The Drinking Philosophers Problem

K. M. CHANDY and J. MISRA
University of Texas at Austin

The problem of resolving conflicts between processes in distributed systems is of practical importance. A conflict between a set of processes must be resolved in favor of some (usually one) process and against the others: a favored process must have some property that distinguishes it from others. To guarantee fairness, the distinguishing property must be such that the process selected for favorable treatment is not always the same. A distributed implementation of an acyclic precedence graph, in which the depth of a process (the longest chain of predecessors) is a distinguishing property, is presented. A simple conflict resolution rule coupled with the acyclic graph ensures fair resolution of all conflicts. To make the problem concrete, two paradigms are presented: the well-known distributed dining philosophers problem and a generalization of it, the distributed drinking philosophers problem.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*concurrency; mutual exclusion; synchronization*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms

Additional Key Words and Phrases: Asymmetry, dining philosophers problem

1. INTRODUCTION

We study the problem of fair conflict resolution in distributed systems. Conflicts can be resolved only if there is some property by which one process in every set of conflicting processes can be distinguished and selected for favorable treatment; that is, a conflict is resolved in favor of the distinguished process. In order to guarantee fairness, the distinguishing property must be such that the process selected for favorable treatment is not always the same. Traditional schemes for fair conflict resolution use priorities assigned to processes [2, 3, 7, 9, 10] or probabilistic selection [5, 8]. We propose a new approach by using the locations of shared resources as a distinguishing property. By introducing auxiliary resources, where needed, and by judiciously transferring resources among processes, we show that all conflicts can be resolved fairly. We propose a paradigm, the *drinking philosophers problem*, which captures the essence of conflict resolution problems in distributed systems. This problem is a generalization of the classical

This work was supported by the Air Force Office of Scientific Research under grant AFOSR 81-0205.
Authors' present addresses: K.M. Chandy, Department of Computer Science, University of Texas at Austin, Austin, TX 78712; J. Misra, Computer Systems Laboratory, Stanford Electronics Laboratories, Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

dining philosophers problem [2, 3]. We present both problems formally in the following sections. This section presents an *informal* introduction to the problem of conflict resolution in distributed systems.

Two or more processes cannot execute certain actions simultaneously: for instance, two processes cannot hold "write locks" on the same data item at the same time. Conflicts arise when two or more processes attempt to carry out such actions simultaneously. The resolution of such a conflict requires that some processes be treated differently from others in the sense that the conflict be resolved *in favor* of some processes and *against* other conflicting processes. If all processes in a set of conflicting processes are indistinguishable (i.e., if every property that holds for one process also holds for the others), then it is impossible to resolve conflicts between them without resorting to random selection. This is because any deterministic algorithm that selects one of the processes for favorable treatment must carry out the selection on the basis of some property that holds for that process and not for the others. Therefore, if we do not wish to use probabilistic algorithms to resolve conflicts, we must ensure the following invariant:

Distinguishability. In every state of the system at least one process in every set of conflicting processes must be distinguishable from the other processes of the set.

An example of a distinguishing property is a process's identity number (provided that it is different from the identity numbers of all processes that it may conflict with).

Fairness. Usually we require not only that conflicts be resolved but also that they be resolved fairly, that is, conflicts should not always be resolved to the detriment of a particular process. If conflicts always occur in the same system state, a deterministic conflict resolution scheme will always resolve conflicts in the same way because the outcome of a deterministic scheme is a function of the system state. In this case conflict resolution will be unfair. Fairness requires that the states that obtain when conflicts occur not always be identical. An example of state information used to ensure that conflicts arise in different system states is *time*, where time may be determined by a centralized, global clock or by distributed logical clocks [7]: every request (which may result in a conflict) is timestamped, and a conflict between two requests is resolved in favor of the one with the smaller timestamp. However, conflicts between processes with equal timestamps must be resolved by using some other distinguishing property (such as process *ID*s). The state information used to ensure fairness may reside in a single process (the centralized solution) or it may be distributed. This paper is about distributed schemes to ensure (1) distinguishability and (2) fairness.

We describe our problem informally by using a graph model of conflict. A distributed system is represented by an undirected graph G with a one-to-one correspondence between vertices in G and processes in the system. Edge (u, v) exists in G if and only if there may be a conflict between (the processes corresponding to) vertices u and v. We assume that there is some mechanism that, in every state of the system, ascribes a precedence ordering to every pair of

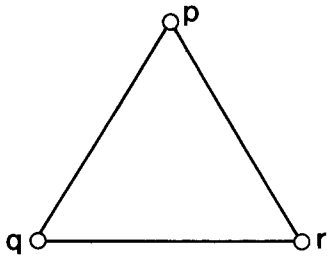

Fig. 1. Graph G.

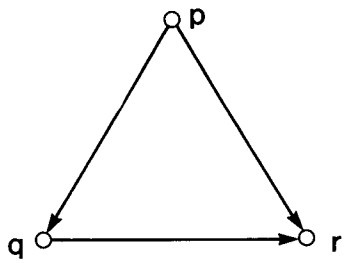

Fig. 2. Graph H.

potentially conflicting processes so that one of the processes in the pair has precedence over the other. If there is a conflict between a pair of processes, the process with the lower precedence must yield to the process with greater precedence in finite time. We represent precedences between pairs of potentially conflicting processes by a *precedence graph* H, which is a graph identical to G except that each edge in G is given a direction in H as follows: An edge in H is directed away from the process with greater precedence toward the process with lesser precedence. For example, Figure 1 shows graph G for a system with 3 processes p, q, and r with the possibility of conflict between any pair of processes. Figure 2 shows graph H for a state of the system in which p has precedence over q and r, and q has precedence over r.

If H is acyclic, then the *depth* of a process in H is a distinguishing property by which a process can be distinguished from all processes that it may conflict with; *depth* of a process p in H is the maximum number of edges on any (directed) path to p from a process without any predecessors. Note that a process with no predecessor has depth 0. It follows that neighbors cannot have the same depth. For example, in Figure 2, the depth of p, q, and r are 0, 1, and 2, respectively.

If H is a cycle, the topology of H does not allow us to distinguish one process from another. We propose an algorithm that ensures that H is acyclic in every state of the system.

The acyclicity of H in every state of the system guarantees distinguishability but does not guarantee fairness. We wish to ensure that every process with conflicts has all its conflicts resolved in its favor in finite time; this requirement can be ensured by a guarantee that every process with conflicts rises to the top (i.e., to zero depth), in H in finite time. By the phrase, a "process p will rise to the top in H," we mean that the state of the system will change, and hence H will change too, so that p will have no predecessor in the precedence graph H at some later state. If p is at depth 0, then any conflict that p has will be resolved in p's favor in finite time because p takes precedence over all of its neighbors.

How should H change? The only way to change H is by changing the directions of the edges. We propose to implement H, and changes to H, by a distributed scheme, where each change in H is made locally at one process. Therefore our requirements are (1) H remains acyclic at all times, (2) H changes in such a manner that every conflicting process eventually rises to the top in H, and (3) each change to H be done locally at a process.

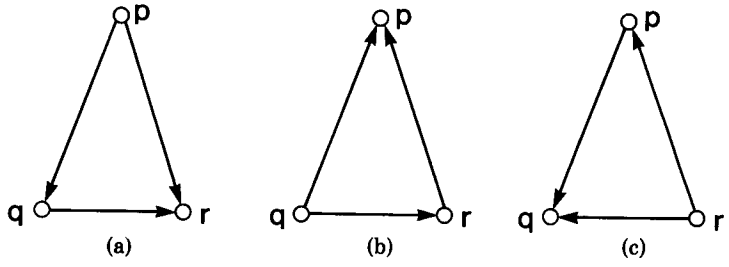

Fig. 3. Example illustrating rule for changing H.

To ensure acyclicity, we employ the following rule for changing H:

Acyclicity Rule. All edges incident on a process p may be simultaneously (i.e., in one atomic action) redirected toward p.

This transformation preserves acyclicity of H because no cycle containing p can be created by the transformation since there is no edge directed away from p after the transformation.

To ensure that every process in a conflict will rise to the top in H eventually we employ the following rule:

Fairness Rule. Within finite time after a conflict is resolved in favor of a process p at depth 0, p must yield precedence to all its neighbors.

This ensures that in the event that process at depth 0 is in conflict it will redirect all incident edges toward itself in finite time. This redirection of edges follows the acyclicity rule.

Example. Consider the precedence graph H shown in Figure 3a, where p, q, and r have depth 0, 1, and 2, respectively. If there are conflicts, then in finite time the directions of all edges incident on p will be reversed to give the precedence graph shown in Figure 3b, in which p, q, and r have depth 2, 0, and 1, respectively. If conflicts persist, in finite time the directions of all edges incident on q will be reversed to give the precedence graph in Figure 3c, in which p, q, and r have depth 1, 2, and 0, respectively.

The *key issue* is to devise a *distributed* implementation of H, as well as the acyclicity and fairness rules. The distributed aspect of the problem makes it nontrivial. The difficulty is that a process has to decide whether to yield or not to yield in a conflict, and the decision has to be made solely on the basis of the process's *local state*. It may not be possible to determine the direction of edges incident on a process only on the basis of the process's local state. Therefore we devise a distributed implementation of H and a scheme by which processes resolve conflicts by making local decisions based on *partial* information of H.

Our goal in this section was to discuss the concepts underlying distributed conflict resolution and the treatment has been informal. The following sections offer a more formal treatment of conflict resolution by defining and solving a specific problem: *The drinking philosopher problem*, which serves as a paradigm of conflict resolution problems.

2. THE DRINKING PHILOSOPHERS PROBLEM (DRINKERS PROBLEM)

The following problem is a generalization of the *dining philosophers problem* [2, 3], which has achieved the status of legend, since it captures the essence of many synchronization problems. Processes, called *philosophers*, are placed at the vertices of a finite undirected graph G with one philosopher at each vertex. A philosopher is in one of 3 states: (1) *tranquil*, (2) *thirsty*, or (3) *drinking*. Associated with each edge in G is a *bottle*.[1] A philosopher can *drink* only from bottles associated with his incident edges. A tranquil philosopher may become thirsty. A thirsty philosopher needs a nonempty set of bottles that he wishes to drink from. *He may need different sets of bottles for different drinking sessions.* On holding all needed bottles, a thirsty philosopher starts drinking; a thirsty philosopher remains thirsty until he gets all bottles he needs to drink. On entering the drinking state a philosopher remains in that state for a finite period, after which he becomes tranquil. A philosopher may be in the tranquil state for an arbitrary period of time.

Two philosophers are *neighbors* if and only if there is an edge between them in G. Neighbors may send messages to one another. Messages are delivered in arbitrary but finite time. Resources, such as bottles, are also encoded and transmitted as messages.

The problem is to devise a nonprobabilistic solution that satisfies the following constraints.

Fairness. No philosopher remains thirsty forever.

Symmetry. All philosophers obey precisely the same rules for acquiring and releasing bottles. There is no priority or any other form of externally specified static partial ordering among philosophers or bottles.

Economy. A philosopher sends and receives a finite number of messages between state transitions. In particular, permanently tranquil philosophers do not send or receive an infinite number of messages.

Concurrency. The solution does not deny the possibility of simultaneous drinking from different bottles by different philosophers.

Boundedness. The number of messages in transit, at any time, between any pair of philosophers is bounded. Furthermore, the size of each message is bounded.

The drinkers problem is a general paradigm for modeling conflicts between processes. Neighboring philosophers will be prevented from drinking simultaneously if they wish to drink from the same bottle—this situation models conflicts for exclusive access to a common file. Neighboring philosophers may drink simultaneously from different bottles—this situation models processes writing into different files.

We must prevent the system from entering states in which neighboring philosophers are indistinguishable. For example, consider philosophers arranged in a ring and a state in which each philosopher is drinking from his "left" bottle—philosophers cannot be distinguished in this state. If all philosophers are drinking from their left bottles and then require both bottles for their next drinking

[1] The solution given in this paper also applies to multiple bottles on every edge. The assumption of one bottle per edge is made for brevity in exposition.

session, then the philosophers must remain thirsty forever because a deterministic algorithm cannot choose between indistinguishable philosophers. However, a system state is certainly possible in which all philosophers hold their left bottles. If we were to disallow this state, we would be disallowing a feasible state in which progress is being made, merely to solve our problem; disallowing feasible states violates our constraint of *concurrency*. We appear to be in a quandary because the constraints of symmetric processes, nonprobabilistic solutions, and concurrency are incompatible for this problem. The solution is to implement precedence graph H by using special "auxiliary" resources. The solution to the dining philosophers problem shows us how to implement H. Therefore we study the dining philosophers problem next. We then study the drinkers problem using the diners problem solution to implementing H.

3. THE DINING PHILOSOPHERS PROBLEM (DINERS PROBLEM)

The diners problem [2] is a special case of the drinkers problem in which every thirsty philosopher needs bottles associated with *all* its incident edges for *all* drinking sessions. We present a solution for this problem with the properties of fairness, symmetry, economy, concurrency, and boundedness. To distinguish between these two problems, we use the following terms for the diners problem, with the corresponding term for the drinkers problem in parentheses: *thinking* (*tranquil*), *hungry* (*thirsty*), *eating* (*drinking*), *fork* (*bottle*). The diners problem disallows neighbors from eating simultaneously. The drinkers problem allows neighbors to drink simultaneously provided that they are drinking from different bottles.

We first present an informal outline of the solution; the next section has a detailed formal description. A fork is either *clean* or *dirty*. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when mailing it (he is hygienic). A fork is cleaned only when it is mailed. An eating philosopher does not satisfy requests for forks until he has finished eating. The key issue is: which requests should a noneating philosopher defer? In our algorithm, a noneating philosopher defers requests for forks that are clean and satisfies requests for forks that are dirty.

This solution to the diners problem suggests an implementation of precedence graph H. The direction of an edge between two neighbors u and v is from u to v (i.e., u has precedence over v) if and only if (1) u holds the fork shared by u and v, and the fork is clean, or (2) v holds the fork, and the fork is dirty, or (3) the fork is in transit from v to u. Observe that the direction (from u to v) of the edge can change only when u starts eating. Furthermore, all edges incident on an eating philosopher are directed toward it. Hence we have an implementation of the acyclicity rule: The direction of edges incident on a process may be changed only in the following way—*all* edges incident on a process may be simultaneously directed toward it.

Initially all forks are dirty and are located at philosophers in such a way that H is acyclic. Hence the following is an invariant: *H is acyclic.*

Immediately upon completion of an eating session, a philosopher yields precedence to his neighbors. A hungry philosopher at depth 0 in H will commence

eating in finite time (because he has precedence over all his neighbors). By induction on depth, a hungry philosopher at depth k, $k \geq 0$, will eat in finite time because he has precedence over all philosophers at greater depth, and all philosophers at smaller depth will yield precedence to it in finite time.

A formal treatment of these arguments is found in the next section.

4. A HYGIENIC SOLUTION TO THE DINERS PROBLEM

4.1 Algorithm

We now give a precise description of the solution outlined in the last section. To simplify our description, we introduce a *request token* for each fork. Only the holder of the request token for fork f can request fork f. A request for a fork is made by sending the corresponding request token to the holder of the fork. It follows then that only one process—the holder of the request token for f—may request fork f and the requested process, after receiving the token, has the next chance to request the fork. Also, if a process holds a fork *and* the request token for the fork then his neighbor (with whom he shares the fork) has an outstanding request for the fork. We introduce the following Boolean variables:

$fork_u(f)$:	philosopher u holds fork f,
$reqf_u(f)$:	philosopher u holds the request token for fork f,
$dirty_u(f)$:	fork f is at u and is dirty,
$thinking_u/hungry_u/eating_u$:	philosopher u is *thinking/hungry/eating*.

We drop the subscripts from the Boolean variables when the context is clear.

Each philosopher follows the rules given below for requesting and sending forks. In each case a rule is written as $g \rightarrow A$, where g is a condition and A is a sequence of actions. These rules constitute our solution to the diners problem. The set of rules forms a single guarded command [4].

(R1) Requesting a fork f:
hungry, $reqf(f)$, $\sim fork(f) \rightarrow$
send request token for fork f (to the philosopher with whom f is shared);
$reqf(f) := false$

(R2) Releasing a fork f:
$\sim eating$, $reqf(f)$, $dirty(f) \rightarrow$
send fork f (to the philosopher with whom fork f is shared);
$dirty(f) := false$;
$fork(f) := false$

(R3) Receiving a request token for f:
upon receiving a request for fork $f \rightarrow$
$reqf(f) := true$

(R4) Receiving a fork f:
upon receiving fork $f \rightarrow$
$fork(f) := true$
$\{\sim dirty(f)\}$

We note that the statement of the diners problem defines transitions among states (*thinking*, *hungry*, *eating*) for a philosopher, and we furthermore have for any philosopher,

$$eating, fork(f) \Rightarrow dirty(f).$$

Initial Conditions

1. All forks are dirty. {$\forall f$, $dirty_u(f)$ *or* $dirty_v(f)$ where u, v are the philosophers who can use fork f}.
2. Every fork f and request token for f are held by different philosophers. {If fork f is shared between philosophers (u, v), then u holds the fork and v the token (i.e., $fork_u(f)$, $reqf_v(f)$, $\sim fork_v(f)$, $\sim reqf_u(f)$), or v holds the fork and u the token.}
3. H is acyclic. {The forks are initially placed in such a manner that H is acyclic.}

4.2 Proof of the Hygienic Solution for the Diners Problem

We show in this section that every hungry philosopher will eat. In addition to this fairness condition, we show that our solution has the properties of symmetry, economy, concurrency, and boundedness.

Fairness

LEMMA 1. *H is always acyclic.*

PROOF. Initially H is acyclic. The directions of edges in H may be affected only when a fork changes its status (dirty or clean) or its location. We will show that every change to H preserves acyclicity. Every transmission of a fork is accompanied by a change in its status from dirty to clean; this does not change the direction of any edge. A fork is dirtied when the philosopher u holding it, eats. In this case u must be holding all other forks associated with edges incident upon it, and they must all be dirty. u cannot then create a cycle in H because all edges upon u are directed toward it. □

THEOREM 2. *Every hungry philosopher eats.*

The following proof is based on the fact that a hungry philosopher requesting a fork that is currently dirty will receive it (from rule R2), and since the fork is clean upon receipt the philosopher will hold it until he eats. A philosopher requesting a fork that is clean must make the request to a philosopher at a smaller depth and, by induction on depth, this philosopher will eat and then dirty the fork, in which case the first argument applies.

PROOF. Recall that the depth of a philosopher in H is the maximum number of edges along a path to that philosopher from one without predecessors. We prove the theorem by induction on depth of a hungry philosopher; the induction amounts to showing that hungry philosophers at depth k in every H eat, provided all hungry philosophers at depths smaller than k in every H eat, for all $k \geq 0$.

We will not do a special analysis for hungry philosophers at depth 0, because that case is subsumed by Case 1, below.

Let u, v be neighbors and u be hungry. We show that u holds or will hold the fork f corresponding to the edge (u, v) and will thereafter continue to hold it until u eats. If u holds the fork currently and holds it continuously until he eats, the result is trivial. Therefore assume that v holds the fork f sometime before u eats next. We do a case analysis on the status of f at the time that v holds the fork. At this time we have: $hungry_u$, $\sim fork_u(f)$, $fork_v(f)$.

Case 1: f is dirty ($dirty_v(f) = true$). If $reqf_u(f)$ holds then u will request f (because precondition of rule R1 will hold) and subsequently $reqf_v(f)$ will hold; otherwise $reqf_v(f)$ already holds. If $eating_v$ holds then at some later point (since eating is finite), $\sim eating_v$ holds, and all other predicates for rule R2 still hold. Therefore rule R2 will be applied by v, and u will eventually hold a clean fork. u will not release a clean fork until u eats.

Case 2: f is clean ($dirty_v(f) = false$). Every fork held by a nonhungry philosopher is dirty because

(a) all forks are dirty initially,
(b) only hungry philosophers receive clean forks, and
(c) all forks held by eating philosophers are dirty.

Since f is clean, the philosopher v holding it must be hungry. Furthermore, because f is clean, (v, u) is an edge in H and hence $depth(v) < depth(u)$. According to the induction hypothesis, v eats and hence dirties f. Case 1 then applies. □

Symmetry. It follows from the description of the algorithm that all philosophers follow the same rules.

Economy. The number of message sends and receives before a state transition is bounded as follows: if d is the number of neighbors of a philosopher, then no more than d requests or forks will be sent or received. More precisely, suppose a philosopher has e dirty forks when he transits to hungry state. Then he must send $d - e$ requests and receive a fork corresponding to each request. In addition, in the worst case, he may lose all e forks he had held initially and therefore have to request and receive them. Assume that a philosopher implements the latter situation by sending a fork and the request for it in one message. Then no more than $2d$ messages are needed before transiting to the eating state. The only messages received in the eating or thinking state are the requests for forks held by the philosopher and hence these do not exceed d. In the best case, a philosopher with permanently thinking philosophers as neighbors will receive no requests for forks and therefore may live a life (think and eat) free of interaction with others.

Concurrency. Our solution does not deny any feasible system state; that is, any state of the system in which neighboring philosophers are not eating is allowable in our solution. This is because the solution does not prevent a philosopher from entering the thinking or hungry state; the only restriction is in entering the eating state, and that is allowable when a hungry philosopher holds all forks, as required by the problem.

Boundedness. There are at most two messages—a fork and a request for a fork—in transit, between any two philosophers.

5. A SOLUTION TO THE DRINKERS PROBLEM

5.1 The Precedence Graph

Our solution to the drinkers problem uses precedence graphs discussed in Section 1. The solution to the diners problem demonstrates a distributed implementation of the precedence graph H. Fairness and the acyclicity of H are ensured by implementation of the fairness and acyclicity rules. It may appear that H provides

a simple resolution mechanism for any type of conflict, including conflict for bottles in the drinkers problem, since any conflict can be resolved in favor of the process with greatest precedence. However, there is a difficulty due to the *distributed* implementation of H. Given only the state of process u we can determine which of neighbors u or v has precedence *if u holds the fork*: If the fork is clean u has precedence, if it is dirty v has precedence. However, *if u does not hold the fork we cannot determine which of u or v has precedence from the state of u alone.* In this case u must make local decisions about holding on to or releasing bottles without using precedence graph H. This issue is discussed next in the context of the drinkers problem.

We use forks to implement H. Forks are *auxiliary resources* in the sense that their sole purpose is to implement precedence graph H. Forks are not part of the drinkers problem specification; they are part of the solution. The real resources in the drinkers problem are bottles. Our philosophers can eat and drink *simultaneously*, and we emphasize that *eating is an artifact of our solution*, used only to guarantee fair drinking. In our solution, the state of a philosopher is a pair (diner's state, drinker's state), where a diner's state is one of thinking, hungry, or eating and a drinker's state is one of tranquil, thirsty, or drinking. Our next step is to define the dining characteristics of our philosophers; the drinking characteristics are specified by the problem. We give rules that ensure that all thirsty philosophers drink in finite time.

Consider the state transitions of a dining philosopher. The only transitions that are decided by the philosopher are thinking-to-hungry and eating-to-thinking; the only transition completely specified by the diners problem is hungry-to-eating (which occurs when a philosopher holds all forks he needs). We now give rules for the dining philosopher to decide the point of the first two transitions.

(D1) *Thinking-to-Hungry Transition*:
A *thinking, thirsty* philosopher becomes *hungry.*

(D2) *Eating-to-Thinking Transition*:
An *eating, nonthirsty* philosopher starts *thinking.*

In the diners problem, a philosopher can think for arbitrary time though he must eat for finite time. Therefore our obligation, arising out of rules (D1) and (D2), is to ensure that each eating period is finite. This is accomplished by the rule (D3) given below.

(D3) *The Conflict Resolution Rule*:
Philosopher u sends a bottle to philosopher v, in response to a request from v, if and only if u does not need the bottle **or** [u is not drinking **and** does not hold the fork for the edge (u, v)].

Note that u's decision to send or hold onto a bottle requested by v depends on whether u holds the fork associated with edge (u, v), and *does not depend on whether u or v has precedence in H.* In particular, u must send the bottle to v if u has precedence over v, but u does not hold the fork associated with edge (u, v). We must show that despite this fact, the algorithm is fair.

The basic idea is this: Suppose u has precedence over v (i.e., (u, v) is an edge in H), but v holds the fork (i.e., the fork is dirty), and suppose u requests a bottle

held by v. We require that u not only request the bottle held by v, but that u also request the fork. We show (from the solution to the diners problem) that in finite time v will yield the fork to u after which it must also yield the bottle to u. Thus, the algorithm ensures that if u has precedence over v in H then *eventually* the conflict resolution rule causes conflicts for bottles between u and v to be resolved in u's favor.

5.2 Algorithm for the Drinkers Problem

Now, we state the algorithm formally. As before, we introduce a request token, *reqb*, for every bottle b. The following Boolean variables are used:

$bot_u(b)$:	philosopher u holds bottle b
$reqb_u(b)$:	philosopher u holds request token for bottle b
$need_u(b)$:	philosopher u needs bottle b
$tranquil_u/thirsty_u/drinking_u$:	philosopher u is *tranquil/thirsty/drinking*

As before, we drop the subscript when the context is understood. From the problem statement we have,

$$tranquil \Rightarrow \forall b[\sim need(b)]$$

State transitions for dining philosopher determined by drinking states are

(D1) *thinking*, *thirsty* → *hungry* := *true*
(D2) *eating*, ~ *thirsty* → *thinking* := *true*

Other actions of the dining philosopher remain unchanged.

Rules for bottle and request transmissions {Let f be the fork corresponding to bottle b, i.e., fork f and bottle b are shared by the same two processes}:

(R1) Request a Bottle:
thirsty, *need*(b), *reqb*(b), ~*bot*(b) →
send request token for bottle b;
reqb(b) := *false*

(R2) Send a Bottle:
reqb(b), *bot*(b), ~[*need*(b) **and** (*drinking* **or** *fork*(f))] →
send bottle b;
bot(b) := *false*

(R3) Receive Request for a Bottle:
upon receiving request for bottle b →
reqb(b) := *true*

(R4) Receive a Bottle:
upon receiving bottle b →
bot(b) := *true*

Initial Conditions

For Dining Philosophers: As before.

For Drinking Philosophers: A bottle and the request token for it are held by different philosophers; that is, if u, v share bottle b, then u holds the bottle and v the token ($bot_u(b)$, $reqb_v(b)$, $\sim bot_v(b)$, $\sim reqb_u(b)$), or v holds the bottle and u the token.

5.3 Proof of Correctness of the Solution to Drinkers Problem

We show that the solution has the desired properties of fairness, symmetry, economy, concurrency and boundedness.

Fairness

LEMMA 3. *Every eating period is finite.*

PROOF. If an eating philosopher is nonthirsty, he completes eating (D2). If philosopher u is eating, he is holding all forks. If he holds a bottle that he needs, he will not release it until he completes drinking, from the precondition of (R2). If he needs and does not hold a bottle that he shares with v, then he holds or will hold the request token for the bottle (same proof as in Case 1 of Theorem 2). He will request the bottle, from (R1), and v will have to send the bottle in finite time (R2) since v does not hold the fork and v can be in drinking state only for finite duration. Therefore u will hold all bottles he needs in finite time. Since u drinks for finite time, u will become tranquil in finite time and, from (D2), u will stop eating in finite time. □

Since every eating period is finite, Theorem 2 applies and we have

COROLLARY 4. *Every hungry philosopher starts eating in finite time.*

THEOREM 5. *Every thirsty philosopher drinks in finite time.*

PROOF. When a philosopher becomes thirsty he is either thinking, hungry, or eating. A (thirsty, thinking) philosopher becomes hungry in finite time (from D1); a hungry philosopher starts eating in finite time (from Corollary 4). Therefore every philosopher who remains thirsty will eat in finite time. The theorem follows from Lemma 3 and the fact (D2) that eating can be terminated only by drinking. □

Symmetry. Follows from the description of the algorithm.

Economy. We first show that a bottle b can travel at most twice between neighbors, u, v, before one of them drinks from b. A bottle is sent in response to a request from a thirsty philosopher. Let (u, v) be a directed edge in H; the bottle will travel at most once from u to v and will then be held by v until v drinks. This is because (1) either v holds a clean fork, which will not be released until after eating (and hence drinking), and therefore the bottle b, which is needed by v will not be released, or (2) u holds a dirty fork, which must have been requested by v (when v became thirsty and hence hungry) and will be mailed, after being cleaned, along with the bottle to v, and then case (1) applies. Hence a bottle can travel at most twice between neighbors before one of them drinks.

LEMMA 6. *There are at most $4qd$ message transmissions for q drinking sessions among all philosophers, where d is the maximum degree (i.e., the maximum number of neighbors) of any philosopher.*

PROOF. There is at most one request (for fork and/or bottle), one transmission of a fork, and two transmissions of a bottle between neighbors before one of them

drinks. Therefore, when a philosopher drinks, there must have been no more than 4 messages per each of its neighbors and hence the result. □

Concurrency. The argument for concurrency is similar to that for the diners problem. We note that no feasible state of the drinkers problem is being eliminated in our solution.

Boundedness. There are at most three messages—request for a bottle and/or fork, a bottle, or a fork—in transit from one philosopher to another at any point.

6. SUMMARY

We have described a distributed implementation of a precedence graph. The changes to the graph are such that the graph is always acyclic. The depth of a process in the graph is the process's distinguishing characteristic. The graph is implemented by the "forks" of the diners problem. Two processes share a fork if they may conflict with one another. The conflict-resolution rule is: A process u yields in a conflict with a process v if and only if u does not hold the fork shared with v. The algorithm ensures that if processes u and v are in conflict, and u has precedence over v in the precedence graph, then the conflict resolution rule will, *eventually*, cause conflicts to be resolved in u's favor.

Many types of conflict can be resolved by using the conflict-resolution rule coupled with our distributed implementation of the precedence graph. For instance, consider the multiple concurrent mutual exclusion problem described next. A critical section in a process has an arbitrary number of colors associated with it (where a color is some attribute of the critical section). The problem is to devise a scheme by which, for each color c, there is at most one process executing a critical section with associated color c. For example, a color may correspond to the privilege of exclusive access to a specific file, and associated with each critical section is the set of files accessed within that section. If all critical sections have the same set of colors, the problem reduces to the classical mutual exclusion problem.

We use our solution to the drinkers problem to solve the concurrent mutual exclusion problem. We use a variant of the drinkers problem in which a pair of philosophers may share an *arbitrary* number of bottles. The bottles are colored, each bottle having precisely one color. A pair of philosophers share *at most* one bottle of a given color. A bottle is specified by the edge it is on (i.e., by the pair of philosophers who share it) and by its color. The set of bottles a thirsty philosopher needs to drink is arbitrary—it may include any bottle he shares. For instance, when philosopher i becomes thirsty, he may need to hold the red bottle shared with j *and* the red bottle shared with k *and* the blue bottle shared with k. If there is precisely one bottle on each edge the problem reduces to the one discussed earlier. We leave it to the reader to show that the algorithm given earlier also applies to the extension to colored bottles.

Given a concurrent mutual exclusion problem, we construct a drinkers problem as follows: Philosophers (processes) i and j share a bottle with color c if and only if both philosophers have critical sections with color c. A process i may enter a critical section with a set of colors ς if and only if, for all colors c in ς, and for all edges e incident on process i, the bottle of color c on edge e is held by philosopher

i. In this case it is obvious that no neighboring philosopher can enter a critical section with a color c in $\underline{c}$.

7. PREVIOUS WORK

The distributed dining philosophers problem (philosophers at the vertices of a graph) and the dining philosophers problem (five philosophers arranged in a ring) appear in [2, 3]. Dijkstra's solutions to the former problem are based on instantaneous atomic transmissions of messages to all neighbors or static fork orderings. Lynch [9] has carried out an extensive analysis of static resource ordering algorithms.

The problem of mutual exclusion among a group of processes in executing their critical sections is a special case of the diners problem: Every process is a neighbor of every other process and execution of a critical section corresponds to eating. Distributed solutions to mutual exclusion using timestamps and process *ID*s to break ties, appear in Lamport [7] and in Ricart and Agrawala [10]. Shared counter variables have been used in [1], for solving the dining philosophers problem.

A symmetric distributed solution to the diners problem appears in Francez and Rodeh [5]. They use an extended form of CSP [6], in which both input and output commands are used in guards.

Lehmann and Rabin [8] give a perfectly symmetric probabilistic algorithm and show that there is no perfectly symmetric nonprobabilistic solution to the diners problem.

ACKNOWLEDGMENT

We thank W.H.J. Feijen and A.J.M. Van Gasteren of Eindhoven University of Technology and Greg Andrews of the University of Arizona for their detailed comments. We are grateful to three unknown referees and to Susan Graham for detailed comments. Conversations with E.W. Dijkstra on this problem were most helpful.

REFERENCES

1. DEVILLERS, R.E., AND LAUER, P.E. A general mechanism for avoiding starvation with distributed control. *Inf. Process. Lett. 7*, 3 (Apr. 1978), 156–158.
2. DIJKSTRA, E.W. Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.
3. DIJKSTRA, E.W. Hierarchical Ordering of Sequential Processes. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds., Academic Press, New York, 1972.
4. DIJKSTRA, E.W. Guarded commands, nondeterminancy and formal derivation of programs. *Commun. ACM 18*, 8 (Aug. 1975), 453–457.
5. FRANCEZ, N., AND RODEH, M. A distributed abstract data type implemented by a probabilistic communication scheme. Tech. Rep. TR-080, IBM Israel Scientific Center, Haifa, Israel, Apr. 1980.
6. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
7. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

8. LEHMAN, D., AND RABIN, M. On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 26–28). ACM, New York, 1981, pp. 133–138.
9. LYNCH, N.A. Fast allocation of nearby resources in a distributed system. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing* (Los Angeles, Calif., Apr. 28–30). ACM, New York, 1980, pp. 70–81.
10. RICART, G., AND AGRAWALA, A. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM 24*, 1 (Jan. 1981), 9–17.

Received May 1983; revised February 1984; accepted May 1984