AN EFFICIENT SOLUTION TO THE DRINKING PHILOSOPHERS PROBLEM AND ITS EXTENSIONS

David Ginat, A. Udaya Shankar, A. K. Agrawala

Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

We introduce a new solution to the drinking philosophers problem, using drinking session numbers. It is simple and easily adjusted to extensions of the problem. The number of messages per drinking session is between zero and twice the number of bottles needed for drinking, a considerable improvement over previous work.

1. INTRODUCTION

The drinking philosophers problem is a paradigm for sharing resources in distributed systems [1]. Contention between processes over "write locks" is abstracted in the problem into contention between philosophers over shared bottles. The problem was posed by Chandy and Misra [1] as a generalization of the dining philosophers problem [3]. Unlike the dining philosophers problem, in which a hungry philosopher needs *all* the forks he shares with his neighbors once he becomes hungry, in the drinking philosophers problem a philosopher may need only a *subset* of the bottles he shares with his neighbors upon becoming thirsty. Different subsets may be needed for different drinking sessions.

Chandy and Misra introduced a solution in which a thirsty philosopher can *concurrently* collect all the bottles he needs and all philosophers obey the same rule for resolving conflicts over shared bottles. Conflicts are resolved using an artificial dining layer below the drinking layer. However, due to the use of an artificial dining layer their solution is not simple and requires extra messages. We present a simple and efficient solution that eliminates this layer, by using drinking session numbers extended with ids. We provide an assertional proof of our solution. We then adjust our solution to extended versions of the problem. The number of messages per drinking session is between zero and twice the number of bottles needed for drinking, a considerable improvement over the result in [1]. The drawbacks of our solution are the lack of upper bound for drinking session numbers, and the use of ids for breaking ties between contending philosophers whose session numbers are equal.

In section 2, we present the problem as posed by Chandy and Misra. In section 3, we present our simple and efficient solution to the problem. In section 4 we provide its assertional correctness proof, and in section 5 we derive its message complexity and explain its improvement over Chandy and Misra's solution. In section 6, we adjust our solution to solve extensions of the original problem.

2. THE BASIC DRINKING PHILOSOPHERS PROBLEM

Philosophers are placed at the nodes of an undirected graph G, with one philosopher at each node. Neighboring philosophers communicate via messages subject to varying but finite transmission delays. Associated with each edge in G is a bottle shared between the two philosophers that are incident on the edge. A philosopher is either *tranquil*, *thirsty*, or *drinking*. A philosopher can be tranquil for an arbitrary period of time. A tranquil philosopher may become thirsty and need a *nonempty subset* of the bottles associated with his incident edges. He remains thirsty until he gets all the bottles he needs, at which point he starts drinking. He drinks for a finite time, after which he becomes tranquil and needs no bottles. A philosopher may need different subsets of bottles in different drinking sessions. Philosophers can drink concurrently from different bottles. The goal is to provide a solution which ensures that no two philosophers drink simultaneously from the same bottle, and that no philosopher remains thirsty forever.

3. ALGORITHM FOR THE BASIC PROBLEM

Philosopher u maintains two nondecreasing integer variables, s_num_u and max_rec_u. s_num_u, referred to as u's session number, identifies u's last drinking session if u is tranquil, u's upcoming drinking session if u is thirsty, and u's current drinking session if u is drinking. max_rec_u indicates the highest session number received by u from his neighbors so far.

(s_num_u, u) is referred to as u's extended session number. $(s\_num_u, u) < (s\_num_v, v)$ iff $s\_num_u < s\_num_v$ or $s\_num_u = s\_num_v$ and $u < v$ [5,8]. Each philosopher has an id which is different from the id's of his neighbors. (Ids of nonneighboring philosophers may be the same.) Thus, if u and v are neighbors, their extended session numbers are never equal.

Let u and v be two philosophers who share bottle b. Associated with bottle b is a request token req_b. Upon becoming thirsty, u sets s_num_u to a value higher than max_rec_u, and if u needs and does not hold the bottle b, u sends to v the request token for b, together with his extended session number, in the request message (req_b, s_num_u, u). u sends request messages *concurrently* to all his neighbors with whom he shares bottles he needs and does not hold. When the neighbor v receives a

request (req_b, s, u), he obeys the following *conflict resolution rule*:

> if v does not need b or (v is thirsty and $(s, u) < (s\_num_v, v)$), then v immediately releases b.

If v does not release b immediately, then he needs b and will release it once he completes drinking from it. v will next use req_b only after he has released b. Thus, when req_b is in transit from v to u, b will either be in transit from v to u ahead of req_b, or will already be at u. s_num_u does not change while u is thirsty. Therefore, once b is released by v, it will not return to v before u drinks from it.

Initially, every philosopher u is tranquil and $s\_num_u = max\_rec_u = 0$. For every bottle b shared between two philosophers, one philosopher has b and the other has req_b.

Each philosopher u obeys the rules given in Table 1 below. Each rule execution is considered an atomic action. In addition to s_num_u and max_rec_u, each philosopher u has the boolean variables $need_u(b)$, $hold_u(b)$, $hold_u(req_b)$, indicating whether u needs b, holds b, and holds req_b, respectively. Below, Send(b) and Send(req_b, s_num_u, u) send the specified message to the philosopher with whom b is shared.

R1: becoming thirsty	when tranquil and "want to drink" do become thirsty; for each desired bottle b do $need_u(b) \leftarrow$ true; $s\_num_u \leftarrow max\_rec_u + 1$
R2: start drinking	when thirsty and holding all needed bottles do become drinking
R3: becoming tranquil, and honoring deferred requests	when drinking and "want to stop drinking" do become tranquil; for each consumed bottle b do [$need_u(b) \leftarrow$ false; if $hold_u(req_b)$ then [Send(b); $hold_u(b) \leftarrow$ false]]
R4: requesting a bottle	when $need_u(b)$, $\neg hold_u(b)$, $hold_u(req_b)$ do Send(req_b, s_num_u, u); $hold_u(req_b) \leftarrow$ false
R5: receiving a request, and resolving a conflict	upon reception of (req_b, s, v) do $hold_u(req_b) \leftarrow$ true; $max\_rec_u \leftarrow \max(max\_rec_u, s)$; if $\neg need_u(b)$ or (thirsty and $(s,v) < (s\_num_u, u)$) then [Send(b); $hold_u(b) \leftarrow$ false]
R6: receiving a bottle	upon reception of (b) do $hold_u(b) \leftarrow$ true

Table 1. Basic Algorithm: Rules for philosopher u

4. PROOF OF CORRECTNESS OF THE BASIC ALGORITHM

The assertions listed below should be viewed as an incremental description of the algorithm's basic properties. These assertions will be proved invariant, i.e., hold at any time during the distributed algorithm's execution.

Conventions. In an assertion, when we say *either A or B* we mean that exactly one of the conditions A or B holds. We assume that messages sent by one philosopher to another arrive in the order sent; a message m is *ahead of* (*behind*) a message n if m was sent before (after) n.

The following invariant properties derive from the problem definition:

A_0: For every bottle b shared between philosophers u and v:
b is either held by u or held by v or in transit between u and v.

A_1: (a) u is not tranquil $\Leftrightarrow$ u needs at least one bottle.
(b) u is drinking $\Rightarrow$ u holds all the bottles he needs and drinks from them.

A_0 and A_1 imply that no two philosophers drink simultaneously from the same bottle.

Let u and v be two neighboring philosophers who share bottle b. The following invariant properties associate the locations of req_b and b:

B_0: req_b is either at u or at v or in transit between u and v.

B_1: req_b is in transit to u $\Rightarrow$ b is either held by u or is in transit to u ahead of req_b.

B_2: req_b is held by u $\Rightarrow$ b is not in transit to u.

B_3: u does not need b $\Rightarrow$ (neither u nor v holds both b and req_b) and ((b is not in transit to u) and (req_b is not in transit to v))

Note that B_1 assures us that b is held by u when he receives req_b.

The following invariant properties relate extended session numbers, request tokens, and bottles:

C_0: b is in transit to u $\Rightarrow$ $(max\_rec_v \geq s\_num_u)$ and
(v does not need b or $(s\_num_u, u) < (s\_num_v, v)$)

C_1: u holds both b and req_b $\Rightarrow$ (u needs b) and $(max\_rec_u \geq s\_num_v)$ and
(u is drinking from b or $(s\_num_u, u) < (s\_num_v, v)$)

C_2: (req_b, s, v) is in transit to u $\Rightarrow$ $(s, v) = (s\_num_v, v)$.

C_3: $s\_num_u \leq max\_rec_u + 1$

The proof of invariance of the above assertions is as follows. All of them hold initially. A_0–A_1 are preserved by every rule. B_0–B_3 are preserved by every rule, assuming A_0 holds before the rule execution. C_0–C_3 are preserved by every rule, assuming A_0, B_0–B_3 hold before the rule execution.

C_3 ensures that R1 does not decrease s_num_u. This yields the next lemma.

Lemma 1. (a) s_num_u and max_rec_u never decrease.
(b) s_num_u does not change while u is thirsty.

Let u and v be two neighboring philosophers who share bottle b. We say that b is *dedicated* to u if u is thirsty and needs b and one of the following holds: (1) b is in transit to u; or (2) u holds b and $(s\_num_u, u) < (s\_num_v, v)$; or (3) u holds b, v does not need b, and $(max\_rec_v+1, v) > (s\_num_u, u)$.

Lemma 2. If b is dedicated to u, then b will not be released by u before he drinks from it.

Proof outline. Consider the period of time while u is thirsty and needs b. From Lemma 1 we know that s_num_u does not change during this period. Given that b is dedicated to u, it can be shown using B_1–B_3 and C_0–C_2 that a request (req_b, s, v) arriving at u during this period must have (s, v) higher than (s_num_u, u), and is therefore deferred. ∎

We now prove that every thirsty philosopher eventually drinks, by introducing a dynamic ordering of the philosophers according to their extended session numbers.

At any time, let the philosophers be grouped into classes, where each class consists of all philosophers with the same extended session number. Order the classes by increasing extended session numbers, and let $pos(u)$ be the position of philosopher u's class in this order. Thus, $pos(u) < pos(v)$ iff $(s\_num_u, u) < (s\_num_v, v)$. We prove progress by an inductive argument on the value of $pos(u)$.

Conventions. We use the temporal operator *leads-to* [2,9]. *A leads-to B* means that if the algorithm is in a state satisfying A, then within a finite number of rule executions it will be in a state satisfying B. We assume that drinking is finite, every message in transit is eventually received, and each of the rules R2 and R4 is fairly implemented, i.e., if it is continuously enabled, it is eventually executed.

Liveness Theorem. Philosopher u is thirsty *leads-to* philosopher u is drinking.

We prove the liveness theorem by proving the following assertion $D(i)$, for any i:

$D(i)$: $pos(u)=i$ and u is thirsty *leads-to* u is drinking.

We prove $D(i)$, for any i, by induction on i.

Proof of $D(1)$. The induction basis. We first prove the following assertion:

E_B: $pos(u)=1$ and u is thirsty and needs bottle b *leads-to* b is dedicated to u.

Let u be a thirsty philosopher who needs bottle b that he shares with philosopher v, and let $pos(u)=1$. $(s\_num_u, u) < (s\_num_v, v)$ since u is in the lowest class, and two neighboring philosophers cannot be in the same class. From Lemma 1, we know that $(s\_num_u, u) < (s\_num_v, v)$ holds continuously during the time that u is thirsty. If b is not already dedicated to u, then b must either be held by v or in transit to v. We show that b will eventually be dedicated to u by examining the possible locations of req_b.

(1) Assume req_b is at v. b cannot be in transit to v, by B_2. Therefore, b is held by v, and from C_1 we can infer that v must be drinking from b. v will complete drinking in finite time, at which point rule R3 will be executed at v, resulting in b being sent to u and hence dedicated to u.

(2) Assume (req_b, s, u) is in transit to v. It will arrive at v in finite time, when b is there (by B_1), and with $(s, u)=(s\_num_u, u)$ (by C_2). Upon reception of req_b, v will either release b immediately and b will become dedicated to u, or v will defer u's request and we are back in (1).

(3) Assume req_b is with u. R4 is continuously enabled at u. Thus, u will eventually send req_b to v and we are back in (2).

(4) req_b cannot be in transit to u, by B_1.

At this point, we can conclude $D(1)$. The number of bottles that u needs for drinking is finite. Each one of these bottles will be dedicated to u and held by him in finite time, after which it will not be released before u drinks, by Lemma 2. Therefore, once all of these bottles are held by u, R2 is continuously enabled at u, and u will eventually start drinking. ∎

Proof of $D(k+1)$ assuming $D(1), \cdots, D(k)$. The induction step. We first prove the following assertion:

E_S: $pos(u)=k+1$ and u is thirsty and needs bottle b *leads-to* b is dedicated to u or u is drinking.

Again, let u be a thirsty philosopher who needs bottle b that he shares with philosopher v, and let $pos(u)=k+1$. $pos(v)\neq k+1$ because u and v are neighbors. If $pos(v)\geq k+2$, then E_S is proved using the same arguments given in E_B's proof. We prove E_S for the case where $pos(v)\leq k$:

If b is not already dedicated to u, then b is not in transit to u. If b is held by u then it may be held by him until he starts drinking, or it may be released by u (upon receiving v's request) while he is still thirsty. The latter reduces to the case where b is not held by u.

If b is not held by u, then b is either held by v or in transit to v. We show that b will eventually be dedicated to u by examining the possible locations of req_b:

(1) Assume req_b is at v. b must be held by v, by B_2, and v needs b, by C_1. If v is thirsty, then he will be drinking in finite time, since we assume $D(1), \cdots, D(k)$. If v is drinking, he will complete drinking in finite time, and b will become dedicated to u.

(2) Assume req_b is not at v. The arguments to show that b will eventually be dedicated to u are the same as those used in E_B's proof under cases (2), (3), (4).

At this point we can conclude $D(k+1)$. Either u will start drinking before all the bottles he needs are dedicated to him, or u will first have all the bottles he needs dedicated to him and then have R2 continuously enabled and start drinking. ∎

5. LOW MESSAGE COMPLEXITY

Let philosophers u and v share bottle b, and consider two consecutive drinking periods from b. If both periods are performed by the same philosopher, then no messages are sent for the second period. If the first period is performed by u and the second by v, then exactly *two* messages are sent for the second period: one transmission of req_b (by v) and one transmission of b (by u). Thus, a drinking session by a philosopher who needs k bottles requires between zero and $2k$ messages.

In Chandy and Misra's solution [1], there is a dining layer that runs concurrently with the drinking layer. Upon becoming thirsty, a philosopher also becomes hungry and needs *all* the forks associated with his incident edges. Thus, a drinking session by a philosophers who has d neighbors and needs k bottles ($k \leq d$) may require up to $2d$ messages for obtaining the forks. Some of these messages may still be in transit even after drinking was completed. Our "single layer" solution does not require these messages. In addition, it does not have complications due to coordinating the drinking and dining layers [7].

We conjecture that the message complexity of our solution is optimal in a model where philosophers have no knowledge about the durations of tranquil periods and drinking sessions. In such a model, a thirsty philosopher who needs a bottle he does not hold must notify his neighbor about his need, and a bottle release should occur no later than one drinking session after a notification is received.

6. EXTENSIONS TO THE BASIC PROBLEM

The first extension to the drinking philosophers problem defined in section 2 is to associate with each edge in G multiple, identical instances of a bottle rather than one instance. A philosopher may need some (but not necessarily all) instances upon becoming thirsty. We call this extension *the multiple-instance extension.*

Consider two neighboring philosophers u and v who share $quantity(b)$ instances of bottle b. We provide each philosopher with his own request token for instances of b. req_{bu} is the request token used by philosopher u and req_{bv} is the request token used by philosopher v. Upon becoming thirsty, u sets s_num_u to a value higher than max_rec_u, and if u does not hold as many instances of bottle b as he needs, he sends the message $(req_{bu}, need_u(b), s\_num_u, u)$ to v, where $need_u(b)$ indicates the total number of instances of b needed by u. Note that $need_u(b)$ is not the additional number of instances that u needs, but the total number he needs. When v receives a request (req_{bu}, n, s, u), he obeys the following *conflict resolution rule*:

> if $n \leq quantity(b) - need_v(b)$ or (v is thirsty and $(s, u) < (s\_num_v, v)$), then v immediately releases as many instances of b as the additional number of instances needed at u.

This conflict resolution rule is a generalization of the rule given for the basic algorithm (in section 3). Note that it is exactly the same rule when $quantity(b)$=1.

We are interested in minimizing the number of messages per drinking session. Therefore, we require that v will release instances of b to u only once before u starts drinking. For this reason, v will release instances to u only at a time when he can give u all the additional instances needed at u. If v does not release instances of b immediately in response to u's request, then he will release them once he completes drinking. When releasing instances of b, v returns req_{bu} to u.

The key point is that when v decides to release instances of b to u, v knows exactly how many additional instances are needed at u and v holds at least that number of instances. The release can occur at one of the following times: (1) when u's request arrives at v and $quantity(b)$ is no less than the sum of the total needs of u and v; or (2) when u's request arrives at v and $quantity(b)$ is less than that sum, but u has priority and v is not drinking yet; or (3) when v completes drinking. At any of these times there are no instances of b in transit between the two philosophers. When no instances are in transit, v can determine the number of instances held by u (it equals $quantity(b)$ minus the number of instances held by v). Knowing the total number of instances needed by u, v can determine the additional number of instances needed by u.

When v receives u's request, there are no instances of b in transit from v to u. If u has priority over v, then there are also no instances in transit from u to v and v holds at least as many instances of b as the number of additional instances needed at u. If v has priority over u, then v either already holds as many instances as the additional need of u or v is thirsty, waiting for instances from u. In the latter case, v will eventually drink and upon drinking completion will have enough instances to release to u.

We now give the algorithm for the multiple-instance extension. The variables maintained by philosopher u are as follows: s_num_u and max_rec_u remain as in the basic algorithm (see section 3); $need_u(b)$ and $hold_u(b)$ are now integers indicating number of instances; $in\_hand_u(req_{bu})$ indicates whether u holds req_{bu}; $in\_hand_u(req_{bv})$ indicates whether u holds req_{bv}, where v is the neighbor with whom u shares instances of b; $hold\_neighbor_u(b)$ indicates the number of instances of b that u knows v to hold, and $need\_neighbor_u(b)$ indicates the total number of instances u knows v to need.

Initially, every philosopher u is tranquil and $s\_num_u = max\_rec_u = 0$. Given $quantity(b)$ instances of bottle b shared between two philosophers, some are with one philosopher and the rest are with the other. Each philosopher holds his own request token for instances of b.

Each philosopher u obeys the rules given in Table 2 below. As in section 3, each rule execution is considered an atomic action and messages regarding instances of a bottle are sent to the philosopher with whom they are shared.

R1': when tranquil and "want to drink" do
become thirsty;
$s\_num_u \leftarrow max\_rec_u + 1$;
for each desired bottle b do
$need_u(b) \leftarrow$ the number of instances of b needed

R2': when thirsty and holding enough instances of all needed bottles do
become drinking

R3': when drinking and "want to stop drinking" do
become tranquil;
for each consumed bottle b do
$[need_u(b) \leftarrow 0$;
if $in\_hand_u(req_{bv})$ then
$[hold\_neighbor_u(b) \leftarrow quantity(b) - hold_u(b)$;
$release_u(b) \leftarrow need\_neighbor_u(b) - hold\_neighbor_u(b)$;
Send ($release_u(b)$ instances of bottle b , req_{bv});
$hold_u(b) \leftarrow hold_u(b) - release_u(b)$;
$in\_hand_u(req_{bv}) \leftarrow$ false]]

R4': when $need_u(b) > hold_u(b)$ and $in\_hand_u(req_{bu})$ do
Send(req_{bu}, $need_u(b)$, s_num_u, u);
$in\_hand_u(req_{bu}) \leftarrow$ false

R5': upon reception of (req_{bv}, n, s, v) do
$in\_hand_u(req_{bv}) \leftarrow$ true;
$max\_rec_u \leftarrow \max(max\_rec_u, s)$;
$need\_neighbor_u(b) \leftarrow n$;
if $n \leq quantity(b) - need_u(b)$
or (thirsty and $(s, v) < (s\_num_u, u)$) then
$[hold\_neighbor_u(b) \leftarrow quantity(b) - hold_u(b)$;
$release_u(b) \leftarrow need\_neighbor_u(b) - hold\_neighbor_u(b)$;
Send ($release_u(b)$ instances of bottle b , req_{bv});
$hold_u(b) \leftarrow hold_u(b) - release_u(b)$;
$in\_hand_u(req_{bv}) \leftarrow$ false]

R6': upon reception of (i instances of bottle b , req_{bu}) do
$hold_u(b) \leftarrow hold_u(b) + i$;
$in\_hand_u(req_{bu}) \leftarrow$ true

Table 2. Multiple-Instance Algorithm: Rules for philosopher u

The liveness proof of the above algorithm is similar to the proof given in section 4 for the basic algorithm, since the use of extended session numbers for resolving conflict is identical in both algorithms. The additional component in the extension's proof involves correctness of the calculations performed by a philosopher determining the number of instances to be released (in rules R3' and R5'). This was intuitively justified earlier. The message complexity of the multiple-instance algorithm is identical to that

of the basic algorithm. That is, at most two messages are transmitted per drinking from instances of a bottle.

Additional Extension. A second extension, referred to as *the multiple-type extension,* is to associate with each edge in G several types of bottles with multiple instances of each type. A philosopher may need instances from some of the types for a drinking session.

The solution to the multiple-type extension is based on the multiple-instance extension. When philosopher u needs additional instances of certain types of bottles he shares with his neighbor v, he notifies v of the total number of instances he (u) needs for each of these types. Philosopher v will obey a conflict resolution rule similar to the one provided for the multiple-instance extension. He will release all the additional instances needed at u in a single message, either immediately upon receiving u's request or upon drinking completion. In this extension too, the message complexity will remain as in the basic algorithm. A detailed discussion of both extensions appears in [4].

7. DISCUSSION

The drinking philosophers problem and its extensions model various paradigms of multiple mutual exclusion. We showed in this paper that resolving conflicts by using session numbers extended with ids provide simple, elegant and efficient solutions to these problems.

The emphasis in our paper is on simplicity, concurrency and minimization of message complexity. A recent work by Styer and Peterson [10] concentrated on minimization of chains of waiting processes. In their solution, as well as in Lynch's solution [6], collection of several "write locks" by a process may not be done concurrently. In addition, the message complexity of their solution is higher than ours. The tradeoffs between concurrent collection of locks, message complexity and the length of "waiting chains" is an interesting question for further research.

REFERENCES

[1] Chandy, M. and Misra, J., "The drinking philosophers problem," *ACM Trans. Prog. Lang. Syst.*, Vol. 6, 4, Oct. 1984, pp. 632-646.

[2] Chandy, M. and Misra, J., "An example of stepwise refinement of distributed programs: quiescence detection," *ACM Trans. Prog. Lang. Syst.*, Vol. 8, 3, July 1986, pp. 326-343.

[3] Dijkstra, E.W., "Two starvation free solutions to a general exclusion problem," EWD 625.

[4] Ginat, D., "Decentralized ordering of contending processes in a distributed system," PhD Thesis, Computer Science Dept., Univ. of Maryland, in preparation.

[5] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, Vol. 21, 7, July 1978, pp. 558-564.

[6] Lynch, N.A., "Fast allocation of nearby resources in a distributed system," *Proc. of the 12th ACM Symp. on Theory of Computing*, 1980, pp. 70-81.

[7] Murphy, S.L. and Shankar, A.U., "A note on the drinking philosophers," *ACM Trans. on Prog. Lang. and Syst.*, Vol. 10, No. 1, pp. 178-188, January 1988.

[8] Ricart, G., and Agrawala, A.K., "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, Vol. 24, 1, Jan. 1981, pp. 9-17.

[9] Shankar, A.U., and Lam, S.S., "Time-dependent distributed systems: proving safety, liveness and real-time properties," *Distributed Computing*, Vol. 2, No. 2, Springer-Verlag, 1987.

[10] Styer, E. and Peterson G.L., "Improved algorithms for distributed resource allocation," *Proceedings of The Seventh Annual ACM Symposium on Principles of Distributed Computing*, August 1988, pp. 105-116.