A New Distributed Resource-Allocation Algorithm with Optimal Failure Locality

Paolo A.G. Sivilotti, Scott M. Pike, and Nigamanth Sridhar
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277

Abstract

Failure locality measures an algorithm's robustness to process failures. We present a new algorithm for the dining philosophers problem — a classic problem in distributed resource allocation — that has optimal failure locality. As a refinement, the algorithm can be easily parameterized by a simple failure model to achieve super-optimal failure locality in the average case.

Keywords: distributed algorithms, fault tolerance, dining philosophers, failure locality.

1 Introduction

The dining philosophers problem is a classic and fundamental resource allocation problem [6]. Although first formulated as a shared-memory concurrency problem, it has since received considerable attention as a distributed conflict-resolution problem [8]. It can be seen as a generalization of the mutual exclusion problem, in which neighboring processes cannot access a shared resource simultaneously. It has many applications in the construction of other distributed resource-allocation algorithms, including drinking philosophers [1] and committee coordination[2].

Two common metrics for evaluating various solutions to the dining philosophers problem are *message complexity* and *response time*. Message complexity is the number of messages sent in the system as a result of a single process requesting access to a shared resource. Response time is the delay between a process requesting access to a shared resource and access being granted to that process.

Recently, Choy and Singh [3] have defined a new metric for evaluating such solutions: *failure locality*. Failure locality measures the robustness of an algorithm in the presence of (process) failures. The *m-neighborhood* of a process p is defined as the set of processes reachable by at most m hops from p in the conflict graph. (The 0-neighborhood of a process p is just p itself, and its 1-neighborhood is p and its immediate neighbors.) A resource-allocation algorithm has failure locality m if the failure of a process only affects processes within its m-neighborhood. Failure localities of 0 and 1 are not possible under a model of computation with weak fairness, fail-stop failures, and reliable but arbitrarily slow channels [4]. Choy and Singh proved an optimal lower bound of 2 by developing an algorithm with failure locality of 2 [4]. Their algorithm assumes FIFO channels and depends on an interrupt-driven scheme.

In this paper, we present a new algorithm for dining philosophers that also achieves the optimal failure locality of 2. Unlike Choy and Singh, however, this algorithm does not require FIFO channels and does not use an interrupt mechanism. The algorithm is based on a dynamic assignment of priorities to processes that ensures progress while allowing lower-priority processes to overtake higher-priority neighbors in the presence of failures.

The rest of this paper is organized as follows. In Section 2, we define the dining philosophers problem and our models of computation and failure. In Sections 3 and 4, we briefly review the hygienic [2] and asynchronous-doorway [3] solutions to this resource-allocation problem and summarize their performance characteristics. Section 5 is the heart of the paper, where we describe our *dynamic threshold-point* solution as a combination of the two solutions previously outlined. An analysis of the performance of this new solution reveals that it improves the failure locality of these algorithms to 2. In Section 6, we describe how the *expected* failure locality can be made super-optimal if a model for the relative likelihood of process failures is known. Finally, in Section 7 we synopsize our findings.

2 Dining Philosophers Problem

2.1 Model of Computation

We assume that processes are distributed, communicating only by asynchronous message passing. Channels are unordered but reliable, delivering messages (with some unbounded delay) without loss, duplica-

tion, or corruption. Processes, however, can fail. We assume a fail-stop model in which a process (without warning) stops executing commands and remains failed forever [5].

2.2 Problem Specification

An instance of the dining philosophers problem can be viewed as a graph in which the nodes represent processes and the edges define the adjacencies between processes (the neighbor relation). This graph is known as the *conflict graph*. We assume that the conflict graph is a subgraph of the communication graph, so each process can communicate directly with each of its neighbors.

Processes are modeled by one of three states: thinking, hungry, or eating. The only permitted transitions are from thinking to hungry, from hungry to eating, and from eating back to thinking. Processes eat for a finite amount of time, but may think indefinitely. Processes control their transitions from thinking to hungry and from eating to thinking, but the conflict-resolution layer controls their transitions from hungry to eating.

A solution to the dining philosophers problem is an algorithm for this conflict-resolution layer that satisfies two conditions:

Safety: No two neighbors eat simultaneously.

Progress: Every hungry process eats eventually.

The safety property can be ensured by the use of *forks*. A fork is a token shared between two neighbors; there is exactly one fork associated with each edge in the conflict graph. No two neighbors can hold the same fork simultaneously. Moreover, a process can eat only if it holds all of its forks. Thus, no two neighbors can eat simultaneously, so safety is guaranteed.

3 Hygienic Algorithm

The hygienic solution [2] to the dining philosophers problem is based on maintaining a partial order of priority among processes. That is, each edge of the conflict graph is given a direction representing priority such that the graph is acyclic. Processes *lower* in the partial order are said to be of *higher* priority. A fork held by a higher priority neighbor is said to be *clean*, while one held by a lower priority neighbor is said to be *dirty*.

When two hungry processes compete for the same fork, the conflict is resolved in favor of the higher-priority process. There is no deadlock because the acyclicity of the partial order prevents the formation of "waits-for" cycles among the processes.

There are two key parts to the hygienic solution. The first is that a higher-priority hungry process never yields to a lower-priority neighbor (*i.e.*, a hungry process never relinquishes a clean fork). The second is that after a process eats, it lowers its priority below that of *all* its neighbors (*i.e.*, it rises above them in the partial order). This operation preserves the acyclicity of the graph. Together, these properties are sufficient to ensure the required progress property.

Informally, the correctness of this algorithm is seen by the following argument. Because there are no cycles, if a process is hungry, there exists a process (perhaps of higher priority) that is hungry and has no hungry higher-priority neighbors. This process can eat. So, if a process becomes hungry, some (possibly different) process will eventually eat. This satisfies weak (global) progress; namely, if some process is hungry, then eventually some process eats.

To satisfy strong (individual) progress, we must ensure that each process that becomes hungry is, itself, eventually given permission to eat. This is proven with a metric. First, let the edges of the partial order point down (*i.e.*, in increasing priority). Now a process y is *reachable* from a process x exactly when there is a directed path from x to y. A good metric for a hungry process u is the number of (higher-priority) processes reachable from u plus the number of (higher-priority) processes reachable from u that are thinking. Since eating always *lowers* the priority of a process below all its neighbors, this metric cannot increase. Furthermore, it is bounded below (by 0, when the process has no higher-priority neighbors). Finally, it is guaranteed to decrease by the observation that if there is a hungry process, there is a reachable highest-priority hungry process that can eat. [2]

This algorithm has the advantage of optimal message complexity ($O(\delta)$) but suffers from poor failure locality, since it allows the formation of extremely long dependency chains. The problem is that a process p never yields to a lower-priority neighbor, even if p is still missing forks from higher-priority neighbors. In the worst case, the length of the dependency chain can be the diameter of the graph, and hence linear in n, the total number of processes.

4 Asynchronous Doorways

4.1 Static Process Priorities

To improve failure locality, Choy and Singh[3] introduce a fault-tolerant fork-collection scheme that breaks long dependency chains. This is achieved using a preemption mechanism based on static process priorities.

Choy and Singh precompute a fixed partial ordering on processes by node-coloring the conflict graph. Using integers to represent colors, a greedy algorithm can assign a color to each node using at most $\delta + 1$ colors. Since neighboring processes always have distinct col-

ors, the resulting partial order guarantees acyclicity. In this scheme, processes with *lower* color have *higher* priority.

To simplify the discussion, let $color(p)$ denote the integer assigned to process p. If p and q are neighbors, and $color(p) < color(q)$, then q is called a *high neighbor* with respect to p. Similarly, p is called a *low neighbor* with respect to q. Thus, the low neighbors of a process p constitute the set of its higher-priority neighbors.

4.2 The Fork-Collection Scheme

Choy and Singh use the low-neighbor set to define a *threshold point* for their fork-collection scheme. When a process p becomes hungry, it sends a fork request to every low neighbor for which p does not already hold the fork. During this period, p yields the shared fork to any requesting neighbor, regardless of that neighbor's priority relative to p.

Once p has collected every low fork, p reaches its threshold point and sends a fork request to every high neighbor for which p does not already hold the fork. While at its threshold point, p defers fork requests from high neighbors; this protects p from being preempted by lower-priority processes. If p receives a fork request from a higher-priority neighbor, however, then p still gets preempted and must yield the requested fork. Since the fork is released to a low neighbor in p's threshold set, p is no longer at its threshold point. Thus, a consequence of preemption is that p must also yield forks to every high-neighbor whose fork request has been deferred.

At this point, the fork-collection scheme starts over and p pursues its threshold point again. If p manages to collect all of its high and low forks, then p immediately proceeds to eating. While eating, p defers all fork requests from high and low neighbors alike.

4.3 Improving Failure Locality

Choy and Singh's fork-collection scheme achieves a constant failure locality of 2, a great improvement over the linear failure locality of the hygienic solution. If a hungry process p has a low neighbor q that has failed while holding the fork, then p will shield all of its remaining neighbors from starvation. Since q is a low neighbor and p can never get the fork from q, p will never reach its threshold point. Consequently, p will always yield forks to requesting neighbors. In this case, the failure locality is 1.

The second case is when a hungry process p has a high-neighbor q that has failed while holding the fork. If p is at its threshold point, it can starve its high neighbors. This can happen for either of two reasons. First, p may have no low neighbors, so it is trivially at its threshold point and cannot be preempted. Second, all of p's low neighbors may be permanently thinking, so p never receives a preempting request from a hungry low neighbor. In either case, p's high neighbors will be unable to reach their threshold points, because they cannot get the fork from their low neighbor p. Consequently, they will yield to any fork request, thereby shielding the rest of the system from q's failure. In this case, the failure locality is 2.

4.4 Problems with Progress

The algorithm as described above is not correct, because it fails to satisfy the progress requirement. With the static ordering of process priorities, high-priority processes maintain their high priority even after eating. Consequently, such processes can cycle around to preempt lower-priority processes infinitely often. This unbounded overtaking can starve lower-priority hungry neighbors.

To overcome this problem, Choy and Singh use the notion of an *asynchronous doorway*. Abstractly, a doorway is a block of code such that a process outside the doorway (*i.e.*, waiting to execute the code) will be blocked until neighbors already past the doorway have exited (*i.e.*, finished executing the code). The asynchronous doorway imposes a new priority hierarchy on processes; namely, any process which is past the doorway has priority over any process outside the doorway. This mechanism prevents unbounded overtaking by blocking higher-priority processes outside the doorway until lower-priority processes past the doorway have eaten. This modification, while ensuring progress, increases the failure locality from 2 to 3, because the wait-for dependency chain now includes the doorway as well.

5 Dynamic Threshold Points

Choy and Singh's fork-collection scheme alone failed to guarantee progress because the partial ordering of process priorities was static. The asynchronous doorway mechanism eliminated unbounded overtaking, but it increased the failure locality from 2 to 3. Our algorithm ensures progress differently by incorporating a dynamic partial ordering on processes. The result is a correct dining philosophers algorithm with an optimal failure locality of 2.

5.1 Algorithm

The dynamic threshold-point algorithm combines the dynamic partial ordering from Chandy and Misra with the concept of threshold points from Choy and Singh. When a process becomes hungry, it attempts to acquire forks from higher-priority neighbors (*i.e.*, the neighbors in its threshold set). When a process holds all forks from its higher-priority neighbors (*i.e.*, all of

its dirty forks), the process is at its threshold point. Processes at their threshold point do not release clean forks. A hungry process that holds all of its forks can eat. After eating, the process lowers its priority.

The key difference between this algorithm and the hygienic solution is that a hungry process will release a requested clean fork if it is not at its threshold point. This ensures low failure locality by breaking long dependency chains.

The key difference between this algorithm and the asynchronous-doorway solution is that the partial order changes over the course of the computation. This ensures progress because high-priority processes lower their priority after eating, preventing unbounded overtaking of lower-priority hungry neighbors.

Below we give the algorithm in a UNITY-like notation [2]. Every action is identified by a label. Every action begins with a guard, written inside curly braces. If the guard is true when the action is (nondeterministically) selected for execution, the corresponding sequence of statements is executed atomically.

Each process p has a variable *state* which can be *thinking*, *hungry*, or *eating*. The relation $N(p,q)$ is true exactly when p and q are neighbors. We write $p < q$ to indicate that $N(p,q)$ and q has higher priority than p. The variable $fork(p,q)$ indicates the location of the fork shared by p and q, while $clean(p,q)$ is a boolean that is true exactly when this fork is clean. Finally, the variable $req(p,q)$ indicates the location of the request token shared by p and q.

The first action, H_p, has a guard that the process p is hungry and is not at its threshold point. This action request the fork from all of p's low neighbors. The action P_p is enabled when a request from a higher-priority neighbor is received. In this case, p relinquishes the fork. As a consequence of this, p is not at its threshold point. The action E_p is enabled when a process may eat. In this case, p has all its forks and no requests from higher priority neighbors. In this action, p begins eating and raises its height above all its neighbors, thus lowering itself in the partial order. R_p is enabled when p holds both a fork and its corresponding request and p is not at its threshold point. If the fork is dirty, the fork is released (to the higher-priority neighbor). If the fork is clean, the fork is also released (since p is not at its threshold point). This action makes clean forks dirty and vice versa.

initially
$(\forall p :: p.state = \text{thinking})$
$(\forall p, q :: clean(p,q) = \textbf{false})$
$(\forall p, q : p < q : fork(p,q), req(p,q) = p, q)$
Priorities form a partial order

always
$p.tp \equiv (\forall q : p < q : fork(p,q) = p)$
$p.h \equiv p.state = \text{hungry}$
$p.t \equiv p.state = \text{thinking}$
$p.e \equiv p.state = \text{eating}$

assign

$H_p :$ $\{ p.h \wedge \neg p.tp \}$
$(\forall q : N(p,q) \wedge fork(p,q) = q \wedge \neg clean(p,q) :$
$req(p,q) := q ;)$

$P_p :$ $\{ req(p,q) = p \wedge fork(p,q) = p$
$\wedge \neg clean(p,q) \}$
$fork(p,q) := q ;$
$clean(p,q) := \textbf{true} ;$
$req(p,q) := q ;$

$E_p :$ $\{ p.h \wedge (\forall q : N(p,q) : fork(p,q) = p \wedge$
$(clean(p,q) \vee req(p,q) = q)) \}$
$p.state := \text{eating} ;$
$(\forall q : N(p,q) : clean(p,q) := \textbf{false}) ;$

$R_p :$ $\{ req(p,q) = p \wedge fork(p,q) = p \wedge \neg p.tp \}$
$fork(p,q) := q ;$
$clean(p,q) := \neg clean(p,q) ;$

5.2 Proof of Correctness

The proof of progress for the dynamic threshold-point algorithm is similar to that for the hygienic algorithm. In particular, hungry processes do not go up in the partial order, since priority decreases only when a process eats. Secondly, a hungry process does not remain at the bottom of the partial order indefinitely. (Recall that processes at the bottom of the partial order have highest priority).

It is this second condition that requires some attention. Unlike the original hygienic solution, requested forks can be relinquished by a process *even though they are clean*. This happens when the process is not at its threshold point. We must show, then, that a hungry process cannot remain at the bottom indefinitely, continually giving up clean forks to lower-priority neighbors.

We first define what is meant by the bottom of the partial order. We say that a process u is at the bottom of the partial order ($u.bot$) exactly when u has no low neighbors that are hungry. We must prove:

$$u.h \wedge u.bot \leadsto \neg(u.h \wedge u.bot)$$

(The $\leadsto$ symbol indicates a leads-to relation.)
The proof uses the following lemmas.

Lemma 1. A process does not remain hungry and at the bottom of the partial order without eventually reaching its threshold point.

$$u.h \wedge u.bot \leadsto \neg(u.h \wedge u.bot) \vee (u.h \wedge u.bot \wedge u.tp)$$

Proof. We show that a hungry process u, remaining at the bottom of the partial order, eventually reaches its threshold point (*i.e.*, u holds all its dirty forks). Assume $u.bot$ holds. Then for every higher-priority neighbor v of u, v is not hungry. There are two cases: either u holds the shared fork (and it is dirty) or v holds the shared fork (and it is clean). In the former case, u holds the dirty fork. In the later case, v holds the clean fork but is not hungry (since $u.bot$). This means that v must not be at its threshold point. Hence, v releases its fork to u. □

Lemma 2. A hungry process at the bottom of the partial order and at its threshold point eventually eats or is no longer at the bottom of the partial order.

$$u.h \wedge u.bot \wedge u.tp \rightsquigarrow \neg(u.h \wedge u.bot)$$

Proof. We show that a hungry process, u, at its threshold point does not remain at the bottom of the partial order without eating.

First assume that u remains not only at the bottom of the partial order *but also* at its threshold point continuously. In this case, u eventually obtains all of its clean forks. This is because, while at its threshold point, u does not relinquish clean forks. Furthermore, clean forks that u is missing are eventually relinquished by the neighboring process (which holds them as dirty forks).

The second case, then, is that u does not remain at its threshold point continuously. For u to exit its threshold point, it must lose one of its dirty forks to a higher-priority neighbor. This can only occur if the higher-priority neighbor is hungry and therefore u is no longer at the bottom of the partial order. □

We are now ready to prove the fundamental property that a hungry process does not remain at the bottom of the partial order.

$$u.h \wedge u.bot \rightsquigarrow \neg(u.h \wedge u.bot)$$

Proof. We show that a hungry process cannot remain at the bottom of the partial order indefinitely. Either it eats (in which case it is no longer hungry) or a higher-priority neighbor becomes hungry (in which case it is no longer at the bottom). This theorem follows from Lemmas 1 and 2.

$$\begin{array}{ll} & \textbf{true} \\ \equiv & \quad \{ \text{ Lemma 2 } \} \\ & u.h \wedge u.top \wedge u.tp \rightsquigarrow \neg(u.h \wedge u.top) \\ \Rightarrow & \quad \{ \ P \rightsquigarrow P \ \} \\ & \neg(u.h \wedge u.top) \vee (u.h \wedge u.top \wedge u.tp) \\ & \quad \rightsquigarrow \neg(u.h \wedge u.top) \\ \Rightarrow & \quad \{ \text{ Lemma 1 and transitivity } \} \\ & u.h \wedge u.top \rightsquigarrow \neg(u.h \wedge u.top) \end{array}$$

□

5.3 Failure Locality

The dynamic threshold-point algorithm uses the same fork-collection scheme (Section 4.2) as the asynchronous-doorway algorithm. In both algorithms, a dependency chain of length 2 can arise when a process at its threshold point waits on a failed high-neighbor. If this dependency chain occurs inside an asynchronous doorway, however, processes blocked outside the doorway will extend the dependency chain to length 3. The dynamic partial ordering, however, does not increase the failure locality, because it does not introduce an additional "wait-for" dependency.

The analysis is similar to Section 4.3. A hungry process p waiting on a failed low-neighbor q will never reach its threshold point. Consequently, p will always yield forks to requesting neighbors, thereby shielding p's neighbors from the failure of q.

Alternatively, a hungry process p waiting on a failed high-neighbor q may have reached its threshold point. Since a process at its threshold point does not release forks, hungry high-neighbors of p will be starved. The "wait-for" chain ends with p's high-neighbors; since they cannot reach their threshold point, they will always yield forks.

6 Expected Failure Locality

Choy and Singh established a lower bound of 2 on failure locality [4]. Our algorithm attains this failure locality in the worst case. It is possible, however, to achieve super-optimal failure locality in the *expected* case by parameterizing the algorithm with a process-failure model.

Define the *failure set* of a process p as the set of processes that can be affected (starved) by p's failure. This set grows and shrinks over the course of a computation. When p does not hold any forks, the size of the failure set is 1 (just p itself). Failure locality can be defined as the height of a directed minimum-spanning tree of the failure set, with the failed process as the root. Often, the cardinality of the failure set is a more relevant metric than the failure locality itself.

Failure sets need not be "symmetric" with respect to their root. In the hygienic solution, for example, the failure set of a process is all its "high descendents". That is, all the processes that can reach the node (where edges are directed and point down in the partial order). Consider a straight-line conflict graph. The failure locality for hygienic is $O(n)$, which occurs when all edges are pointed down and the node at the bottom of the partial order fails.

The *expected* cardinality of the failure set, however, may be quite a bit smaller. Consider a random orientation of the edges up and down. When a process fails, the size of the failure set is 1 plus the number of "high descendents". If the probability of an "up"

edge is p, the expected cardinality of the failure set is $1/(p-p^2)-1$. For example, with $p = 1/2$, the expected cardinality of the failure set is 3.

Our algorithm creates a failure set that includes the high neighbors of the failed process. In addition, the failure set includes the low neighbors and their high neighbors (*i.e.*, failure locality of 2). On average, then, the algorithm gives an approximate failure locality of $1+w$, where w is the proportion of low neighbors of the failed process.

Therefore, on average the algorithm performs with *better* than optimal failure locality if w is strictly less than 1. In the limit, if w is kept close to 0, then the expected failure locality of the algorithm is 1. That is, if processes can be guaranteed to fail when they have no low neighbors, the algorithm has super-optimal failure locality. Of course, such a guarantee is too strong in our fail-stop model.

Consider the situation, however, where process failures occur according to some known distribution (*e.g.*, negative exponential). Somehow, we would like to keep the processes that are most likely to fail below as many of their neighbors as possible.

To improve the average performance, we make use of the *unhygienic* variation of the dining philosophers algorithm [7]. in which requested forks may arrive dirty. In this solution, a partial order is maintained by assigning an integer height to each process such that neighbors have different heights. After eating, a process must increase its height by some positive amount, but *it is not required to rise above all, or indeed any, of its neighbors.* The only requirements on the new height of the process is that it be (i) greater than the previous height and (ii) different from the height of each neighbor. The first requirement ensures progress by preventing a philosopher from eating infinitely often at the expense of a neighbor, while the second requirement preserves the acyclicity of the graph.

With this modification, failure-prone processes can be kept (on average) below their neighbors by increasing their height comparatively little when they eat. This modification leads to better expected failure locality.

7 Conclusion

In this paper, we have presented a new algorithm for the dining philosophers problem. This algorithm uses concepts from two known algorithms: (i) from the hygienic algorithm, dynamic priority encoded in clean and dirty forks; (ii) from the asynchronous-doorway algorithm, threshold points that permit lower-priority neighbors to overtake higher-priority neighbors. This combination results in an algorithm with optimal failure locality. Unlike Choy and Singh's optimal solution, our algorithm does not require FIFO channels or interrupt mechanisms. Furthermore, this algorithm can be easily parameterized with a simple failure model to achieve super-optimal *expected* failure locality.

There are several promising avenues for future investigation. The notion of a "threshold set" can be generalized to include any subset of lower-priority neighbors. In particular, when the empty set is used, the result is precisely the hygienic algorithm. In this sense, the algorithm presented here can be viewed as a generalization of the hygienic solution.

The analytical treatment of expected failure locality indicates that there is considerable utility in this refinement. We intend to investigate this utility experimentally with simulation studies.

References

[1] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[2] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[3] Manhoi Choy and Ambuj K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems*, 17(3):535–559, May 1995.

[4] Manhoi Choy and Ambuj K. Singh. Localizing failures in distributed synchronization. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):705–716, July 1996.

[5] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[6] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, New York, 1968.

[7] Rajeev Joshi and Jayadev Misra. Maximally concurrent programs. Technical Report TR-99-15, The University of Texas at Austin, Austin, Texas 78712, April 1999.

[8] N. A. Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 70–81, New York, 1980. ACM.