DINING PHILOSOPHERS WITH MASKING TOLERANCE TO CRASH FAULTS

A Thesis

by

VIJAYA K. IDIMADAKALA

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

MASTER OF SCIENCE

December 2006

Major Subject: Computer Science

DINING PHILOSOPHERS WITH MASKING TOLERANCE TO CRASH FAULTS

A Thesis

by

VIJAYA K. IDIMADAKALA

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

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Scott M. Pike
Committee Members,	Andreas Klappenecker
	Madhav Pappu
Head of Department,	Valerie Taylor

December 2006

Major Subject: Computer Science

ABSTRACT

Dining Philosophers with Masking Tolerance to Crash Faults. (December 2006)

Vijaya K. Idimadakala, B.Tech, National Institute of Technology, Warangal, India
Chair of Advisory Committee: Dr. Scott M. Pike

We examine the tolerance of dining philosopher algorithms subject to process crash faults in arbitrary conflict graphs. This classic problem is unsolvable in asynchronous message-passing systems subject to even a single crash fault. By contrast, dining can be solved in synchronous systems capable of implementing the perfect failure detector $\mathcal{P}$ (from the Chandra-Toueg hierarchy). We show that dining is also solvable in weaker timing models using a combination of the trusting detector $\mathcal{T}$ and the strong detector $\mathcal{S}$; Our approach extends and composes two currents of previous research. First, we define a parametric generalization of Lynch's classic algorithm for hierarchical resource allocation. Our construction converts any mutual exclusion algorithm into a valid dining algorithm. Second, we consider the fault-tolerant mutual exclusion algorithm (FTME) of Delporte-Gallet, et al., which uses $\mathcal{T}$ and the strong detector $\mathcal{S}$ to mask crash faults in any environment. We instantiate our dining construction with FTME, and prove that the resulting dining algorithm guarantees masking tolerance to crash faults. Our contribution (1) defines a new construction for transforming mutual exclusion algorithms into dining algorithms, and (2) demonstrates a better upper-bound on the fault-detection capabilities necessary to mask crash faults in dining philosophers.

ACKNOWLEDGMENTS

This thesis is the product of a series of meetings with my advisor Dr. Scott Pike. He has been my biggest motivator and has helped me solve every little problem that came along the way, eventually helping me grasp the rich complexity of this work. He has always been forthright in his comments and his criticism and has been extremely patient in correcting me and steering me in the right direction.

I would like to acknowledge each and every member of the TARGET team for sitting through endless hours of meetings for providing valuable suggestions on improving my work. Special thanks to Kaustav and Yantao who reviewed my thesis document and gave useful feedback.

I would also like to thank my parents and my sister who have been there for me through good times and bad and who have always motivated me to achieve the goals that I set for myself.

Finally, I would like to thank my friends at TAMU for making my stay here a very memorable one.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The main goal of this thesis is to isolate the impact of process crash faults in distributed systems so that they do not starve other correct processes. In a distributed system, the ability to localize the impact of faults becomes paramount as the size of the system grows. We develop a specific technique which is used to isolate the impact of crash faults in a distributed system. We consider the generalized dining philosophers problem which is a model of static resource allocation problems in distributed environments. Our goal is to weaken the requirements needed to be satisfied by the environment to achieve masking tolerance. Masking tolerance implies that the failure of any process does not effect any other processes in the system. Our algorithm helps in proving a new upper bound on the system requirements needed to mask crash faults in the generalized dining philosophers problem. We show the correctness of our algorithm by proving that the algorithm satisfies the specified safety and progress properties of dining under weak exclusion. This chapter gives a brief overview of the motivation, scope and goals of this work.

A. Localizing faults in a distributed system

Faults can occur in a distributed system in a variety of ways. Since the system is distributed, there are several machines in different locations. Although the probability of an individual process crashing is very low, if we consider a sufficiently large system, the probability of some process crashing in the system becomes very high. If these crashes could have a cascading effect in the system, it would result in several systems

The journal model is *IEEE Transactions on Automatic Control.*

being affected. A typical example of a cascading failure is the ARPANET (pre-cursor of the present-day Internet) crash on October 27, 1980. This crash was a result of bit-corruption at one local node in the network. A network-wide collapse was precipitated as a result of bit-corruption at a single node though this should not have happened.

It is important to note that the goal of this work is not to prevent the occurrence of crashes. Instead, we concentrate on the system failures that are precipitated by such crashes. Our goal is to limit the propagation of these failures in the system. In fact, the ultimate goal here is to isolate these failures so that the crash of a process does not affect any other correct processes.

B. Resource allocation problems

In a distributed environment, resources are shared by several users because of their scarce availability. A resource in a distributed system could imply either physical resources like a printer or logical resources that are acquired at run time like CPU cycles, shared variables or bandwidth.

Every process needs a subset of the existing resources in the system to perform its task (to execute). Note that this subset might also be an empty set. Each process needs to acquire all the resources in this subset before it can actually execute. We assume that a process can execute independently using private (unshared) resources, but may require access to public (shared) resources to execute a distinguished code segment, also known as the critical section. We also assume that for every process, the execution of critical section takes finite time. Once the process is done with the execution of its critical section, it gives up all the resources that it uses and hence these resources can be re-allocated to other waiting processes. An important aspect to be noted here is that no two processes can simultaneously use the same resource. Hence access to the resources is mutually exclusive. We also note that if the distributed system is fair, every process which wants to perform a task eventually gets to do so. Hence if a process requests a set of resources to enter its critical section, in a fair system, it will eventually get all these resources.

Typically, systems cycle among four states: (i) performing local actions without using shared resources, (ii) requesting shared resources and acquiring them, (iii) executing a critical section task and (iv) exiting the critical section to release the resources held. Systems which actually do so are also called *non-terminating reactive systems*.

We define and discuss three different resource allocation problems in the coming subsections: the mutual exclusion problem, the dining philosophers problem and the drinking philosophers problem. For each of these problems, we consider the following about the underlying system. The system has a set of processes, D_1, D_2, ... , D_n and a set of resources R. Every process D_i requires a subset of these resources S_i to actually execute its critical section.

$S_i = \{R_1,\ R_2,\ ....\ ,\ R_m\} \subseteq \mathrm{R}$

Every process can be in one of the following four states:

1. Thinking

2. Hungry

3. Eating and

4. Exit.

Each of these states is described below:

1. Thinking: This is a state where processes execute local code without utilizing shared resources. Processes can remain in their *thinking* state forever or move to the *hungry* state to start acquiring resources.

2. Hungry: This is a state where processes start requesting and acquiring the resources that they need to execute their critical section. From the *hungry* state, processes move to the *eating* state.

3. Eating: In this state, processes execute the *critical* section code after acquiring all the resources that they need. Processes done with eating move to the *exit* state.

4. Exit: In this state, processes execute the exit code to relinquish all their resources. They then move to the *thinking* state. Observe that the transitions from one state to another can only progress as specified in figure 1. Hence, *Thinking-Hungry-Eating-Exit-Thinking* is the only sequence of states that any correct process can follow if the execution is well-formed.

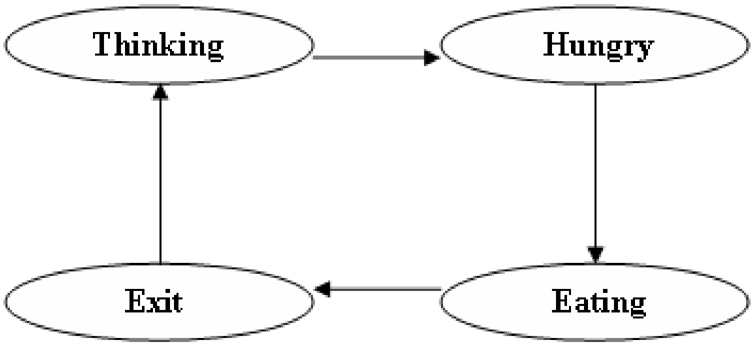


Fig. 1. Our dining model

Definition: Conflict graph

In a conflict graph, every node represents a unique process and an edge between two nodes denotes a resource conflict between the two processes which represent these nodes. A resource conflict between two processes implies that the intersection of the resource requirements of these two processes is non-empty.

Definition: Crash fault

A process is said to have crashed by faulting if it ceases execution without warning and never recovers in the infinite suffix of execution.

A *correct* process never crashes. A process is said to be *faulty* if it has or will crash in a given run. Any process that has not crashed yet is said to be *live*. Thus, every correct process is always live, and every faulty process is live only during the prefix prior to crashing.

1. The mutual exclusion problem

The mutual exclusion problem was first defined and solved by Edsger Dijkstra in 1965 [9]. In the mutual exclusion (*MX*) problem, every process requires the same subset of resources to eat. Hence two processes cannot be eating simultaneously in the system. Observe that the intersection of the resource requirements of any two processes in the system is non-empty. Hence in the conflict graph, there is an edge between every pair of processes in the system. Hence the conflict graph for a *MX* problem is a complete graph.

An algorithm is said to solve the mutual exclusion problem if it satisfies the following properties:

Safety: No two live neighbors eat simultaneously.

Progress:

(1)*Starvation freedom:* If no correct process eats forever, then every correct hungry process eventually eats.

(2) *Unobstructed exit:* Every correct exiting process eventually thinks.

Note that the progress requirement that we specify here is stronger than the standard progress requirement for mutual exclusion: No-deadlocks can occur in *MX*. Our specification also requires that lockout cannot occur in the system. Hence if a correct process becomes hungry, some time later, this process has to eat.

2. The dining philosophers problem

The dining philosophers problem was first proposed by Edsger Dijkstra in 1971 [8]. The conflict graph of this problem is a ring. Hence every process has a conflict of resources with two other processes in the system. We consider a generalized version of this dining philosophers problem where the conflict graph is any arbitrary graph. Note that at any future reference in this work, the term 'dining' refers to the generalized dining philosophers problem.

We assume the following about the *thinking* and *eating* states: Every diner can think indefinitely. Every correct diner that starts eating, eats only for a finite time. Based on these assumptions, we need our dining solution to satisfy the following properties:

Safety: No two live neighbors eat simultaneously.

Progress:

(1)*Starvation freedom:* If no correct process eats forever, then every correct hungry process eventually eats.

(2) *Unobstructed exit:* Every correct exiting process eventually thinks.

Observe that the safety and progress properties of *MX* and dining are exactly same. *MX* is only a special case of dining where the conflict graph is complete. Every *MX* solution can also solve dining. This is possible because the conflict graph of dining can be turned into a conflict graph of *MX* by trivially adding edges between every pair of nodes. The downside of using a *MX* solution to solve dining is the degradation in the concurrency. Specifically, the number of processes that can eat simultaneously reduces drastically to 1 in *MX* whereas dining allows more processes to eat concurrently. Hence as the total number of processes in the system grows, the importance of dining solutions compared to *MX* solutions increases drastically.

a. Wait chains and starvation in dining

A wait chain consists of a series of processes, each of which is waiting on a subsequent process to release a resource that it needs to eat. A typical wait chain is shown in figure 2. Here, process D_1 is waiting on D_2 to release R_2. Similarly, D_2 is waiting on D_3 to release R_3. We also have D_3, D_4, D_5 waiting on their subsequent processes to release resources. Now we consider the scenario where processes can crash. Hence suppose that D_6 crashes. Because of the wait chain, D_5 never gets to eat because it is waiting on D_6 to relinquish R_6. A correct hungry process that can never eat is said to *starve*. Hence D_5 starves. Now, D_4 is waiting on D_5 to finish eating. However, since D_5 starves, D_4 also starves. This also results in the starvation of D_3, D_2, and D_1. Hence starvation propagates throughout the wait chain and every process in it eventually starves.

Intuitively, it is easy to see why D_5 cannot eat if D_6 crashes because the intersection of their resource requirements is not empty. However, the other processes in the system, despite not having any resources in common with D_6 still get to starve. The main goal of this thesis is to present an algorithm that prevents starvation of correct processes due to the failure of other processes in the system.

3. The drinking philosophers problem

The drinking philosophers problem was proposed and solved by Chandy and Misra [6]. It is an extension of the dining problem where each process may need a different set of resources every time it becomes hungry. In drinking, although the conflict graph is still static, the resource allocation model is dynamic, in so far as each diner can dynamically determine different subsets of resources for different critical sections.

This thesis mainly deals with the mutual exclusion and dining variants of the

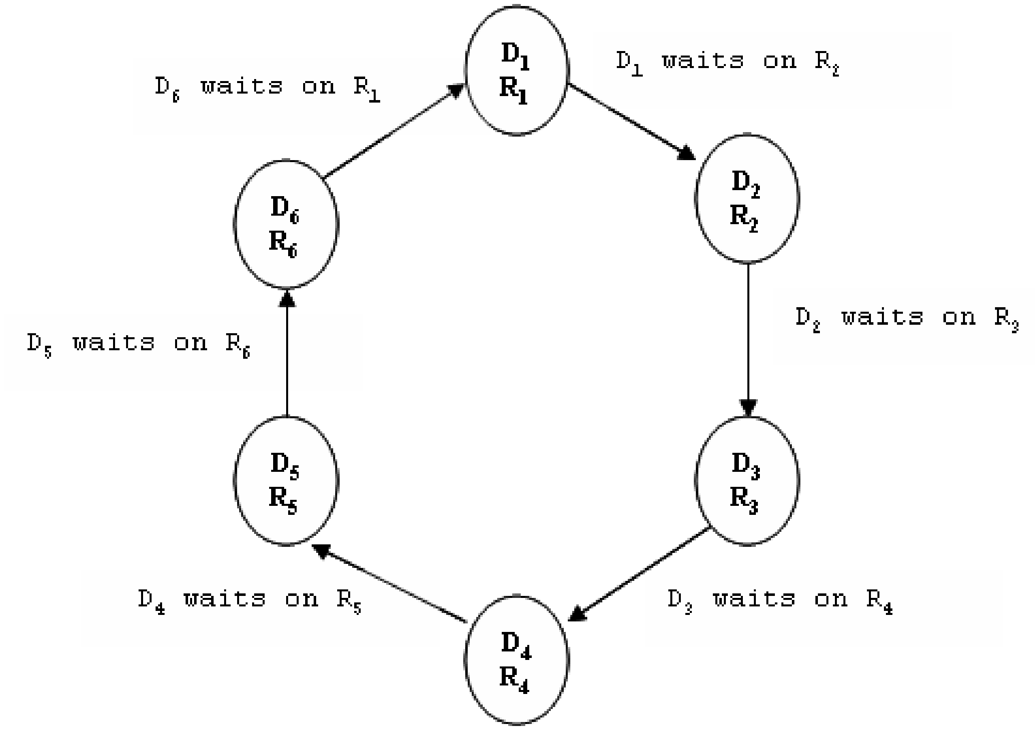


Fig. 2. A wait chain

resource allocation problems. Hence we do not study the drinking philosophers in detail here.

C. System model

In the following sub-sections, we define our system model and explain the rationale behind some of our assumptions.

1. Asynchronous message-passing system

We assume that our system is an asynchronous message-passing system. In such systems, the following properties hold:

(1)There are no bounds on end-to-end message delay.

(2)There are no bounds on relative processor speeds.

We also assume that the communication channels are reliable, in so far as messages are neither lost, duplicated, nor corrupted. More specifically, in reliable channels every message sent to a correct process will be eventually received. Non-duplication of messages implies that every message sent is only received once. Finally, every message that is received on such channels is one that was previously sent by some process. Hence there are no orphan messages.

Fault Model: Processes can fault only by crashing. A process faults by crashing if it ceases execution without warning and never recovers in the infinite suffix of execution.

2. Weak exclusion

There are different models of mutual exclusion that fit different application domains. In [20], several variants of mutual exclusion are explained. We consider two of these in this work: Strong exclusion and weak exclusion.

Under strong exclusion, no two neighbors can eat simultaneously regardless of whether one (or both) of them has crashed. Strong exclusion models systems where resources can be permanently corrupted by crash faults and, as such, cannot be meaningfully utilized thereafter. By contrast, weak exclusion requires only that no two *live* neighbors eat simultaneously. In particular, a live process can eat concurrently with any crashed neighbor. Weak exclusion models systems where resources are recoverable and can be re-allocated in the wake of crash faults.

We consider the model of weak exclusion in our dining problem.

3. Fault locality

We have seen in figure 2, how several correct processes can potentially starve because of the crash fault of a single process in the system. This is due to the wait chains that can be formed in dining. We now give the definition of a *failing* process:

Definition: Failure

A process is said to *fail* if it does not satisfy its specifications. Typically, starving processes are said to have failed because they are correct hungry processes which never get to eat thus violating their progress specification (starvation freedom).
The main motivation behind this work is to limit the number of failures in the system that are precipitated by the crash of a process. In fact, our final goal is to ensure that no process fails in the system. We now describe the metric which is used to measure the impact of a crash fault in the system. This metric is *fault locality* and it was described by Choy and Singh [2].

Fault locality is described as follows: A dining algorithm is said to have a fault locality of *k* if every process which is outside the *k-neighborhood* of a crashed process never starves i.e. all hungry processes which are at least *k+1* hops from any crashed process eventually get to eat. Improving the fault locality even by a value of 1 is significant because of the exponential factor involved. Note that if the maximum degree of every node in the conflict graph is δ, a fault locality of *k* (also denoted as FL k) would imply that the number of processes that could potentially starve is δ^k. If we reduce the fault locality to *k-1*, the number of processes that could potentially starve reduces to δ^{k-1}. As such, small improvements in fault locality translate into large improvements in the number of processes potentially affected by any given crash.

FL 0 implies that the crash of a process does not affect any other processes in the system. It is also referred to as *masking tolerance.*

D. Existing results

1. Impossibility of masking tolerance in a purely asynchronous system

The problem of solving dining with FL 0 is impossible in a purely asynchronous system. This result is implied by two stronger results. First, Choy and Singh [3] proved that the weaker tolerance property of FL 1 is unattainable in purely asynchronous message-passing systems. Later, Pike and Sivilotti [21] proved that FL 0 was also unattainable in stronger computational models of partial synchrony. The inherent difficulty in such systems is the inability to reliably distinguish a process which has crashed from processes that are merely slow. However, algorithms have been constructed in purely asynchronous systems which are able to achieve a fault locality of 2 in the presence of crash faults [2] [6] [7].

2. Fault local 0 dining: Solvable in purely synchronous systems

However, dining is solvable in purely synchronous systems with reliable communication channels. In a purely synchronous system, we assume that the following is true:
(1) Bounds on end-to-end message delay exist and are known.
(2) All processes take steps at the same rate.

Hence in a purely synchronous system, if a process crashes, all its neighbors will be able to detect for sure that it has crashed. This can be done by assuming that all correct processes continually send "I am live" messages at the end of each round of execution to all their neighbors. Hence if a process P_i does not receive any message at the end of one round of execution from another process P_j, P_i can conclude that P_j has crashed. The key idea here is that at the end of every round, in a purely synchronous system, even not receiving a message gives information. This is because every process expects to receive a message at the end of each round. Hence crashes can be reliably detected in synchronous systems. Note that since we consider the model of weak exclusion, we are assuming that the resources are recoverable in our system. Hence, once the crash of a process is reliably detected by other correct processes, it is possible for these correct processes to utilize the resources which were being used by the crashed process without violating weak exclusion. Correct hungry processes do not have to wait on crashed processes once they detect the failure of the crashed process. Hence dining can be solved with FL 0 in these systems.

Recent results by Pike and Sivilotti [21], have shown that a failure locality of 1 can be achieved in partial synchrony under certain assumptions, specifically, by using the unreliable failure detectors of Chandra and Toueg [4]. We explore the path of finding the minimum assumptions needed for achieving masking tolerance (FL 0) and show that it can be achieved in a partially synchronous system subject to crash faults using the concept of unreliable failure detectors and more specifically a novel detector called the trusting detector proposed by Delporte-Gallet, et al. [10].

E. Contributions

Our contributions in this work are two-fold:

(1) We propose a new compiler that takes as input, any mutual exclusion algorithm and provides a dining algorithm as output

(2) We demonstrate a better upper-bound on the fault-detection capabilities necessary to mask crash faults in dining philosophers.

F. Methodology

Our methodology extends and composes two currents of previous research. First, we construct a parametric generalization of Lynch's classic algorithm for hierarchical resource allocation (HRA algorithm) [17][18]. Our construction converts any mutual exclusion algorithm into a valid dining algorithm. It works as a modular compiler which takes as input any *MX* solution and outputs a valid dining solution. A construction is *modular* if it includes or uses modules which can be interchanged as units without disassembly of the module. Our compiler is modular because it uses an underlying module that is a black-box implementation of *MX*. We clearly define the interface to this module without knowing any details of how the module was originally constructed. Any correct *MX* solution can be used as the module.

Second, we use the fault-tolerant mutual exclusion algorithm (FTME) of Delporte-Gallet, et al., [10] which uses the trusting detector $\mathcal{T}$ and the strong detector $\mathcal{S}$ of the Chandra-Toueg hierarchy [4] to mask crash faults in any environment. We instantiate our dining construction with FTME, and prove that the resulting dining algorithm guarantees masking tolerance to crash faults.

CHAPTER II

RELATED WORK

Modular construction has previously been used in the construction of efficient dining algorithms. Injong Rhee [22] came up with a modular construction that takes as input any resource allocation algorithm and provides as output, a new resource allocation algorithm with better response time in a distributed message-passing system. The modular algorithm M uses a subroutine S in the following way: the critical region of S is primarily used to lock out competing processes while they schedule themselves for resource access. The actual resource allocation is not done in the critical region of S. The author establishes the fact that this modular construction bounds the response time to be a function of δc, where δ is the maximum number of conflicting processes at any time during the execution of the algorithms and c is the maximum time that a process is in its critical region. Moreover, this construction can be used to generate various resource allocation algorithms. If the input S to this construction is a dining algorithm, the output algorithm M becomes a drinking algorithm which allows more concurrency than the dining algorithm.

Comparing this solution to our generalization, we first observe that our goal is to limit the fault locality of the resulting solution, whereas the goal of Rhee's work is to improve the algorithm in terms of response time and message complexity. Also, our algorithm takes any solution to the mutual exclusion algorithm (could be any MX, dining or drinking solution) and produces a new dining algorithm as output, whereas the above solution actually results in different outputs depending on the input subroutine.

Another modular algorithm has been proposed by Lynch and Welch [26]. This algorithm solves the drinking philosophers problem given any arbitrary dining solution as input. The primary difference between this work and ours is that, our modular construction preserves the fault locality of the underlying mutual exclusion algorithm if it is FL 0. However, it has been observed that this construction by Lynch and Welch increases the fault locality by 1.

A solution has also been proposed to the dining philosophers problem by Arora and Nesterenko [16] that can tolerate malicious crashes. A *malicious crash* is one in which the faulty process takes a finite number of arbitrary steps before halting. Their solution achieves its tolerance through a combination of stabilization and crash failure locality. Their work considers the shared memory model which they claim only simplifies the presentation details. However, they claim that their results still hold when translated to the message-passing environments.

Pike and Sivilotti [21] came up with solutions to the dining philosophers problem that achieve a crash locality of 1 in distributed message-passing systems. They used unreliable failure detectors or oracles that were proposed by Chandra and Toueg [4]. Pike and Sivilotti provided transformations that took existing dining solutions, added the eventually perfect detector $\diamond\mathcal{P}$ of the Chandra-Toueg hierarchy, and achieved fault locality 1 dining solutions. They also showed that $\diamond\mathcal{P}$ is the weakest detector among the Chandra-Toueg detectors to achieve fault locality 1 dining solutions under weak exclusion. Our work is an extension to this work where we try to find the weakest detector to achieve masking fault tolerance (FL 0).

CHAPTER III

THE HIERARCHICAL RESOURCE ALLOCATION(HRA) ALGORITHM

This chapter describes the HRA algorithm by Lynch [17][18] and proves that the algorithm satisfies safety and progress. This algorithm (HRA) is so called because of the assumption of a total-ordering on all the resources in the system. All the resources are ordered according to some static, hierarchical ranking. This ordering is needed to prevent deadlocks in the system. However, this condition can be relaxed to a partial ordering on all the resources, provided that it projects to a total ordering on the individual resource requirements of each diner.

A. The algorithm

Let R denote the set of all the resources and D_1, D_2, ... , D_n denote all the diners in the system. As stated before, all the resources in R are totally ordered. Each individual resource is represented as R_i. Every resource R_i has a unique queue Q_i associated with it. Q_i contains a list of process IDs of all the diners that are currently waiting on R_i. The diner at the head of Q_i holds R_i or is in the critical section with respect to R_i. Every diner D_i requires a subset of these resources S_i to eat.
$S_i = \{R_1, R_2, ... , R_m\} \subseteq$ R

Also observe that the head of each queue is unique. The hungry protocol is given in algorithm 1 and is explained below.

Upon becoming hungry, D_i first enqueues its ID into the queue Q_1 of its least ranked resource R_1 and waits until it reaches the head of Q_1. When it reaches the head of Q_1, D_i holds the resource R_1. Now, D_i enqueues itself into the resource queue Q_2 and waits until it reaches the head of Q_2. This procedure of enqueueing into a resource queue and waiting until it reaches the head, goes on until D_i reaches the head of the resource queues of all the resources in S_i. Then D_i starts eating.

```
Algorithm 1: The Hungry Protocol
    Upon: Diner $D_i$.hungry
    for (j = 1; j $\leq$ m; j++)
        enqueue into resource queue $Q_j$;
        Wait-until at the head of $Q_j$;
    $D_i$.state := eating;
Eating Invariant: $D_i$.eating $\Rightarrow$ for all $R_j \in S_i$, <i> at the head of $Q_j$
```

The exit protocol of the HRA algorithm is given in algorithm 2 and is explained below.

```
Algorithm 2: The Exit Protocol
    Upon: Diner $D_i$.Exit
    for (j = 1; j $\leq$ m; j++)
        dequeue from resource queue $Q_j$;
    $D_i$.state := thinking;
Thinking Invariant: $D_i$.thinking $\Rightarrow$ for all $R_j \in S_i$, <i> not in $Q_j$
```

Upon finishing eating, D_i goes into the *exit* section in which it dequeues itself from all resource queues in which it was present. D_i then transits to *thinking.*

B. Proof of correctness

Lynch proves the correctness of the HRA algorithm by showing that it satisfies the safety and progress properties of dining that were specified in Chapter I.

1. Safety

Safety: No two live neighbors eat simultaneously

Proof: In any dining algorithm, safety is violated if two live processes which are neighbors are eating simultaneously. This is violation of weak exclusion which is the safety criteria that we specify in our model. Lynch proves by contradiction that HRA does not violate safety. Assume that two live neighbors D_i and D_j are eating simultaneously thereby violating safety. Since D_i and D_j are neighbors in the conflict graph, the intersection of their resource requirements is non-empty. Hence they have at least one resource that they both need to start eating. Let us assume that this resource is R_k. In the HRA algorithm, a diner can move from its *hungry* state to eating only when it is at the head of all of its resource queues. Since D_i is eating, by the above requirement and the eating invariant specified in Algorithm 1, <i> has to be at the head of the resource queue for the resource R_k. However, D_j is also eating. Hence by the eating invariant, <j> should also be at the head of the resource queue for R_k which is not possible, because the head of each such queue is unique. Hence we derive a contradiction. Thus the assumption that the live neighbors D_i and D_j violate safety is false and safety (weak exclusion) is proved.

2. Progress

Starvation freedom: If no correct process eats forever, then every correct hungry process eventually eats.

The progress requirement for Lynch's algorithm states that if any process becomes hungry at some time t_0, it will eat at some later time $t_1 > t_0$. Note that for this to be guaranteed, Lynch makes some assumptions about the behavior of the processes in their *hungry* and *exit* sections. More specifically, Lynch assumes that the eating time of every correct process is finite. Every process which starts eating stops after a finite time. Moreover, Lynch also assumes that unobstructed exit holds.

Definition: Dining with no-deadlock

A dining algorithm with no-deadlock satisfies the following progress property: If a correct process becomes hungry, then eventually some correct process eats.

Definition: Dining with no-lockout

A dining algorithm with no-lockout satisfies the starvation freedom property specified in Chapter I.

To prove progress, we need to show that:

(1) HRA prevents deadlocks from occurring in the system.

(2) HRA guarantees no-lockout.

a. No deadlocks

Deadlocks can occur if and only if all of the following necessary conditions are satisfied: [5]

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock can never occur in a system if any of the above conditions cannot hold. We show that the circular wait condition can never occur in the system thereby preventing a deadlock.

Circular wait is defined as follows: the processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list and the last process in the list is waiting for a resource held by the first process. An illustration of circular wait is shown in figure 3.

Waits-for: Process A is said to be waiting on process B if A is waiting to obtain some resource to enter its *critical* section and that resource is currently being used (or held) by B. The relationship between A and B is said to be a *waits-for* relationship.

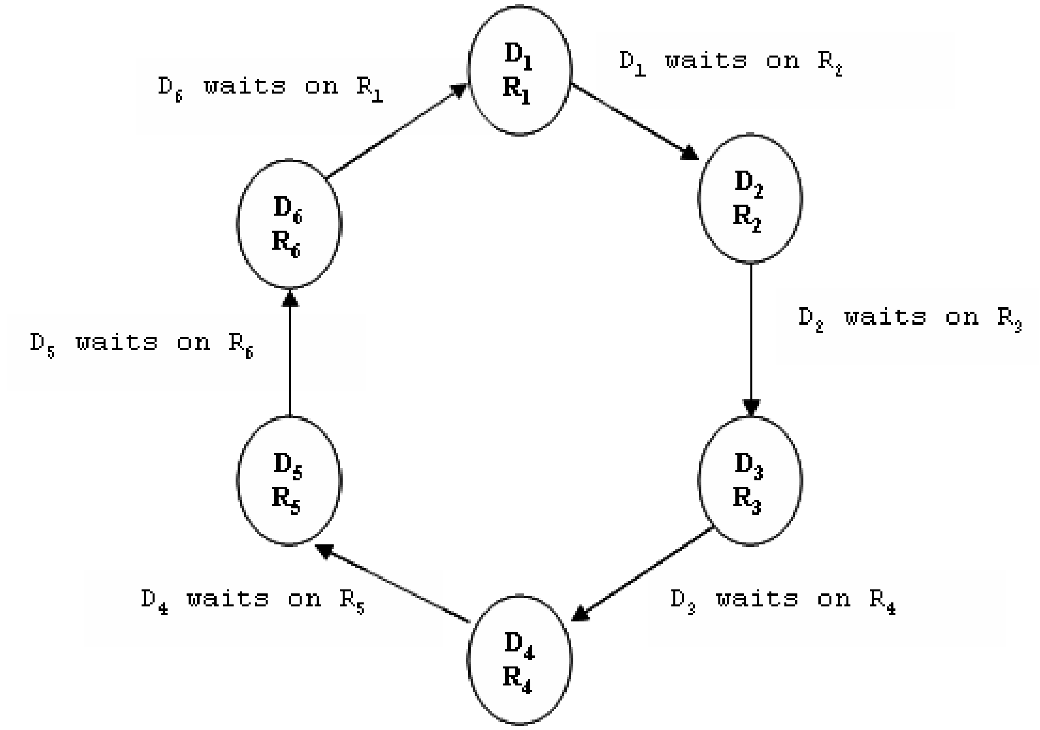


Fig. 3. A circular wait chain. D_i holds R_i and waits for R_{i+1}. D_n holds R_n and waits for R_1

Note that *waits-for* is transitive. Hence, if A *waits-for* B and B *waits-for* C, then due to transitivity, A also *waits-for* C.

Consider a set of two or more processes in the system D_1, D_2, ..., D_{k+1}. Consider the following scenario for circular wait. D_1 holds R_1 and *waits-for* R_2, D_2 holds R_2 and *waits-for* R_3, and so on. D_{k+1} holds R_{k+1} and *waits-for* R_1. We show by contradiction that such a scenario can never occur in the HRA algorithm. Because of the total-ordering on all the resources, we can conclude the following about the resources:

$R_1 < R_2$ (D_1 obtained R_1 before R_2)

$R_2 < R_3$ (D_2 obtained R_2 before R_3)

$R_3 < R_4$ (D_3 obtained R_3 before R_4)

....

$R_{k+1} < R_1$ (D_{k+1} obtained R_{k+1} before R_1)

By applying the transitivity property of the '$<$' relation on the above set of inequalities, we conclude that $R_1 < R_1$. However, this violates the irreflexivity property of the total-order. Thus we derive a contradiction thereby implying that no-deadlock can occur in the HRA algorithm.

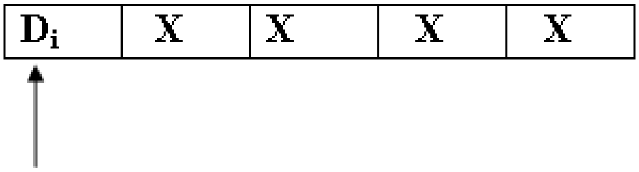
b. No lockouts

The no-lockout condition requires that any process P_i which becomes hungry at time t_0 will start eating at some time $t_1 > t_0$. To prove this, Lynch in [18], shows that there is an upper bound on the worst-case waiting time for any process. This proof is based on the observation that the queues in Lynch's algorithm are FIFO in nature. Assume that there is a metric which is the sum of the maximum queue lengths of the queues of all the resources that a diner needs to eat. From the time a process becomes hungry, this metric keeps decreasing with every advancement made by the process in any queue. Note that this metric can never increase because of the FIFO nature of the queues.

Using the FIFO nature of the queues, Lynch proposes and solves a set of recurrence relations to obtain a bound on the worst-case response time (T_u) of a hungry process. This implies that any process which becomes hungry will only have to wait for a maximum time T_u before it can start eating. Hence process P_i which becomes hungry at t_0 will start eating no later than $t_1 = t_0 + T_u$. Hence lockout cannot occur in a system with an upper bound on the waiting time and so the HRA algorithm guarantees *no-lockout*.

C. Observations

Analyzing the HRA algorithm, we observe the role performed by the queue at each resource. The queue for a particular resource helps in providing mutually exclusive access to the resource for the diner whose process *ID* is at the head of the queue. Observe that in figure 4, D_i is at the head of R_m and hence controls R_m. Moreover, the FIFO nature of the queues also guarantees that no process can overtake other processes within the same queue. Another observation here is that the FIFO nature of the queue is used in showing that no lockout can ever occur in the system, thus preventing the starvation of any hungry diner.



D_i is at the head of the queue. Hence D_i controls R_m. D_i is in its critical section wrt R_m.

Fig. 4. Resource queue for R_m where D_i is the head

If we assume that progress is guaranteed, the sole purpose of using the queue is to ensure mutually exclusive access to the resource. Hence if we have an algorithm which solves mutual exclusion ensuring progress, we can replace the queues with this algorithm and still ensure that HRA functions correctly. This is the intuition behind the generalization which is explained in the next chapter.

D. Chapter notes

1. HRA and 2-phase locking

2-phase locking (2PL) is a lock-based concurrency control technique which is used in database transactions to maintain the integrity of the database. In lock-based techniques, every data object in the database has a lock associated with it. This can be requested and held by a transaction. The transaction releases the lock after it finishes using the data object. The concept of two-phase locking has been adopted from [25].

Two-phase locking has two phases, a *growing* phase and a *shrinking* phase. In the growing phase, a transaction starts requesting locks while holding any lock it gains. When the transaction holds all the locks it needs, it is said to be at its *lock point.* After performing the necessary operations, the transaction proceeds to unlock all the locks that it holds. Observe the similarity between the HRA algorithm and the two-phase locking. A transaction is similar to a process and the data objects represent the resources that a process needs. Acquiring a lock is similar to reaching the *critical* section of a particular resource. A transaction (process) performs its operations only after it acquires all the locks (resources) it needs.

Several variations of 2PL exist. In one such technique, there is no total ordering on the data objects. Hence there is the possibility of a deadlock occurring in the system. In such systems, there is an overhead involved to detect the occurrence of deadlocks and to take corrective measures for the correct operation of the system. Other techniques assume a total ordering on the data objects and thereby prevent deadlocks. These are exactly similar to the HRA algorithm.

CHAPTER IV

GENERALIZING THE HIERARCHICAL RESOURCE ALLOCATION ALGORITHM

As we have seen in Chapter III, the HRA algorithm works by associating a queue with each resource and making every process request resources in rank order. We have also observed that the use of queues is actually not necessary in the HRA algorithm. Any mutual exclusion algorithm that ensures progress also serves the purpose.

A. Using a *MX* solution instead of queues

Instead of associating a queue with each resource, we use an underlying algorithm which provides mutually exclusive access to the resource. If we assume that the *MX* algorithm is safe, it will ensure that there can only be at most one process for each resource. Also, assume that the *MX* algorithm guarantees progress. Hence every process that becomes hungry for that resource is guaranteed to eat with respect to that resource in finite time. Hence we require a *MX* algorithm that satisfies the following properties:

Safety: No two live neighbors eat simultaneously.

Progress:

(1)*Starvation freedom:* If no correct process eats forever, then every correct hungry process eventually eats.

(2) *Unobstructed exit:* Every correct exiting process eventually thinks.

Also observe that *any* solution to *MX* that satisfies the above properties is sufficient for our construction. Our modular construction takes as input a black-box implementation of the *MX* solution and constructs a dining algorithm from it.

B. The generalized HRA algorithm

The generalized HRA algorithm works as follows: Consider a set of diners in the system, $D_1.....D_n$. We also assume that there is a set of resources R in the system. Every diner D_i requires a subset of these resources S_i to eat.

$S_i = \{R_1, R_2, ... , R_m\} \subseteq$ R

Also assume that every diner in the system cycles between the four states, <*Thinking*>, <*Hungry*>, <*Eating*> and <*Exit*> as explained in Chapter I. Hence every process has a local variable *state* to which it assigns its current state. We associate a mutual exclusion group MX_i with each resource R_i. A process which is eating in MX_i currently has privileged access to R_i. A diner D_i becomes hungry by changing its state from *thinking* to *hungry.*

For every diner, we assume that there exists a *proxy* for each resource that it needs. Hence for a diner D_i with resource requirements S_i, it contends in a total of $|S_i|$ *MX* groups. Hence we assume that there are $|S_i|$ proxies for every diner. The relationship between a diner and its proxies is specified by the following properties:

Dependent crash properties:

(1) If a diner crashes, all its proxies are assumed to have crashed.

(2) If at least one of the proxies of a diner crashes, the diner is assumed to have crashed. Consequently, by (1), all the other proxies are also assumed to have crashed.

A plausible implementation that satisfies the above property is as follows: A diner is a set of actions. A proxy is another set of actions. A process executes the actions of both the diner and its proxies. Every diner has a unique process associated with it and only processes can crash. Hence if a process crashes, both the diner and the set of proxies associated with it crash.

1. The hungry protocol

D_i becomes hungry and changes its state to *hungry* which sets off the hungry protocol. D_i sets the state of p_{i1}, the proxy for its least ranked resource R_1, to *hungry* in the mutual exclusion group MX_1. D_i waits until p_{i1} starts eating in MX_1. Then D_i sets the state of p_{i2}, the proxy for its second least ranked resource to *hungry.* This procedure of setting a proxy's state to *hungry* and waiting until the proxy starts eating, continues until the proxy of D_i's highest ranked resource p_{im} starts eating in its mutual exclusion group MX_m. At this moment, all the proxies of D_i are eating in their respective exclusion groups. Hence D_i changes its state to *eating* and starts eating. The code at each process is shown in Algorithm 3 below:

Algorithm 3: The Hungry Protocol

```
Upon: Diner $D_i$.hungry
for (j = 1; j $\leq$ m; j++)
    In $MX_j$, set $p_{ij}$.state := hungry;
    *Wait-until* $p_{ij}$.eating;
$D_i$.state := eating;
```

Eating Invariant: D_i.eating $\Rightarrow$ for all $R_j \in S_i$, p_{ij}.eating

2. The exit protocol

The diner D_i changes its state to *exit* once it is done eating. This action sets off the exit protocol. When this happens, D_i sets the state of the proxy for its least ranked resource p_{i1} to *exit.* The proxy p_{i1} changes its state to *thinking.* Once this is done, D_i then repeats the same with the proxy of the next higher ranked resource. Again this continues until p_{im}, the proxy for the highest ranked resource of D_i changes its state from *exit* to *thinking.* Then D_i changes its own state from *exit* to *thinking.* The code at each process is shown below in Algorithm 4.

Algorithm 4: The Exit Protocol

```
  Upon: Diner $D_i$.exit
  for (j = 1; j $\leq$ m; j++)
    In $MX_j$, set $p_{ij}$.state := exit;
    Wait-until $p_{ij}$.thinking;
  $D_i$.state := thinking;
```

Thinking Invariant: D_i.thinking $\Rightarrow$ for all $R_j \in S_i$, p_{ij}.thinking

C. Proof of correctness

In order to prove the correctness of the generalized HRA, we assume that we are provided with a mutual exclusion algorithm that satisfies safety and progress. The following properties are assumed of the underlying *MX* algorithm:

Safety: No two live neighbors eat simultaneously.

Progress:

(1)*Starvation freedom:* If no correct process eats forever, then every correct hungry process eventually eats.

(2) *Unobstructed exit:* Every correct exiting process eventually thinks.

There are four steps in the proof of our generalized HRA and each of it is explained below.

Step 1: We first prove the safety of the generalized HRA. To prove this we use the safety of the underlying *MX* algorithm. Note that we can use the safety property of mutual exclusion here because it holds without making any assumptions about the eating time of the processes in the exclusion groups.

Step 2: Theorem 1 - Unobstructed exit of generalized HRA

Secondly, we prove that the generalized HRA algorithm has unobstructed exit. We use the unobstructed exit of the underlying *MX* algorithm in this proof. Also note that we can use the unobstructed exit of *MX* because it also does not make any assumptions about the eating time of the processes in the exclusion group.

Step 3: Lemma 1 - No correct proxy eats forever in any exclusion group
To use the starvation freedom property of the *MX* solution, we need to provide some guarantees on the eating time of the correct proxies in any exclusion group. In other words, no correct proxy can eat forever in any exclusion group. Recall that the starvation freedom property of *MX* holds only if this assumption is satisfied. The dining algorithm uses the *MX* algorithm as a subroutine. It acts as a client of the *MX* solution. Hence it is the responsibility of the dining algorithm to ensure that the input assumptions of the *MX* algorithm are satisfied. Thus it is imperative that we prove Lemma 1 before we could use the starvation freedom property of the underlying *MX* algorithm in theorem 2.

Step 4: Theorem 2 - Starvation freedom of generalized HRA
Finally, we show that the generalized HRA algorithm guarantees starvation freedom. To prove this we use the starvation freedom property of the underlying *MX* solution. However, observe that the usage of this underlying property is valid only because we proved Lemma 1 which guarantees that the input assumptions of the underlying *MX* solution are satisfied.

1. Safety: No two live neighbors eat simultaneously

Proof: We prove by contradiction that the generalized HRA algorithm is safe. Assume that safety is violated in the generalized HRA algorithm. This means that there are at least two live neighbors in the conflict graph, D_i and D_j, which share some common resource R_k and are eating simultaneously. D_i is eating. Hence by the eating invariant specified in Algorithm 3, all the proxies of D_i are eating. Observe that the resource R_k is in S_i. Hence p_{ik}, the proxy of D_i, is eating in the mutual exclusion group MX_k. By hypothesis, D_j is also eating. Hence again by the eating invariant specified in Algorithm 3, all the proxies of D_j are eating. Observe that the resource R_k is also in S_j. Hence the proxy of D_j, p_{jk} is eating in MX_k.

From the above argument, it can be seen that the mutual exclusion group R_k has two live proxies p_{ik} and p_{jk} eating simultaneously. However, this violates the safety of the underlying *MX* algorithm which guarantees that only one live process can be eating in one exclusion group. Hence our assumption that safety is violated in the generalized HRA algorithm was wrong. Hence the algorithm guarantees safety. This completes *Step 1* of our proof structure. We now prove *Step 2* of our proof. This is the progress property – unobstructed exit.

2. Progress

We need to prove the following two progress properties:

(1) *Theorem 1:* Every correct exiting process eventually thinks.

(2) *Theorem 2:* If no correct process eats forever, then every correct hungry process eventually eats.

a. Theorem 1: Every correct exiting process eventually thinks.

Proof: We know that the underlying *MX* algorithm has an unobstructed exit section. Hence any correct process which starts exiting in the underlying *MX* algorithm eventually thinks. We continue to assume that every diner D_i has resource requirements S_i.

We define the following metric for every diner D_i and prove that each step taken by the algorithm reduces the value of the metric. We also show that the metric eventually reaches a lower bound, at which time the diner actually moves from its *exit* section to its *thinking* section.

Metric

M(D_i)= $<$ | D_{ix}.state=*Eating* |, | D_{ix}.state=*Exit* |, | D_{ix}.state=*Thinking* | $>$

where | D_{ix}.state=Q | denotes the number of proxies in the Q section

The metric is totally ordered lexicographically. We define $< x_1, x_2, x_3 >$ to be strictly lesser than $< y_1, y_2, y_3 >$ if and only if:

$x_1 < y_1$ *or*

$x_1 = y_1$ *and* $x_2 < y_2$ *or*

$x_1 = y_1$ *and* $x_2 = y_2$ *and* $x_3 < y_3$

Observe that the initial value of the metric when a diner D_i enters its *exit* section is $< k, 0, 0 >$ where k is the total number of resources that the diner needs to eat. This is because, by the eating invariant of Algorithm 3, all the proxies of D_i, p_{ix}, will be eating at this stage. We prove that the metric keeps decreasing after each iteration of the exit protocol (Algorithm 4) and finally reaches the value $< 0, 0, k >$ at which time, the diner D_i can start thinking.

Let the value of the metric after the $n-1^{th}$ iteration of the exit protocol (Algorithm 4) be,

M(D_i) = $< x_{n-1}, y_{n-1}, z_{n-1} >$

Now we need to prove that the metric decreases after the n^{th} iteration of the exit protocol. We start with the first line of the 'for' loop in the algorithm where D_i sets p_{in}.state to *exit.* The metric value changes to $< x_{n-1} - 1, y_{n-1} + 1, z_{n-1} >$. This is because one of the proxies, p_{in}, changes its state from *eating* to *exit.* Hence the number of eating proxies reduces from x_{n-1} to $x_{n-1} - 1$ and the number of proxies in their *exit* sections increases from y_{n-1} to $y_{n-1} + 1$.

D_i waits until the proxy p_{in} changes its state to *thinking* (the second line of the 'for' loop). This eventually happens in finite time because of the progress property (2) [unobstructed exit] of the underlying mutual exclusion algorithm. Hence when the iteration ends, p_{in}.state= *thinking.* Hence the new value of the metric would be, $< x_{n-1} - 1,\ y_{n-1},\ z_{n-1} + 1 >$. This is because the number of proxies in the *exit* section reduces by 1. Hence it changes from $y_{n-1} + 1$ to y_{n-1}. Also, the number of proxies in the *thinking* section increases by 1 to $z_{n-1} + 1$, since a proxy has changed its state from *exit* to *thinking.*

Hence the new value of the metric at the end of the n^{th} iteration is:

$\mathrm{M}(D_i) = < x_n,\ y_n,\ z_n >$ where $x_n = x_{n-1} - 1$, $y_n = y_{n-1}$, and $z_n = z_{n-1} + 1$

According to the definition of the metric given above, $< x_n,\ y_n,\ z_n >$ is strictly lesser than $< x_{n-1},\ y_{n-1},\ z_{n-1} >$. Hence after the n^{th} iteration of the algorithm, the metric value strictly decreases.

We have also proved that at the end of each iteration, the value of the metric changes such that $< x_n,\ y_n,\ z_n >$ becomes $< x_n - 1,\ y_n,\ z_n + 1 >$. Hence the metric starting at $< k,\ 0,\ 0 >$ after k iterations will become $< 0,\ 0,\ k >$, which implies that all the proxies of the diner are thinking. Observe from the exit protocol and the thinking invariant that this condition is sufficient for the diner D_i to move from its *exit* state to its *thinking* state.

Thus we show that there exists a metric value which decreases with each step of the exit protocol and eventually reaches a lower bound, at which point the diner moves from its *exit* to its *thinking* state. Hence it has been proved that once initiated, the exit protocol will eventually change the state of the diner to *thinking.* Thus, the unobstructed exit of dining has been proved.

Hence we have proved *Step 2* of the proof of correctness. The next step, *Step 3*, is to prove *Lemma 1.* Observe that *Lemma 1* is fundamentally, the most important proof in our thesis. It is the basis on which the rest of the proof is structured. Theorem 2 can only be proved after Lemma 1 because of the usage of the starvation freedom of *MX* in the proof of Theorem 2.

b. Lemma 1: No correct proxy eats forever in any exclusion group

Proof: The proof is by contradiction. Assume that there exists some run α of the algorithm in which some proxy of a correct diner D_i, eats forever in a *MX* group of some resource. Also assume that R_h is the highest ranked resource in which the proxy of any diner D_i eats forever in this run α. Note that there could be other processes eating forever in other exclusion groups. But R_h is the highest ranked resource in which such a bad scenario can ever occur in the run α. Hence we claim that:

Claim 1: p_{ih}.state = *eating* (forever)

There are only two possible scenarios in which the above can hold and we consider each of them separately here:

Case 1: R_h is the highest ranked resource needed by D_i

Since p_{ih} is already eating, the wait condition on Line 4 of the 'for' loop of the hungry protocol (Algorithm 3) is satisfied. Since this is the highest ranked resource needed by D_i, the control exits the 'for' loop. The next statement that is executed changes the state of D_i to *eating*. We assume that the eating time of a dining process is finite. Hence the action that starts the exit protocol will eventually be executed. Once the exit protocol starts, we know from Theorem 1 (unobstructed exit of dining) that the process eventually starts thinking in finite time thereby forcing all its proxies to also start thinking because of the thinking invariant. Hence p_{ih}.state = *thinking* eventually. However, this violates *Claim 1.* Hence we derive a contradiction to our claim that p_{ih} is *eating* forever.

Case 2: R_h is not the highest ranked resource needed by D_i

Assume that there is some next higher ranked resource R_z in which D_i is *hungry.*

Claim 2: p_{iz}.state = *hungry* or will be *hungry* eventually

Again there are only two possible ways in which the above scenario can happen:

(1) Some correct proxy eats forever in MX_z

(2) No correct proxy eats forever in MX_z

We consider each of them in two different sub-cases below:

Case 2a: Some correct proxy eats forever in MX_z

Let there be a proxy of another correct diner D_j which is eating forever in MX_z thereby preventing p_{iz} from progressing in MX_z.

p_{jz}.state = *eating* (forever).

However, R_z is ranked higher than R_h. This is because D_i requests R_z after it acquires R_h and from the hungry protocol, we know that resources are only requested in rank order from lowest to the highest ranked resource. Hence R_z is a resource which is ranked higher than R_h but has a proxy (p_{jz}) eating forever in it. However this violates our assumption that R_h is the highest ranked resource in which a proxy can eat forever. Hence we derive a contradiction to our assumption.

Case 2b: No correct proxy eats forever in MX_z

If no correct proxy eats forever in the mutual exclusion group MX_z, the progress property of *MX*, starvation freedom, is applicable since the input assumptions of *MX* are satisfied. Hence by the starvation freedom of the MX algorithm, the hungry proxy p_{iz} eventually gets to eat in MX_z. p_{iz}.state = *eating* (eventually).

Also observe that in this case, no correct proxy eats forever in MX_z because of the maximality of MX_h. In other words, R_z is ranked higher than R_h and we know that R_h is the highest ranked resource in which a correct proxy can eat forever. Hence p_{iz} has to eventually stop eating and change its state to *thinking*. However, by the exit protocol, this can only happen if the diner has started its exit code. This is because the state of a proxy can change from *eating* to *thinking* only in the exit code. Hence the exit code should have initiated. Once the exit protocol starts, we know from Theorem 1 (unobstructed exit of generalized HRA) that the process has to eventually think thereby forcing all its proxies to also think. Hence p_{ih}.state = *thinking* eventually. However, this violates *Claim 1* that p_{ih}.state is *eating* forever. Hence we derive a contradiction to our claim.

Hence from the above arguments, it can be observed that our claims are violated in all possible scenarios. Our assumption that there is a bad run α in which some correct proxy eats forever in some exclusion group must be false. We conclude that no such run exists. Hence our lemma is proved.

Thus we have finished proving *Step 3* of our proof. The final step, *Step 4*, is very similar to *Step 2* but the only important issue here is that since we have already proved *Lemma 1*, we can use the starvation freedom property of the underlying *MX* solution in this proof.

c. Theorem 2: If no correct process eats forever, then every correct hungry process eventually eats.

Proof: We assume the correctness of the underlying mutual exclusion algorithm. Note that the input assumptions of the *MX* algorithm are satisfied by lemma 1. Hence any correct process which becomes *hungry* in the underlying *MX* algorithm eats in finite time. We continue to assume that every diner D_i has resource requirements S_i.

We define the following metric for every diner D_i and prove that each step taken by the algorithm reduces the value of the metric. We also show that the metric eventually reaches a lower bound where in the diner actually moves from its *hungry* section to its *eating* section.

Metric

$\mathrm{M}(D_i) = < \mid D_{ix}.\mathrm{state}=\textit{Thinking} \mid, \mid D_{ix}.\mathrm{state}=\textit{Hungry} \mid, \mid D_{ix}.\mathrm{state}=\textit{Eating} \mid >$ where

$\mid D_{ix}$.state$=Q \mid$ denotes the number of proxies in the Q section

Observe that the metric is totally ordered lexicographically. We define $< x_1,\ x_2,\ x_3 >$ to be strictly lesser than $< y_1,\ y_2,\ y_3 >$ if and only if:

$x_1 < y_1$ *or*

$x_1 = y_1$ *and* $x_2 < y_2$ *or*

$x_1 = y_1$ *and* $x_2 = y_2$ *and* $x_3 < y_3$

Observe that the initial value of the metric when a diner D_i becomes *hungry* is $< k,\ 0,\ 0 >$ where k is the total number of resources needed by the diner. This is because, by the thinking invariant of algorithm 4, all the proxies of D_i, p_{ix}, will be thinking. We prove that the metric keeps decreasing after each iteration of the hungry protocol and finally reaches the value $< 0,\ 0,\ k >$ at which time, the diner D_i can start eating.

Let the value of the metric after the $n-1^{th}$ iteration of the 'for' loop of the hungry protocol (Algorithm 3) be,

M$(D_i) = < x_{n-1},\ y_{n-1},\ z_{n-1} >$

Now we need to prove that the metric decreases after the n^{th} iteration of the algorithm. We start with the first line of the 'for' loop in algorithm 3, where D_i sets p_{ix}.state to *hungry*. The metric value changes to $< x_{n-1} - 1,\ y_{n-1} + 1,\ z_{n-1} >$. This is because one of the proxies, p_{in}, changes its state from *thinking* to *hungry*. Hence the number of thinking proxies reduces from x_{n-1} to $x_{n-1} - 1$ and the number of hungry proxies increases from y_{n-1} to $y_{n-1} + 1$.

D_i waits until the proxy starts eating (the second line of the 'for' loop in algorithm 3). This eventually happens in finite time because of the starvation freedom of the underlying mutual exclusion algorithm. Note that we can use the starvation freedom property of the *MX* solution only because we have already proved Lemma 1. Hence when the iteration ends eventually, p_{in}.state=*eating*. Hence the new value of the

metric will be,
$< x_{n-1} - 1, y_{n-1}, z_{n-1} + 1 >$. This is because the number of proxies in the *hungry* section reduces by 1. Hence the number of *hungry* proxies changes from $y_{n-1} + 1$ to y_{n-1}. Also, the number of eating proxies increases by 1 to $z_{n-1} + 1$ since a proxy has moved from its *hungry* section to its *eating* section.
Hence the new value of the metric is:
M(D_i) = $< x_n, y_n, z_n >$ where $x_n = x_{n-1} - 1$, $y_n = y_{n-1}$, and $z_n = z_{n-1} + 1$

According to the definition of the metric given above,
$< x_n, y_n, z_n >$ is strictly lesser than $< x_{n-1}, y_{n-1}, z_{n-1} >$. Hence after the n^{th} iteration of the algorithm, the metric value strictly decreases.

We have also proved that at the end of each iteration, the value of the metric changes such that $< x_n, y_n, z_n >$ becomes $< x_n - 1, y_n, z_n + 1 >$. Hence the metric starting at $< k, 0, 0 >$ after k iterations will become $< 0, 0, k >$ which implies that all the proxies of the diner are eating. However, note that by the eating invariant of algorithm 3, at this stage, the diner D_i can start eating.

Thus we show that there exists a metric value which decreases with each iteration of the hungry protocol and eventually reaches a lower bound when the diner can eat. Hence the theorem is proved.

Hence we have proved theorem 2 which completes the final step in our proof.

D. Conclusions

So far, we have constructed a correctness preserving compiler which takes as input any correct mutual exclusion solution and outputs a correct dining solution. This is illustrated in figure 5.

Observe that the fault locality of *MX* solutions must be either 0 or 1.

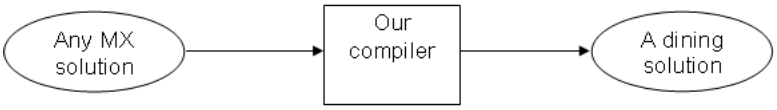


Fig. 5. Our compiler

(1) FL 0 (good tolerance): Every correct hungry process eventually eats.

(2) FL 1 (bad tolerance): Since every process is a neighbor of every other process in *MX*, a fault locality of 1 implies that if a process crashes in the system, all other processes could potentially starve. Hence FL 1 *MX* solutions are not considered in our work.

The next chapter deals with the tolerance properties of our compiler. Specifically, we study the effects of using a FL 0 *MX* solution as input. We prove that our compiler actually preserves tolerance if the tolerance of the underlying *MX* solution is good (FL 0). This implies that using a FL 0 *MX* solution as input, our compiler produces a FL 0 dining solution.

CHAPTER V

DINING WITH FAILURE LOCALITY 0

So far, we have seen how the HRA algorithm by Lynch can be generalized to use a correct *MX* solution instead of queues. We have also seen that any mutual exclusion algorithm can be converted to a dining algorithm using this generalization. We have constructed a compiler which takes as input any correct mutual exclusion algorithm and gives as output, a correct dining algorithm. We have also proved in Chapter IV that our compiler is correctness preserving. In this chapter, we evaluate the tolerance properties of our compiler. Specifically, we study the effects of using a FL 0 (failure locality 0) *MX* solution as input. We prove that our compiler actually preserves tolerance if the tolerance of the underlying *MX* solution is good (FL 0). This implies that using a FL 0 *MX* solution as input, our compiler produces a FL 0 dining solution.

A. Tolerance preservation

We now consider two different environments and analyze the tolerance properties of our compiler.

1. Environment: No process crashes occur

In this environment, by using any correct *MX* algorithm, our generalization produces a correct dining algorithm. This is because, in an environment where no process crashes, all processes are correct. Hence we only require the following properties to be guaranteed by the underlying *MX* algorithm.

Safety: No two neighbors eat simultaneously

Progress:

(1) *Starvation freedom*: If no process eats forever, then every hungry process eventu-

ally eats.

(2) *Unobstructed exit*: Every exiting process eventually thinks.

Observe that these properties are sufficient because every process in the system is a correct and a live process. Since no process crashes in this environment, tolerance properties of our compiler are not applicable here.

2. Environment: Processes crash

We assume that the input *MX* algorithm has FL 0. We now consider the tolerance of our output dining solution when a process crashes.

Recall the properties that we have already proved of the dining solution:

(1) No two *live* neighbors (diners) eat simultaneously [Safety]

(2) Every *correct* exiting process(diner) eventually thinks [Theorem 1]

(3) No *correct* proxy eats forever in any exclusion group [Lemma 1]

(4) If no *correct* process(diner) eats forever, then every *correct* hungry process(diner) eventually eats [Theorem 2]

Observe that all the four properties above only deal with correct and live diners(and their proxies). We only make assumptions about the behavior of *correct* diners and prove properties which are valid only for *correct* and *live* diners(and their proxies). We now consider the crash of a diner in every possible state and evaluate the tolerance of our dining solution:

Case 1: Thinking

Assume that a thinking diner D_i crashes. Observe that properties (1), (2) and (4) stated above only deal with correct and live diners. Since D_i is a crashed diner, it is neither correct nor live. Hence it does not violate any of these properties. Now we consider property (3) stated above. By the thinking invariant (Algorithm 4), we know that every proxy of D_i is thinking because D_i is thinking. Also, from the *dependent* *crash property* (1) specified in Chapter IV, we know that if D_i crashes, all its proxies are assumed to have crashed. From this argument, we observe that all of D_i's proxies are thinking and are assumed to have crashed. Property (3) makes assumptions about correct proxies which eat. Since none of the proxies of D_i are correct or eating, they do not violate property (3). Hence we have shown that none of the four properties above are violated if a thinking diner crashes. Hence by property (4), every correct hungry process eventually eats (irrespective of other processes crashing in the system). Hence we conclude that the dining solution has FL 0.

Case 2: Hungry

Assume that a hungry diner D_i crashes. Observe that properties (1), (2) and (4) stated above only deal with correct and live diners. Since D_i is a crashed diner, it is neither correct nor live. Hence it does not violate any of these properties. Now we consider property (3) stated above. Since the diner is executing the hungry protocol, at most one of its proxies is hungry in an exclusion group, zero or more proxies are eating in some other exclusion groups and zero or more proxies are thinking. Also, from the *dependent crash property* (1) specified in Chapter IV, we know that if D_i crashes, all its proxies are assumed to have crashed. Observe that the thinking proxies do not violate property (3) [follows from Case 1]. The one hungry proxy which crashes is neither correct nor eating. Hence it does not violate property (3). The proxies which crash while eating, now eat forever. However, they are not *correct* proxies. Hence they also do not violate property (3). We thus conclude that property (3) is not violated by any of the proxies of D_i. Hence we have shown that none of the four properties above are violated if a hungry diner crashes. Again by the same argument as in Case 1, we conclude that our dining solution has FL 0.

Case 3: Eating

Assume that an eating diner D_i crashes. Observe that properties (1), (2) and

(4) stated above only deal with correct and live diners. Since D_i is a crashed diner, it is neither correct nor live. Hence it does not violate any of these properties. Now we consider property (3) stated above. By the eating invariant (Algorithm 3), we know that every proxy of D_i is eating because D_i is eating. Also, from the *dependent crash property* (1) specified in Chapter IV, we know that if D_i crashes, all its proxies are assumed to have crashed. From this argument, we observe that all of D_i's proxies are eating and are assumed to have crashed. Property (3) makes assumptions about correct proxies which eat. Since none of the proxies of D_i are correct, they do not violate property (3). Hence we have shown that none of the four properties above are violated if an eating diner crashes. Again by the same argument as in Case 1, we conclude that our dining solution has FL 0.

Case 4: Exit

Assume that an exiting diner D_i crashes. Observe that properties (1), (2) and (4) stated above only deal with correct and live diners. Since D_i is a crashed diner, it is neither correct nor live. Hence it does not violate any of these properties. Now we consider property (3) stated above. Now, since the diner is executing the exit protocol, at most one of its proxies is in its exit state in an exclusion group, zero or more proxies are eating in some other exclusion groups and zero or more proxies are thinking. Also, from the *dependent crash property* (1) specified in Chapter IV, we know that if D_i crashes, all its proxies are assumed to have crashed. Observe that the thinking and the eating proxies do not violate property (3) [follows from Cases 1 & 2]. The one exiting proxy which crashes is neither correct nor eating. Hence it does not violate property (3). We thus conclude that property (3) is not violated by any of the proxies of D_i. Hence we have shown that none of the four properties above are violated if an exiting diner crashes. Again by the same argument as in Case 1, we conclude that our dining solution has FL 0.

Hence we have shown that the crash of a diner in any possible state does not affect other diners in the system. Hence we prove that our dining solution has FL 0 and that our compiler preserves the tolerance (produces an output FL 0 dining) if the tolerance of the input *MX* algorithm is good (FL 0).

3. A sample execution showing FL 0 dining

We now illustrate with the help of a sample execution, how our dining solution allows correct diners to eat with crashed neighbors by using the properties of the underlying *MX* solution. Consider the conflict graph shown in figure 6. There are four diners in the system, D_1, D_2, D_3, D_4. The resource requirements of each diner are shown in the figure. Every diner has a proxy in the exclusion group of every resource that it needs. We will consider a run of our dining algorithm in this configuration and show how the crash of a diner does not prevent other diners from eating. Let the diner D_1 become hungry. Since the diner is a live diner, by the hungry protocol, it sets its proxy in the exclusion group of its least ranked resource p_{11} to hungry. Proxy p_{11} eventually gets to eat in MX_1 or crashes. Assume that it eats. Once p_{11} starts eating, D_1 then sets the proxy of its next higher ranked resource p_{12} to hungry. Again p_{12} has to eat in MX_2 or crash. We assume that p_{12} eventually starts eating. D_1 then sets p_{13} to hungry. Assume that at this time, the diner D_4 becomes hungry. By the hungry protocol of our algorithm, D_4 sets the state of the proxy of its least ranked resource p_{41} to hungry. At this point, let the diner D_1 crash thereby resulting in the crash of all its proxies. The scenario is shown in figure 7 below.

Consider the situation in the exclusion group, MX_1. There is a correct hungry proxy (p_{41}) and there is a crashed eating proxy (p_{11}). However, no correct proxy is eating forever in MX_1. Hence by the starvation freedom property of *MX*, every correct hungry proxy has to eventually eat. Hence p_{41} eventually gets to eat in MX_1.

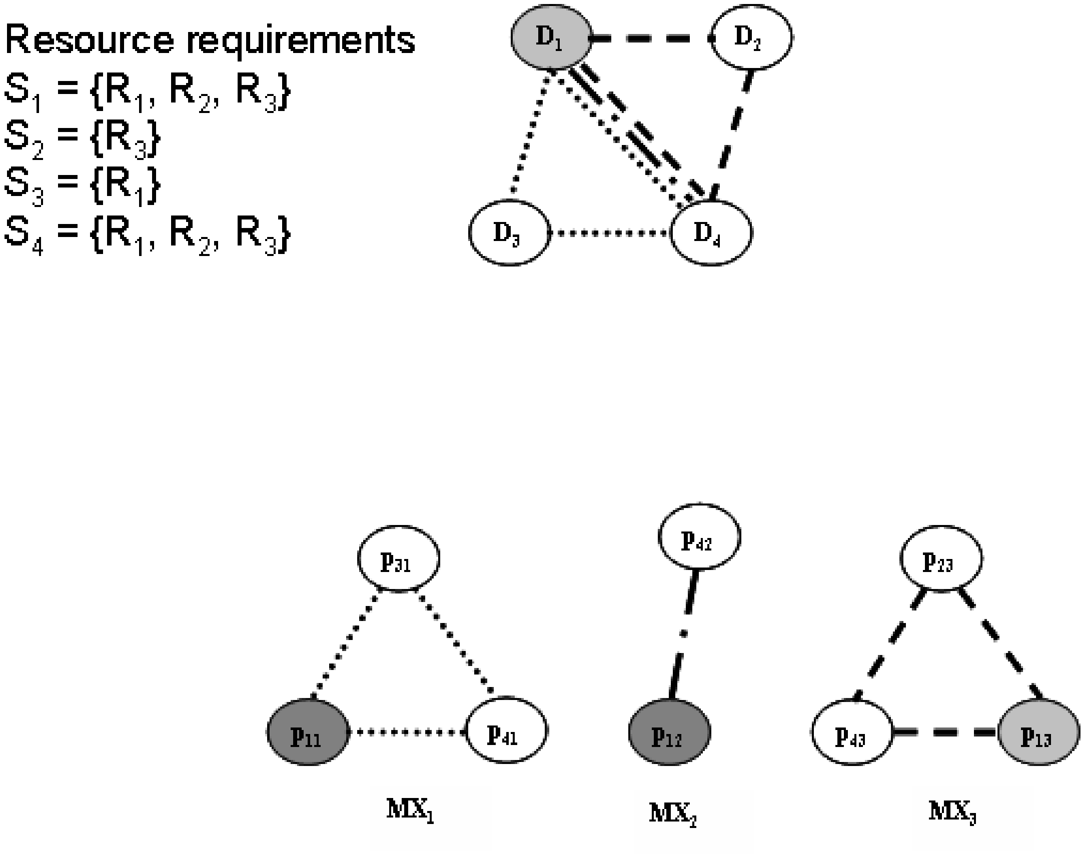


Fig. 6. A sample execution showing FL 0: Step 1

D_4 then sets the state of p_{42} to hungry which eventually starts eating by the same argument as above. Hence p_{43} is then set to hungry. It starts eating eventually at which time D_4 converts its state to eating. Hence the correct hungry diner (D_4) eats despite having a crashed neighbor (D_1) supporting the starvation freedom of our dining solution. Note that this example only illustrates that a correct diner can eat despite having a crashed neighbor who is eating. It does not specifically prove that the dining solution has FL 0.

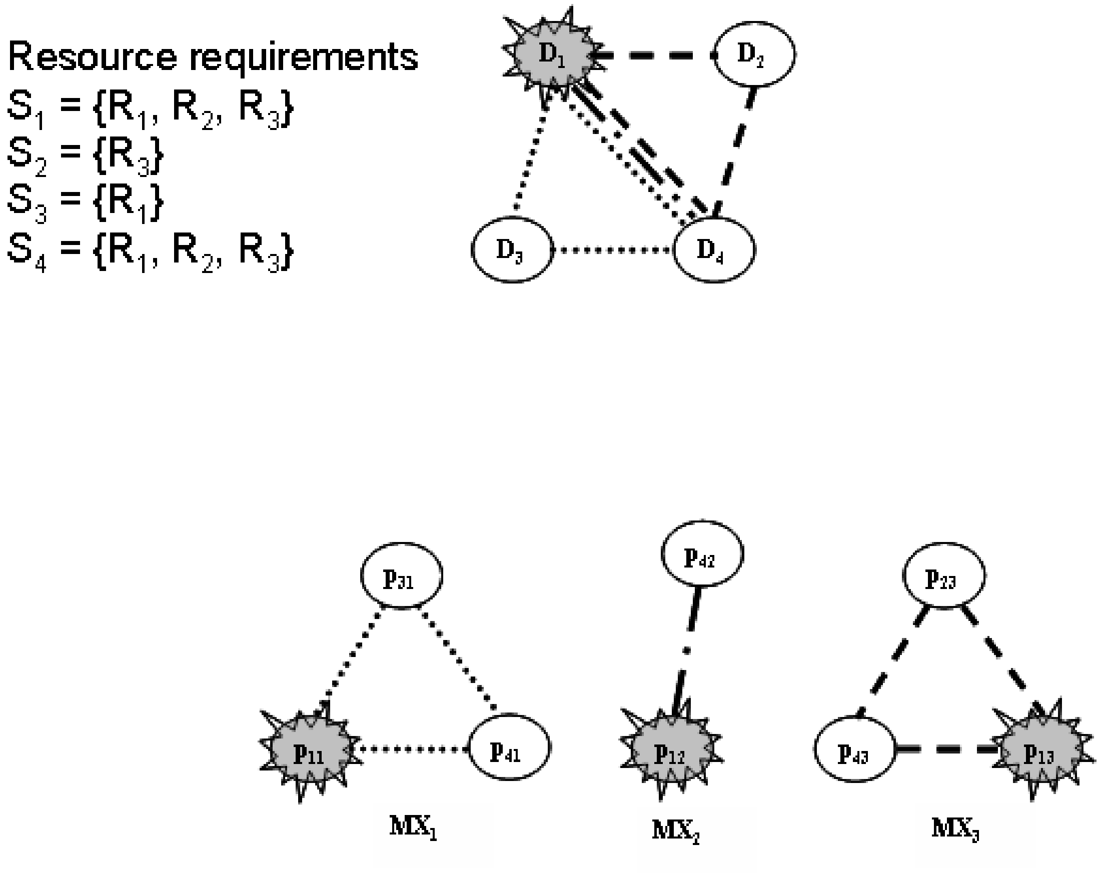


Fig. 7. A sample execution showing FL 0: Step 2

B. Conclusions

In this chapter, we have proved an additional property of our compiler: tolerance preservation for good inputs. In the next chapter, we discuss the minimum assumptions on synchrony that are required to construct a FL 0 mutual exclusion solution. The next chapter explains the construction of a specific FL 0 mutual exclusion solution, fault-tolerant mutual exclusion (FTME) which was solved by Delporte-Gallet, et al., [10] using the concept of unreliable failure detectors in an environment that does not require full synchrony. Observe that the next chapter only serves the purpose of supporting our assumption that there exists a FL 0 *MX* solution which can be used as input to our compiler.

CHAPTER VI

FAULT-TOLERANT MUTUAL EXCLUSION

This chapter explores FL 0 mutual exclusion solutions. We first show that FL 0 can be trivially achieved in full synchrony. As explained in Chapter I, crash faults can be reliably detected in such systems. Since resources are recoverable (weak exclusion), correct processes can seize resources from processes known to have faulted (crashed). Hence correct processes never wait indefinitely on crashed processes. This in turn will result in FL 0 *MX* solutions. However, we now explore the minimum assumptions of synchrony to achieve FL 0 *MX*. Full synchrony, though sufficient, is too strong a requirement on the underlying environment. Delporte-Gallet, et al.,[10] proved that full synchrony is not necessary. They solved FTME in an asynchronous system augmented with unreliable failure detectors. The rest of the chapter introduces the concept of unreliable failure detectors and explains the FTME solution of Delporte-Gallet, et al. [10].

A. Unreliable failure detectors

The asynchronous model of computation does not make any timing assumptions on the system. This becomes important because applications based on this model are highly portable compared to those which incorporate specific timing assumptions. Moreover, timing assumptions are usually only probabilistic. Hence they can rarely be assumed to be totally accurate. However, with no timing assumptions, there is an inherent difficulty in distinguishing a crashed process from one that is very slow.

Chandra and Toueg first introduced the concept of unreliable failure detectors[4]. These failure detectors are distributed in nature. Each process has a local failure detector module which it can query to get information regarding the failures of other processes in the system. The purpose of the failure detector module is to monitor the rest of the processes in the environment and add a process to the suspect list if it suspects the process to have crashed. A failure detector can be unreliable. Hence it can make mistakes. These mistakes are of two types: false negatives and false positives. A *false negative* is an instance where a process which was actually crashed was not suspected. A *false positive* is an instance where a correct process is incorrectly suspected by the failure detector.

Chandra and Toueg classified the failure detectors on the basis of two properties: Completeness and Accuracy[4]. The completeness property restricts the false negatives that can be made. Specifically, it deals with ensuring that in the infinite suffix of any run of the algorithm, all the incorrect processes in the system are suspected by the failure detectors. The accuracy property restricts the false positive mistakes that can be made by the failure detector, *i.e.*, it deals with the correct processes incorrectly suspected by the detector and tries to restrict these mistakes. These are defined more formally in the next section.

1. Completeness

Chandra and Toueg classified completeness properties into strong and weak categories as described below:

Strong completeness: Eventually every process that crashes is permanently suspected by every correct process.

Weak completeness: Eventually every process that crashes is permanently suspected by some correct process.

2. Accuracy

Chandra and Toueg defined four accuracy properties but for our purposes we define and discuss only three of these.

Perpetual strong accuracy: No process is suspected before it crashes by any live process. This is the strongest requirement for accuracy. There is also an eventual version of accuracy called the eventual strong accuracy defined below.

Eventual strong accuracy: For every run, there exists a time after which correct processes are not suspected by any other live process.

Weak accuracy: Some correct process is never suspected by any live process.

Note that a failure detector can trivially satisfy the property of completeness by initially and permanently suspecting all processes in the system. Also, a failure detector can also trivially satisfy accuracy by not suspecting any process ever. Hence completeness and accuracy are not very useful only by themselves. However, together they accurately describe a useful failure detector.

3. Failure detector classes

Failure detectors have been classified on the basis of the completeness and accuracy properties that they satisfy. The three detectors of significance to this work and their properties are described below.

The perfect detector ($\mathcal{P}$): This detector satisfies the two properties, strong completeness and strong accuracy.

The eventually perfect detector($\diamondsuit\mathcal{P}$): This detector satisfies the two properties, strong completeness and eventual strong accuracy.

The strong detector($\mathcal{S}$): This detector satisfies the two properties, strong completeness and weak accuracy.

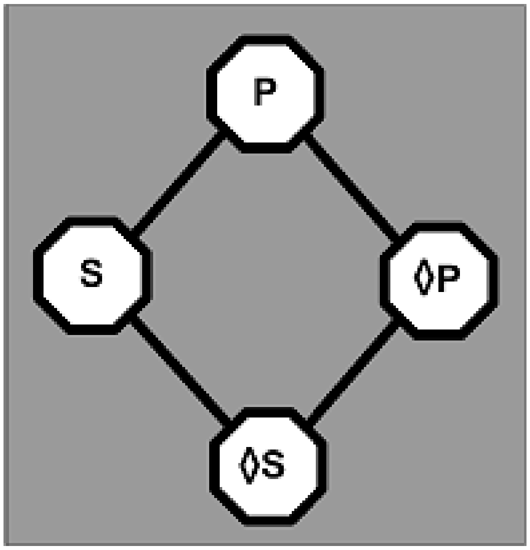


Fig. 8. The Chandra-Toueg hierarchy of failure detector classes

The Chandra-Toueg hierarcy of failure detectors is shown in the figure 8. Note that detectors higher in the hierarchy are more powerful. Hence $\mathcal{P}$ is the most powerful detector. $\mathcal{S}$ and $\Diamond\mathcal{P}$ are incomparable. This is because, $\mathcal{S}$ makes an unbounded number of mistakes over time. It can repeatedly suspect a correct process any number of times. However, $\Diamond\mathcal{P}$ makes an unbounded number of mistakes over space. Initially, it can suspect all the correct processes in the system. But after it converges, it does not make any mistakes. $\Diamond\mathcal{S}$ is the weakest detector among these four but we do not consider it in this work.

a. Optimal localities

In the following discussion, we assume that the underlying environment is purely asynchronous and every process in the system has access to a local module of the specified detector.

The perfect detector,$\mathcal{P}$, will detect every process crash accurately in finite time due to strong completeness and strong accuracy. As we have seen in Chapter I, accurate detection of every process crash is sufficient to solve dining with FL 0 with weak exclusion. Hence $\mathcal{P}$ is sufficient to achieve FL 0 dining. This work shows that $\mathcal{P}$ is not necessary for FL 0 dining.

The eventually perfect detector, $\diamondsuit\mathcal{P}$, has been known to achieve FL 1 dining (result from Pike and Sivilotti[21]). Pike and Sivilotti[21] proved that $\diamondsuit\mathcal{P}$ is necessary but not sufficient for FL 0 dining. Hence we need a stronger detector than $\diamondsuit\mathcal{P}$ to achieve FL 0.

The strong detector, $\mathcal{S}$, is known to solve dining with FL 2. Pike and Sivilotti[21] prove that 2 is the optimal locality that can be achieved with $\mathcal{S}$. It is interesting to note that adding $\mathcal{S}$ to a purely asynchronous environment does not improve the failure locality. We have already seen in Chapter I that it is possible to achieve FL 2 in an asynchronous environment even without failure detectors.

From these optimal localities, it becomes obvious that, to solve dining with FL 0, we need a detector that is stronger than $\diamondsuit\mathcal{P}$ but weaker than $\mathcal{P}$. We now introduce the trusting detector.

4. The trusting detector

A new failure detector that does not belong to the class of the Chandra–Toueg hierarchy has been defined by Delporte-Gallet, et al.[10]. They define *trust* as the negation of suspicion. Hence if a process is suspected by a failure detector, it implies that the detector does not trust that process. Delporte-Gallet, et al., introduced the detector called the *trusting detector*, $\mathcal{T}$, and it is characterized by the following properties: Strong completeness, eventual strong accuracy and trusting accuracy. Trusting accuracy is defined below:

Trusting accuracy:

Every process P_j that stops being trusted by a process P_i is crashed.

Essentially $\mathcal{T}$ could also be defined as follows: $\mathcal{T} = \Diamond\mathcal{P}$ + Trusting accuracy

The idea behind $\mathcal{T}$ is that $\mathcal{T}$ stops trusting some process if and only if that process is known to have crashed in the system. This is illustrated in figure 9.

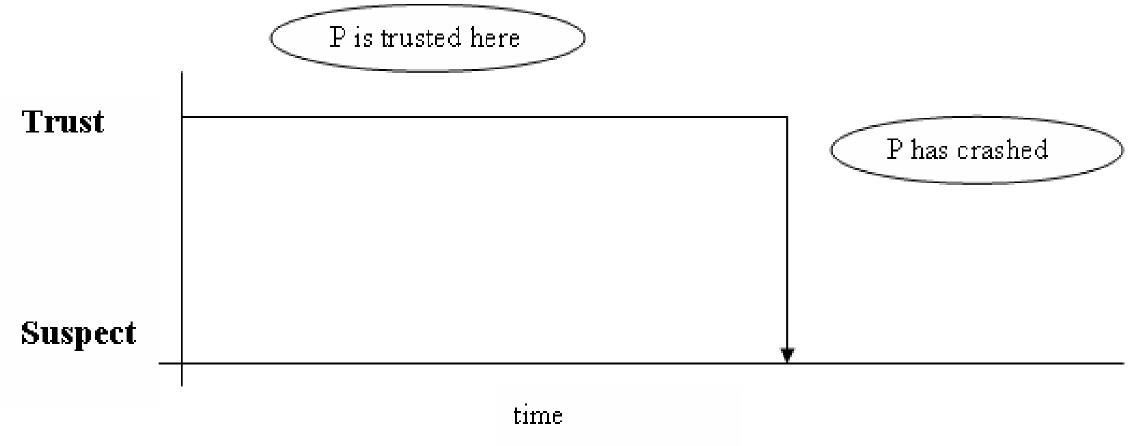


Fig. 9. Output of the Trusting detector with respect to some crashed process P. Note that if there is a down-edge in the output of T for some process, it implies that the process has crashed.

a. $\mathcal{T}$ vs. $\Diamond\mathcal{P}$

From the definition of $\mathcal{T}$, it can be seen that $\mathcal{T}$ has an additional property compared to $\Diamond\mathcal{P}$, the trusting accuracy. With $\Diamond\mathcal{P}$, suspicion can never be considered as knowledge of a process crash. However, with $\mathcal{T}$ suspicion can imply knowledge of a crash if the process was previously trusted. Also, $\mathcal{T}$ can only make a bounded number of mistakes because it can only trust a faulty process once. However, $\Diamond\mathcal{P}$, can make an unbounded number of mistakes before it converges. Hence $\mathcal{T}$ is strictly stronger than $\Diamond\mathcal{P}$. A formal proof is presented in [10].

b. $\mathcal{T}$ vs. $\mathcal{P}$

Though $\mathcal{T}$ is strictly stronger than $\Diamond\mathcal{P}$, it can still make mistakes. Specifically, it can initially suspect correct processes. However, $\mathcal{P}$ cannot make such mistakes because of its strong accuracy. Hence $\mathcal{P}$ is strictly stronger than $\mathcal{T}$. A formal proof is presented in [10].

B. Fault-tolerant mutual exclusion(FTME)

FTME denotes a mutual exclusion solution with failure locality 0 (or masking tolerance). Delporte-Gallet, et. al., [10] solved the *MX* problem and achieved FL 0 in an asynchronous environment augmented with the failure detector $\mathcal{T} + \mathcal{S}$. Their solution satisfies the safety and progress properties of *MX* specified in Chapter I. Their solution and its relevance to our work is explained in the next subsection.

Observe that for a *MX* solution, FL 1 implies that if a process crashes in the system, every other process can potentially starve. This is because all the processes are in the *1-neighborhood* of the crashed process and the conflict graph in *MX* is complete.

1. FTME and weak exclusion

In the FTME solution by Delporte-Gallet, et al.[10], the authors assume that if a process crashes in its critical section, it automatically exits its critical section. This is because, their safety property of strong exclusion does not allow processes to eat simultaneously even with crashed neighbors. However, we observe that this is equivalent to assuming a weaker model of exclusion (weak exclusion) without making any assumptions about the crashed processes. Specifically, we allow correct processes to eat with crashed neighbors. Hence we contend that the underlying exclusion model of FTME is equivalent to weak exclusion.

2. Overview of the FTME solution

The FTME solution assumes that every process is provided with two primitives which together implement the total order broadcast. These are *to-broadcast()* and *to-deliver()*. The output of a trusting detector is available at each process in the system. The concept of total order broadcast has been explained by [10] and it satisfies the following properties:

(1) validity: Any message m that is *to-broadcasted* by some correct process i, is eventually *to-delivered* by i

(2) agreement: If some correct process *to-delivers* a message m, then every correct process eventually *to-delivers m.*

(3) integrity: If a message was previously *to-broadcast*, then it is *to-delivered* at most once.

(4) total-order: If the order of message delivery at some process is m followed by *m'*, it has to be the same at all other processes.

Note that the assumption of total-order broadcast is not too strong because we assume that there exists $\mathcal{T}$ in the system and it has been proved by Chandra and Toueg[4] that total order broadcast can be implemented in any system which has at least $\Diamond\mathcal{P}$. Since we have already shown that $\mathcal{T}$ is stronger than $\Diamond\mathcal{P}$, the assumption of total order broadcast is justified.

The key idea in the solution is that upon becoming hungry, a live process has to become trusted by some correct process before it can start competing for the resource. To implement this, the authors assume the presence of the strong detector in addition to the trusting detector. By the properties of the strong detector (specifically, weak accuracy), there should be at least one correct process in the system which is never suspected by any other live process. Upon becoming hungry, a process P_i sends trust requests to all processes in the system and waits until it receives acknowledgements from all the processes not being suspected currently by the strong detector at P_i. Note that by the strong completeness of the strong detector, every faulty process is eventually suspected by the strong detector at P_i. All other correct processes will eventually trust P_i because of the eventual strong accuracy property of the trusting detector and hence will respond to the trust request. Now, once P_i gets trusted by all processes not in the output of its own strong detector, P_i is bound to be trusted by the one correct process P_j that is never suspected by the strong detector.

P_i then draws a ticket which is a tuple (i, r_i) where i is the process number and r_i is the number of times that i has run the hungry protocol. P_i then to-broadcasts this ticket number to all the processes in the system. Processes are served in the order of their ticket numbers. Note that ticket numbers are unique because process numbers are included in the ticket.

The advantage of using *total order broadcast* is that candidates can be served only in the order in which they have requested the critical section. This is because all tickets are delivered in exactly the same order at every process. Every ticket at a process P_i is processed sequentially, in the order of its receipt. Upon processing a ticket, if P_i receives its own ticket, it enters its critical section. However, if the ticket belongs to some other process P_k, P_i waits until it receives a message that P_k has exited its critical section or that P_k has crashed. After finishing eating, P_i sends a message to all other processes that it has exited its critical section.

We now need to consider the scenario where some process crashes in its critical section. Observe that there is at least one correct process, P_j, that is never suspected by $\mathcal{S}$ at P_i due to the weak accuracy property of $\mathcal{S}$. Also P_i has to be trusted by all processes that its strong detector does not suspect. Consequently, P_i is trusted by P_j before it actually eats. Hence if P_i crashes thereafter, P_j will stop trusting P_i eventually because of the strong completeness property of $\mathcal{T}$ at P_j. However, since P_i was previously trusted by P_j, by trusting accuracy, P_j detects the crash of P_i accurately using the down edge shown in figure 10. This crash can then be broadcasted to the entire network. Upon receiving this crash message, other waiting processes make progress as explained in the previous paragraph. The authors also prove the correctness (safety and progress) of this solution.

3. Necessity of the trusting detector

The authors also prove that no detector weaker than $\mathcal{T}$ can solve the FTME problem for every environment where there are no restrictions on the number of faults that can occur in the system. They do this by showing that if some algorithm A solves FTME using a detector D, a reduction algorithm can be given which converts D to $\mathcal{T}$. This shows that D is at least as strong as $\mathcal{T}$. Hence $\mathcal{T}$ is necessary to solve FTME.

a. The reduction algorithm(sketch)

The authors assume that there are n concurrent instances of FTME algorithm where n is the total number of processes in the system. Hence there is one critical section for each process in the system. Every correct process gains access to its critical section and eats forever in its critical section. If a process crashes in its critical section, some other correct process then gets to eat in it.

Input: We are given some algorithm A that solves the MX problem using some detector D.

Output: We need to provide a valid history of $\mathcal{T}$ in terms of the suspected processes as the output.

Procedure: Initially output the set of all the processes in the system. This is a valid output for the trusting detector since it can suspect every process initially. Now every process becomes hungry to gain access to its unique critical section. Hence for each process i, there is a mutual exclusion group MX_i in which it is the only process that is hungry. Now, because of the correctness of the algorithm A (specifically, the starvation freedom property), the process i has to eventually get to eat in MX_i or crash.

If it crashes, since $\mathcal{T}$ suspects the process anyway, our output is valid for $\mathcal{T}$. If i is a correct process, it eventually gets to eat in MX_i. It then broadcasts a message to every process saying that it is in its critical section in MX_i. Once this happens, i is removed from the suspect list of all processes which receive this message from i. Every process j that receives this message from i also becomes hungry in MX_i. If i crashes in MX_i some time later, one of the other hungry processes, j, gets to eat in MX_i because of the starvation freedom property of A. This is because j is a correct hungry process in MX_i and no correct process is eating forever in MX_i. Once j gets to eat in MX_i, it broadcasts the crash of i to every process in the system. Upon receipt of this message, every process puts i back on the suspect list and it remains there forever because it has crashed. Hence given any algorithm that solves FTME, this construction gives a valid output for the trusting detector. Hence it can be seen that the trusting detector is necessary to solve this problem.

C. Observations and conclusions

Delporte-Gallet, et al.,[10] solved the MX problem and gave FL 0 solutions in two different environments.

1. An environment with a majority of correct processes and the trusting detector $\mathcal{T}$.

2. An environment with no restrictions on the number of crashes along with the detectors $\mathcal{T}$ and $\mathcal{S}$. Note that there should be at least one correct process in the system in this scenario to implement $\mathcal{S}$.

We now observe the implications of these assumptions in our dining topology. Consider the first solution with $\mathcal{T}$ and a majority of correct processes in each underlying exclusion group. Assume that our conflict graph is a ring. Hence every process shares a resource with its left and right neighbors. Every resource is shared by two processes. Each underlying exclusion group has two processes. If we require a majority of them to be correct, we are assuming that both processes in every exclusion group are correct. This reflects to an assumption that every process in the system is correct. This is not very reasonable because, though we now have a FL 0 dining solution, we are assuming that no process ever crashes. Hence our solution is irrelevant.

Hence we consider the second solution in our work. This means that in our dining topology, we have to assume that there is at least one correct process in every underlying exclusion group. This translates to an assumption that each resource in the system has at least one correct process that would require this resource to eat. This is a reasonable assumption and hence we consider the second solution in our construction.

We have already proved in Chapter V that our compiler is tolerance preserving. Hence by using the second solution of Delporte-Gallet, et al., which solves FTME, we now conclude that we have a FL 0 dining solution with the assumption that every resource in the system has at least one correct process that would require this resource to eat.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

A. Conclusions

The contribution of this work is two-fold. Firstly, we have constructed a compiler that gives as output, a solution to the generalized dining philosophers problem by using any underlying solution to the mutual exclusion problem as input. Using our compiler, every possible *MX* algorithm that has been constructed so far can be converted into a dining algorithm. Hence, we have managed to construct a host of new dining solutions. Moreover our compiler preserves the fault locality of dining if the locality of the *MX* algorithm is 0.

We then used the FTME solution of Delporte-Gallet, et al., as input to our compiler to achieve a FL 0 dining solution in a partially synchronous environment. To our knowledge, this is one of the first dining solutions to mask crash faults in any environment which is weaker than full synchrony. Weakening the requirements on the environment by not having to make any specific timing assumptions makes our solution more portable.

B. Future work

Observe that every dining algorithm also solves mutual exclusion. This is because *MX* is only a special case of dining. Hence we could potentially give another dining algorithm as input to our compiler. The consequences of this action could be studied as future work. A potential goal would be to extend the generalized HRA algorithm and use the oracle $\diamondsuit$P such that any dining algorithm supplied as input will be transformed into a dining algorithm with failure locality 1. Note however, that the input assumptions will need to be strengthened in this scenario. We need to ensure that the input dining algorithm is *abortable*. Abortable dining implies that the diner should be able to abort while in any state to get back to its thinking state. We also need to demonstrate that abortable dining algorithms actually exist by showing how some classic dining solutions can be converted into abortable variants.

Another different extension to this work could be in finding the weakest detector to solve dining with FL 0 in an environment where there are no restrictions on the number of processes that can fault at any time. We have seen that $\mathcal{T} + \mathcal{S}$ can solve dining with FL 0 in any environment. Hence it is already known that $\mathcal{T} + \mathcal{S}$ is sufficient. However, the necessity of $\mathcal{T} + S$ is still an open question and can be considered future work.

Also observe that we consider the performance of our algorithm only with respect to the tolerance metric. Analyzing the message complexity and the response time of our solution have also been left as future work. Observe that there are actually several variants of the *MX* solution. There have been *fast MX* solutions [14][24][27] which guarantee a response time that is proportional to the number of hungry processes in the system. Using these *MX* algorithms as input, the performance of our compiler and the output dining algorithm in terms of response time can be studied as future work. There are also bounded-overtaking *MX* algorithms[1][23]. Bounded-overtaking implies that once a process becomes hungry, there is a bound on the number of times it can be overtaken by other processes in the system before it can get to eat. It ensures that the system is fair to all competing processes. The performance of our compiler when given a bounded-overtaking *MX* solution as input could also be considered as future work. Specifically, it would be interesting to see if an input bounded-overtaking *MX* solution would result in a dining algorithm which also guarantees bounded-overtaking with a larger bound.

REFERENCES

[1] A. K. Alagarsamy, "A mutual exclusion algorithm with optimally bounded bypasses," *Information Processing Letters*, vol. 96, no. 1, pp. 36–40, 2005.

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

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

[4] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM(JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

[5] E. G. Coffman, M. J. Elphick and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971.

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

[7] K. Mani Chandy and J. Misra, *Parallel Program Design: A Foundation*. Boston, MA: Addison-Wesley, 1988.

[8] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

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

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

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

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

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

[14] L. Lamport, "A fast mutual exclusion algorithm," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 1–11, 1987.

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

[16] M. Nesterenko and A. Arora, "Dining philosophers that tolerate malicious crashes," *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pp. 172–179, 2002.

[17] N. A. Lynch, *Distributed Algorithms*. San Fransisco, CA: Morgan Kaufman, 1996.

[18] N. A. Lynch, “Fast allocation of nearby resources in a distributed system,” *Proceedings of the 12th ACM Symposium on Theory of Computing*, pp. 70–81, 1980.

[19] N. A. Lynch, “Upper bounds for static resource allocation in a distributed system,” *Journal of Computer and System Sciences(JCSS)*, vol. 23, no. 2, pp. 254–278, 1981.

[20] S. M. Pike, “Distributed resource allocation with scalable crash containment,” Ph.D. dissertation at the Ohio State University, Department of Computer Science & Engineering, 2004.

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

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

[23] M. O. Rabin, “N-Process mutual exclusion with bounded waiting by 4 log 2 N - valued shared variable,” *Journal of Computer and System Sciences(JCSS)*, vol. 25, no. 1, pp. 66–75, 1982.

[24] E. Styer, “Improving fast mutual exclusion,” *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing(PODC)*, pp. 159–168, 1992.

[25] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems.* New York, NY: McGraw-Hill, 1994.

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

[27] J. Yang and J. H. Anderson, “A fast, scalable mutual exclusion algorithm,” *Distributed Computing*, vol. 9, no. 1, pp. 51–60, 1995.

APPENDIX A

HAPPENS-BEFORE

In this appendix, we provide an alternate proof of progress for Lynch's original HRA algorithm. Specifically, we prove that in the HRA algorithm, no deadlocks can occur between any pair of processes in the system. Note that this is only a part of the proof and does not by itself hold as proof of progress for the HRA algorithm. We do not prove the no-lockout property of the HRA algorithm. We only prove that deadlocks cannot occur between any two processes in the system. The impossibility of multiple processes deadlocking is not proved here.

We use the concept of a *happens-before* relationship in this proof.

Using *happens-before*

The '*happens-before*' relationship has been defined by Leslie Lamport[17]. It is described below:

For two events A and B, we say that A happens before B or $A \rightarrow B$ if and only if either of the following conditions is satisfied:

1. If A and B occur at the same process and A occurs before B at that process, then $A \rightarrow B$ is true

2. If A is an event where some message is sent at a process and B is an event at a different process where that message sent at A is received, then $A \rightarrow B$ is true.

happens-before satisfies transitivity i.e. if there are three events A, B and C such that $A \rightarrow B$ and $B \rightarrow$ C, then $A \rightarrow$ C.

happens-before defines an irreflexive partial order, *i.e.*, if neither $A \rightarrow B$ nor $B \rightarrow A$, then A and B are said to be concurrent events.

To prove: Circular wait cannot occur with the HRA algorithm for distributed resource allocation.

We show by contradiction that there cannot be a circular wait in Lynch's HRA. Consider two diners D_i and D_j and two resources R_m and R_n. Assume that circular wait exists between the two processes. Without loss of generality, assume that D_i is ahead of D_j in the resource queue of the resource R_m. Similarly, D_j is ahead of D_i in the resource queue of the resource R_n.

Consider the following events:

E_{im} - Enqueue of D_i into the resource queue for resource R_m

E_{in} - Enqueue of D_i into the resource queue for resource R_n

E_{jm} - Enqueue of D_j into the resource queue for resource R_m

E_{jn} - Enqueue of D_j into the resource queue for resource R_n

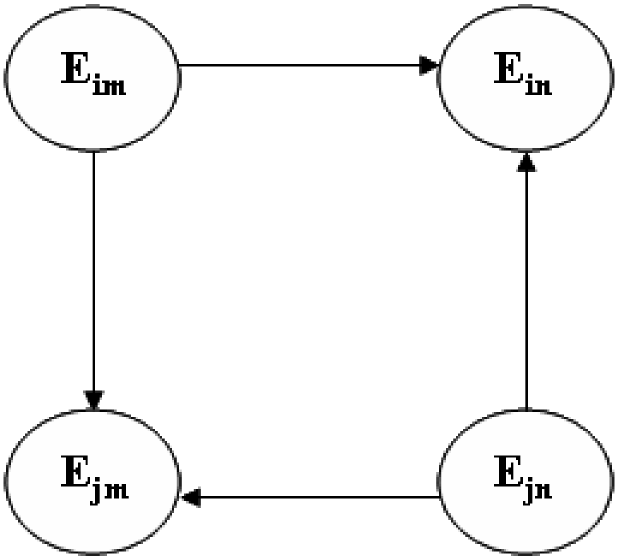


Fig. 10. Using *happens-before.* The relationships between various events are illustrated.

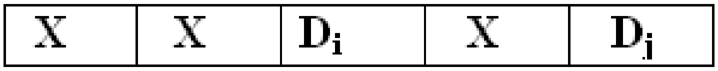
These events are linked with the following *happens-before* relationships:

Relationship: $E_{im} \rightarrow E_{in}$

Reasoning: Two events occurring at the same process and one occurring before the other.

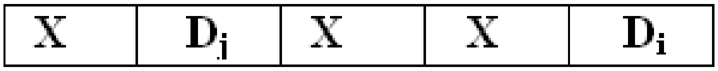
Relationship: $E_{jn} \rightarrow E_{jm}$

Reasoning: Two events happening at the same process and one occurring before the other.



X	X	D_i	X	D_j

Fig. 11. Resource queue for R_m



X	D_j	X	X	D_i

Fig. 12. Resource queue for R_n

The resource queues of R_m and R_n are shown in figures 11 and 12. From these, we can also observe that the following relationships exist between the events:

Relationship: $E_{im} \rightarrow E_{jm}$

Reasoning: D_i lies before D_j in the queue for R_m and processes cannot overtake one another in the queue.

Relationship: $E_{jn} \rightarrow E_{in}$

Reasoning: D_j lies before D_i in the queue for R_n and processes cannot overtake one another in the queue.

Without loss of generality, we assume that there is a total ordering on all the resources and that $R_m < R_n$ in the resource ordering. Hence we *should* have the following relationships between events.

Relationship: $E_{jm} \rightarrow E_{jn}$

Reasoning: A process which requires both resources R_m and R_n needs to get to the head of the resource queue for R_m before enqueueing into the resource queue for R_n. Hence it should enqueue itself into R_m first.

From this relation, and the relation $E_{jn} \rightarrow E_{jm}$, we apply transitivity to get
$E_{jm} \rightarrow E_{jm}$
But this violates the irreflexivity of the *happens-before* relation. Hence we derive a contradiction. Thus it has been proved that no circular wait can occur in the HRA algorithm between any two processes.

VITA

Vijaya K. Idimadakala

Address:

Department of Computer Science,
Texas A&M University,
College Station, TX – 77840

Education:

2000-2004: B.Tech, Computer Science & Engineering, National Institute of Technology, Warangal, A.P, India.

2004-2006: M.S Computer Science, Texas A&M University, College Station, TX.

Experience:

2004-2005: Graduate Assistant Non-Teaching, Texas A&M University, College Station, TX.

2005-2006: Graduate Research Assistant, Texas A&M University, College Station, TX.