t \le t_4$. It is clear that both failure detector histories provide identical information about their respective failure patterns prior to time t_4.

Furthermore, H and H' are both admissible histories of a $\Diamond\mathcal{P}$ detector over failure patterns F and F', respectively. To see this, note that in H, the only incorrect process, p, is suspected permanently (strong completeness), and the only correct process, q, is never suspected after time 0 (eventual strong accuracy). In history H', no process crashes (so strong completeness is satis£ed vacuously), and no correct process is suspected after time t_4 (eventual strong accuracy).

Recall that A does not have access to the actual failure pattern of a given run, but only to the information provided by its failure detector modules. We have shown that H and H' are admissible histories for $\Diamond\mathcal{P}$, and that both histories provide identical information prior to time t_4. Thus, even though runs R and R' have failure patterns that require distinctly different behavior, these runs are indistinguishable to algorithm A over the £nite pre£x in question.

Since A is deterministic, it executes the same actions in both runs prior to time t_4. Consequently, when process q gets scheduled to eat at time t_3 in run R (as required by failure locality 0), q must also be scheduled to eat at time t_3 in run R'. This action, however, violates mutual exclusion by eating simultaneously with a live neighbor p. The assumption that A solves $\mathcal{FL}_0$ dining philosophers using $\diamond\mathcal{P}$ has been contradicted. Thus, no such algorithm exists. ■

8. The Weakest Detector for $\mathcal{FL}_1$ Dining

In this section, we prove that $\diamond\mathcal{P}$ is the weakest failure detector in the Chandra-Toueg hierarchy capable of implementing $\mathcal{FL}_1$ dining algorithms. Theorem 2 establishes that $\diamond\mathcal{P}$ was actually a necessary assumption in our transformations. The proof approach draws on the structure of the Chandra-Toueg hierarchy, which we pause to review.

In Chandra and Toueg's original classi£cation of unreliable failure detectors [13], each class was de£ned by two properties: *completeness* and *accuracy*. Each property is either *weak* or *strong*; additionally, accuracy is either *perpetual* or *eventual*. Altogether, there are eight distinct classes. Four pairs of these classes are known to be equivalent (via mutual reduction), however, so the resulting hierarchy becomes a lattice with four nodes, as illustrated in Fig. 4.

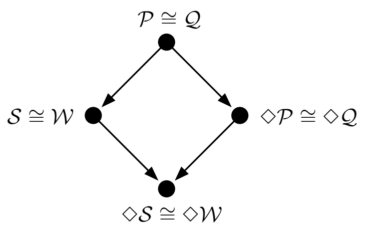

$\mathcal{P} \cong \mathcal{Q}$
$\mathcal{S} \cong \mathcal{W}$
$\diamond\mathcal{P} \cong \diamond\mathcal{Q}$
$\diamond\mathcal{S} \cong \diamond\mathcal{W}$

Figure 4. The Chandra-Toueg hierarchy consists of four equivalence classes of failure detectors. Failure detector classes higher in the lattice are strictly stronger than detector classes lower in the lattice.

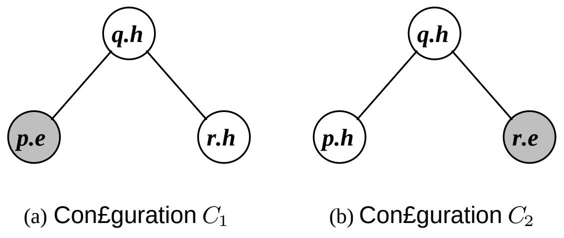
The perfect and quasi-perfect failure detectors ($\mathcal{P}$ and $\mathcal{Q}$) are strongest (*i.e.*, most reliable). The eventually strong and eventually weak detectors ($\diamond\mathcal{S}$ and $\diamond\mathcal{W}$) are the weakest. The intermediate equivalence classes are *incommensurable*; some problems solvable by one cannot be solved by the other, and vice versa. These classes include the strong and weak failure detectors ($\mathcal{S}$ and $\mathcal{W}$), and the eventually perfect and eventually quasi-perfect detectors ($\diamond\mathcal{P}$ and $\diamond\mathcal{Q}$).

To obtain our result, we will prove that no algorithm can solve $\mathcal{FL}_1$ dining using the strong failure detector S. Since the weak detector W is computationally equivalent to S, no algorithm can solve $\mathcal{FL}_1$ dining using W either. The impossibility result also extends down the lattice to the classes $\diamond S$ and $\diamond W$, since both are strictly weaker than S.

Strong Failure Detectors. S satis£es strong completeness (just like $\diamond\mathcal{P}$); thus, every process that crashes is eventually and permanently suspected by every correct process. Unlike $\diamond\mathcal{P}$, however, S satis£es only *weak accuracy*, which requires that *some correct process is never suspected*. Thus, whereas $\diamond\mathcal{P}$ may wrongfully suspect every correct process £nitely many times, S cannot wrongfully suspect some correct process even once. On the other hand, $\diamond\mathcal{P}$ must eventually stop suspecting all correct processes, but S may suspect other correct processes in£nitely often or even permanently so.

Theorem 2: *$\diamond\mathcal{P}$ is the weakest failure detector in the Chandra-Toueg hierarchy suf£cient for solving $\mathcal{FL}_1$ dining.*

Proof. Suppose a deterministic algorithm A solves $\mathcal{FL}_1$ dining using the strong failure detector S. We will force a contradiction by constructing a malicious run where A violates $\mathcal{FL}_1$-Progress. Speci£cally, we will prove by induction that there exists an admissible run of A where no process crashes and yet some hungry process never eats.



(a) Con£guration C_1 (b) Con£guration C_2

Figure 5. System Con£gurations for Theorem 2

Let $\Pi = \{p, q, r\}$ be a system where q is a neighbor of both p and r, but p and r are not neighbors of each other. Consider a con£guration C_1 where q and r are hungry while process p is eating, as in Fig. 5(a). We will show that q can starve in algorithm A. The proof is by induction on the number of times (since q last became hungry) that the system, Π, enters con£guration C_1 before q eats.

Base Case: We require a partial run of A where q becomes hungry, but the system reaches C_1 before q eats. The following schedule suf£ces: p becomes hungry, p begins eating, q becomes hungry, r becomes hungry. In this pre£x, the number of times Π enters C_1 before q can eat is 1.

Inductive Step: In a given partial run, suppose process q becomes hungry and that the system enters con£guration C_1 at least k times while q is still waiting to eat. We must show that this partial run can always be extended so that Π re-visits C_1 at least once more before q eats.

Consider the failure detector history H_S where p and r always suspect each other, but q always suspects p *and* r. That is, for all $t \in \mathcal{T}$: $H_S(p,t) = \{r\}$, $H_S(r,t) = \{p\}$, and $H_S(q,t) = \{p,r\}$. Surprisingly, H_S is an admissible history of the strong detector S for *any* failure pattern of Π in which q never crashes. For all such patterns, q is the correct process that is never suspected in H_S (weak accuracy). Since p and r are suspected initially and permanently, H_S satis£es strong completeness too, regardless of when p or r crash. If neither crashes, strong completeness holds vacuously. Under weak accuracy, permanent suspicion does not imply failure. Thus, the failure pattern in which no process crashes can starve q, because every partial run is indistinguishable from a similar partial run in which p or r have just crashed while eating. Fig. 6 applies this observation to our inductive step.

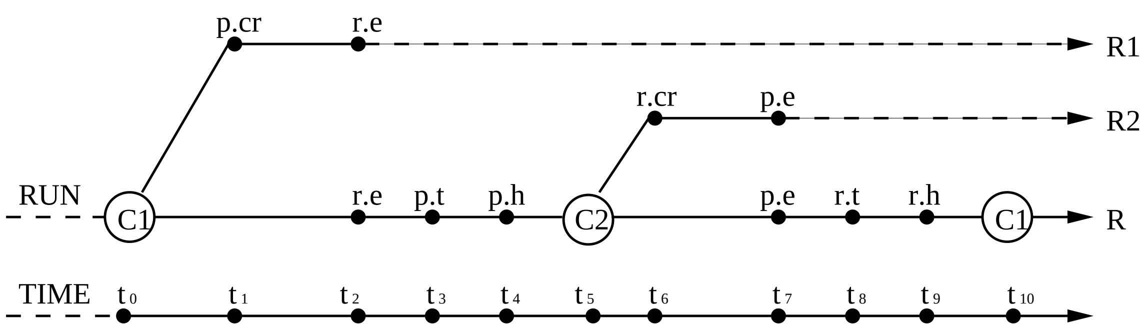


Figure 6. Any partial run up to C_1 can be extended to re-visit C_1 before q eats.

Let $t_0 \in \mathcal{T}$ be the time at which Π entered con£guration C_1 for the k^{th} time while q was still hungry (see Fig. 6). The partial run prior to t_0 can be extended into two possible futures. In extension R_1, p crashes at time t_1 while eating. By hypothesis, A is $\mathcal{FL}_1$, so (hungry) process r must eventually eat in this run. Recall, however, that the failure detector history, H_S, over this pre£x can have the same behavior over run R where p does not crash. Thus, A must also allow r to eat in run R even though p has not crashed. Once r begins eating at time t_2, process p can transit from eating to thinking and back to hungry again. This precipitates con£guration C_2 at time t_5 (see Fig. 5(b)). Since either p or r eats at every moment between C_1 and C_2, their neighbor q remains hungry and does not eat before time t_5.

Now the situation is reversed, with p being hungry while r is eating, and there being two possible futures of this run: extension R_2 where r crashes at time t_6, and extension R where r does not crash. $\mathcal{FL}_1$ dining requires that p be allowed to eat in the former, but again, history H_S has the same behavior in both extensions. Thus, once p begins eating in run R at time t_7, r can cycle from eating to thinking to hungry again to precipitate the original con£guration C_1. As before, either p or r eats at every moment between con£gurations C_2 and C_1, so, again, process q remains hungry and does not eat before time t_{10}, when the system Π enters con£guration C_1 for the $k + 1^{st}$ time.

By induction, there exists a crash-free run where system Π enters con£guration C_1 in£nitely many times before q gets to eat. This violates $\mathcal{FL}_1$ -Progress, because q is a starving process without any crashed neighbors. Thus, our assumption that A achieves $\mathcal{FL}_1$ dining using S was a contradiction. Since no such algorithm exists, $\diamond\mathcal{P}$ is, in fact, the weakest Chandra-Toueg failure detector capable of $\mathcal{FL}_1$ dining. ■

9. Conclusions

The contribution of this paper is two-fold. First, we close the failure-locality complexity gap between asynchronous and synchronous dining solutions; we show that $\mathcal{FL}_1$ dining can be achieved using $\diamond\mathcal{P}$. This result applies to any partially synchronous system in which $\diamond\mathcal{P}$ can be implemented [12], [13]. Our second contribution is a general strategy for transforming off-the-shelf dining solutions into $\mathcal{FL}_1$ algorithms. We illustrated this heuristic by applying it to three algorithmically diverse dining solutions.

Finally we have shown that our failure locality results are strict in the sense that $\diamond\mathcal{P}$ does not permit the construction of dining algorithms with locality better than 1. Moreover, $\diamond\mathcal{P}$ is the weakest detector in the Chandra-Toueg hierarchy capable of implementing crash-local-1 dining algorithms. These results close the locality complexity gap under weak mutual exclusion, and improve the crash tolerance of resource allocation algorithms for many practical networks.

References

[1] Manhoi Choy and Ambuj K. Singh, "Ef£cient fault tolerant algorithms for resource allocation in distributed systems," in *24th ACM Symposium on Theory of Computing*, May 1992, pp. 593–602.

[2] Edsger W. Dijkstra, "Co-operating sequential processes," in *Programming Languages*, F. Genuys, Ed., pp. 43–112. Academic Press, 1968.

[3] Nancy Lynch, "Fast allocation of nearby resources in a distributed system," in *12th ACM Symp. on Theory of Computing*, 1980, pp. 70–81.

[4] K. Mani Chandy and Jayadev Misra, "The drinking philosopher's problem," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pp. 632–646, Oct. 1984.

[5] B. Awerbuch and M. Saks, "A dining philosophers algorithm with polynomial response time," in *31st Annual Symposium on Foundations of Computer Science*, IEEE, Ed., 1990, vol. 1, pp. 65–74.

[6] K. Mani Chandy and Jayadev Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, MA, USA, 1988.

[7] Manhoi Choy and Ambuj K. Singh, "Ef£cient fault-tolerance algorithms for distributed resource allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17, no. 3, pp. 535–559, 1995.

[8] Yih-Kuen Tsay and Rajive L. Bagrodia, "An algorithm with optimal failure locality for the dining philosophers problem," in *8th International Workshop on Distributed Algorithms, WDAG '94*, Gerard Tel and Paul M. B. Vitányi, Eds. 1994, vol. 857 of *LNCS*, pp. 296–310, Springer.

[9] Paolo A. G. Sivilotti, Scott M. Pike, and Nigamanth Sridhar, "A new distributed resource-allocation algorithm with optimal failure locality," in *Proceedings of the 12th Int'l Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov 2000, vol. 2, pp. 524–529.

[10] Mikhail Nesterenko and Anish Arora, "Dining philosophers that tolerate malicious crashes," in *22nd Int'l Conference on Distributed Computing Systems*, 2002, pp. 172–179.

[11] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer, "On the minimal synchronism needed for distributed consensus," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 77–97, 1987.

[12] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer, "Consensus in the presence of partial synchrony," *J. Assoc. Comput. Mach.*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[13] Tushar Deepak Chandra and Sam Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[14] C. Fetzer, M. Raynal, and F. Tronel, "An adaptive failure detection protocol," in *Paci£c Rim International Symposium on Dependable Computing*, 2001, pp. 146–153.

[15] Flavin Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[16] Mikhail Nesterenko, *Personal Correspondence*, 2003.

[17] Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.