A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems

MUKESH SINGHAL, MEMBER, IEEE

Abstract—**This paper presents a heuristically-aided algorithm to achieve mutual exclusion in distributed systems which has better performance characteristics than the algorithm of Suzuki and Kasami [36], which in turn is an improvement over the well-known Ricart and Agrawala algorithm for mutual exclusion [33]. The proposed algorithm makes use of state information, which is defined as the set of states of mutual exclusion processes in the system. Each site maintains information about the state of other sites and uses it to deduce a subset of sites likely to have the token. Consequently, the number of messages exchanged for a critical section invocation is a random variable between 0 and n ("n" is the number of sites in the system). We show that the algorithm achieves mutual exclusion and is free from deadlock and starvation. We discuss the effects of a site crash and a communication medium failure on the proposed algorithm and suggest methods of recovery from these failures. We study the performance of the algorithm by a simulation technique (and by an analytic technique for low and heavy traffics of requests for critical section execution). The proposed algorithm also serves as an illustration of the effectiveness of heuristic techniques applied to problems where a system must satisfy strong constraints such as database consistency or mutual exclusion.**

Index Terms—**Consistency, deadlock, fairness, heuristic techniques, network partitioning, mutual exclusion, performance, starvation, token passing.**

I. Introduction

IN THE problem of mutual exclusion, concurrent access to a shared resource by several uncoordinated user requests is serialized to secure the integrity of the shared resource. It requires that the actions performed by a user on a shared resource must be atomic. That is, if several users concurrently access a shared resource, then the actions performed by a user, as far as other users are concerned, must be instantaneous and indivisible such that the net effect on the system is as if user actions were executed serially as opposed to in interleaved manner [25].

A distributed system consists of a collection of geographically dispersed autonomous sites, say S_1, S_2, $\cdots$, S_n, which are connected by a communication network. Enslow [12] has characterized a distributed system to contain the following five components: a *multiplicity* of general-purpose resources, a *physical distribution* of the resources, a *high-level operating system, system transparency,* and *cooperative autonomy.*

Manuscript received October 28, 1986; revised April 8, 1988. This work was supported by the Department of Computer and Information Science, The Ohio State University.

The author is with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

IEEE Log Number 8926765.

The problem of mutual exclusion arises at several places in distributed systems whenever concurrent access to shared resources is involved. For example, in directory management, where an update to a directory must be done atomically because if updates and reads to a directory proceed concurrently, reads may result in getting inconsistent information.

The problem of mutual exclusion becomes much more complex in distributed systems because of the lack of shared memory and common physical clock and because of unpredictable message delays. Over the last few years, several algorithms to achieve mutual exclusion in distributed systems have been proposed, e.g., [3], [5], [16], [19], [23], [26], [33], [34], [36]. These algorithms can be divided into two classes. The algorithms in the first class [19], [26], [33], [34] are assertion based and require two (or more) successive rounds of message exchanges among the sites. A site enters into its critical section (CS) only after an assertion becomes true at that site. The assertion has a property that it becomes true only at one site at any time; therefore, only one site can be in its CS at any time. For example, in the Ricart-Agrawala algorithm [33], the assertion is "A site has received a REPLY message for its REQUEST messasge from all other sites."

The algorithms in the second class [5], [23], [36] are token based where a unique token (also known as the PRIVILEGE message [36] or the ticket [23]) is shared among the sites such that possession of the token gives a site the authority to execute its CS. Singular existence of the token implies the enforcement of mutual exclusion in distributed systems. In the mutual exclusion algorithm of Suzuki and Kasami [36], which is an improvement over the widely known Ricart-Agrawala algorithm [33], if a site interested in executing its CS does not have the token, it broadcasts request messages for the token to all other sites. When the site holding the token receives the request message, it sends the token to the requesting site. Thus, this algorithm requires "n" messages per invocation of mutual exclusion as compared to $2*(n - 1)$ messages per invocation of mutual exclusion by the Ricart-Agrawala algorithm. The Suzuki-Kasami algorithm [36] is one of the best known mutual exclusion algorithms when message traffic and delay both are concerned.

Objective of the Paper: Previous distributed mutual exclusion algorithms share a common property that they are deterministic in the sense that they do not use state information of the system, which can be obtained by exchanging messages or by implicit means, to guide their actions. For example, in the mutual exclusion algorithms in [5] and [36], a site which is requesting its CS sends a request message for the token to all

other sites. In this paper, we present a distributed mutual exclusion algorithm which is an improvement over the mutual exclusion algorithm of Suzuki and Kasami [36] and which makes use of the state of the system to optimize its course of actions.

In the developed algorithm, each site maintains state information about the system. Whenever a site intends to invoke mutual exclusion, it uses a heuristic[1] to guess from its available state information what sites of the system are probably holding or are likely to have the token and sends token request messages only to those sites rather than to all the sites. Consequently, the proposed algorithm requires fewer messages per mutual exclusion execution as compared to the algorithm of Suzuki and Kasami [36] without sacrificing any delay in granting the CS. We carry out a performance study of the algorithm to illustrate the improvement in the performance which results due to the use of state information and the heuristic.

Effectiveness of heuristics techniques has been demonstrated in the problems which have stochastic nature such as distributed scheduling [6], [10], [11], [18], [38], [31], network routing [4], [28], multiaccess channels [27], [37], and distributed problem solving [2]. However, applicability of heuristic techniques to problems which must satisfy strong and sharply defined constraints, such as database consistency [13] or mutual exclusion [33], is yet to be shown. The proposed algorithm not only improves the performance of the Suzuki–Kasami algorithm, but also demonstrates that heuristic techniques can indeed be advantageously applied to deal with uncertainty in state information in the problems where systems must satisfy strong and sharply defined constraints.

The rest of the paper is organized as follows. In the next section, we develop background material. Section III contains a description of the proposed mutual exclusion algorithm. In Section IV, we discuss the correctness of the algorithm. Section V describes results on the performance of the algorithm. In Section VI, we discuss the consequences of communication medium and site failures on the algorithm and propose methods for recovery from these failures.

II. Preliminaries

State of the System: A site can be in the state of "requesting its CS," "executing its CS," "not requesting the CS," or "not requesting its CS but possessing the token" (also referred to as *holding the idle token*). The state of the system is the set of the states of all the sites. A site maintains arrays (called state vectors) to store the state of the sites of the system. Sites can disseminate state information by exchanging messages. Due to the reasons mentioned earlier, information in the state vectors (i.e., picture of the state of the system available to a site) may be partially out-dated, thus incorrect.

Since the algorithm relies upon the state information to guide its actions, fast dissemination of changes in system state is a desirable requirement. This is because fast dissemination of state information reduces the imprecision and uncertainty in the state information available to a site and thus enhances the credibility/success of its decisions. However, there is a tradeoff between the cost of fast dissemination of the state information and the cost of poor decisions made due to out-dated/incorrect state information available to sites. The cost of fast dissemination of state information is mostly in terms of more messages or larger size messages, and the cost of poor decisions can be degraded response to users (in case of load balancing), large message traffic (in case of mutual exclusion), long intersite communication delays (in case of network routing), etc. It is difficult to exactly quantify the tradeoff between these variables because solutions to the mathematical models of these systems are quite intricate.

In the algorithm, we implicitly pass the state information in the messages used to achieve mutual exclusion. That is, we do not exchange any additional messages to disseminate the state information. Rather, we use the messages required to request the token and the token itself to pass the state information. For example, when a site S_i receives a token request message from site S_j, S_i can infer that S_j is in the state of requesting the CS.

Complicating Factors: Although the proposed idea seems fairly simple, coming up with a heuristic and rules for exchanging the state information which guarantee fairness and liveness properties and which are efficient at the same time is nontrivial. This is because a site may heuristically select some sites for the purpose of sending request messages such that none of them has the token and none of them gets the token for a long time (or none of them may ever see the token). Or the sites can become divided into groups such that a site in a group sends token request messages only to the sites in the same group; consequently, all groups except the one which has the token, starve forever—a form of deadlock [8]. In order to be free from deadlock and starvation, the heuristic and the rules to update the state information should satisfy the following conditions.

1) If a site is not requesting its CS, then it should have information about at least one site which is requesting its CS.

2) If a site is waiting to execute its CS, then in finite time it should know the site which is executing its CS or a site which is requesting its CS and knows the site executing its CS.

3) A site should not retain the token indefinitely when some other site is requesting its CS.

III. Description of the Algorithm

We assume that sites of the system do not share any memory and communicate completely by message passing. Message propagation delay is finite but unpredictable and between any pair of sites, messages are delivered in the order they are sent. Initially, we assume that the underlying communication medium is reliable and sites do not crash. Consequences of communication medium and site failures and recovery are discussed later in Section VI.

A. Sequence Number

Sites use sequence numbers to distinguish between a current token request and an old delayed token request. Every site keeps a counter. When a site has to execute its CS, it

[1] Due to lack of perfectly synchronized physical clocks at sites and unpredictable and finite message propagation delays, state information at sites may be out-dated and inconsistent [24]. A heuristic is used to cope with the uncertainty in the state information.

TABLE I
STATES OF A SITE

State Symbol	State Description
R	The site is requesting the CS
N	The site is not requesting the CS
E	The site is executing the CS
H	The site is holding the idle token (i.e., the site has the token but is not requesting the CS)

increments its counter and sends the updated value (called "sequence number") in token request messages. Each site keeps a record of the highest known sequence number (along with the latest known state information) of each site. By comparing the sequence number in a received message with the latest known sequence number of its sender site, a site can determine if the message is carrying an obsolete request for the token.

B. States of a Site

Table I defines the states of a site.

C. Data Structure

Each site maintains two state vectors to store the state information of the system. The first vector, called state vector (SV), stores the latest known states of the sites. The second vector, called sequence number vector (SN), maintains the highest known sequence number for each site. Similarly, the token contains two vectors—one to store states (TSV) and another to store sequence numbers (TSN). Since entries of state vector TSV can take only values N and R, the state vector TSV can be a binary array where values 0 and 1, respectively, indicate states "not requesting" and "requesting."

Comments on Token Size: Note that the token, which consists of two vectors, is much larger than a token request message, which only consists of the id and current sequence number of a requesting site. In fact, most token-based mutual exclusion algorithms (an exception is the algorithm by Le Lann [22]) require a similar kind of data structure in the token. For example, in the token-based mutual exclusion algorithm of Ricart–Agrawala [5], the token contains an integer array to store sequence numbers of the sites (exactly the same as TSN of our case); in the Suzuki–Kasami algorithm [36], the token not only contains an integer array to store sequence numbers of sites (which is the same as TSN) but also contains a queue of the ids of the sites which are requesting the token (similar to TSV but of variable size). Note that queue size can vary from 0 to $n - 1$ depending upon statistical fluctuations and load of CS requests. In heavy load of CS requests, the size of the queue will be $n - 1$ most of the time because all the sites will be requesting the CS most of the time.

Since state vector TSV can be accommodated in n bits of space (recall that it can be implemented as a binary array), the token size in the proposed algorithm is somewhat (by n bits) larger than that of the Ricart–Agrawal algorithm [5] and is comparable to or smaller than that of the Suzuki–Kasami algorithm [36] (because n bits take less space that $n - 1$ integers). Note that in all the three algorithms, the token is considerably bigger than a token request message. However, the token is exchanged once roughly for every $n - 1$ exchanges of token request messages. Therefore, the percentage of bigger messages will be low. Furthermore, a measurement study done by Lazowska *et al.* [21] indicates that in a message exchange, the overhead of packaging and unpackaging data is much more than the overhead of transmitting the data over a communication network. Therefore, unless message size is very large (such that it requires special buffering techniques or its has to be broken into smaller packets), a large token message should not pose any serious problem in these algorithms.

D. Heuristic

The heuristic should be such that given locally available state information of the system, a site should be able to guess a set of sites which are likely to have the token. If the heuristic is highly focused, then the set is likely to be small which may result in a miss (we define a *miss* to be a situation where none of the chosen sites has the token). Otherwise, the probability of a miss will be low, but the efficiency (e.g., message traffic and delay in granting permission to enter the CS) is likely to suffer.

There are several heuristics to guess the location of the token which immediately come to mind. For example, in one obvious heuristic, a site sends/relays a token request message only to the site to which it forwarded the token last. However, it can be easily visualized that in this heuristic a token request message may end up in an endless chase of the token, and user response time will be poor due to serial propagation of the token request. We propose to use the following heuristic: "All the sites for which the state vector SV entries are "R," are in the probable set, and a requesting site sends request messages to all such sites."

The rationale behind the heuristic is that for a site S_i, SV[j] $= R$ if site S_j recently informed S_i of its intention to enter the CS. However, it should be noted that the success of a heuristic also depends upon the correctness/credibility of the state information it is operating on.

E. Dissemination of State Information

In the algorithm, sites do not require additional messages to disseminate state information and use the messages needed to achieve mutual exclusion to disseminate the state information. For example, when a site receives a request message from a site, it updates its state vector to reflect that the latter site is in the state of requesting the CS. Likewise, the token carries information about the state of the system and disseminates the information as it hops around among the sites—sites update their state vectors against the information in the token (see Section III-G-1).

F. Initialization

The system starts in the following initial state:

For a site S_i ($i = 1$ to n),
[SV[j] $= N$ for $j = n$ to i; SV[j] $= R$ for $j = i - 1$ to 1;
SN[j] $= 0$ for all j}
For token, TSV[i] $= N$ and TSN[i] $= 0$ for all i.
The token is initially at site S_1 and at S_1, SV[1] $= H$.

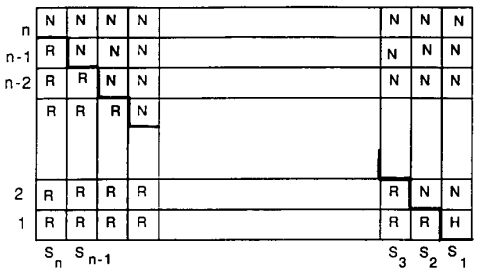

Fig. 1. Initial state of state vectors (SV's).

Thus, initially site S_i, $1 \leq i \leq n$, thinks that sites S_n, S_{n-1}, $\cdots$, S_{i+1} are not requesting the token and thinks that sites S_{i-1}, S_{i-2}, $\cdots$, S_1 are requesting the token and one of these sites has the token. If we stagger the state vectors SV of all the sites side by side, the initial system state looks as shown in Fig. 1.

In topological sense, "R" 's among the SV's of the sites form a staircase pattern—number of "R"s decreases in stepwise manner from left to right. If sites are arranged from S_n to S_1 in descending order, then each site thinks that all the sites to its left are in the state "N" while all the sites to its right are in the state "R." Conversely, for a site S_i, all the sites to its left (S_n, S_{n-1}, $\cdots$, S_{i+1}) think that S_i is in state "R" and all the sites to its right (S_{i-1}, S_{i-2}, $\cdots$, S_1) think that S_i is in state "N."

G. The Algorithm

Site S_i executes the following three steps to invoke mutual exclusion:

Step 1: If SV[i] $\neq H$, then
{SV[i] := R; SN[i] := SN[i]+1; [Rule 1]
send request message(i, SN[i]) to all sites S_j for which SV[j] = R}
Step 2: Wait until S_i has the token; [Rule 2]
SV[i] := E; execute CS;
Step 3: SV[i] := N; TSV[i] := N; [Rule 3]
Update state vectors using the update rules (described in Section III-G-1),
If SV[none] = R, then SV[i] := H
else
{Using the arbitration rule (described in Section III-G-2), determine which site should get the token next and send it the token}.

Note that at step 1 of the algorithm, if SV[i] = H, then S_i does not increment SN[i]. Therefore, multiple consecutive executions of CS by a site result in that site's sequence number (SN) to increase at most by one. However, by no means does this cause one site to monopolize the execution of CS. This is because after every CS execution, a site checks its state vector SV to determine if any other site is requesting CS (step 3) and sends the token to one of the sites requesting CS, if there are any.

If every execution of the CS by site S_i is to be reflected in its SN[i], then we need to simply change step 1 of the algorithm in the following way:

Step 1: SN[i] := SN[i] + 1; [Rule 1]
If SV[i] $\neq H$, then
{SV[i] := R; send request message(i, SN[i]) to all sites S_j for which SV[j] = R}.

1) Update Rules: When a site completes the execution of its CS, it updates its state vectors and token vectors. The information in token vectors is used to refresh the information in a site's state vectors, and vice versa. The update rules are as follows.

For i = 1 to n do
If SV[i] = R and TSV[i] = N then [Rule 4]
{ cases
SN[i] = TSN[i] : SV[i] := N;
SN[i] > TSN[i] : TSV[i] := R;
TSN[i] := SN[i];
SN[i] < TSN[i] : SV[i] := N;
SN[i] := TSN[i] }
else
if SV[i] = N and TSV[i] = R then [Rule 5]
{ cases
SN[i] = TSN[i] : cannot occur;
SN[i] > TSN[i] : cannot occur;
SN[i] < TSN[i] : SV[i] := R;
SN[i] := TSN[i] }
else
if SV[i] = N and TSV[i] = N then [Rule 6]
{ cases
SN[i] = TSN[i] : do nothing ;
SN[i] > TSN[i] : cannot occur;
SN[i] < TSN[i] : SN[i] := TSN[i] }
else
/* SV[i] = R and TSV[i] = R */ [Rule 7]
{ cases
SN[i] = TSN[i] : do nothing ;
SN[i] > TSN[i] : cannot occur;
SN[i] < TSN[i] : SN[i] := TSN[i] }.

Explanation: The update rules essentially compare the state vectors at the site performing updates with the token vectors to determine who (the site or the token?) has more current information about the state of sites and restore the out-dated entries of these vectors with more current ones. This involves comparing respective entries of the vectors SN and TSN. For example, if SN[i] > TSN[i], then the site performing the updates has more current information about site S_i and TSV[i] and TSN[i] should be properly set; if SN[i] < TSN[i], then the token has more current information about site S_i and SV[i] and SN[i] should be properly set. The case SN[i] = TSN[i] is subtle. In this situation, if SV[i] = R and TSV[i] = N, then site S_i has executed its request corresponding to sequence number SN[i] and the update performing site is not aware of this and its SV[i] should be set to N.

Note that the update rules can be condensed into the following single rule.

If SN[i] > TSN[i], /* Token has out-dated information about S_i */
then {TSV[i] := SV[i]; TSN[i] := SN[i]}
else /* Site doing update has out-dated information about S_i */
{SV[i] := TSV[i]; SN[i] := TSN[i]};

Nonetheless, to enhance the understanding of the algorithm and its proof of correctness, we have given detailed rules for update, too.

2) Arbitration Rules: A site uses arbitration rules to determine which of many requesting sites should get the token[2] next. Fairness of the algorithm depends upon the fairness of its arbitration rules. Ideally, arbitration rules should be such that the token is not granted to a site twice (or many times) while other sites are still waiting for the token. We propose the following two arbitration rules which complement each other.

1) Grant the token to a requesting site with the lowest sequence number.[3] This scheme favors sites which have executed their CS's least frequently and discourages sites which have executed their CS's heavily. In this scheme, a site which has not executed its CS for a long time can hold up other sites by repeatedly getting the token while its sequence number is low. This scheme is desirable when all sites have balanced traffic of CS requests.

2) A site grants the token to the nearest requesting site. Site S_{i+1} is the nearest and site S_{i-1} is the farthest to site S_i (except when $i = 1$ and $i = n$). This scheme guarantees that a site cannot get the token twice while some other site is waiting for it (unless some site's request message has suffered a terribly long communication delay). This scheme will maintain fairness even when sites have unbalanced traffic of CS requests.

3) Request Message Handler: The request message handler at a site intercepts and processes incoming request messages. On the receipt of a request message, it takes actions such as updating state vectors and sending a request message or the token to other sites. The request message handler at site S_i is given below.

/* Site S_i is handling a request message(j, SeqNo) */
if SN[j] ≥ SeqNo, then {discard the message} /* the message is out-dated */
else
case SV[i] of
N : { SV[j] := R; SN[j] := SeqNo} [Rule 8]
R : { If SV[j] ≠ R, then {SV[j] := R; send a request message to S_j} [Rule 9]
SN[j] := SeqNo }
E : { SV[j] := R; SN[j] := SeqNo} [Rule 10]
H : { SV[j] := R; SN[j] := SeqNo; TSV[j] := R; TSN[j] := SeqNo; [Rule 11]
$SV\{i\}$:= N; send the token to site S_j}.

Explanation: When site S_i receives a current request message from site S_j, it updates its state vectors to reflect that S_j is requesting the token. If S_i is also requesting the token and it has not sent a request message to S_j previously (this will be true if SV[j] ≠ R before S_i has updated its state vectors), then it sends a request message(i, SN[i]) to S_j (rule 9). This action is consistent with the heuristic because now S_j also belongs to the set of sites which are likely to have the token. Intuitively, since S_j is also requesting the token and may get it before S_i gets it, it is logical for S_i to convey its request to S_j. It is also likely that S_j gets the token before any other site whom S_i has already sent request messages, gets it.

Note that the request message handler and three steps of the algorithm (Section III-G) access shared data structures, viz., SV and SN. For correctness, we require that execution of the request message handler and three steps of the algorithm (except wait for token and CS execution at step 2) mutually exclude each other.

H. Discussion

The algorithm trivially achieves mutual exclusion due to the singular existence of the token. However, an interesting feature of the algorithm is that a token requesting site can get the token in finite time without communicating with *all* the sites of the system. This feature is largely due to the staircase pattern which has a property that every site knows the site holding the token. Also, rules for information exchange and updating state/token vectors in the algorithm are so designed that the staircase pattern is preserved among the sites even after they have executed the CS any number of times. (Only the position of sites in the staircase pattern changes.) The reader should try some simple cases: in the simplest case, after a site S_i has executed its CS (starting in the initial configuration), the staircase pattern is satisfied but now S_i occupies the rightmost position and S_1 is second from the right.

During normal operation, a snapshot of the system may not satisfy the staircase pattern at any instant because the state of the system is dynamic; that is, sites are invoking mutual exclusion, messages are in transit, and sites are updating their state vectors. However, the system will satisfy the staircase pattern whenever there is no leftover work in the system, i.e., whenever all the pending requests for mutual exclusion have been executed and all the request messages have been received and processed.

IV. Correctness of the Algorithm

We show that the algorithm achieves mutual exclusion and is free from deadlock and starvation. In doing so, we refer to

[2] Note that TSV[j] = "R" implies that site S_j is requesting the token (see Lemma 1, Appendix). Therefore, a site can safely schedule the token to any site with an "R" entry in TSV.

[3] The sequence number of a site reflects how many times that site has executed its CS.

the results stated in the Appendix. For a proof of these results, the reader should consult [35].

A. Mutual Exclusion

Since a site captures the token before entering its CS and retains it during the execution, the singular existence of the token in the system trivially guarantees the existence of mutual exclusion.

B. Deadlock

In the algorithm, a deadlock occurs when sites are divided into noncommunicating groups such that a site in a group never sends a token request message to a site in another group, i.e., the sites in a group are in circular wait for the token (except the group which has the token). To show that such a situation never occurs and a requesting site eventually gets the token, the following two conditions must hold.

[1] "A site requesting its CS eventually sends a request message to the site executing its CS or to a site which is making a request for its CS and has sent its request for token to the site executing its CS."

A result of the above condition, the token vector TSV will eventually have an "R" entry for a token requesting site. The result of Lemma 3 (Appendix) shows that this condition is met by the algorithm if the system starts in the initial configuration (i.e., staircase pattern). Since the initial configuration is conserved subsequently (Theorem 2), this condition is met later, too.

[2] "A site with an "R" entry in the token vector TSV gets the token in finite time." This condition is met by both the arbitration rules discussed in Section III-G-2. Hence, the algorithm is deadlock free.

C. Freedom from Starvation and Fairness

Starvation occurs when few sites repeatedly execute their CS's while other sites wait indefinitely for their turns to do so. The algorithm is starvation free because if some sites are repeatedly executing their CS's, after finite number of executions, their sequence numbers will exceed those waiting to execute their CS's. Consequently, the token will eventually be granted to one of the waiting sites (the first arbitration rule, Section III-G-2). The second arbitration rule imposes a logical ordering among the sites for the purpose of selecting a site to schedule the token next. Under this arbitration rule, the token is scheduled in round-robin fashion among the sites waiting for the token, which guarantees that a few sites do not monopolize the token on a local basis while others wait. Therefore, under the second arbitration rule, once all of site S_i's request messages have been processed by its recipient sites, no other site can execute the CS twice (or more number of times) before site S_i has executed the CS.

V. Performance

Since the operations of mutual exclusion algorithms are very complex and quite difficult to analyze mathematically [15], we studied the performance of the proposed algorithm by a simulation technique. In addition, we carried out an analytic study of the performance of the algorithm for low and heavy traffic conditions whose results agree with the results of the simulation.

TABLE II
RESULTS OF THE SIMULATION

λ	M	W	λ	M	W
0.010	5.44	1.822	0.099	9.25	8.445
0.020	5.47	1.856	0.100	9.45	8.877
0.040	5.57	1.981	0.105	9.93	9.839
0.060	5.76	2.259	0.110	9.97	9.939
0.080	6.28	3.043	0.120	9.99	9.971
0.090	6.99	4.208	0.150	9.99	9.992
0.095	7.76	5.493	0.200	10.00	9.996
0.096	7.97	5.970	0.300	10.00	9.998
0.097	8.26	6.496	0.500	10.00	10.00
0.098	8.75	7.441	1.000	10.00	10.00

A. Model

The performance model used in this paper is similar to the one used in [32]. A similar performance model has been used in [9]. We assume that requests for CS execution arrive at a site according to Poisson distribution with parameter λ. Message propagation delay between any two sites is constant (T). The time taken by a site to execute its CS (i.e., the time spent inside the CS) is assumed to be constant (E), because each site is likely to execute almost the same number of instructions inside the CS. A site processes the requests for CS one by one, and there is only one CS in the system. In this study, we considered only the following two performance measures:

1) M: The average number of messages exchanged per CS execution.

2) W: The average delay in granting the CS, which is the span of time between the instant a request invokes the mutual exclusion algorithm and the instant when the request enters the CS.

B. Simulation Results

Simulation experiments were carried out for a homogeneous system of ten sites for various values of the traffic of CS requests (λ). Message propagation delay (T) among sites[4] was taken as 1.0, CS execution time (E) was taken as 0.0002. The values of the parameters chosen here are consistent with [32].

Collected performance measures, viz., number of messages per CS execution and delay in granting the CS, are probabilistic in nature. For these measures, we collected their averages by observing the values of these variables for several CS executions. To increase the accuracy of results, we ensured that the simulations reached a steady state (this was done by collecting statistics after every 500 CS executions and thus identifying a steady state) and by taking a large number of observations after steady state was reached. To increase the confidence in the simulation results, we computed a 95 percent confidence interval for delay W in granting CS. The confidence interval was found to be quite narrow; that is, the error bound is very small. The results of the simulation are summarized in Table II, where if the unit of time is

[4] Note that we have assumed that the system is logically fully connected; that is, any site can send a message to any other site. So, the topology of the underlying communication network is unimportant.

"seconds," then λ is in "CS requests/second" and the average delay in granting CS is in "seconds."

Note that when traffic of CS requests is low to medium, the proposed algorithm performs better than the Suzuki and Kasami algorithm for mutual exclusion [36] as far as M is concerned. (From the standpoint of delay in granting CS, both algorithms perform similarly.) However, when traffic of CS requests increases, the performance of the proposed algorithm gradually worsens, and at high λ, its performance is comparable to the performance of the Suzuki–Kasami algorithm [36] which does not use state information or a heuristic.

Explanation of Simulation Results: The reason the performance of the proposed algorithm degrades with increasing the traffic of CS requests (λ) is as follows: when λ increases, the system state changes more rapidly (because now sites enter into the states R, E, H, N more frequently) than can be disseminated. Consequently, the likelihood of a site having out-dated state information increases, thereby reducing the power of the heuristic. For example, when λ increases, more and more sites are in state R on the average because a site enters state R quickly and because it stays in that state longer due to increased W (because interference also increases). Since a site which is requesting CS sends a token request message to all the sites which it thinks are in state R, a site which is requesting the CS ends up sending more and more token request messages as λ increases. When the traffic of CS requests reaches a point where on the average all the sites always have a pending request for CS execution, every invocation of mutual exclusion ends up sending $n - 1$ token request messages (this happens when $\lambda \geq 0.2$ in the simulation).

When a site receives a request for CS and it does not have the token, delay in granting the CS will be $2*T + \Phi(\lambda)$ where $2*T$ accounts for roundtrip message delay and $\Phi(\lambda)$ accounts for delays due to interference because there may be other conflicting/concurrent requests for CS execution. As traffic of requests for CS execution increases, interference increases because more and more sites will execute the CS between two successive CS executions by a site, causing $\Phi(\lambda)$ to monotonically increase with λ. This is the reason why delay in granting CS increases with λ. When the traffic of CS requests reaches a point where on the average all the sites always have a pending request for the CS, then $\Phi(\lambda) = (n - 2)*T$ because all other sites execute the CS between two successive CS executions by a site (this starts happening when $\lambda \approx 0.2$ in the simulation).

To sum up, when traffic of CS requests is low to medium, the performance of the proposed algorithm is better than that of the Suzuki–Kasami algorithm [36]. However, the performance degrades as the traffic of CS requests increases and asymptotically reaches the performance of the Suzuki–Kasami algorithm [36]. When traffic of CS requests is high, the system state changes at a much faster pace than can be disseminated, causing the heuristic to work on more out-dated and incorrect state information. A smarter heuristic may be needed in the conditions of high load. For example, when the load is so high that all the sites always have a pending request for the token, it may be clever to send a token request message just to one site (rather than to all the sites as is dictated by the current heuristic). This is because every site always has a pending request for the token and will eventually get it. With this heuristic, a system with high load will behave as if the token is hopping among all the sites in round-robin fashion—each invocation of mutual exclusion will require only two messages.

C. Analytic Study

As mentioned earlier, due to the complex nature of mutual exclusion algorithms, it is very difficult to mathematically analyze their performance. In [15], Gravey and Dupis have analyzed the performance of two mutual exclusion algorithms for a distributed system consisting of two sites. They found that when the number of sites is more than two, an analytic performance study of mutual exclusion algorithms becomes intractable, by currently available tools, due to rapid growth of the state space of the underlying Markov chain. However, in limiting cases (e.g., "low traffic" and "heavy traffic" of CS requests), it is possible to analyze the performance of mutual exclusion algorithms [9]. Next, we analyze the performance of the proposed algorithm for these limiting cases.

Low Traffic: In case of low traffic of CS requests, most of the time only one or no request for CS will be present in the system. Consequently, the staircase pattern will be reestablished between two consecutive requests for the CS and there will seldom be interference among the CS requests from different sites.

When traffic of CS requests is low, sites will send 0, 1, 2, $\cdots$, $(n - 1)$ number of request messages with equal likelihood (assuming uniform traffic of CS requests at sites) to request the token. This is because state vectors SV of the sites have 0, 1, 2, $\cdots$, $(n - 1)$ R's respectively, in the staircase configuration. Therefore, the mean number of request messages sent per CS execution for this case is $= (0 + 1 + 2 + \cdots + (n - 1))/n = (n - 1)/2$. Since one message is required to send the token, the average number of messages exchanged per CS execution is $(n - 1)/2 + 1 = (n + 1)/2$. At low traffic, there will be very little interference among the requests from different sites. So, the mean delay in granting the CS will be equal to one roundtrip message propagation delay ($2T$).

However, both results should be reduced by a factor ($\approx 1 - 1/n$) because sometimes a requesting site will already have the token, thus resulting in no message transmission and no delay in granting the CS. Simulation study corroborates both results.

Heavy Traffic: Since in heavy traffic conditions, all the sites will always be making a request for the token (thus, will always be in the state R), the heuristic will cause a site to send token request messages to all other sites. Consequently, the average number of messages exchanged per CS execution will be n. Also, in heavy traffic conditions, the token will seldom be idle; that is, when a site finishes with the execution of its CS, it will immediately send the token to some other site (step 3 of the algorithm). Therefore, every T (strictly speaking $T + E$) units of time, a site will get the token and execute its CS resulting in the mean delay in granting CS of $n*T$. Simulation study also corroborates these results. (Note that the rate of CS executions, or system throughput, is $1/T$ in heavy traffic

conditions because a site executes the CS every T units of time.)

D. Adaptivity in Heterogeneous Traffic Patterns

An interesting feature of the algorithm is its adaptiveness to the environments of heterogeneous traffic of CS requests (λ) to optimize the performance (number of messages exchanged per CS invocation). In nonuniform traffic environments, the sites with higher traffic of CS requests will position themselves towards the lower end of the staircase pattern. That is, a site with higher traffic of CS requests will tend to have a fewer number of R's in its state vector SV. Also, at a high-traffic site, if $SV[j] = R$, then S_j is also a high-traffic site (this comes intuitively from the result of Lemma 2 of the Appendix). Consequently, high-traffic sites mostly send request messages to other high-traffic sites and seldom send request messages to sites with low traffic. This adaptivity results in further reduction in the number of messages and delays in granting the CS in environments of heterogeneous traffic.

VI. Failure Considerations

So far we have assumed that a site does not fail and the communication medium is reliable. In real life, sites and communication media are prone to failure. Site and communication medium failures may result in loss of a message, loss of information at a site, or partitioning of the system. In this section, we discuss the impact of site failure and communication link failure on the functioning of the proposed algorithm and suggest ways to cope with these failures. We show that when a failure occurs, the system reorganizes itself such that it continues to function without any (prolonged) disruptions.

Note that failures can be simple to catastrophic in nature. They can have malicious consequences on the system. For example, a site may not follow the mutual exclusion algorithm properly—it may falsely set the entries in token vectors (TSV and TSN) or it may start generating duplicate tokens deliberately (or due to a software bug). The communication medium may deliver a message with errors—it may deliver a request message(j, SeqNo) instead of the message(i, SeqNo) to site S_k. Consequently, site S_k may send the token to site S_j even though S_j is not requesting the token.

There are so many states in which a system can fail and it may not always be possible to accurately reconstruct the system state at the time of crash. Therefore, it is a very difficult task to give an exhaustive crash recovery procedure (i.e., a procedure which can recover the system from all conceivable failures) for the algorithm. Consequently, in this paper we do not consider catastrophic failures. In the remainder of this section, we discuss the consequences of message loss, site crash, and network partitioning on the operation of the algorithm and present methods for recovery from these failures.

A. Loss of Messages

We assume that either the communication medium delivers a message correctly or it does not deliver it. We can implement this easily by having the receiver of a message discard the message if it is found to contain an error (of course, we have to use an error detecting code).

Loss of a token request message does not seriously affect the operation of the algorithm and just delays the execution of the CS by a site (unless all the token request messages sent by a site get lost). Low-level protocols of the communication medium can use time-out and retransmission mechanisms to handle message losses. Loss of the token (which can be because a site which has the token crashes and does not remember that it had the token when it recovers) is serious. All token-based mutual exclusion algorithms have these problems for the following reasons.

1) Declaration of the token loss requires special care because if the declaration is false and an additional token is generated, then the condition of mutual exclusion will be violated.

(We assume that if a requesting site has not received the token for a specified period of time, it times out and declares that the token is lost [22]. To be safer, a site which times out on token wait, consults other sites of the system before declaring that the token is lost.)

2) When the token is declared to be lost, exactly one site should generate the token.

(In a simple scheme, we can designate a site to generate the new token in the wake of token losses. However, for reasons of reliability and autonomy, it is desirable that the token generation be done in a distributed manner. That is, there is no *a priori* fixed site to generate the token and in the wake of a token loss, all the sites elect a site to generate the new token.)

The problem of electing a site in distributed environments is discussed in [14]. Several algorithms to elect a site to generate a new token have appeared in [7], [17], [22], and [30], and the best algorithm requires $O(n \log n)$ messages in the worst case.

B. Site Failures

We assume that a failed site does not behave maliciously; that is, we do not consider Byzantine failures [20], [29]. An up site does not lie, faithfully executes the mutual exclusion algorithm, and honestly interacts with other sites. A failed site stops responding to incoming messages and it may lose its state information (i.e., the contents of state vectors SV and SN). We also assume that when a site fails, other sites are informed of it (by the communication medium or by other sites) or a site learns about it via time-outs. For the sake of ease of exposition, we will consider only single site failures.

A site can be in any of the following three phases: *Running*—the site is responding to incoming messages normally and invoking the mutual exclusion algorithm properly; *Failed*—the site is neither executing nor responding to incoming messages (i.e., it discards all incoming messages); and *Recovering*—the site is executing and running the recovery procedure. It may not respond to incoming request messages in this phase. When a site fails, the system can be in so many states that it impossible to give a crash recovery procedure which will work in all situations. Next, we discuss recovery methods for several site-crash situations which cover all important cases. We also present a crash recovery

procedure which consistently restores a recovering site in the staircase pattern with other sites.

Case I: In the first case, at the time of failure, the failed site did not have the token and did not have any pending request for the token (i.e., it was in state N).

Since the failed site did not have the token when it crashed and never gets the token while it is crashed, the token continues to exist among the rest of the sites. Also, while that site is down, other sites continue to satisfy the initial configuration and have all the properties discussed in Section IV. This is because when up sites learn that a site has crashed, they set their SV[crashed site] to N; consequently, the crashed site is eventually pushed to the highest end of the staircase pattern. Therefore, in this situation a site's failure does not disrupt the operation of the rest of the system.

When a site recovers from a crash, its state vectors must be restored to an appropriate state and the site must be reintegrated with the rest of the system. Next, we present a crash recovery procedure which performs these functions.

Crash Recovery Procedure: In the recovering phase, a site (say S_i) runs the following crash recovery procedure.

1) Send Recovering(S_i) message to all other sites. (When a site receives the Recovering(S_i) message, it sets its SV[i] := N.)

2) Set all entries of its state vector SV to "R" and all entries of its state vector SN to 0.

3) If a site crashes while in the middle of the crash recovery procedure, it restarts the procedure from the beginning when it comes up.

Explanation: A Recovering(S_i) message serves two purposes: first, it informs other sites that S_i has come up, and second, it makes other sites set their SV[i] to N (it is possible that some sites did not learn that S_i was down before they had received the Recovering(S_i) message). Setting all entries of SV to R at step 2 is compatible with all other sites setting their SV[i] to N. Also, when S_i comes up, it has no idea about the location of the token; therefore, when it has to request the token, it should send request messages to all other sites (setting all entries of SV to R by S_i achieves this). Setting all entries of SN to 0 by S_i does not create any problem because when S_i gets the token next, update rules will appropriately set the SN from TSN.

Lemma: The crash recovery procedure brings up a crashed site correctly.

Proof: We must show that the crash recovery procedure fixes a recovering site such that the entire system resumes functioning correctly. As a result of step 1, when other sites receive the Recovering(S_i) message, they set their SV[i] to N. At step 2, site S_i sets all of its SV entries to R. Consequently, site S_i fits itself into the staircase configuration (at the highest end) with the rest of the sites and the system resumes functioning normally.

If a site crashes while in the middle of running the crash recovery procedure, it does not disrupt the functioning of other sites. This is because if other sites, as a result of the receipt of Recovering messages, have set SV[crashed site] to N, it only prevents these sites from sending token request messages to the crashed site (which is all right because the crashed site will not respond to the request messages anyway).

Case II: In the second case, a site crashes with the token (i.e., in the state E or H).

In this case we require a somewhat trickier mechanism, such as generating another token to recover from the failure. When sites learn that a site has crashed, they set their SV[crashed site] to N. Consequently, the crashed site is pushed to the highest end of the staircase pattern. Sites use time-outs to determine that the token has disappeared from the system and initiate a token recovery procedure to generate the new token. When the failed site recovers from the crash, it restores its state vectors to an appropriate state and reintegrates itself with the rest of the system by using the crash recovery procedure described for case I.

Case III: In the third case, at the time of failure, the failed site did not have the token, but it did have a pending request for the token (i.e., it was in state R).

In this case, two situations can arise. In the first situation, all the up sites may learn that a particular site is down before any one of them has sent the token to it. When a site S_i learns that site S_j is down, S_i sets its SV[j] to N. Therefore, site S_j is pushed to the highest end of the staircase pattern and no site sends the token to it (unless S_j comes up and sends out token request messages).

In the second situation, an up site had sent the token to the down site before it learned that the site it sent the token is down. Note that the second situation is similar to case II provided the failed site had not yet recovered when the token arrives. In this situation, the system recovers from the token loss as in case II (i.e., the sites time-out to discover that the token is lost and run a token recovery algorithm). However, if the failed site had recovered before the token arrives, then the system continues to function normally. When the crashed site comes up, it reintegrates itself with the rest of the system by executing the crash recovery procedure described for case I.

Case IV: In the fourth case, at the time of failure, the failed site (say S_i) held a "current" request for the token from another site (say S_j). This happens when S_j sends out a token request message to S_i before it has learned that S_i is down.

In this case, if in addition to site S_i, S_j has sent token messages to other sites, then no complication arises because there are other sites in the system which hold the token request of site S_j. However, complications arise when S_i is the only site to which S_j sent a token request message. This is because as a result of the crash, S_i loses the information that S_j was requesting the token from it. Consequently, S_j thinks that it has conveyed its token request to an appropriate set of sites and waits for the token, but it will not get the token because no site knows about its request for the token. (There is a remote possibility where S_i updates the token vectors to reflect that S_j is requesting the token and dispatches the token to some site before S_i crashes. In this situation, the token contains the information that S_j is requesting the token even though S_i has crashed.)

When a site learns that the *only* site to which it sent a token request message has failed, it handles this situation by sending

token request messages to all other sites of the system. This way the site guarantees that its request will become known to a site which eventually gets the token. When the failed site comes up, it recovers by executing the crash recovery procedure described for case I.

C. Network Partitioning

A distributed system is partitioned if it gets divided into two or more groups of sites such that intergroup communication is cut off. System partitioning occurs due to communication link failures, site failures, or a combination of both. We assume that partitioning occurs in a clean way; that is, either all the messages from the sites in one partition to the sites in other partitions reach their destinations or none do.

Handling Network Partitioning: The main issue in the face of network partitioning is to ensure that sites in different partitions do not concurrently access the shared resource [1]. In the proposed algorithm, this is simple to achieve because at most one partition can contain the token, and only the sites in that partition will execute the CS so long as the system remains partitioned. Sites in the partition which has the token assume that only the sites in that partition are up (i.e., they do not send token request messages and the token to the sites in other partitions).

Recovery from Network Partitioning: A partition which does not have the token is referred to as an *inactive partition*. A partition which has the token is referred to as the *active partition*. Merger of two inactive partitions is easy to handle—all the sites in both partitions have to be notified that they are now a part of a bigger (inactive) partition and have to be informed about the new sites that joined the partition.

Merger of an inactive partition with the active partition, however, requires a somewhat elaborate mechanism because sites in the inactive partition must fit themselves into the staircase pattern with the sites in the active partition so that all sites in the resulting (active) partition are able to execute the CS normally. When an inactive partition IP = $\{S_{i1}, S_{i2}, \cdots, S_{im}\}$ and the active partition AP = $\{S_{j1}, S_{j2}, \cdots, S_{jn}\}$ merge to form a bigger active partition, the following protocol is executed to reintegrate the sites in partition IP with the sites in partition AP:

All the sites in AP set SV$[i] = N$ for $\forall S_i \in$ IP; that is, sites in the active partition think that sites in the inactive partition are not requesting the token. All the sites in IP set SV$[j] = R$ for $\forall S_j \in$ AP; that is, sites in the inactive partition think that sites in the active partition are requesting the token. (Note that after these two steps have been executed, sites in partition IP have fitted themselves in the higher end of the staircase pattern with the sites in partition AP. Now what is left is that sites in IP should arrange themselves in the staircase pattern. This is done next.) Site S_{i1} sends a RECOVERING_PART($i1$) message to all other sites in partition IP. Upon the receipt of a RECOVERING_PART($i1$) message, site S_{ik}, $2 \le k \le m$, sets SV$[i1] = N$, and in addition, site S_{i2} sends RECOVERING_PART($i2$) messages to all sites in the set $\{S_{i3}, S_{i4}, \cdots, S_{im}\}$. Upon the receipt of a RECOVERING_PART($i2$) message, a site S_{ik}, $3 \le k \le m$, sets SV$[i2] = N$, and in addition, site S_{i3} sends RECOVERING_PART($i3$) messages to all sites in the set $\{S_{i4}, S_{i5}, \cdots, S_{im}\}$ and so on until S_{im} receives the RECOVERING_PART($im - 1$) message from S_{im-1} and sets its SV$[im - 1] = N$. State vectors SN of sites in IP get updated when these sites get the token.

If all sites in IP have a unique ordering $S_{i1}, S_{i2}, \cdots, S_{im}$, then we can alternatively use the following method to perform the reintegration process: all sites in AP set SV$[i] = N$ for $\forall S_i \in$ IP as before. Site S_{ik}, $1 \le k \le m$, in IP set SV$[j] = R$ for $\forall S_j \in$ AP $\cup \{S_{ik+1}, S_{ik+2}, \cdots, S_{im}\}$; that is, a site S_{ik} in the inactive partition thinks that all the sites in the active partition and sites $S_{ik+1}, S_{ik+2}, \cdots, S_{im}$ in the inactive partition are requesting the token. Note that after these two steps have been executed, the sites in partition IP have fitted themselves toward the higher end of the staircase pattern with the sites in partition AP.

VII. Concluding Remarks

We have presented a heuristically-aided algorithm for achieving mutual exclusion in distributed systems, where sites do not share a global memory and communicate solely by passing messages over a communication network. A novelty of the algorithm is that sites use information about system state and a heuristic to determine their course of actions in order to achieve improved performance (reduced message traffic on the network). We have shown that the algorithm achieves mutual exclusion and is free from deadlock and starvation. We have discussed the impact of site and media failures on the operation of the algorithm and have presented crash recovery procedures for these failures.

We have conducted a performance study of the proposed algorithm using an event-driven simulation. The results of the performance study reveal that when traffic of CS requests is low to medium, the heuristic algorithm presented here performs better than the algorithm of Suzuki and Kasami, which is one of the most efficient mutual exclusion algorithms, from the standpoint of message traffic. (Both perform identically as far as delay in granting request is concerned.) However, in case of heavy traffic of CS requests, performance enhancement is marginal because the system state changes at a much faster pace causing the heuristic to work on more outdated and incorrect knowledge of the state of the system. A smarter or perhaps adaptive heuristic may circumvent this problem, but we have not addressed this issue in this paper.

Besides showing that we can improve the performance of the Suzuki–Kasami algorithm by using state information, we have demonstrated that heuristics can be advantageously applied to deal with uncertainty in state information in problems where systems must satisfy a strong and sharply defined constraint, e.g., mutual exclusion. However, the heuristic presented in this paper is simple and should be treated as an initial step in showing the effectiveness of heuristics in problem domains which have sharply defined constraints. It is felt that heuristic techniques can effectively applied to many problems in the domain of distributed database systems such as concurrency control, crash recovery, etc. The author hopes that the research presented in this paper will trigger concurrent investigations in these directions.

APPENDIX

In this Appendix, we present results on the properties of the algorithm which are conducive in proving its liveness properties. We only state the results in the form of lemma/theorem, and a formal proof of these results can be found in [35].

Notations:

R_i^j : Denotes that at site S_i, SV[j] = R.
N_i^j : Denotes that at site S_i, SV[j] = N.
$/S_i/$: The number of N entries in the state vector SV of site S_i.
NS : The set of all the sites that have not made a request to execute their critical sections since the beginning.
ES : The set of all the sites that have executed their critical sections at least once since the beginning.

In this notation, the initial configuration of the system state can be represented as

R_i^j, for $j = i - 1, i - 2, \cdots, 2, 1$ and $1 \leq i \leq n$,
and N_i^j, for $j = i, i + 1, \cdots, n$ and $1 \leq i \leq n$.

Totally Ordered Sequence:

Sites $S_n, S_{n-1}, \cdots, S_1$ are said to form a "totally ordered sequence" if

$$/S_n/ < /S_{n-1}/ < \cdots < /S_1/.$$

Note that the initial system state forms a totally ordered sequence.

Lemma 1: Token vector TSV never has any false R entry.

Lemma 2: If the system starts in the initial configuration and all sites in the set ES have executed their critical sections exactly once and all the request messages have been received and processed then

a) $\forall S_i \in ES$ AND $\forall S_j \in NS\ R_i^j$ holds,
and
b) $\forall S_i \in ES$ AND $\forall S_j \in NS\ N_i^j$ holds.

Lemma 3: If the system starts from the initial configuration and sites $S_{i1}, S_{i2}, \cdots, S_{im}$ are making requests for the token and site S_{ie} is executing its critical section and all request messages have been received and processed then

a) At $\forall S_i$, $i = i1, i2, \cdots, im$, SV[J] = R, for $J = i1, i2, i3, \cdots, im$, and ie,
and
b) At site S_{ie}, SV[J] = R, for $J = i1, i2, i3, \cdots, im$, and SV[$ie$] = E.

Lemma 4: If sites $S_{im}, S_{im-1}, \cdots, S_{i1}$ just executed their critical sections in this order and all request messages sent by them have been received and processed, then

a) $\forall j, 2 \leq j \leq m, R_{ij}^{ik}, \quad k = j - 1, j - 2, \cdots, 1,$
and
b) $\forall j, 1 \leq j \leq m - 1, N_{ij}^{ik}, \quad k = j + 1, j + 2, \cdots, m.$

Lemma 5: If the system starts from the initial configuration and all the sites in the set $NS = \{S_{im}, S_{im-1}, \cdots, S_{i1}\}$ initially satisfy the condition $/S_{im}/ < /S_{im-1}/ < \cdots < /S_{i1}/$ and have received all messages directed to them, then the following two conditions hold:

a) $\forall j, 1 < j \leq m, R_{ij}^{ik}$ for $k = j - 1, j - 2, \cdots, 1,$
and
b) $\forall j, 1 < j \leq m, N_{ij}^{ik}$ for $k = j + 1, j + 2, \cdots, m.$

Theorem 1: If the system starts with the initial configuration and at any instant

a) all sites $S_{im}, S_{im-1}, \cdots, S_{i1}$ have executed their CS's once in this sequence,

b) all request messages have been received and processed,

c) sites $S_{jr}, S_{jr-1}, \cdots, S_{j1}$ never made a request and $/S_{jr}/ < /S_{jr-1}/ < \cdots < /S_{j1}/$, and

d) no site is making a request for token, then sites $S_{jr}, S_{jr-1}, \cdots, S_{j1}, S_{im}, S_{im-1}, \cdots, S_{i1}$ form a totally ordered sequence.

Note: Theorem 1 implies that if the system starts from the initial state and at any instant some sites have executed their critical sections only once and no request message and the token message is pending and no site is executing its critical section, then all sites of the system form a totally ordered sequence.

Theorem 2: If the system starts from the initial configuration and at any instant

a) some (or all) sites have executed their critical sections any number of times,

b) all request messages have been received and processed, and

c) no site has its token request pending or executing its critical section, then all sites of the system form a totally ordered sequence at that instant.

Note: Theorem 2 proves the conservability of the initial configuration.

ACKNOWLEDGMENT

The author is thankful to Prof. A. K. Agrawala of the University of Maryland for providing him with valuable critique and suggestions on this research. The author wishes to thank Dr. B. Lindsay of IBM Almaden Research Center and anonymous referees whose comments on the earlier versions of the paper were helpful in improving the quality of the presentation and enhancing the technical contents of the paper. The author is also deeply indebted to IEEE TC editor Dr. D. Fishman whose painstakingly detail comments on the manuscript were instrumental in improving the quality of the final version. He was also kind enough to handle the processing of the paper despite the fact that his term expired in the middle of review cycle.

REFERENCES

[1] D. Barbara and H. Garcia-Molina, "Mutual exclusion in partitioned distributed systems," Tech. Rep. CS-001, Dep. Comput. Sci., Princeton Univ., July 1985.
[2] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, pp. 226–244, Apr. 1978.
[3] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, pp. 185–222, June 1981.
[4] R. R. Boorstyn and A. Livne, "A technique for adaptive routing in networks," *IEEE Trans. Commun.*, pp. 474–480, Apr. 1981.
[5] O. S. F. Carvalho and G. Roucairol, "On mutual exclusion in computer networks, technical correspondence," *Commun. ACM*, Feb. 1983.

[6] T. Casavant and J. G. Kuhl, "A formal model of distributed decision-making and its application to distributed load balancing," in *Proc. 6th Int. Conf. Distribut. Comput. Syst.*, May 1986, pp. 232-239.
[7] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Commun. ACM*, pp. 281-283, May 1979.
[8] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surveys*, pp. 66-78, June 1971.
[9] A. Dupis, G. Hebuterne, and J.-M. Pitie, "A comparasion of two mutual-exclusion algorithms for computer networks," in *Proc. 1st Workshop Modeling Perform. Eval. Parallel Syst.*, Grenoble, France, Dec. 1984.
[10] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Software Eng.*, pp. 662-675, May 1986.
[11] K. Efe, "Heuristic models of task assignment in distributed systems," *IEEE Computer*, pp. 50-56, June 1982.
[12] P. H. Enslow, "What is a 'distributed' data processing system?," *IEEE Computer*, pp. 13-21, Jan. 1978.
[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notion of consistency and predicate locks in a database system," *Commun. ACM*, pp. 624-633, Nov. 1976.
[14] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. Comput.*, pp. 48-59, Jan. 1982.
[15] A. Gravey and A. Dupis, "Performance evaluation of two mutual exclusion distributed protocols via Markovian modeling," in *Proc. 6th IFIP Workshop Protocol Specification, Testing, Verification*, Montreal, P.Q., Canada, June 10-13, 1986.
[16] J. Helary, N. Plouzeau, and M. Raynal, "A distributed algorithm for mutual exclusion in an arbitrary network," Tech. Rep. 496, INRIA, Centre de Rennes, France, Mar. 1986.
[17] D. S. Hirschberg and J. B. Sinclair, "Decentralized extrema-finding in circular configurations of processes," *Commun. ACM*, pp. 627-628, Nov. 1980.
[18] A. Kumar, M. Singhal, and M. Liu, "A model for distributed decision making: An expert system for load balancing in distributed systems," in *Proc. 11th Annu. Int. Comput. Software Appl. Conf.*, Tokyo, Japan, Oct. 7-9, 1987.
[19] L. Lamport, "Time, clocks and ordering of events in distributed systems," *Commun. ACM*, pp. 558-565, July 1978.
[20] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Programming Languages Syst.*, pp. 228-234, July 1982.
[21] E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel, "File access performance of diskless workstations," *ACM Trans. Comput. Syst.*, pp. 238-268, Aug. 1986.
[22] G. LeLann, "Distributed systems—Towards a formal approach," *Inform. Proc. 77*. North-Holland, 1977, pp. 155-160.
[23] G. LeLann, "Algorithms for distributed data sharing systems which use tickets," in *Proc. 3rd Berkeley Workshop Distribut. Data Management Comput. Networks*, Aug. 1978, pp. 259-272.
[24] G. LeLann, "Motivation, objective, and characteristics of distributed systems," in *Distributed Systems-Architecture and Implementation*, Lampson *et al.*, Eds. New York: Springer-Verlag, 1981, pp. 1-9.
[25] D. B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," in *Proc. ACM Conf. Language Design Reliable Software, SigPlan Notices*, Mar. 1977, pp. 128-137.
[26] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, pp. 145-159, May 1985.
[27] R. A. Maule and A. Kandel, "A model for an expert system for medium access control in a local area network," *Inform. Sci.*, vol. 37, pp. 39-83, 1985.
[28] J. M. McQuillan, "The new routing algorithm for the ARPANET," *IEEE Trans. Commun.*, pp. 711-718, May 1980.
[29] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, pp. 228-234, Apr. 1980.
[30] G. L. Peterson, "An O(n log n) unidirectional algorithm for the circular extrema problem," *ACM Trans. Programming Languages Syst.*, pp. 758-762, Oct. 1982.
[31] K. Ramamritham, J. Stankovic, and W. Zhao, "Meta-level control in distributed real-time systems," in *Proc. 7th Int. Conf. Distribut. Comput. Syst.*, W. Berlin, West Germany, Sept. 23-25, 1987.
[32] G. Ricart and A. K. Agrawala, "Performance of a distributed network mutual exclusion algorithm," Tech. Rep. TR-774, Dep. Comput. Sci., University of Maryland, College Park, MD, Mar. 1979.
[33] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, Jan. 1981, pp. 9-17.
[34] B. Sanders, "The information structure of distributed mutual exclusion algorithms," *ACM Trans. Comput. Syst.*, Aug. 1987, pp. 284-299.
[35] M. Singhal, "On the application of AI in decentralized control: An illustration by mutual exclusion," in *Proc. 7th Int. Conf. Distribut. Comput. Syst.*, W. Berlin, West Germany, pp. 232-239, Sept. 21-25, 1987.
[36] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, pp. 344-349, Nov. 1985.
[37] F. Tobagi and V. B. Hunt, "Performance analysis of analysis of a carrier sense multiple access with collision detection," in *Proc. Local Area Network Symp.*, Boston, MA, 1979.
[38] W. Zhao and K. Ramamritham, "Distributed scheduling using bidding and focussed addressing," in *Proc. Symp. Real-Time Syst.*, Dec. 1985, pp. 103-111.

Mukesh Singhal (S'81-M'87) was born in India on May 8, 1959. He received the Bachelor of Engineering degree in electronics and communication engineering with honors from the University of Roorkee, Roorkee, India, in 1980.

He joined the Department of Computer Science, University of Maryland, College Park, in January 1981, where he received the Ph.D. degree in May 1986 for work in design and analysis of concurrency control algorithms in distributed database systems. Since February 1986, he has been an Assistant Professor of Computer and Information Science, The Ohio State University, Columbus. From 1981 to 1985, he served as a Teaching/Research Assistant and Instructor at the Department of Computer Science, University of Maryland, College Park. His current research interests include distributed database systems, distributed systems, and performance modeling of distributed systems.