Reprinted from INFORMATION AND COMPUTATION
All Rights Reserved by Academic Press, New York and London

Vol. 95, No. 1, November 1991
Printed in Belgium

Fault-Tolerant Critical Section Management in Asynchronous Environments

AMOTZ BAR-NOY*

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598

DANNY DOLEV

IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 and
Computer Science Department, Hebrew University, Jerusalem, Israel

DAPHNE KOLLER[†]

Computer Science Department, Stanford University, Stanford, California 94305

AND

DAVID PELEG[‡]

Department of Applied Mathematics, The Weizmann Institute, Rehovot 76100, Israel

The paper deals with the problem of managing a fault-tolerant critical section in a completely asynchronous distributed network. The existence of a solution to this problem should be contrasted with a basic result of Fischer, Lynch, and Paterson, proving that in a completely asynchronous network, "nontrivial agreement" cannot be achieved even when only a single "benign" processor failure is possible. We present solutions to several versions of the critical section problem in this model. Denote by t the maximum number of possible faulty processors. Processors are allowed to fail while in the critical section, and therefore the critical section must have at least $t+1$ slots. In the case where the slots are identical we present two algorithms which require $t+1$ slots. The first is very simple, but requires every non-faulty processor to use the critical section infinitely often. The second solution allows non-faulty processors to quit. For distinct slots we present an algorithm that requires $2t+1$ slots. © 1991 Academic Press, Inc.

* This work was carried out while this author was visiting Stanford University. Supported in part by a Weizmann fellowship, by Contract ONR N00014-88-K-0166, and by a grant of Stanford Center for Integrated Systems.

[†] This work was carried out while this author was a student in the Computer Science Department, Hebrew University, Jerusalem.

[‡] Part of this work was carried out while this author was visiting Stanford University. Supported in part by a Weizmann fellowship, by Contract ONR NOOO14-88-K-0166, and by a grant of Stanford Center for Intregrated Systems.

0890-5401/91 $3.00

1. Introduction

An important issue in the theory of distributed systems is the extent to which processor cooperation and coordination can be achieved in the presence of faults. There are several parameters influencing this question. The first major parameter is the level of synchronism that exists in the system. A basic result [FLP] states that in a completely asynchronous system, a collection of $n \geqslant 3$ processors cannot deterministically achieve "nontrivial consensus" in a faulty environment, even if at most one processor may fail, and even when this can only be a *benign fail-stop* fault (i.e., a faulty processor may only stop functioning completely at some stage). This result and later stronger versions of it [DDS] characterize agreement as a "possibly too powerful" goal, and force us to limit ourselves to weaker forms of processor cooperation, hoping that these will be sufficient for executing various common tasks within such a system.

In this paper we study the ability to achieve weak forms of cooperation in a completely asynchronous message passing evironment. The paper deals with various algorithms for handling a basic task that requires a certain degree of processor cooperation—a controlled access to a shared resource. This task is sometimes called *critical section management.* Sometimes it is necessary to achieve *mutual exclusion* for accessing the resource, i.e., at most one processor can be in the critical section at any time. This goal is obviously unachievable when processors may fail while inside the critical section. We consider an extension of the problem in which there are several copies or slots of the resource, and the number M of such slots bounds the number of processors allowed to concurrently access the resource. This models a common situation in parallel operating systems [PS, Ru], and was introduced first in [FLBB].

Note that most previous studies of the critical section problem assumed a shared memory (cf. [R]) and that no processor fails in the critical section. Failure within a critical section was studied in [DGS]. Recently, other achievable goals in faulty asynchronous message passing networks received some attention (cf. [ABDKPR, BW, BMZ, DLPSW, K]).

When possible access methodologies for a *multi-slot* resource are considered, there are two viable alternatives. One approach in designing the access algorithms asserts that a processor's responsibility is limited only to ensuring itself the right to enter the critical section, and it is not required to locate and secure itself a particular slot. This approach allows processors to view the critical section as a "black box," containing equal, externally indistinguishable slots. A more demanding approach requires the processor to be responsible for the entire assignment process, including finding itself a specific slot and making sure that this slot is not occupied by any other processor at the same time. Here, the processor views each slot as a distinct entity although all the slots might be functionally equivalent and interchangeable; i.e., a system with a number of identical servers, where the process has to choose a specific server. This is a common situation in operating systems.

The two approaches can be illustrated by considering the different procedures of buying a ticket for a bus ride or a flight. In the first case, the passenger needs only to make sure that there is a room on the bus, but not to reserve a particular seat. In the second case, it is necessary to have a seat assignment before boarding the aircraft. This seemingly insignificant distinction turns out to have a considerable influence on complexity and algorithmic issues.

The *identical-slot critical section* problem (*identical CS*, for short) can be formalized by imposing the following three requirements:

1. *Exclusion*: At most M of the processors are in the CS at any given time.
2. *Non-starvation*: Every non-faulty processor that wants to enter the CS eventually succeeds.
3. *Fairness*: If a non-faulty processor p enters the CS, then it is among the first M processors according to a given priority rule.

Note that the fairness requirement alone does not prevent non-starvation. It could be the case that a processor has priority to access the CS but still cannot do this. In the sequel two priority rules are discussed in detail. Both rules are based on $\# p$, the actual number of times that p has ever used slots of the CS. This is a natural criterion in an asynchronous system, where more widely-used criterions (e.g., which process attempts to enter the critical section first) are very difficult to formalize. These rules imply the following two variants of the identical CS problem:

- *The Global Identical CS*: The variant based on the rule that processor p has higher priority than processor q whenever $\langle \# p, p\rangle < \langle \# q, q\rangle$.[1]
- *The Transient Identical CS*: The variant based on the rule that processor p has higher priority than processor q only if $\langle \# p, p\rangle < \langle \# q, q\rangle$, p wants to enter the CS, and furthermore, q knows that fact.

An apparent limitation of the first rule is that it forces non-faulty processors to use the critical section infinitely often; if M non-faulty processors stop entering the critical section at some stage, then some time afterwards they will reach the highest priority and deadlock the system. This limitation

[1] Throughout the paper, whenever we compare two tuples $\langle a_1, ..., a_m\rangle$ and $\langle b_1, ..., b_m\rangle$, we assume a lexicographical ordering with the first component being the most significant.

does not exist with the second rule. On the other hand, when the second rule is used sometimes a processor can effectuate its priority only after some other processors know that it wants to enter the critical section.

A third variant of the problem is the *distinct-slot critical section* problem (*distinct CS*, for short). In this variant there are M distinguished slots $1, ..., M$, and the exclusion property is replaced by:

1′. *Distinction*: No two processors are simultaneously in the same slot.

Solving these versions of the critical section problems when all the processors are non-faulty is not difficult. In this paper we solve them in the presence of faulty processors, where the fault model assumed is *fail-stop*. In this model, a faulty processor may suddenly stop functioning, regardless of the state it is in. In particular, a processor may fail while in a critical section, as well as in the process of sending messages. Some of the algorithms can be extended to worse kinds of faults, such as Byzantine faults, where the faulty processors may be malicious and even collude to prevent a correction solution.

Throughout, n denotes the number of processors in the system and t is a prescribed upper bound on the number of faulty processors. These values are known to all the processors and the processors are named $1, ..., n$.

Solutions to the CS problems should strive to minimize M. However, there may be as many as t faulty processors in the system, and each of them might stop functioning while inside the critical section. The asynchronous model implies that one cannot distinguish between a faulty processor and a slow one. Therefore, any algorithm needs at least $t+1$ slots in order to prevent starvation. This proves:

PROPOSITION 1.1. *Any algorithm for either of the critical section problems requires $M \geqslant t+1$ slots.*

Clearly, if $M \geqslant n$ we can dedicate a distinct slot to each processor and trivially meet all the requirements. Thus Proposition 1.1 is complemented by the following:

PROPOSITION 1.2. *There exists an algorithm for either of the critical section problems using $M = n$ slots.*

In the case of the Byzantine fault model, we can prove a stronger lower bound on M. According to Proposition 1.1, there are at least $t+1$ slots which are open to competition to all processors. If we assume that a faulty processor can enter the critical section without any of the other processors being aware of the situation, then the t faulty processors can secretly take up t additional slots, thus creating a situation where there are $2t+1$ processors in the critical section at the same time. This proves:

PROPOSITION 1.3. *In the presence of Byzantine faults, any algorithm for either variant of the identical critical section problem requires* $M \geqslant 2t+1$ *slots.*

A few of the algorithms presented for the identical CS problem overcome Byzantine faults, as they are with t additional slots. We note that the bound on the number of non-faulty processors in the critical section remains the same in both environments. The t additional slots are required for the faulty processors only.

Another consideration besides the number of required slots is the amount of memory and communication used by the algorithm. The definition of fairness ncessitates the usage of an internal memory proportional to the number of processors and the number of times that the processors have accessed the critical section. Nevertheless, it does not impose the use of messages of that size. The issue of message complexity will not be discussed in this paper.

Let us now list the results presented in the paper. For the global identical CS problem, our basic algorithm requires $M=t+1$, matching the above lower bound. This algorithm is very simple and the size of the messages is only one bit. The algorithm overcomes Byzantine faults as well, using $2t+1$ slots. In this algorithm each processor is required to maintain information about the usage of the critical section by all the other processors.

Next, we present an algorithm for the transient identical CS problem. In this algorithm each processor p stores locally only $\# p$, the number of times it has used the critical section, and collects additional information only when it wants to use the CS. This algorithm also requires $M=t+1$ slots, but the size of its messages is proportional to the number of times processors have accessed the critical section. Another, simpler, algorithm for the transient identical CS problem, which requires $M=2t+1$ slots is described. The later overcomes Byzantine faults using $3t+1$ slots.

For the distinct CS case we provide an algorithm that uses $M=2t+1$ slots. This algorithm again requires the processors to access the critical section infinitely often and use internal memory and message size proportional to the numbers $\# p$. Also, there is still a gap between our upper and lower bounds for M in this problem, as we cannot prove any lower bound other than that of Proposition 1.1.

The general strategy for solving the distinct CS problem is somewhat similar to that presented in the renaming algorithm of [ABDKPR], where selecting a slot is analogous to deciding a new name. A major difference between the two problems is that in the renaming problem, the entire process is done only once, and processors do not change their names once they decide on them. The need to repeatedly access the CS and the non-starvation requirement prevent us from using the solution to the renaming problem.

The system model is the standard asynchronous model [FLP, DDS]. Each processor has a message buffer modeled as an unordered set; sending a message to processor p is represented as appending the message to p's buffer. In each step the processor either receives or sends messages, but not both (i.e., we assume non-atomic receive/send). When receiving, it reads some arbitrary (possibly empty) subset of the messages in its buffer; when sending, it can only transmit a message to a single processor. There are no restrictions or assumptions on the order in which messages are received, nor are there any restrictions on the order in which processors take steps, except that each none-faulty processor takes an infinite number of steps during any infinite run. In addition, every message sent from a non-faulty processor to another non-faulty processor will eventually be received.

Section 2 describes the solution for the global identical CS problem. The two algorithms for the transient identical CS problem are presented in Section 3. The solution for the distinct CS problem is given in Section 4.

2. The Global Identical CS Problem

This section presents a simple algorithm named **GICS** for the global identical CS problem with $M = t + 1$. The result matches the lower bound of Proposition 1.1. The algorithm requires every processor to attempt entering the critical section infinitely often, in order to guarantee non-starvation.

The main idea of our algorithm is that upon leaving the CS, a processor sends a one-bit message notifying the others. Each processor p maintains a vector C_p of counters. The qth entry of this vector, $C_p(q)$, accumulates the number of notifications received by p from every other processor q. The following simple fact makes these estimates useful by relating them to the actual number of times (denoted by $\# q$) each processor q entered the critical section.

Fact 2.1. For every two processors p, q, at any time,

1. $C_p(q) \leqslant \# \equiv q$, and
2. $C_p(p) = \# p$.

In every counter vector, C_p, the processors q are ordered dynamically by the pairs $\langle C_p(q), q \rangle$. Note that this is a *logical* ordering, not a physical one. We refer to it as the *local ordering* of p. The *local rank* of a processor

q in a vector C_p (in this local ordering) is denoted by $R_p(q)$. Also, the *global rank* of a processor q in the global ordering of the pairs $\langle \#q, q \rangle$ is denoted by $R_{\#}(q)$.

COROLLARY 2.1. *For every processor p, at any time, $R_p(p) \geqslant R_{\#}(p)$.*

Proof. Suppose that $R_{\#}(p) = i$. This means that there are exactly $i-1$ other porcessors q such that $\langle \#q, q \rangle < \langle \#p, p \rangle$. By Fact 2.1, for each such processor q,

$$\langle C_p(q), q \rangle \leqslant \langle \#q, q \rangle < \langle \#p, p \rangle = \langle C_p(p), p \rangle.$$

This completes the proof since then necessarily $R_p(p) \geqslant i$. ▮

These relationships imply that the local estimates made by a processor p about its global rank are conservative, in the sense that it always ranks itself no lower than its real place. Thus if the estimates maintained by p indicate that its rank is $t+1$ or less, then it can safely enter the critical section.

Let us now give a formal description of the algorithm.

ALGORITHM GICS. /* For a processor p */

```
1. /* Initialization */
   Create a vector $C_p$ of length $n$. Set each entry to 0.
2. /* An attempt to enter the CS */
   (a) If you receive a message "1" from $q$, then $C_p(q) \leftarrow C_p(q) + 1$.
   (b) If $R_p(p) \leqslant t + 1$ then goto 3.
3. /* Entering the critical section */
   (a) Enter the critical section.
   (b) Upon leaving the CS:
       send "1" to everyone; $C_p(p) \leftarrow C_p(p) + 1$; and goto 2.
```

In order to prove the correctness of the algorithm we need to show that the algorithm guarantees exclusion, non-starvation, and fairness.

LEMMA 2.1 (Exclusion). *In every run of algorithm* **GICS**, *at most $t+1$ processors are in the critical section at any given time.*

Proof. Assume to the contrary that $t+2$ or more processors are present in the critical section at a certain time, in some run of the algorithm. Let p be the processor with the largest global rank $R_{\#}(p)$ among these processors at that time. Necessarily $R_{\#}(p) \geqslant t+2$. It follows from Corollary 2.1 that also $R_p(p) \geqslant t+2$. Hence p should not have entered the critical section; a contradiction. ▮

LEMMA 2.2 (Non-starvation). *In every run of algorithm* **GICS** *every non-faulty processor enters the critical section an infinite number of times.*

Proof. Assume, seeking to establish a contradiction, that starvation has occurred in some run of the algorithm and let p be the processor with the minimal global rank $R_{\#}(p)$ among the starved processors. Eventually, for every non-starved non-faulty processor q, $\#q > \#p$, because every non-starved non-faulty processor uses the critical section infinitely often. At some later time the appropriate notifications reach p and are reflected in C_p, i.e., $C_p(q) > C_p(p)$ for every non-starved non-faulty processor q. There are at most t faulty processors whose messages may not reach p from some point on. Therefore the local rank of p, $R_p(p)$, eventually becomes $t+1$ or smaller, and p should enter the CS; a contradiction. ▮

LEMMA 2.3 (Fairness). *In any run of algorithm* **GICS**, *if a processor p enters the critical section, then its global rank satisfies $R_{\#}(p) \leqslant t+1$.*

Proof. If p enters the CS then its local rank satisfies $R_p(p) \leqslant t+1$. By Corollary 2.1 this is true also globally (i.e., $R_{\#}(p) \leqslant t+1$). ▮

Theorem 2.1 follows from Lemmas 2.1, 2.2, and 2.3.

THEOREM 2.1. *Algorithm* **GICS** *solves the global identical CS problem with $t+1$ slots.*

Note that if $M = 2t+1$, this algorithm is correct even when the faulty processors are malicious, as long as a non-faulty processor can always identify the immediate sender of any message it receives.

Though this algorithm achieves our definition of fairness, it is only weak fairness, as it does not ensure that processors enter the critical section in the right order. Under this definition, a processor can wait arbitrary long (though finite) amount of time, while processors with lower priority enter the critical section. A slightly stronger notion of fairness requires that if a non-faulty processor p enters the CS, then every processor q with higher priority than p enters the CS when it receives all the messages in transit for it. Note, that under this definition there might be also some time in which the fairness is not perfect. It is possible that a processor enters the CS before some other with higher priority, but this is unavoidable in a completely asynchronous system. The GICS algorithm can easily be extended to achieve this notion of fairness, using simple message forwarding. This extension is only valid in a fail-stop fault model.

3. The Transient Identical CS Problem

3.1. *The $M = t + 1$ Algorithm*

Algorithm **GICS**, presented in the previous section, has two main drawbacks. First, it requires every processor to try to enter the critical section infinitely often, in order to guarantee non-starvation. Second, each processor has to handle every message it receives. The correctness of the algorithm depends heavily on a processor's updating its data structure upon receiving every message. Without this update it cannot reflect the state of other processors. Algorithm **TICS**, described below, solves the CS problem with the transient fairness property and does not have these drawbacks.

In algorithm **TICS**, processors that do not want to access the CS are asked only to reply by sending some *acknowledge* message, and do not need to maintain any information about other processors. The algorithm requires $t + 1$ slots. Whenever a processor intends to use the CS, it registers itself by sending an appropriate message to every processor. Only processors that at present want to use the CS need to keep track of how many times each processor has visited the CS. Every other processor stores only the number of times it has previously visited the CS.

In the previous algorithm **GICS**, whenever a processor finds itself ranked $t + 1$ or less in the global ordering of the pairs $\langle \# q, q \rangle$, it may safely enter the critical section. The transient rule for fairness does not allow us to use such a simple criterion. A processor needs to inform others that it intends to access the CS. Similarly, before entering the CS, it has to make sure that no processor of higher priority has changed its state. Thus, the process of entering the CS is composed of two rounds of acknowledgment collection. This process is best described by identifying special states through wich the processor has to go. Each processor is initially in PASSIVE state. A processor p that wishes to enter the CS first changes its state into REGISTERING and sends announcements informing all other processors of its wish. It then has to await acknowledgements for its announcement. These acknowledgments enable p to collect information regardig other processors' states. It switches into the state TRYING when it finds itself ranked $t + 1$ or less among the processors that want to enter the CS. Upon entering state TRYING, p has to start a second round of sending announcements and awaiting acknowledgements. If, while collecting these acknowledgements, p learns of any higher priority processor that changed its state, it has to return to state REGISTERING and go through the entire process once again. The delicate part of the algorithm is to guarantee the Exclusion Property.

Let us now give a slightly more formal definition of the various states and messages used in the algorithm. Every processor can be in one of four states:

- PASSIVE (not interested at the moment)—encoded by 3.
- REGISTERING (to enter the CS)—encoded by 2.
- TRYING (to enter the CS)—encoded by 1.
- ACCESSING (at present in the CS)—encoded by 0.

There are two types of messages sent by processors. Announcement messages of the form "$\langle S, c \rangle$," where S is the current state of the sender and c is its counter, or acknowledgment messages of the form "$\langle S, c, S', c' \rangle$" as a reply to an announcement message "$\langle S', c' \rangle$," where S and c are defined as above.

During any run of the algorithm processors may send the same announcement message more than once. Therefore, they need to be able to associate each acknowledgment with the appropriate announcement in order to recognize when an acknowledgment to the current announcement is received. This can be achieved by either adding a counter to messages, or assuming FIFO on the lines and counting the acknowledgments received. It can also be solved by transmitting an announcement only after the acknowledgment to the previous announcement is received. Applying the last method to the algorithm does not require storing all outstanding announcements; it is sufficient to remember the last one. Throughout the algorithm we assume that one of these methods is applied. Hence, a processor eventually receives an acknowledgment to its last announcement from any non-faulty processor.

While a processor p attempts to enter the CS, it maintains three vectors, K_p, S_p and C_p, each of length n, containing information about the other processors. The vector CD_p is as in the previous section. The qth entry indicates whether q has acknowledged knowing that p is in a REGISTERING or TRYING state (encoded by $K_p(q)=1$), or such an acknowledgement has not arrived p yet (encoded by $K_p(q)=0$). Throughout the run of the algorithm, each processor maintains information about itself (even when it is in state PASSIVE). The initial values are $K_p(p)=1$, $S_p(p)=3$, $C_p(p)=0$. Thus, every processor starts in a PASSIVE state with a zero counter.

Denote by DB_p the *database* that processor p holds, i.e., the above three vectors. In every database DB_p the processors q are ordered dynamically by the quadruples

$$\langle K_p(q), S_p(q), C_p(q), q \rangle.$$

The rank of a processor q in a database DB_p (in this ordering) is denoted by $R_p(q)$.

Each processor is instructed by the algorithm to respond to certain messages arriving while it is in certain states, but is allowed to ignore these messages while being in other states. Consequently, the description of the algorithm prefixes each instruction by the states in which that instruction is applicable.

```
ALGORITHM TICS /* For a processor $p$ */

1.  /* Initialization */
    Create vectors $K_p$, $S_p$ and $C_p$ of length $n$. $K_p(p) \leftarrow 1$; $S_p(p) \leftarrow 3$;
    $C_p(p) \leftarrow 0$.
2.  In every state:
    /* acknowledgements and book-keeping */
    if you receive "$\langle s, c \rangle$," from $q$ then
    (a) Send "$\langle s, c, S_p(p), C_p(p) \rangle$" to $q$.
        if not in state PASSIVE and $c \geqslant C_p(q)$ (not an old message) then
        $C_p(q) \leftarrow c$; $S_p(q) \leftarrow s$.
3.  In state PASSIVE:
    if you want to enter the CS then
    (a) Change your state to REGISTERING ($S_p(p) \leftarrow 2$).
    (b) Send "$\langle S_p(p), C_p(p) \rangle$" to every processor.
    (c) For every processor $q$ initialize the vectors:
        $K_p(q) \leftarrow 0$; $S_p(q) \leftarrow 0$; $C_p(q) \leftarrow -1$.
4.  In state REGISTERING:
    (a) If you receive "$\langle s, c, s', c' \rangle$" from $q$ such that $s = S_p(p)$ and
        $c = C_p(p)$, then $K_p(q) \leftarrow 1$; $C_p(q) = c'$; $S_p(q) = s'$.
    (b) If $R_p(p) \leqslant t+1$ then
     i.   Change your state to TRYING ($S_p(p) \leftarrow 1$).
     ii.  For every $q$, $K_p(q) \leftarrow 0$.
     iii. Send "$\langle S_p(p), C_p(p) \rangle$" to every processor.

5.  In state TRYING:
    (a) If you receive "$\langle s, c, s', c' \rangle$" from $q$ such that $s = S_p(p)$ and
        $c = C_p(p)$, then $K_p(q) \leftarrow 1$; $C_p(q) \leftarrow c'$; $S_p(q) \leftarrow s'$.
    (b) If an announcement message was received from some $q$ such that
        $\langle C_p(q), q \rangle < \langle C_p(p), p \rangle$, then
         i.   Change your state to REGISTERING ($S_p(p) \leftarrow 2$).
         ii.  For every $q$, $K_p(q) \leftarrow 0$.
         iii. Send "$\langle S_p(p), C_p(p) \rangle$" to every processor.
    (c) If $R_p(p) \leqslant t+1$ then
         i.  Change your state to ACCESSING ($S_p(p) \leftarrow 0$).
         ii. enter the CS.
6.  In state ACCESSING:
    upon leaving the CS:
```

(a) Change your state to PASSIVE ($S_p(p) \leftarrow 3$).
(b) $C_p(p) \leftarrow C_p(p) + 1$.
(c) Send "$\langle S_p(p), C_p(p) \rangle$" to every processor.

LEMMA 3.1 (Exclusion). *In every run of algorithm* **TICS** *at most $t+1$ processors are in the critical section at any given time.*

Proof. Assume to the contrary that there is a set Z of $t+2$ processors in the critical section at a certain time in some run. Let p be the last processor from this set that changed its state from REGISTERING to TRYING before accessing the critical section. Since p accessed the critical section, there must be a processor q in the set Z such that according to the data in p's vectors just before switching from TRYING to ACCESSING

$$\langle K_p(p), S_p(p), C_p(p), p \rangle < \langle K_p(q), S_p(q), C_p(q), q \rangle.$$

As $K_p(p) = 1$ we conclude that $K_p(q) = 1$ and $S_p(q) \geqslant S_p(p) = 1$. On the other hand, $S_p(q)$ was extracted by p from an acknowledgement sent by q. This acknowledgment was sent in response to an announcement sent by p after switching into state TRYING (in Step 4(b)). Since p was the last to change its state into TRYING, it follows that q was already TRYING or ACCESSING, i.e., $S_p(q) \leqslant 1$. Thus, necessarily $S_p(q) = S_p(p) = 1$. Hence it should be the case that $\langle C_p(p), p \rangle < \langle C_q(q), q \rangle$. But then if q had received p's announcement while being in state TRYING, the algorithm instructs q (in Step 5(b)) to change its state back to REGISTERING and retry. Thus if q is in the CS now, it must have switched back into TRYING after p had already done so, contradicting the assumption that p was the last to switch from REGISTERING to TRYING. ∎

LEMMA 3.2 (Non-Starvation). *In every run of algorithm* **TICS** *every non-faulty processor that wants to enter the critical section eventually succeeds.*

Proof. Assume to the contrary that starvation has occurred in some run of the algorithm. Let p be the non-faulty processor with the smallest pair $\langle \# p, p \rangle$ among the starved processors. Eventually, for every non-faulty processor q, $\langle C_p(q), q \rangle$ will be greater than $\langle C_p(p), p \rangle$. When this happens, p will no longer return from state TRYING to state REGISTERING, and therefore will access the CS after all the non-faulty processors acknowledge its trying announcement; a contradiction. ∎

LEMMA 3.3 (Fairness). *In every run of algorithm* **TICS**, *if a non-faulty processor p enters the CS, then at the time p enters the critical section, there* *is a slot available for every processor with higher priority that wants to use the critical section.*

Proof. If $\langle \# q, q\rangle < \langle \# p, p\rangle$ and q wants to enter the CS and p knows that, then by definition q has higher priority than p. If $S_p(q) \leqslant S_p(p)$, then q appears before p in p's database, and p takes q into account (and leaves it a slot) when it decides to enter the CS. If $S_p(q) > S_p(p)$, then since q is not in state PASSIVE, necessarily p's state is TRYING. But then when p gets q's announcement, it will return to state REGISTERING (Step 5(b)), which reduces to the first case. ▮

Theorem 3.1 follows from Lemma 3.1, 3.2, and 3.3.

THEOREM 3.1. *Algorithm* **TICS** *solves the transient identical CS problem with* $t+1$ *slots.* ▮

3.2. *The* $M = 2t+1$ *Algorithm*

In algorithm **TICS**, state TRYING is necessary because the CS has only $t+1$ slots. In the case where $M = 2t+1$, one can implement the transient rule for fairness without state TRYING, i.e., with only one round of announcements and acknowledgements. The necessary modifications involve canceling Steps 4(b)(ii), 4(b)(iii), and 5 of the algorithm; in Step 4(b)(i), instead of entering state TRYING, the processor directly switches into state ACCESSING. We refer to this modified algorithm as algorithm **TICS**-1.

In order to prove that algorithm **TICS**-1 is correct, it suffices to prove the exclusion property. The proofs for the non-starvation and fairness properties remain as for algorithm **TICS**.

LEMMA 3.4 (Exclusion). *In every run of algorithm* **TICS**-1 *at most* $2t+1$ *processors are in the critical section at the same time.*

Proof. Assume to the contrary that there are $2t+2$ processors in the critical section at a certain time in some run. Construct the following directed graph over the set of the processors that are at the critical section. The directed arc $\langle p, q\rangle$ is in the graph if in p's database $\langle K_p(p), S_p(p), C_p(p), p\rangle < \langle K_p(q), S_p(q), C_p(q), q\rangle$. It is impossible that in this graph the arcs $\langle p, q\rangle$ and $\langle q, p\rangle$ occur together (but it might be that there is no arc between p and q).

Each procesor draws at least $t+1$ outgoing arcs from itself, otherwise it cannot enter the CS. Therefore, there exists at least one processor with indegree at least $t+1$ which should prevent it from entering the CS; a contradiction. ▮

Theorem 3.2 follows from Lemmas 3.4, 3.2 and 3.3.

THEOREM 3.2. *Algorithm* **TICS**-1 *solves the identical CS problem with* $2t+1$ *slots.* ▌

When $M=3t+1$, Algorithm **TICS**-1 is correct even when faulty processors are malicious, as long as a non-faulty processor can identify the immediate sender of any message it receives. Algorithm **TICS** cannot overcome Byzantine faults, because a faulty processor can force a non-faulty processor to continually retry entering the CS without success (i.e., switching between the states TRYING and REGISTERING).

4. THE DISTINCT CS PROBLEM

In this section we present an algorithm named **DCS** for the distinct CS problem using $M=2t+1$ slots. Following Proposition 1.2 we assume that $n>2t+1$. The set of slots is denoted by $S=\{1, ..., 2t+1\}$. Throughout the execution of the algorithm each processor p maintains three vectors X_p, J_p, and C_p, each of n entries, containing information about the system's status. The processors dynamically update their vectors by exchanging them with all the others. Specifically, the information kept by p is the following:

1. $X_p(q)$—a slot suggested by q.
2. $J_p(q)$—a running counter of suggestions.
3. $C_p(q)$—the number of times processor q has previously used the CS, according to p's knowledge.

Initially, the vectors held by p are set to the appropriate null values. Denote by DB_p the database that processor p holds, i.e., the above three vectors. In addition, p maintains a collection U_p of n databases, such that $U_p(q)$ is the last database that p has received from q, and $U_p(p)$ is p's current database.

In every database DB_p the processors q are ordered dynamically by the pairs $\langle C_p(q), q\rangle$. The rank of a processor q in a database DB_p (in this ordering) is denoted by $R_p(q)$. The set $\text{left}_p(q)$ contains all processors in DB_p with rank less than or equal to that of q (i.e., $\text{left}_p(q)=\{q' \mid R_p(q') \leqslant R_p(q)\}$).

DEFINITION 4.1. Suppose p holds the database DB_p. The database DB is a *supporting* database for DB_p if it contains identical information about all the processors in $\text{left}_p(p)$.

Since t processors might be faulty, a processor cannot expect to get messages from more than $n-t-1$ other processors. Thus, after receiving $n-t-1$ supporting versions of its database from other processors, it is useless to wait for more information (which might never arrive), and the processor should take some action. This observation leads us to define the notion of a *left-stable database.*

DEFINITION 4.2. A database DB_p is *left-stable with respect to* p in a given run of the algorithm if p has $n-t$ supporting databases in its collection of databases, U_p. The database DB is *left-stable* if it is left-stable w.r.t. some processor p.

The process of selecting a slot and entering the CS can be sketched as follows. A processor p is required to exchange information with other processors until it reaches a left-stable database DB_p, and then to suggest a slot based on this stable information. Again, p exchanges information with other processors until it reaches a left-stable database. Now p has to review its suggestion by checking whether it currently collides with suggestions made by other processors. If there are no collisions, the processor p decides on its slot and proceeds to enter the critical section. Otherwise, it has to suggest a new slot and repeat the whole process.

The general strategy of algorithm **DCS** is thus somewhat similar to that of the renaming algorithm of [ABDKPR], and selecting a slot is analogous to deciding a new name. A major difference between the two problems is that in the renaming problem, the entire process is done only once, and processors do not change their names once they decide on them. This simplifies the solution by allowing stabilization on the entire database. The need to repeatedly recompute stable databases while processors change their priorities every once in a while is responsible for the additional complication of having to consider only the "lower" part of the database.

We need a certain partial ordering on databases. This ordering reflects the accumulation of knowledge by the processors. Intuitively, $\mathrm{DB}_q > \mathrm{DB}_p$ means that DB_q is more updated than DB_p. The ordering is defined as follows.

DEFINITION 4.3. The information about processor r is more updated in DB_q than in DB_p, denoted by $\mathrm{DB}_q \geqslant_r \mathrm{DB}_p$, if

$$\langle C_q(r), J_q(r)\rangle \geqslant \langle C_p(r), J_p(r)\rangle.$$

In order to suggest a new slot, p should know all the slots that are suggested and that appear in any of these supporting databases. We define free(DB) for any database DB as the list of the slots that do not appear as suggestions in its slot-suggestions vector X, and free(p, U_p) as the list of the slots that appear in free(DB_q) in every supporting database DB_q that appears in the collection U_p.

```
ALGORITHM DCS /* For a processor $p$. */
1. /* Initialization */
   Construct an initial $\mathrm{DB}_p$ and $U_p$. Set all entries to 0.
2. /* A new attempt to enter the CS */
   (a) Send $\mathrm{DB}_p$ to every other processor.
   (b) $U_p(p) \leftarrow \mathrm{DB}_p$.
3. Wait until you receive a message $\mathrm{DB}_q$ from some processor $q$.
   (a) /* test if $\mathrm{DB}_q$ is more updated */
       i. $U_p(q) \leftarrow \mathrm{DB}_q$.
       ii. For every processor $r$ such that $\mathrm{DB}_q \geqslant_r \mathrm{DB}_p$:
           update $C_p(r) \leftarrow C_q(r)$; $J_p(r) \leftarrow J_q(r)$; $X_p(r) \leftarrow X_q(r)$.
       iii. $U_p(p) \leftarrow \mathrm{DB}_p$.
       iv. If $\mathrm{DB}_p$ has been modified, send it to every other processor.
   (b) /* $p$ tests if it has more support */
       If the number of supporting databases in $U_p$ is $n-t$,
       then goto 4 else goto 3.
4. /* $\mathrm{DB}_p$ is a left-stable database */
   If a slot $X_p(p)$, has previously been suggested, and this slot is different
   from any suggested slot $X_q(r)$ for any $r$ and any $q$ such that $\mathrm{DB}_q \in U_p$
   is a supporting database for $\mathrm{DB}_p$, then goto 5, else goto 6.
5. /* Entering the critical section */
   (a) Enter slot number $X_p(p)$ of the critical section.
   (b) Upon releasing this slot and leaving the CS:
       $X_p(p) \leftarrow 0$; $C_p(p) \leftarrow C_p(p)+1$; $J_p(p) \leftarrow 0$; and goto 2.
6. /* otherwise, needs to suggest a new slot */
   (a) If $R_p(p) > \min\{t+1, |\mathrm{free}(p, U_p)|\}$ /* no suggestion possible */
       then: $X_p(p) \leftarrow 0$; and goto 2.
   (b) $X_p(p) \leftarrow$ the $R_p(p)$th slot in free($p$, $U_p$).
   (c) $J_p(p) \leftarrow J_p(p)+1$.
   (d) Goto 2.
```

As in Section 2, in order to prove the correctness of the algorithm we need to show that distinction, non-starvation, and fairness properties are preserved.

LEMMA 4.1 (Distinction). *At most one processor is in slot number i at any given time.*

Proof. Assume to the contrary that there exists a time T such that processors p and q are in the same slot in the CS. The algorithm implies that $X_p(p) = X_q(q)$, where X_p (respectively, X_q) is the vector of slots suggestions held by p (respectively, q) when deciding to enter the critical section. Let U_p and U_q be the sets of databases they respectively maintained when they decided to enter the CS. The assumption $n > 2t$ implies that

$(n-t)+(n-t)>n$. Therefore, there exists a processor r such that $U_p(r)$ is in the set of the $n-t$ supporting databases of p and $U_q(r)$ is in the set of the $n-t$ supporting databases of q. Let T_p and T_q be the times at which r sent $U_p(r)$ and $U_q(r)$, respectively. Without loss of generality assume that $T_p \leqslant T_q \leqslant T$. Processor p did not change its suggestion $X_p(p)$ between time T_p and T (otherwise, $U_p(r)$ would not be counted as a supporting database). Therefore, $X_p(p)$ appears in $U_p(r)$ by time T_p and on. The definition of T_q implies that $X_p(p)$ appears as the suggested slot of p in $U_q(r)$ and, hence, $X_q(q)$ could not have passed the test in step 4 of the algorithm; a contradiction. ▮

LEMMA 4.2 (Non-starvation). *Every non-faulty processor enters the critical section an infinite number of times.*

Proof. We prove the claim by assuming the opposite and deriving a contradiction. Given an infinite run, we classify the processors of P as follows. Let P_g be the set of non-faulty processors that access the critical section infinitely often. Let P_1 be the set of non-faulty processors p that enter the critical section only a finite number of times during the run (i.e., reach a final value $\# p$), but get infinitely many left-stable vectors afterwards. Together, these two sets form the collection of active processors, $P_a = P_g \cup P_1$. Further, let P_2 be the set of non-faulty processors p that reach a final value of $\# p$ and obtain only a finite number of left-stable vectors during the run. Let P_f denote the set of processors that become faulty during the run. These two sets form the collection of passive processors, $P_b = P_f \cup P_2$. See Fig. 1.

P_b passive processors		P_a active processors	
P_f faulty processors	P_2 final $\#p$ and finite left-stable vectors	P_g access the CS infinitely often	P_1 final $\#p$ and infinite left-stable vectors

FIG. 1. The partition of processors.

The contradiction assumption assumes the existence of a run in which $P_1 \cup P_2 \neq \emptyset$. From some point on, all the databases DB_p held by the processors satisfy the following properties:

1. All processors q in P_b have reached their final $C_q(q)$ value, obtained their last left-stable database and made a suggestion based on it (hence their entries to not change afterwards).

2. All processors q in P_1 have reached their final $C_q(q)$ value.

3. For every processor $q \in P_g$, $C_q(q)$ is larger than any of the final $C_r(r)$ values of the processors r in $P_b \cup P_1$.

Hereafter we refer to every database with these properties as a *limit database*. Note that for all limit databases DB_p, the rank $R_p(q)$ of any processor $q \in P_b \cup P_1$ is fixed. We refer to these ranks as *limit ranks*. In particular, let p_0 be the processor whose limit rank R_0 is the smallest in $P_1 \cup P_2$. For any limit database DB_p of a non-faulty processor, the subdatabase $\mathrm{left}_p(p_0)$ (all processors in DB_p with rank less than or equal to that of p_0) is fixed and contains p_0 and possibly some processors from P_f. Since $|P_f| \leqslant t$ it follows that $R_0 \leqslant t+1$.

CLAIM 4.1. $p_0 \in P_1$.

Proof. Assume to the contrary that $p_0 \in P_2$. Consider the point of time by which all databases held by the processors are limit databases. In all these databases, only processors from P_f might appear to the left of p_0, and the information on these processors does not change. Therefore, at some later point p_0 will obtain a left-stable database again; a contradiction. ∎

Let Y_b denote the set of final slots suggested by the passive processors in P_b, and let $Y_a = S - Y_b$. Intuitively, Y_a is the set of slots into which the active processors of P_a continuously attempt to enter.

CLAIM 4.2. $|Y_a| \geqslant R_0$.

Proof. Assume $|Y_a| < R_0 \leqslant t+1$. Since $|P_f| \leqslant t$ and $|S| = 2t+1$, some slots must be suggested by some processors of P_2. Let $p \in P_2$ be the processor with the smallest limit rank among those processors in P_2 whose final state includes a suggestion $X_p(p) \neq 0$. Since in all limit vectors the rank of every active processor in P_a is at least R_0, according to step 6(a) no suggestions will be made and eventually all these processors will set their suggestions to 0 and never change it. Therefore sometime later p will obtain yet another stable vector; a contradiction. ∎

Assume that Y_a is ordered, and let $Y_a = \{y_1, y_2, ...\}$. For every limit database DB and for every slot $y \in \text{free(DB)}$, denote by $f(y)$ its index in free(DB). Clearly $f(y_i) \leqslant i$.

There is a time after which every suggestion made by processors in P_a is based on a limit database. Hence, there is a later time at which p_0 holds a left-stable database DB_L in which every suggested slot was suggested based on a limit database.

CLAIM 4.3. *In every left-stable database* DB *obtained by* p_0 *after* DB_L, *either* $y_{R_0} \in \text{free(DB)}$ *or* y_{R_0} *is suggested only by* p_0.

Proof. Assume to the contrary that y_{R_0} appears in DB as a slot suggested by some $q \in P_a$, $q \neq p_0$. Then q suggested y_{R_0} according to some left-stable limit database DB_q. But then $f(y_{R_0}) \leqslant R_0$ in free(DB_q), so q could not have suggested it, as its rank in DB_q is strictly larger than R_0. ▌

Therefore, upon seeing DB_L, p_0 either decides immediately on y_{R_0} and enters this slot (in case y_{R_0} appears as its suggested slot in DB_L) or it suggests y_{R_0} now and decides it upon obtaining the next left-stable vector.

It follows that p_0 does enter the critical section once again, contradicting the assumption that p_0 has reached its final value of $\# p_0$. This completes the proof of Lemma 4.2. ▌

LEMMA 4.3 (Fairness). *If a processor p enters the critical section then its global rank satisfies* $R_{\#}(p) \leqslant t+1$.

Proof. The same as the proof of Lemma 2.3. ▌

Theorem 4.1 follows from Lemmas 4.1, 4.2 and 4.3.

THEOREM 4.1. *Algorithm* **DCS** *solves the distinct CS problem with* $2t+1$ *slots.* ▌

We do not have any lower bound for the number of slots needed for the distinct CS problem. The difficulties in constructing a better upper bound arise from the fact that processors cannot distinguish between slow processors and faulty processors. It seems that a processor must leave t slots for the faulty processors, in case they have higher priorities, and t slots for slow processors that might have higher priorities.

RECEIVED December 12, 1988; FINAL MANUSCRIPT RECEIVED March 1, 1990

REFERENCES

[ABDKPR] H. ATTIYA, A. BAR-NOY, D. DOLEV, D. KOLLER, D. PELEG, AND R. REISCHUK, Achievable cases in an asynchronous environment, *J. Assoc. Comput. Mach.*, to appear.

[BMZ] O. Biran, S. Moran, and S. Zaks (1988), A combinatorial characterization of the distributed tasks whcih are solvable in the presence of one faulty processor, in "Proc. 7th ACM Symp. of Principles of Dist. Computing," pp. 263–273.

[BW] M. F. Bridgland and R. J. Watro (1987), Fault-tolerant decision making in totally asynchronous distributed systems, *in* "Proc. 6th ACM Symp. of Principles of Dist. Computing," pp. 52–63.

[DDS] D. Dolev, C. Dwork, and L. Stockmeyer (1987), On the minimal synchronism needed for distributed consensus, *J. Assoc. Comput. Mach.* **34**, 77–97.

[DGS] D. Dolev, E. Gafni, and N. Shavit (1988), Toward a non-atomic era: *L*-exclusion as a test case, *in* "Proc. 19th ACM SIGACT Symposium on Theory of Computing," pp. 78–92.

[DLPSW] D. Dolev, N. A. Lynch, S. Pinter, E. Stark, and W. E. Weihl (1986), Reaching approximate agreement in the presence of faults, *J. Assoc. Comput. Mach.* **33**, 499–516.

[FLBB] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin (1979), Resource allocation with immunity to limited process failure, *in* "Proc. 20th Symp. on Foundations of Comp. Science," pp. 234–254.

[FLP] M. J. Fischer, N. A. Lynch, M. S. Paterson (1985), Impossibility of distributed consensus with one faulty processor, *J. Assoc. Comput. Mach.* **32**, 374–382.

[K] D. Koller (1986), "Token Survival: Resilient Token Algorithms," M.Sc. Thesis, Hebrew University.

[PS] J. L. Peterson and A. Silberschatz (1985), "Operating Systems Concepts," 2nd. ed., Chaps. 8, 9, 13, Addison–Wesley, Reading, MA.

[R] M. Raynal (1986), "Algorithms for Mutual Exclusion," North Oxford Academic Publishers.

[Ru] L. S. Rudolph (1981), "Software Structures for Ultra Parallel Computing," Ph.D. dissertation, Courant Institute, New York University, 1981.

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium