ON THE ADVANTAGES OF FREE CHOICE:

A SYMMETRIC AND FULLY DISTRIBUTED SOLUTION TO THE DINING PHILOSOPHERS PROBLEM (Extended Abstract)

by
Daniel Lehmann
and
Michael O. Rabin*

Mathematics Institute
Hebrew University
Jerusalem, Israel

Abstract

It is shown that distributed systems of probabilistic processors are essentially more powerful than distributed systems of deterministic processors, i.e., there are certain useful behaviors that can be realized only by the former. This is demonstrated on the dining philosophers problem. It is shown that, under certain natural hypotheses, there is no way the philosophers can be programmed (in a deterministic fashion) so as to guarantee the absence of deadlock (general starvation). On the other hand, if the philosophers are given some freedom of choice one may program them to guarantee that every hungry philosopher will eat (with probability one) under any circumstances (even an adversary scheduling). The solution proposed here is fully distributed and does not involve any central memory or any process with which every philosopher can communicate.

1. Introduction

Since the notion of a probabilistic algorithm was introduced in [10], the idea has been used in different fields to provide algorithms which are more efficient than the deterministic algorithms known to solve the same problem. Recently ([11] and [12]), the second author has applied the same idea to some problems of concurrency control and cooperation for distributed systems. His results will appear elsewhere. We present here an application of this idea to the dining philosophers problem and exhibit a probabilistic solution for this problem which guarantees, with probability one, that every hungry philosopher eventually gets to eat. We feel this application is interesting in many ways.

* Michael Rabin is Visiting Professor of Computer Science and Vinton Hayes Senior Fellow at Harvard University, Aiken Computation Lab., Cambridge, Mass. 02138, for the 1980-81 academic year.

Concurrent programming seems to be a field particularly well suited to probabilistic algorithms, and the idea of an operating system built out of probabilistic processes and which performs correctly with a high probability is very attractive.

For the first time, it provides an example of a problem which can be solved by probabilistic processors but provably cannot be solved by deterministic processors.

As we shall see later, the system of probabilistic processors may be proved to behave correctly with probability one and not just with a probability which is as close to one as one likes.

The realm of proofs of correctness for concurrent processes is not yet well known. As the reader will realize proofs of correctness for probabilistic distributed systems are extremely slippery; in fact the proof presented here (hopefully correct) is only the last one in a sequence of incorrect proofs. We hope that the present exercise will provide some hints as to what the important concepts are and what could be the adequate formal systems for such proofs. In particular, we have occasion to introduce in Section 7 an ordering in time of complex activities of processes in a distributed system. It turns out that it is impossible to define a natural total ordering on these activities. But, the activities of processes which share external variables are totally ordered in time, and this property suffices for our proof of correctness. We hope that this methodology concerning the handling of time will be useful in other contexts as well.

Our analysis of the interplay between probabilistic ideas and the area of large systems of simple processors suggests an application to the theory of biological systems where all three features of randomness, large number of components and simplicity of those components appear.

The protocols presented here for the synchronization of the philosophers are the first really distributed solution to the dining philosophers problem (N. Francez and M. Rodeh [4] have, concurrently, proposed such a solution written in the language CSP [6]; in their solution the processes do not use random draws). We feel that our protocols provide an elegant solution to the problem and that the ideas presented here should be useful to solve other problems in the area of concurrency control and cooperation between asynchronous processes.

2. The Dining Philosophers Problem

In [3] E. Dijkstra proposed a problem in concurrent programming which has, since then, been considered as a classical problem, the dining philosophers problem. This problem is interesting not so much on account of its practical importance but because it is a paradigm for a large class of concurrency control problems.

The problem will be presented first informally and then in a more precise way. Suppose a number of philosophers are sitting around a circular table. The life of a philosopher consists mainly of thinking. When a philosopher thinks, she does not interact with her colleagues. But, from time to time, it may happen that a philosopher gets hungry from too much thinking. He then wants to eat from a bowl of food, which a kind benefactor maintains full and which is located in the middle of the table. To eat he needs two chopsticks (wisdom is essentially oriental in this tale). He will then try to pick up the two chopsticks which are closest to him. One chopstick is located at his right, just in the middle between him and his right neighbor (there is only one chopstick between any two adjacent philosophers) and another one is located between him and his left neighbor. In all, there are only as many chopsticks as there are philosophers. A philosopher may only pick up one chopstick at a time and obviously he cannot pick up a chopstick which is already in the hand of a neighbor. If a hungry philosopher cannot eventually get both chopsticks adjacent to him, for example, if each time he tries to pick up a chopstick it happens to be in the hand of the appropriate neighbor, then the philosopher starves. If a hungry philosopher gets to hold both his chopsticks at the same time, he eats (without releasing his chopsticks), eventually satisfies his material needs and puts down both his chopsticks.

In a more precise way a philosopher goes indefinitely through a cycle: thinking, trying, eating and so on ad infinitum. To eat a philosopher needs exclusive access to two resources each of which is shared with a neighbor. A philosopher may die only while he is thinking.

The problem is to describe a system of protocols for the philosophers, chopsticks and possibly some other entities, which will behave in the way indicated above (especially that at any time each chopstick is in at most one hand) and which will ensure that, with varying degrees of strength, philosophers will be able to eat. Thus we will be talking, not of one, but of several problems and solutions.

3. Constraints and Properties of the Solutions

We shall now describe some constraints on the class of solutions we are willing to consider. Such constraints will be justified both by aesthetic and practical considerations.

Our first constraint is that we are interested only in truly distributed systems, i.e., systems in which there is no central memory or central process to which every other process may have access. Indeed, we are looking for systems in which the only active agents are the philosophers who do not communicate directly with each other and in which chopsticks are represented by passive cells (memories) which may be accessed only by the two adjacent philosophers. N. Francez and M. Rodeh have recently considered this constraint and remarked that none of the solutions published so far (e.g., [1], [2], [3], [5], [6], [7] and [8]) satisfied this criterion. Indeed, they all use some kind of central scheduler which regulates the eating of the philosophers. Francez and Rodeh [4] propose a solution which is truly distributed, in the language CSP [6].

A second constraint that we impose is that all philosophers be identical (we could call this the layman's point of view). This is a very natural assumption if we think of a very large number of very simple philosophers, so simple, in fact, that they could not even possibly remember an identification number, personal to each philosopher (assume we have more than n philosophers, each of whom can hold fewer than logn bits). We also restrict our attention to initial configurations which are symmetric: we assume that, in the beginning, all philosophers are in the same state and all shared variables hold the same value. If it were not for such an assumption, it would be easy to code different protocols for the different philosophers in the initial values of the shared variables (or the initial states of the different philosophers); protocols similar to those described in Section 7 would also do the job if the initial values of the shared variables are favorable. In short, we are interested in large distributed systems of simple identical processors.

Our goal is to find protocols for the philosophers which will satisfy the two constraints explained above and will allow the philosophers to eat. If the system is such that every hungry philosopher eventually gets to eat, then we shall say that it is lockout-free. We shall exhibit such a system. But as a first step we will build a system which enjoys only a weaker non-starvation property: if, at any time, there is a hungry philosopher, say Plato, then at some later time some philosopher (not necessarily Plato) will eat. A system which enjoys this property will be said to be deadlock-free. In other words, a deadlock-free system only guarantees that not all hungry philosophers starve.

A word on our assumptions concerning the synchronization of the different processors. We are not assuming anything concerning the respective rates of activity of the different philosophers, or the overall scheduling. We may not, therefore, exclude the possibility of an adversary scheduler who would, for example, do his best to keep Plato from eating, by awaking him only when one of his neighbors is eating. We allow this adversary scheduler to make use of all information about the system, including the result of random draws performed by processes, the values of the shared variables and the value of the private variables of each protocol. This is an extremely severe assumption which ensures that the protocols presented here have very strong correctness properties. Other works, in different situations, make less severe assumptions ([4] and [11]). This adversary scheduler, though, is not allowed to use information about the results of future random draws. We have to allow for the possibility of an adversary scheduler because we assume that the interactions between philosophers that we describe are only the visible part of an iceberg of complex relations which we do not know about and are not willing to study. This is also a very sound principle of system design: we are to assume that the worst is certain.

We use variables shared by two adjacent philosophers and assume that both philosophers will never access a common variable of theirs exactly at the same time (they have exclusive access to the variable) and that every philosopher that requests access to a variable will eventually get access to it. In other words, we assume that the problem of synchronizing the access of a number (here, only two) of processes to a shared variable is solved. The justification for this assumption is that the length of time taken by an atomic action (reading, writing or reading and writing) is very small compared with the rate of activity of a philosopher. Therefore the density of accesses is very small and we may assume either that conflicts do not occur, or that they are taken care of by the hardware. To fix ideas, we are assuming that we are dealing with four different orders of magnitude of time slices. The smaller one is that of the atomic action on a variable (only the shared variables are of interest), say, a nanosecond. The second one is the rate of activity of a philosopher: the idle time between each activation which is, say, of the order of a millisecond. The third is the time needed for a meal which is, say, of the order of a second. A philosopher is therefore willing to suffer a number of failures before he may eventually get to eat. The last one is the lifetime of the system itself which we may assume to be of the order of hours. Since, at the previously mentioned rates, each process participating in an actual system will perform many millions of atomic actions, the properties we are about to claim for our systems in terms of unending computations, in practice apply also to actual systems.

We are therefore justified in assuming that a philosopher may, in one move and without risk of being disturbed by or of disturbing a neighbor, check that a chopstick is down on the table and pick it up. As will be seen later the picking up and putting down of a chopstick will be expressed by a change in the value of a shared variable.

4. Deterministic Solutions

A very simple argument will now show that there is no solution to the problem, satisfying the constraints mentioned above, in which the philosophers are deterministic processes.

Theorem 1. There is no deterministic, deadlock-free, truly distributed and symmetric solution to the dining philosophers problem.

Proof. Suppose there is a deterministic, truly distributed and symmetric solution. We shall define a scheduler which will allow no philosopher to eat, showing in this manner that no such solution may be deadlock-free. For the proof's sake, let us number the philosophers in cyclic order from 1 to n (this is an external naming and the philosophers themselves are not aware of their own name). The scheduler will activate each philosopher for a single atomic action in the order 1 to n, then repeat another similar round in which the philosophers 1 to n are activated in turn, and so on. The claim is that, if the configuration is symmetric with respect to all philosophers at the beginning of a round, then the configuration will again be symmetric at the end of the round. Full details of the proof will appear elsewhere.

The deterministic solution proposed by Francez and Rodeh in [4] seems to contradict the claim we just made. The solution to this apparent contradiction is that there is no truly distributed deterministic implementation of CSP. L. Lamport [9] seems to have been the first to notice this fact, and Theorem 1 above, together with the CSP protocols proposed in [4], constitutes a formal proof of this fact. Thus any truly distributed implementation of CSP must be probabilistic and, in such an implementation, even terminating programs (for the semantics of CSP) terminate only with probability one. Such a probabilistic implementation of CSP is proposed in [4].

5. The Free Philosopher's Algorithm

The gist of our idea is the following: since the problem with any deterministic solution is the symmetry which could keep recurring, we need a way to break this symmetry. We shall incorporate free choice into the protocols of the individual philosophers, letting the laws of probability ensure that, with probability one, the symmetry will be broken.

We propose the following process for each philosopher. In the program below, the function R is the reflection function on {Right,Left}.

```
while true
 do think;
   do trying:=true or die od;
   while trying
     do draw a random element s of {Right,Left};
           ***with equal probabilities ***
        wait until s chopstick is down
          and then lift it;
        if R(s) chopstick is down
          then lift it;
            trying:=false
          else
            put down s chopstick
        fi
     od;
   eat;
   put down both chopsticks
     *** one at a time, in an arbitrary order ***
 od.
```

Definition. A *schedule* S, for n philosophers, is a function which assigns to every past behavior of the philosophers, the philosopher Pi whose turn is next to be active, i.e., to perform an atomic action. Under past behavior up to any given time, we mean the complete sequence of atomic actions and random draws with their results, up to that time.

Following [11], for us a schedule is not merely a fixed sequence of activations but, rather, is a mapping which makes the next action depend on the whole past behavior. This captures the idea that, for any specific system, what will happen next depends on the whole history of past successes and failures of the processes to gain access to shared resources, as well as on what has happened internally within the processes. Unlike [11], we do include under past history the results of random draws already made.

For a given schedule S and specific outcomes of the random draws D (D is an infinite sequence of elements of the set {Right,Left}), we get a particular computation C = COM(S,D), which is an infinite sequence of atomic actions. Note that a computation is unending and embodies the total life-span of the system. We shall use the term finite computation to denote a finite sequence of atomic actions. The ith element of a computation C is the atomic action which takes place at time i. Note that we assume that no two atomic actions take place exactly at the same time in C; this restriction could be easily lifted to allow atomic actions of different processes, as long as they do not concern the same shared variables, to take place exactly at the same time.

Definition. A computation C is proper if, in C, every philosopher is activated an infinite number of times. A schedule S is called proper if, for every sequence D of outcomes of the random draws, the computation COM(S,D) is proper.

It follows from the explanations found after the program, that, if a schedule S is proper then, in every possible computation C = COM(S,D), no philosopher quits while trying to eat, eating, or exiting.

On the space of all possible outcomes of random draws D we impose the uniform probability distribution. The function COM then associates with every schedule S a probability distribution on the space of all computations, the probability of a set E of computations being defined as the probability of the set of sequences of random draws D such that COM(S,D) is in E.

In the sequel we shall make no assumption on S, except that it is proper. Our theorems, thus, ensure that certain properties hold for every individual proper schedule. We do not assume a probability distribution on the space of schedules.

Our goal is to show that, in the system of the free philosophers, a deadlock may occur only with probability zero. We shall first define precisely the events in question.

Definition. A deadlocked computation C is a computation for which there exists a point t in time, at which at least one philosopher is trying to eat, but after which no philosopher eats. A philosopher Pi is locked-out (or starving) in a computation C, if there exists a time t at which Pi is trying to eat, and after which Pi never eats.

For a fixed proper schedule S, the event of being a deadlocked computation has a well defined probability (the proof is left to the reader). Denote DL(S) = Pr(D : C=COM(S,D) is deadlocked). We actually want to prove that, for every proper schedule S, DL(S) = 0.

The following lemmas refer to two philosophers, Plato and Aristotle, where Plato is seated next and to the left of Aristotle.

Lemma 1. If Plato picks up a chopstick an infinite number of times but Aristotle picks up a chopstick only a finite number of times, then, with probability one, Plato eats an infinite number of times.

The exact meaning of the lemma is that the event of Plato eating an infinite number of times has probability one relative to the event described in the hypotheses. The claim is meaningful only for those schedules which attach a positive probability to the hypotheses, and it should be understood that the lemma applies only to those schedules. Proofs for this and the following lemmas are omitted.

Lemma 2. In a deadlocked computation, every philosopher picks up a chopstick an infinite number of times, with probability one.

Lemma 3. Let F be a finite computation consisting of t steps, such that, at time t, both Plato and Aristotle are trying to eat, Plato's last random draw was Left and Aristotle's last random draw was Right. Consider all (infinite) computations C which are continuations of F. Then, with probability not less than one half: at least one of Plato or Aristotle picks up a chopstick just a finite number of times in C, or at least one of them gets to eat in C, after his last random draw in F and no later than two random draws after his last random draw in F.

To each time instant there corresponds a configuration of latest random draws. We shall say that a configuration, A, and a later configuration B, are disjoint if each philosopher has, between A and B, performed a random draw.

Lemma 4. If every philosopher picks up a chopstick an infinite number of times, and if, at time t, the configuration of last random draws is A, then there will arise, with probability one, a later configuration B, disjoint from A, in which some philosopher's last random draw is Left while his right neighbor's last random draw is Right.

We now get to the main theorem concerning the free philosophers.

Theorem 2. For every proper schedule S, DL(S) = 0.

Proof. We shall prove the theorem by contradiction. Assume that DL(S) > 0. We may then talk about the probability of events relative to a deadlock. By Lemma 2, with probability one (relative to the event of deadlocked computation), every philosopher performs an infinite number of random draws. By Lemma 4, there will arise, with probability one, an infinite sequence of disjoint configurations of last random draws satisfying the hypotheses of Lemma 3: say A0, A1,...,An... . By Lemma 3, some philosopher eats between An and An + 2, for every n, with probability one. We have shown that, relative to the event of deadlocked computations, non-deadlocked computations have probability one. We conclude that a deadlock may occur only with probability zero. QED

6. Lockouts are Possible

As indicated in the introduction one would like a lockout-free system. It may be shown that the system proposed above is not lockout-free.

Theorem 3. The system of the free philosophers is not lockout-free.

C.A.R. Hoare [5] has proposed a measure of the quality of a solution to the dining philosophers problem: the size of the longest chain of starving philosophers that may occur. Though it is possible that the protocols proposed above are quite satisfactory in practice, we shall show that a schedule may, with probability one, starve all but one philosopher.

Theorem 4. For the system of n free philosophers, there is a schedule which starves, with probability one, $n-1$ philosophers.

The previous theorem throws light on why the proof of Theorem 2 had to be delicate. No local reasoning would succeed in showing that one of a chain of philosophers sitting next to each other will get to eat.

We shall now offer another solution which guarantees that, with probability one, there will be no lockout, i.e, nobody will starve.

7. The Courteous Philosopher's Algorithm

The possibility for lockouts demonstrated in Section 6 is due to the fact that a philosopher Pi may be discourteous enough to pick up his neighbor's chopstick (on line 6), even if that neighbor is trying to eat and Pi has alread eaten *after* his neighbor's most recent meal. By using additional values for the variables shared by neighboring philosophers, we can ensure courteous behavior and obtain a lockout-free system. The courteous philosopher is defined by the following process.

```
var left-signal,right-signal : {On,Off};

*** Left-signal is shared with left neighbor.  ***
*** It is initially set to Off and is set to On ***
*** when one becomes hungry and restored to    ***
*** Off only after eating.                     ***
*** The left neighbor may read it but not      ***
*** change it.                                 ***
*** He refers to it as right-neighbor-signal.  ***
*** Symmetrically for right-signal.            ***

read only var left-neighbor-signal,right-neighbor
signal : {On,Off};

*** Left-neighbor-signal is left neighbor's    ***
*** right signal.                              ***

var left-last,right-last : {Left,Neutral,Right};

*** left-last is shared with left neighbor and ***
*** both may change it.                        ***
*** It indicates who ate last : left from chop-***
*** stick or right from chopstick.  It is      ***
*** initially on Neutral.                      ***
*** Left-last is the same as left neighbor's   ***
*** right-last.                                ***
```

```
1  while true
2   do think;
3     do trying:=true
4          left-signal:=On; right-signal:=On
5      or die
6     od;
7     while trying
8      do
9        draw a random element s of {Right,Left};
           *** with equal probabilities ***
10       wait until   s chopstick is down
11                 and
12                   (  s-neighbor-signal = Off
13                   or s-last = Neutral
14                   or s-last = s)
15         and then lift s chopstick
16       if R(s) chopstick is down
17         then lift it;
18           trying:=false
19         else
20           put down s chopstick
21       fi
22      od;
23    eat;
24    left-signal:=Off; right-signal:=Off;
25    left-last:=Right; right-last:=Left;
26    put down both chopsticks
        *** one at a time in any order ***
27  od.
```

The following proof requires an ordering in time of the meals of the philosophers. It turns out that while there is no immediate natural way to define a global order on the meals, we are able to say when a philosopher's meal preceded or followed his neighbor's meal. This local ordering suffices for our proof. The methodology used here for dealing with time in systems of concurrent processes may be useful, with appropriate modifications, in other contexts.

When a philosopher eats he goes through the following sequence of actions: picking up a first chopstick (line 15), picking up a second chopstick (line 17), setting his left-last variable to Right (line 25), setting his right-last variable to Left (line 25), and putting down both his chopsticks (line 26). If, while performing the sequence described above, a philosopher picks up his second chopstick (line 17) at time t1 and puts down the first of the chopsticks he releases at time t2, we shall say that his corresponding meal-interval is $[t1,t2]$ (this implies $t1 < t2$).

Definition. We shall say that meal-interval $[t1,t2]$ precedes meal-interval $[t3,t4]$ ($[t1,t2] < [t3,t4]$) if $t2 < t3$.

Remark 1. The relation "precedes" is antireflexive and transitive.

Remark 2. If $[t1,t2]$ is a meal-interval of Plato, then between time t1 and time t2, Plato is the only philosopher who has access to Plato's left-last and right-last variables (since Plato holds both his chopsticks during this interval of time and no philosopher ever changes his "last" variables unless he holds both his chopsticks) and therefore at time t2 Plato's left-last variable is equal to Right and his right-last variable to Left.

Remark 3. If, in a computation C, $[t1,t2]$ is a meal-interval of Plato and [t3,t4] a meal-interval of Plato and [t3,t4] a meal-interval of Aristotle, then either $[t1,t2] < [t3,t4]$ or $[t3,t4] < [t1,t2]$ (since a philosopher has both his chopsticks in hands during any meal-interval of his, and no two neighbors may each have both their chopsticks in hands at the same time).

Theorem 5. If S is a proper schedule for a system of courteous philosophers, then, with probability one, a computation $C = COM(S,D)$ is lockout-free.

8. Conclusions

The solution to the dining philosophers problem presented here suggests an approach to the general question of programming methodology which seems to be opposed to the prevalent one, as illustrated in particular in [1]. There, the reproducible behavior of programs is advocated as a necessary condition for debugging and it is claimed that, for systems to be reliable, they must be built out of components which themselves have a reproducible behavior. Here the reliability of the system is guaranteed even though the component processes may have a totally irreproducible behavior.

References

[1] Brinch Hansen, P. Operating Systems Principles. Prentice-Hall 1973.

[2] Brinch Hansen, P. Distributed processes, a concurrent programming concept. CACM 21, 11 (November 1978).

[3] Dijkstra, E.W. Hierarchical ordering of sequential processes, Operating Systems Techniques, Academic Press 1972.

[4] Francez, N. and Rodeh, M. A distributed abstract data type implemented by a probabilistic communication scheme. I.B.M. Israel Scientific Center TR-080, April 1980 (to be presented at 21st Annual Symposium on F.O.C.S., Syracuse Oct. 1980).

[5] Hoare, C.A.R. Towards a theory of parallel programming, Operating Systems Techniques, quoted above.

[6] Hoare, C.A.R. Communicating sequential processes. CACM 21, 8 (August 1978).

[7] Holt, R.C., Graham, G.S., Lazowska, E.D., and Scott, M.A. Structured concurrent programming with operating systems applications, Addison-Wesley 1978.

[8] Kaubisch, W.H., Perrot, R.H., and Hoare, C.A.R. Quasiparallel programming. Software and Experience, Vol. 6 1976, pp. 341-356.

[9] Lamport, L. Private communication, 1978.

[10] Rabin, M.O. Theoretical impediments to artificial intelligence, Information Processing 74 (Jack L. Rosenfeld ed.), pp. 615-619.

[11] Rabin, M.O. N-process synchronization by 4.logN-valued shared variable, Technical Report Forschungsinstitut fuer mathematik, ETH Zuerich, March 1980, 21st Annual F.O.C.S. Symposium (1980).

[12] Rabin, M.O. The choice coordination problem Memorandum No. UCB/ERL M80/38, Univ. of Calif. Berkeley, August 1980.