Stabilizing Dining with Failure Locality 1

Hyun Chul Chung[1,⋆] , Srikanth Sastry[2,⋆⋆] , and Jennifer L. Welch[1,⋆]

[1] Texas A&M University, Department of Computer Science & Engineering
{h0c8412,welch}@cse.tamu.edu
[2] CSAIL, MIT
sastry@csail.mit.edu

Abstract. The dining philosophers problem, or simply dining, is a fundamental distributed resource allocation problem. We propose two algorithms for solving stabilizing dining with failure locality 1 in asynchronous shared-memory systems with regular registers. Since this problem cannot be solved in pure asynchrony, we augment the shared-memory system with failure detectors. Specifically, we introduce the local anonymous eventually perfect failure detector $?\diamond\mathcal{P}^1$, and show that this failure detector is sufficient to solve the problem at hand.

1 Introduction

The *dining philosophers problem* [1, 2], or simply *dining*, is a fundamental distributed resource allocation problem, in which each process repeatedly needs simultaneous exclusive access to a set of shared resources in order to enter a special part of its code, called the *critical section*. The sharing pattern is described by an arbitrary "conflict" graph, each edge of which corresponds to a resource shared by the two processes corresponding to the endpoints of the edge.

In large scale and long-lived systems, the likelihood of some process failing at some point is high, thus sparking interest in crash fault-tolerant dining. The ideal case would be for the algorithm to isolate each crashed process such that it does not impact any other correct processes in the system. If the ideal case cannot be achieved, restricting the impact of the crash failure to a local neighborhood would still be desirable. *Failure locality* [3, 4] is a metric that realizes this concept; it is the maximum distance in the conflict graph between a crashed process and any other process that is blocked from entering its critical section.

In addition to crash failures, we take into account the presence of transient failures. Transient failures correspond to unexpected corruptions to the system

⋆ The work of Hyun Chul Chung and Jennifer L. Welch was supported in part by NSF grant 0964696.

⋆⋆ The work of Srikanth Sastry was supported in part by NSF Award Numbers CCF-0726514, CCF-0937274, and CNS-1035199, and AFOSR Award Number FA9550-08-1-0159. His work was also partially supported by Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

state; the system can be in an arbitrary state after a transient failure occurs. Algorithms tolerant of transient failures are also known as *stabilizing* algorithms.

In this paper, we consider stabilizing failure-locality-1 dining where we require that (1) eventually, no two neighbors in the conflict graph enter their corresponding critical sections simultaneously, and (2) each correct process that is trying to enter its critical section eventually does so if it is at least two hops away from any other crashed process in the conflict graph.

We consider an asynchronous shared-memory system where processes communicate through read/write operations on shared *regular* registers. Regularity states that each read operation returns the value of some overlapping write operation or of the latest preceding write operation.

Choy and Singh [4] showed that any asynchronous algorithm that solves dining must have failure locality at least 2.[3] This implies that failure-locality-1 dining cannot be solved in pure asynchrony. To circumvent this lower bound, we augment the system with failure detectors [5], system services that provide information about process crashes that need not always be correct. Specifically, we introduce the local anonymous eventually perfect failure detector $?\diamond\mathcal{P}^1$ and show that this failure detector is sufficient to solve the problem at hand.

We propose two algorithms that solve stabilizing failure-locality-1 dining. The first algorithm is inspired by the Hierarchical Resource Allocation (HRA) algorithm [6] and the second algorithm is inspired by the Asynchronous Doorway (ADW) algorithm [3]. Both algorithms utilize stabilizing mutual exclusion subroutines which can be implemented using regular registers (e.g., Dijkstra's stabilizing token circulation algorithm using regular registers [7]). By presenting two algorithms, we observe that there exists multiple methods to solve stabilizing failure-locality-1 dining. This follows the case of solving the original dining philosophers problem: the HRA, ADW, and Hygienic algorithm presented in [2], [3], and [8], respectively, constitute the three major methodologies in solving the original dining philosophers problem.[4]

Dining algorithms that consider both crash fault tolerance and stabilization are presented in [10–12]. The dining algorithms in [10, 11] achieve failure locality 2. A wait-free (failure-locality-0) dining algorithm is presented in [12] which utilizes the $\diamond\mathcal{P}$ failure detector.[5] We fill in the gap between wait-freedom and failure-locality-2 by presenting two failure-locality-1 stabilizing dining algorithms that utilize $?\diamond\mathcal{P}^1$. The $?\diamond\mathcal{P}^1$ failure detector can be implemented using $\diamond\mathcal{P}$ in asynchronous systems. This means that $?\diamond\mathcal{P}^1$ is at most as powerful as $\diamond\mathcal{P}$.

Our Contribution: We present the problem specification for stabilizing failure-locality-1 dining. This specification is the first to consider both failure

[3] Although the failure-locality-2 lower bound in [4] is proved for asynchronous message-passing systems, it also applies to asynchronous shared-memory systems.

[4] Notice that we did not include the Hygienic approach into our stabilizing failure-locality-1 dining agenda. The Hygienic-based crash fault-tolerant dining algorithms that we are aware of (e.g., [9]) use unbounded memory which is problematic for stabilizing algorithms.

[5] $\diamond\mathcal{P}$ satisfies the following: eventually, (1) every crashed process is suspected by every correct process, and (2) no correct process is suspected by any correct process.

locality 1 and stabilization. We present the first two stabilizing failure-locality-1 dining algorithms in asynchronous shared-memory systems using failure detectors along with regular registers. The proposed algorithms are modular in the sense that they utilize stabilizing mutual exclusion subroutines.

2 System Model and Problem Definition

We consider a system that contains a set Π of n (dining) processes, where each process is a state machine. Each process has a unique incorruptible ID and is known to all the processes in the system. For convenience, we assume that the IDs form the set $\{0, \ldots, n-1\}$; we refer to a process and its ID interchangeably. There is an undirected graph G with vertex set Π, called the *(dining) conflict graph*. If $\{i, j\}$ is an edge of G, then we say that i and j are *neighbors*.

The *state* of a process i is modeled with a set of local variables. Each process i has a local variable $diningState_i$ through which it communicates with the user of the dining philosophers algorithm. The user sets $diningState_i$ to "hungry" to indicate that it needs exclusive access to the set of resources for i. Sometime later, the process should set $diningState_i$ to "eating", which is observed by the user. While $diningState_i$ is "eating", the user accesses its critical section. When the user is through eating, it sets $diningState_i$ to "exiting" to tell i that it can do some cleaning up, after which i should set $diningState_i$ to "thinking". This sequence of updates to $diningState_i$ can then repeat cyclically.

Process i has another local variable $?\diamond\mathcal{P}^1_i$ through which it communicates with the failure detector $?\diamond\mathcal{P}^1$. This variable is set to true or false at appropriate times by the failure detector and is read (but never set) by process i. The behavior of $?\diamond\mathcal{P}^1$ is that after some time, $?\diamond\mathcal{P}^1_i$ is always false if i has no crashed neighbors and is always true if i has at least one crashed neighbor.

The processes have access to a set of shared single-writer single-reader (SWSR) registers that satisfy the consistency condition of regularity, through which they can communicate. Reads and writes on such registers are not instantaneous. Each operation is invoked at some time and provides a response later. *Regularity* means that each read returns the value of some overlapping write or of the latest preceding write. If there is no preceding write, then any value can be returned. When a process invokes an operation on a shared register, it blocks until receiving the response. For each process, invocations and responses occur in pairs (invocation first, response second) unless the process crashes after an invocation but before receiving a response. This implies that each operation response must be preceded by an invocation for that operation.[6]

Certain subsets of processes synchronize among themselves using mutual exclusion modules (i.e., subroutines). For any mutual exclusion module X, the participants in X are all neighbors of each other in the dining conflict graph.

[6] For each process i, invocations/responses occurring in pairs prevent i from being in a state in which, after a transient fault occurs, i is waiting for a response without having a preceding invocation to a register. This is a common assumption for stabilizing algorithms that involve read/write operations on shared registers. (e.g., [7, 13, 14])

For each mutual exclusion module X in which it participates, (dining) process i has a local variable $X.mutex_i$. Process i, at an appropriate time, sets $X.mutex_i$ to "trying" when it needs access to the corresponding critical section. Subsequently, the mutual exclusion module should set $X.mutex_i$ to "critical". When i no longer needs the critical section for X, i sets $X.mutex_i$ to "exiting", and at some later point the module X should set the variable to "remainder". This sequence of updates to $X.mutex_i$ can then repeat cyclically. Note that such stabilizing mutual exclusion algorithms exist considering asynchronous shared-memory systems with regular registers (e.g. a variation of Dijkstra's stabilizing token circulation algorithm using regular registers in [7]). This implies that, by assuming that processes have access to mutual exclusion modules, we are not assuming anything more than asynchronous shared-memory systems with regular registers.

Correctness condition: Our task is to design a distributed algorithm for the (dining) processes in Π such that every execution has a suffix in which the following four properties hold:

- Well-formedness: For all $i \in \Pi$, $diningState_i$ is set to "eating" only if the current value is "hungry", and $diningState_i$ is set to "thinking" only if the current value is "exiting".
- Finite Exiting: For each correct $i \in \Pi$, $diningState_i$ is not forever "exiting".
- Exclusion: If i and j are both correct and are neighbors, then $diningState_i$ and $diningState_j$ are not both equal to "eating" in any system state.
- FL-1 Liveness: If $i \in \Pi$ is correct and all its neighbors are correct, then if $diningState_i$ is "hungry" in some state, there is a later system state in which $diningState_i$ is "eating".

Here is an explanation for how our pseudocode maps to this model of executions. Pseudocode is presented as a set of guarded commands.[7] If a guard is continuously true, then eventually the corresponding command is executed. Each command includes at most one shared register operation. If a command includes a shared register operation, then this is actually two instantaneous steps: the first step ends with the invocation of the operation, and the second step begins with the response of the operation. If a command does not include a shared register operation, then it corresponds to a single instantaneous step.

For the complete system model and problem specification, see [15].

3 HRA-Based Stabilizing Dining

In this section, we use multiple mutual exclusion modules described in Section 2 to construct a stabilizing failure-locality-1 dining algorithm. The algorithm is inspired by the hierarchical resource allocation (HRA) algorithm from [6]. The complete algorithm description and the correctness proof can be found in [15].

[7] Guarded commands have the following format: $\{guard\} \rightarrow command$. For each guarded command of process i, the guard is a predicate on i's state and the command is a block of code; the command is executed only if the guard is true.

Algorithm 1 HRA-based stabilizing FL-1 dining algorithm; code for process i

⟨ **Variables** ⟩

1: local variable $diningState_i \in \{thinking, hungry, eating, exiting\}$;
2: local variable $?\diamond\mathcal{P}_i^1 \in \{T, F\}$;
3: $\forall R_x \in C_i$: local variable $M_x.mutex_i \in \{remainder, trying, critical, exiting\}$;

⟨ **Program Actions** ⟩

4: $\{(diningState_i = thinking) \vee (diningState_i = exiting) \vee$ Action $D.1$
$(badSuffix(\mathcal{C}_i) \vee ((diningState_i \neq eating) \wedge (?\diamond\mathcal{P}_i^1))\} \rightarrow$
5: **for all** $R_x \in C_i$ **do**
6: **if** $M_x.mutex_i = critical$ **then**
7: $M_x.mutex_i \leftarrow exiting$;
8: **if** $\neg?\diamond\mathcal{P}_i^1$ **then**
9: $diningState_i \leftarrow thinking$;

10: $\{(diningState_i = hungry) \wedge (csPrefix(\mathcal{C}_i) \neq \mathcal{C}_i) \wedge$ Action $D.2$
$(\neg?\diamond\mathcal{P}_i^1) \wedge (\neg badSuffix(\mathcal{C}_i))\} \rightarrow$
11: $R_x \leftarrow currentMutex(\mathcal{C}_i)$;
12: **if** $M_x.mutex_i = remainder$ **then**
13: $M_x.mutex_i \leftarrow trying$;

14: $\{(diningState_i = hungry) \wedge (csPrefix(\mathcal{C}_i) = \mathcal{C}_i))\} \rightarrow$ Action $D.3$
15: $diningState_i \leftarrow eating$;

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

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

The pseudocode of the actions is given in Algorithm 1. The pseudocode is self-explanatory using the definitions of the following three functions.

Three functions: For each diner i, for each $R_x \in C_i$, we introduce three functions $csPrefix$, $currentMutex$, and $badSuffix$ which are functions of a process's state and are used in specifying the guards for the three actions.

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

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

Function badSuffix. The boolean function $badSuffix(\mathcal{C}_i)$ is T if and only if there exists some clique R_x in the suffix of $\mathcal{C}_i$ following $currentMutex(\mathcal{C}_i)$ such that $(M_x.mutex_i = trying) \vee (M_x.mutex_i = critical)$ (i is either trying or in the critical section of M_x).

4 ADW-Based Stabilizing Dining

We have also designed a stabilizing failure-locality-1 dining algorithm that is inspired by the asynchronous doorway (ADW) algorithm [3]. In the original ADW algorithm, each process shares a single token called a *fork* with each of its neighbors. For a hungry process i to eat, it must first enter the *doorway* by obtaining permission from all of its neighbors through a ping-ack protocol. Only after i enters the doorway, it requests for the missing forks. Also, while i is inside the doorway, i does not give its neighbors permissions to enter the doorway. The hungry process i can start to eat if it is both inside the doorway and holds all forks shared between itself and its neighbors. After eating, i satisfies all deferred requests and exits the doorway. In our algorithm, we simulate the ping-ack protocol using two mutual exclusion modules per neighboring processes and the fork activities using one mutual exclusion module and two SWSR regular registers per neighboring processes. The complete algorithm and its correctness proof can be found in [15].

References

1. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica **1**(2) (1971) 115–138
2. Lynch, N.A.: Fast allocation of nearby resources in a distributed system. In: Proc. of 12th ACM Symposium on Theory of Computing (1980) 70–81
3. Choy, M., Singh, A.K.: Efficient fault-tolerant algorithms for distributed resource allocation. ACM TOPLAS. **17**(3) (1995) 535–559
4. Choy, M., Singh, A.K.: Localizing failures in distributed synchronization. IEEE Trans. Parall. Distrib. Syst. **7**(7) (1996) 705–716
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2) (1996) 225–267
6. Lynch, N.A.: Upper bounds for static resource allocation in a distributed system. Journal of Computer and System Sciences **23**(2) (Oct 1981) 254–278
7. Dolev, S., Herman, T.: Dijkstras self-stabilizing algorithm in unsupportive environments. In: Workshop on Self-Stabilizing Systems. (2001) 67–81
8. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM TOPLAS **6**(4) (1984) 632–646
9. Pike, S.M., Song, Y., Sastry, S.: Wait-free dining under eventual weak exclusion. In: Proc. of 9th ICDCN. (2008) 135–146
10. Nesterenko, M., Arora, A.: Tolerance to unbounded byzantine faults. In: Proc. of 21st IEEE SRDS. (2002) 22–29
11. Nesterenko, M., Arora, A.: Dining philosophers that tolerate malicious crashes. In: Proc. of 22nd IEEE ICDCS. (2002) 172–179
12. Sastry, S., Welch, J.L., Widder, J.: Wait-free stabilizing dining using regular registers. In: Proc. of 16th OPODIS. (2012) 284–299
13. Hoepman, J.H., Papatriantafilou, M., Tsigas, P.: Self-stabilization of wait-free shared memory objects. J. Parall. & Distribut. Comput. **62**(5) (2002) 818–842
14. Johnen, C., Higham, L.: Fault-tolerant implementations of regular registers by safe registers with applications to networks. In: Proc. of 10th ICDCN. (2009) 337–348
15. Chung, H.C., Sastry, S., Welch, J.L.: Stabilizing dining with failure locality 1. Dept. of Comput. Sci. & Eng., Texas A&M Univ. Technical Report 2013-10-1 (2013)