The Information Structure of Distributed Mutual Exclusion Algorithms

BEVERLY A. SANDERS
University of Maryland

The concept of an information structure is introduced as a unifying principle behind several of the numerous algorithms that have been proposed for the distributed mutual exclusion problem. This approach allows the development of a generalized mutual exclusion algorithm that accepts a particular information structure at initialization and realizes both known and new algorithms as special cases. Two simple performance metrics of a realized algorithm can be obtained directly from the information structure. A new failure recovery mechanism called local recovery, which requires no coordination between nodes and no additional messages beyond that needed for failure detection, is introduced.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; C.4 [**Computer Systems Organization**]: Performance of Systems—*reliability, availability, and servicability*; D.4.1 [**Operating Systems**]: Process Management—*mutual exclusion*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Failure-tolerant algorithms

1. INTRODUCTION

In distributed systems, the lack of both shared memory and a common clock results in more complex mutual exclusion algorithms than the synchronization primitives typically found when processes share memory. This complexity has resulted in the introduction of several algorithms to solve the mutual exclusion problem. Our goal is to provide a unifying framework that allows comparison of these algorithms and points the way to the design of new ones.

The system under consideration consists of N processes, each with a unique identification that communicates asynchronously via message passing. The processes have a simple structure in which they repeatedly alternate between computations outside a critical section and computations inside a critical section. The nodes in the distributed system logically are completely connected so that a process can send a message to any other process. Message arrivals may occur at any time and are handled serially when they arrive. Either a process may be interrupted on a message arrival to handle the message, or a separate process

This work was partially supported by NASA Goddard Space Flight Center under grant NAGS 235 and NASA Ames Research Center under grant NCC 2-414.
Author's present address: Institut für Informatik, ETH Zentrum, CH-8092 Zürich, Switzerland.

may be created for message handling. In the latter case, shared variables are protected in a local critical section. The communication subsystem is assumed to be reliable, and messages between any pair of processes are delivered in the order they were sent. No assumptions are made about the delay between the time a message is sent and received, the time spent in a critical section, or the time spent in computations outside the critical section, except that they are finite.

The design of a mutual exclusion algorithm consists of defining the protocols used to coordinate entry into the critical section. These protocols consist of the local variables at each process, the entry code that is executed before entering the critical section, the exit code that is executed on leaving the critical section, and the message handling code that is executed when a message is received asynchronously from another process.

Many algorithms that fit this model have been proposed [1, 2, 5, 6, 7]. They tend to differ in their communication topology and also in the amount of information processes maintain about other processes. The concept of an information structure is the unifying principle in our approach and describes which processes maintain state information about other processes, and from which processes information must be requested before entering a critical section. In the next section, we formally define the information structure, and then we give a generalized algorithm using messages with simple semantics that realizes different known and new algorithms, depending on the information structure. Necessary and sufficient conditions for an information structure to realize an algorithm that guarantees mutual exclusion are stated. Assuming that the information structure is static, we discuss performance issues and failure recovery. Finally, an example of an algorithm with a dynamic information structure is given.

2. INFORMATION STRUCTURES

The information structure of a particular algorithm describes which processes maintain state information about other processes, and for each process, the set of processes from which information or permission should be requested before it enters the critical section. As viewed by other processes, a process can be in one of three states, either IN-CS (executing inside the critical section), NOT-IN-CS (executing outside the critical section), or WAITING (executing or blocked in the entry code). These states correspond to the states that are relevant to mutual exclusion and can be determined by communication with other processes. Messages will have simple semantics and indicate changes of state, requests for permission, and so forth.

Formally, the information structure can be described by a pair of subsets of process IDs associated with each process. The *inform set* for process i is denoted I_i and the *request set* for i is denoted R_i. In the generalized algorithm given in the next section, a process sends a message to every process in its inform set whenever it changes state from IN-CS to NOT-IN-CS or from NOT-IN-CS to WAITING. Before entering the critical section, each process must request and receive permission from every process in its request set.

It is convenient to also define for each process the *status set*, S_i. The status sets are determined by the inform sets where $j \in S_i$ if $i \in I_j$. The status set S_i indicates the set of processes about which process i maintains information.

The inform and request sets define the *information structure* of a mutual exclusion algorithm. Two different situations can occur. In one, the information structure is static, that is, it does not change during the normal execution of the algorithm. In this case, the entire information structure can be known to all processes. It is necessary to change the information structure when a processor fails, and failure recovery can be viewed as a mechanism for revising the information structure to take into account the removal of a process. The new information structure must satisfy the correctness conditions. Dynamic information structures change during normal evolution of the algorithm, and the global information structure is not necessarily known to all nodes. Specifying the information structure now requires both the specification of the initial inform and request sets and also the rules by which they change.

3. THE GENERALIZED MUTUAL EXCLUSION ALGORITHM

3.1 Policy for Conflict Resolution

Although we are more interested in the information structure, a realization of a mutual exclusion algorithm must also specify a policy for conflict resolution. There is always more than one candidate for the policy, but the policy chosen must have the property that two processes resolving the same conflict must be able to resolve it the same way. For the special case of a centralized algorithm, a single process resolves all conflicts, and any policy that is a function of information available to this process can be used. Typically, the policy chosen is first come, first served (FCFS) based on the times of arrival of information requests at the process. Since the focus of this paper is on a different problem, only one conflict resolution mechanism, time stamp-based priorities [4] will be considered.

Time stamps evolve as follows: Each process maintains a variable that contains the largest time stamp that process has seen yet, and this value is updated after the receipt of each message from other processes and is included with every message that is sent. When a process needs to choose a time stamp for itself, the time stamp is equal to the largest time stamp + 1, and the largest time stamp variable is updated. Conflicts are resolved by choosing the process with the lowest time stamp. Ties are broken using a predetermined ordering of the process ID.

3.2 The Algorithm

A generalized algorithm that guarantees mutual exclusion is presented below.

The entry code performs the following actions:

(1) The process chooses a time stamp and sends a REQUEST message along with the time stamp to every process in the request set R_i.

(2) The process then waits for a GRANT message from every process in R_i. Process i then enters the critical section.

The exit code performs the following:

(1) Send a RELEASE message to all processes in the inform set I_i.

When a process receives a message from another process, it updates its local variables and takes action depending on which type of message was received.

Messages are handled serially. If message handling for mutual exclusion is performed by a separate, concurrent process, then shared variables are protected by local critical sections. The local data maintained by the process include a priority queue containing a list of processes (along with their time stamps) from which a REQUEST has been received but a GRANT has not yet been sent. A variable CSSTAT, which indicates process i's best knowledge of the status of the critical section, is maintained. It indicates that the critical section is free or contains the identity of a process in the status set that has been sent a GRANT message, but from which no RELEASE message has been received.

On receiving a REQUEST message

1. The ID of the sending process is placed on the priority queue.
2. If CSSTAT indicates that the critical section is free, then a GRANT is sent to the process on top of the queue and the entry is removed from the queue. If the recipient of the GRANT is in S_i, then CSSTAT is set to indicate that the process is in the critical section.

When a RELEASE message is received:

1. CSSTAT is set to FREE.
2. A GRANT is sent to the process on top of the queue, if one exists, and the entry is removed from the queue. If the recipient of the GRANT is in S_i, then CSSTAT is set to indicate that the process is in the critical section.
3. Step 2 is repeated until CSSTAT indicates that a process is in the critical section or until the queue is empty.

In the algorithm described above, processes request permission from the processes in their request set. Permission is granted unless some process that is *in the status set of the process receiving the request* is currently assumed to be in the critical section. A REQUEST message can be interpreted to mean "According to your information, is it OK for me to enter the critical section?" A GRANT message can be interpreted as "According to my information, it is OK to enter the critical section" When a GRANT is sent to a process that is not in the status set, the fact that a GRANT is sent and that the process may be in the critical section is not remembered by the process that sent the GRANT. The information structure must be defined so that the state of each process is kept track of somewhere, and so that every process will somehow find out about the state of every other process before entering the critical section. If j is in the status set S_i of process i, then i will assume that j is IN-CS if CSSTAT $= j$, is WAITING if j is in the queue, or is NOT-IN-CS if neither condition is satisfied.

Figure 1 illustrates several known algorithms described in terms of their information structure. A solid arc from i to j indicates that $j \in I_i$ and $j \in R_i$. A dashed arc from i to j indicates that $j \in R_i$, but $j \notin I_i$. The algorithm in Figure 1a has a single node that arbitrates admission to the critical section [1, 6]. The algorithm in Figure 1b corresponds to a completely distributed, symmetrical algorithm in which each process requests permission from every other process before entering the critical section [7]. Figure 1c corresponds to an algorithm in which each process requests permission from a predetermined subset of processes [5]. The request sets and inform sets of each process are identical and are chosen so that for all pairs of processes i and j, $I_i \cap I_j \neq \emptyset$. It was claimed

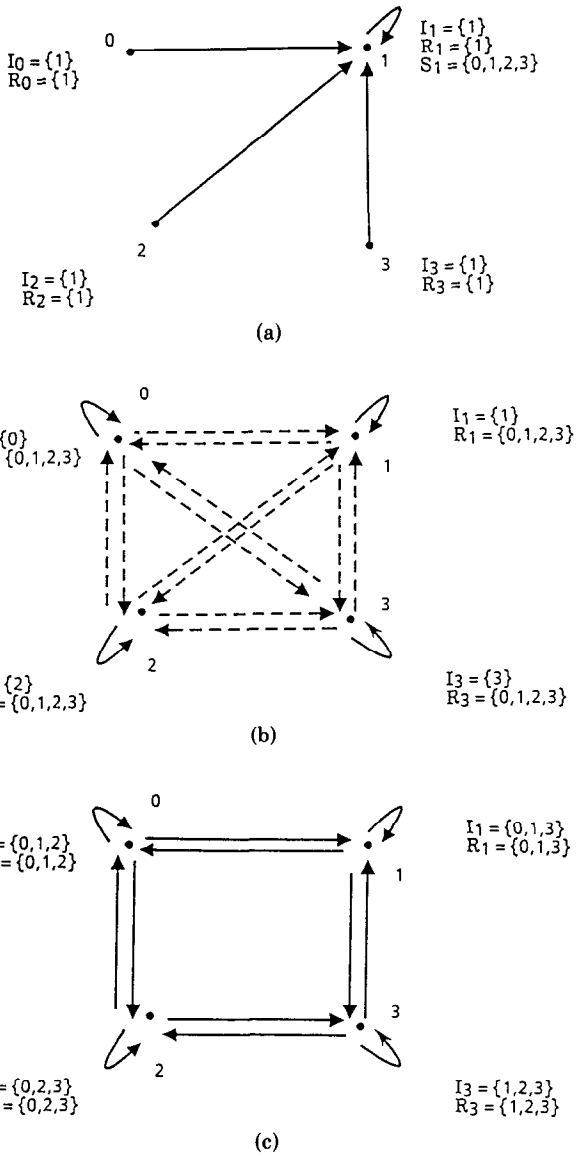

Figure 1

that the symmetrical configuration that minimizes the number of messages per critical section entry arranges the subsets to correspond to lines in a finite projective plane and results in $c(N)^{1/2}$ messages where c is a constant between three and five.

Figure 2 illustrates a new algorithm. This algorithm would be appropriate in situations in which two processes have significantly more frequent requests for entrance into the critical section than the others, which minimizes the number of messages needed for critical section entry by those two processes and treats

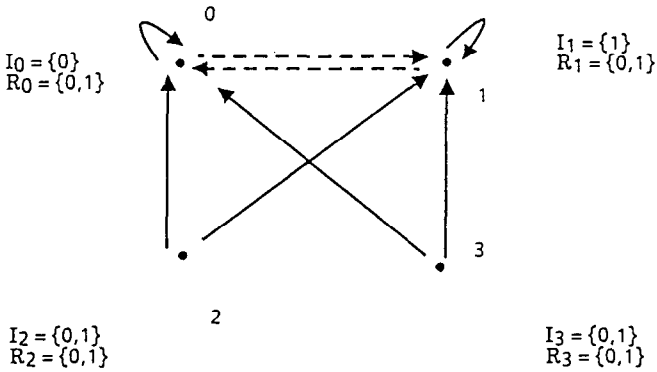

Figure 2

them symmetrically. In many proposed algorithms, symmetry is considered an essential feature of the problem. This example indicates a situation in which a nonsymmetrical algorithm may be preferable. The generalized algorithm presented here allows the system to be configured optimally with respect to the workload of that system.

The generalized algorithm may deadlock under certain information structures, but this problem will be deferred to Section 5 where a modified version that can recover from potential deadlocks will be introduced. First we will give a necessary and sufficient condition for the information structures to guarantee mutually exclusive access to the critical section, then modify the algorithm to detect and recover from possible deadlock situations.

4. NECESSARY AND SUFFICIENT CONDITIONS

Arbitrary choices of inform and request sets do not necessarily result in a correct algorithm. We will require that each process maintains information about its own state and give necessary and sufficient conditions for an information structure to yield a correct algorithm.[1]

THEOREM. *Given that $i \in I_i$ for all i, properties a and b together are necessary and sufficient for a realized algorithm to guarantee mutual exclusion.*

a. I_i is a subject of R_i.

b. $\forall i, j$ either $I_i \cap I_j \neq \emptyset$ or both $j \in R_i$ and $i \in R_j$.

Property *a* requires that a process request permission from every process in its inform set before entering the critical section. The result is that processes in the inform set will always know beforehand about potential state changes into

[1] The following theorem, giving a sufficient condition when it is not required that $i \in I_i$ for all i, is proved in [8]. Conditions a and b together are sufficient to realize an algorithm that guarantees mutual exclusion.

1a. I_i is a subset of R_i.

1b. For all processes $j \neq i$, either $I_i \cap I_j \neq \emptyset$ or all of $i \in I_i$, $j \in I_j$, $i \in R_j$, and $j \in R_i$ are satisfied.

the critical section. Property b guarantees that process i has information about every other process before entering its critical section, either by communicating directly with the process or by communicating with a process that maintains up-to-date information about the process. Together these properties are necessary and sufficient to ensure correct operation of the generalized algorithm in a distributed environment. They are satisfied for the examples given previously. The requirements for a correct algorithm in [5], once placed in this framework, correspond to property a and the first predicate of property b.

PROOF. *Sufficiency*: First a simple lemma is stated.

LEMMA. *If $j \in I_i$, then j will set CSSTAT = i before i enters the critical section and reset CSSTAT = FREE after i leaves the critical section.*

PROOF OF LEMMA. This follows from the description of the algorithm and property a. □

Now consider any pair of processes i and j. Property b requires that the inform and request sets of i and j satisfy at least one of two conditions.

The first is that there is some node, say k, such that k is in the inform sets of both i and j. Since property a requires that a node in the inform set is also in the request set, i and j will both request permission from k before entering the critical section. Process k will not send a GRANT message to i while its CSSTAT variable indicates j is in the critical section and vice-versa. By the lemma, CSSTAT will always indicate that j is in the critical section when it is in the critical section, and therefore mutual exclusion is guaranteed.

The second possible situation allowed by property b is that i and j both maintain state information about themselves and request information from each other. A conflict between i and j could occur in two different ways. In the first, both i and j choose a time stamp before receiving a request from the other process. The lower priority (larger time stamp) process will immediately send a GRANT to the higher priority (smaller time stamp process). The higher priority process will not send a GRANT until it has entered and left the critical section. The second scenario occurs when a process, say j, decides to enter the critical section after it has received a REQUEST from and sent a GRANT to i. The time stamp chosen by j will always be larger than i's time stamp so i will not send a GRANT to j until it has left the critical section. □

Necessity: The necessity of property a is obvious in the context of the generalized algorithm. The necessity of property b will be proved by considering all possible situations that are not included in the theorem and showing that mutual exclusion may be violated in these cases. Given that I_i is a subset of R_i, I_j is a subset of R_j, and $I_i \cap I_j = \emptyset$, then the following relationships between I_i, R_j, I_j, and R_j are mutually exclusive and cover all the following possible situations:

(1) $R_i \cap R_j = \emptyset$;

(2) $R_i \cap I_j = \emptyset$ and $I_i \cap R_j = \emptyset$, but $R_i \cap R_j \neq \emptyset$;

(3) $R_i \cap I_j = \emptyset$, but $R_j \cap I_i \neq \emptyset$;

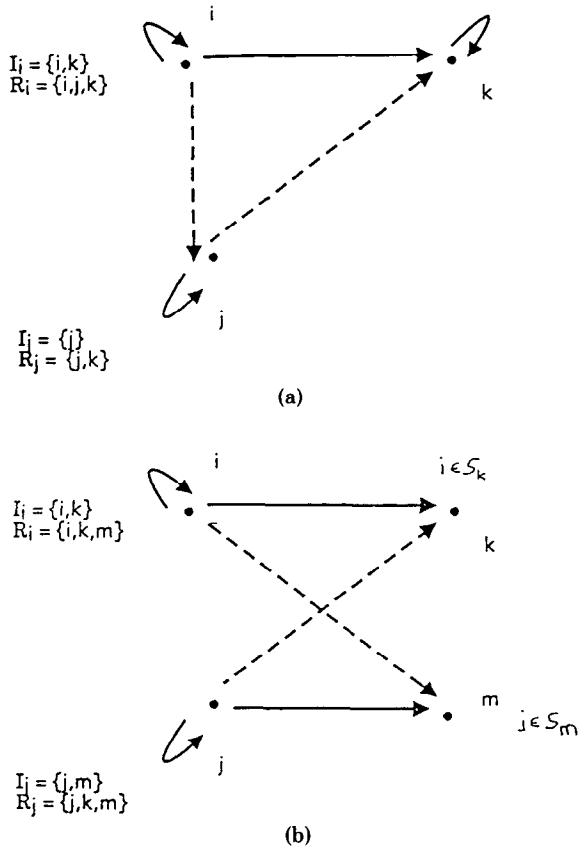

Figure 3

(4) $R_j \cap I_i \neq \emptyset$, and $R_i \cap I_j \neq \emptyset$, where this can be further broken down into three different situations.
   (4a) $i \in R_j \cap I_i$, and $j \in R_i \cap I_j$;
   (4b) $k \in R_j \cap I_i$, and $j \in R_i \cap I_j$, where $k \neq i$ and $k \neq j$ (and the symmetrical situation obtained by replacing i with j and j with i);
   (4c) $k \in R_j \cap I_i$, and $m \in R_i \cap I_j$, where $k \neq m$ and both are different from i and j.

Situation 4a satisfies property *b* as stated in the theorem. We will give counterexamples showing the violation of mutual exclusion for situations 4b and 4c and argue that mutual exclusion can also be violated in situations 1 to 3, thus proving the necessity of property *b*.

The situation in case 4b is shown in Figure 3a. The inform sets for i and j are $I_i = \{i, k\}$ and $I_j = \{j\}$. The request sets are $R_i = \{i, j, k\}$ and $R_j = \{j, k\}$, where $k \neq i$ and $k \neq j$. Process j's information about i's state is obtained indirectly via process k, whereas i's information about j is obtained directly. Let the critical section be free, but, at approximately the same time, i selects a time stamp of

say 1, and j selects a time stamp of 2. Now suppose the following events occur in the order given. A REQUEST from j arrives at k, which will immediately send a GRANT back to j. A REQUEST from i arrives at j, which will respond with a GRANT since i has the higher priority. The REQUEST from i arrives at k, which will immediately reply with a GRANT. When the GRANT from k arrives at j, j will enter the critical section. Eventually, the GRANTs that have been sent from j and k will arrive at i, and i will enter the critical section, thus violating mutual exclusion.

The situation corresponding to 4c is shown in Figure 3b. In this configuration, $I_i = \{i, k\}$, $I_j = \{j, m\}$, $R_i = \{i, k, m\}$, and $R_j = \{j, k, m\}$. Here, $k \neq m$, and k and m are different from both i and j. In this situation, mutual exclusion will be violated if i and j both request admission to the critical section at approximately the same time, and i's REQUEST to m arrives before j's REQUEST, and j's REQUEST to k arrives before the REQUEST from i. Cases 1 to 3 offer even less opportunity for cooperation than the situation in case 4. In case 3, mutual exclusion may be violated if i attempts to enter the critical section while j has control; in case 2, if either i or j is in the critical section, the other may enter; in case 1, there is no overlap between the sets of processes i and j communicate with before entering a critical section at all. □

5. DEADLOCK FREE ALGORITHM

Although the conflict resolution policy has been chosen so that all conflicts will be, in principle, resolved in the same way by different decision makers, the presence of unpredictable communication delays cause processes to sometimes violate this in reality. For example, consider the configuration of Figure 4. The inform sets $I_0 = \{0\}$, $I_1 = \{1\}$, and $I_2 = \{0, 1, 2\}$ and the request sets $R_0 = \{0, 1\}$, $R_1 = \{0, 1\}$, and $R_2 = \{0, 1, 2\}$ satisfy the conditions given in the theorem for an algorithm that guarantees mutual exclusion. Suppose the critical section is free, and 0 and 2 decide to request entrance at about the same time. Process 0 chooses a time stamp of, say 1, and process 2 chooses 2. If 2's REQUEST arrives at 1 first, process 1 will send a GRANT to 2 and set CSSTAT to indicate that 2 is in the critical section. When process 0's REQUEST arrives at 1, it will be deferred until process 1 receives a RELEASE from 2. On the other hand, 2's request to 0 will be deferred until process 0 has entered and released the critical section, since 0 has the lower time stamp. The algorithm is deadlocked at this point. Process 1, owing to communication delays, made a wrong decision.

The key to modifying the algorithm to recover from deadlocks is that process 1, for example, will know that the system has entered an unsafe state if, after sending a GRANT to a process, it receives a request from a higher priority process. When this happens, steps can be taken to revoke the GRANT and recover from a deadlock if it has occurred.

The modified algorithm that can recover from deadlock situations is described below. Three new message types, FAIL, INQUIRE, and YIELD are introduced, and the algorithms for handling the other messages are modified. A process waiting for GRANT messages may now cancel a GRANT that has been received and wait for another one, before entering the critical section.

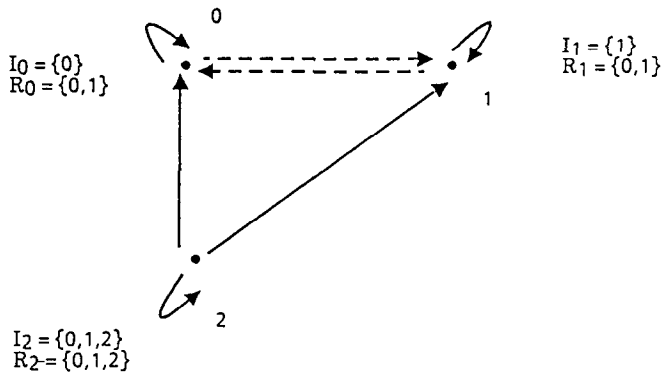

Figure 4

On receiving a REQUEST message

1. The process ID is placed on the priority queue.
2. If CSSTAT indicates that the critical section is not free, then the time stamp of the process in the critical section is compared with that of the requesting process. If the time stamp of the process in the critical section is smaller, then a FAIL message is sent to the requesting process. Otherwise, an INQUIRE message is sent to the process indicated by CSSTAT, unless one has already been sent. A FAIL message is sent to any process in the priority queue with a larger time stamp that has not yet been sent a FAIL message.
3. If CSSTAT indicates that the critical section is free, then a GRANT is sent to the process on top of the queue and the process is removed from the queue. If the recipient of the GRANT is in S_i, then CSSTAT is set to indicate that the process is in the critical section.

When a RELEASE message is received

1. CSSTAT is set to FREE.
2. A GRANT is sent to the process on top of the queue if one exists, and the process is removed from the queue. If the recipient of the GRANT is in S_i, then CSSTAT is set to indicate that the process is in the critical section.
3. Step 2 is repeated until CSSTAT indicates that a process is in the critical section or until the queue is empty.

When an INQUIRE message is received

1. The process checks the messages received from processes in its request set.
2. If it has received a FAIL message from any process, or if it has sent a YIELD to any process and not yet received a new GRANT, it revokes the GRANT from the INQUIREing process. A YIELD is sent to the INQUIREing process.

When a YIELD message is received

1. CSSTAT is marked FREE, and the YIELDing process is returned to the priority queue in the appropriate location.
2. The process proceeds as if a RELEASE message had been received.

The modified algorithm has the same necessary and sufficient conditions on the information structures to guarantee mutual exclusion. When the information structure is appropriately chosen, this algorithm is similar to the deadlock free

algorithm given in [5]. The algorithm in [5], however, neglects to send a FAIL message to the process involved in an INQUIRE/YIELD exchange when a REQUEST with a smaller time stamp arrives and could deadlock in this situation. It should be noted that the centralized algorithm [1, 6], described in Figure 1a, has no possibility of deadlock since there is a single decision point. The completely decentralized algorithm [7], described in Figure 1b, is also a special case. In this situation, each process contains only itself in its status set. Deadlock can be made impossible by waiting until grants have been received from all other processes before "sending" a grant to one's self. (This requires a modification to the requirement that all messages are handled serially; while waiting for GRANTS from other processes, the node must be able to receive them.) Alternatively, by always yielding when an inquire has been received (inquire messages in this case are always sent to one's self) the need for FAIL messages can be eliminated. A more sophisticated algorithm could make use of this information to avoid sending the new messages introduced for deadlock recovery, when it can be determined through knowledge of the global information structure that deadlocks will not occur.

6. PERFORMANCE ISSUES

In this section the performance of an algorithm is studied as a function of its information structure. The interest here is on simple performance measures that can be computed a priori. The two performance measures of interest are the number of messages required per critical section entry and the synchronization delay, defined below.

6.1 Number of Messages per Critical Section Entry

The number of messages required for critical section entry depends on the number of potential deadlocks that occur during the actual evolution of the algorithm. It is possible, however, to compute upper and lower bounds on the number of messages required as functions of the cardinality of the inform and request sets at each node.

Define: NM_i = number of messages per critical section entry by node i,

then a lower bound on the number of messages needed for a critical section entry can easily be computed.

$$|I_i - \{i\}| + 2(|R_i - \{i\}|) \leq NM_i.$$

The lower bound counts the REQUEST messages sent to and GRANT messages received from every process in R_i. It also includes the RELEASE messages sent to every process in I_i. Messages from a process to itself are not counted, and it is assumed that all processes include themselves in their inform sets.

The upper bound includes the additional messages that may be needed to recover from a potential deadlock and is given by

$$NM_i \leq |I_i - \{i\}| + 2(|R_i - \{i\}|) + DM$$

$$\text{where} \quad DM = \sum_{j \in R_i} r_j \quad \text{and} \quad r_j = \begin{cases} 1 & \text{if } S_j = \{j\} \text{ or } \{i, j\}, \\ 4 & \text{otherwise.} \end{cases}$$

The sequence of INQUIRE, YIELD, and new GRANT is associated with the process whose REQUEST induced it. The original grant has been canceled by the YIELD message and has been counted with the original REQUEST. This message exchange will occur with at most one member of the status set of each process j that is a member of R_i. If S_j contains only j or i and j, then these extra messages will not be generated. Therefore, at most zero or three INQUIRE, YIELD, and new GRANT messages are generated for each process in R_i. The extra message in each term of DM is a possible FAIL message. Note that it is possible for a REQUEST that induces an INQUIRE exchange to later receive a FAIL message from that node if a request with a smaller time stamp arrives during the INQUIRE exchange.

The actual number of messages per critical section entry will tend toward the lower bound when conflicts for access to the critical section are infrequent. A centralized information structure will realize an algorithm that can always achieve the lower bound, if the algorithm does not attempt deadlock recovery (i.e., the algorithm realized it is centralized). If not, there will still be at most two extra messages per critical section entry, since the process to whom an INQUIRE is sent will always have achieved entrance to the critical section and will not YIELD or require a new GRANT. For the completely distributed information structure, $S_j = \{j\}$ for all j and $DM = N - 1$.

6.2 Synchronization Delay

In this section, we consider the synchronization delay for various algorithms. This measure is defined for every pair of processes, and D_{ij} is the maximum number of sequential messages required after j leaves the critical section before i can enter the critical section. Process i is the next process and is assumed to be blocked while waiting for the critical section when j leaves.

$$D_{ij} = \begin{cases} 2 & \text{if } I_i \cap I_j \neq \varnothing, \\ 0 & \text{if } i = j, \\ 1 & \text{otherwise.} \end{cases}$$

The average synchronization delay for a process can be expressed as

$$D_i = \frac{1}{n-1} \sum_{j=1}^{N} D_{ij}.$$

Taking the average over all processes D, we see that for the centralized algorithm of Figure 1a, $D = 2(N-1)/n$ (note that this approaches 2 as N grows large), whereas for the completely decentralized algorithm, $D = 1$. Other information structures will give values for D ranging between 1 and 2. This is an example of the trade-off between the number of messages and the synchronization delay that often occurs in distributed decision problems. This phenomenon has received thorough treatment in [3].

The new, nonsymmetric algorithm in Figure 2 is an example of how these performance measures can be useful, particularly in a system where the demand for critical section entrance is not homogeneous. In particular, we assume that the demands for critical section entrance for nodes 0 and 1 are much higher than for the other nodes, and that we also want nodes 0 and 1 to be treated fairly with

respect to each other. In this configuration, $NM_0 = NM_1 = 2$ (since nodes 2 and 3 enter the critical section very rarely, a deadlock situation is unlikely, and we take the number of messages per critical section entry to be approximately equal to the lower bound.) Furthermore, $D_{0j} = D_{1j} = 1$ for $j = 2$ and 3, and $D_{01} = D_{10} = 1$.

7. FAILURE TOLERANCE

Failure tolerance is often considered to be one of the inherent advantages of a distributed system, since when one processor fails there are others that can continue operating. On the other hand, if necessary information is lost with a failed processor, future decisions may be incorrect unless the information can be reconstructed. If a process is waiting for some action to be taken by a process on a failed processor, unless that failure is detected, the first process may be waiting forever.

When a processor fails, a failure-tolerant mutual exclusion algorithm should be able to "mend" itself in such a way as to maintain mutual exclusion and continue operating with the remaining processors. Questions arise as to when a failure needs to be detected, who needs to know about the failure, and what should be done about it. The first two questions are answered in terms of the information structure of the algorithm. Recovery involves modifying the information structure in a way that eliminates the failed process and still satisfies the necessary conditions for a correct algorithm.

A process i needs to know about the failure of another process j in either of the two following situations:

(1) $j \in R_i$, and i wants to enter the critical section.
(2) $j \in S_i$, and CSSTAT $= j$.

For example, consider the centralized algorithm (Figure 1a). Processes 0, 2, and 3 need to know if 1 has failed when they wish to enter a critical section. Otherwise, they would wait forever for a GRANT from 1. Process 1 needs to know that 0 has failed, only if 0 fails while in the critical section. Similarly, 1 needs to know about the failure of 2, only if 2 fails while in a critical section, and likewise for 3. Processes 0, 2, and 3 do not need to know about the failure of each other.

7.1 Local Recovery

Mechanisms for recovery after the failure of a process have been proposed for the completely distributed mutual exclusion algorithm [7] and for the centralized algorithm [1, 6]. These mechanisms produce a new algorithm with the same topology as the original one and require a significant amount of cooperation between processes. A process that detects failure notifies all other processes, and in the centralized case, an algorithm to determine whether a new leader is invoked. In all cases, the problem of what happens if the failure is detected by more than one process must be considered and tends to result in complex recovery mechanisms.

A different approach to failure recovery is presented here. This approach is called local recovery, since it requires no cooperation or additional communication between processes than what is needed for failure detection. A process monitors the status of another process when it is in one of the situations described above with respect to that process. This can be done by means of timeouts and "are you up?" messages, or by implementing system routines that when called would monitor a specified processor j for failure and would interrupt the calling routing if a failure is detected. The conditions of the theorem are assumed to be satisfied by the original information structure that is known to all processes.

On detecting the failure of process j, process i takes the following actions:

(1) Delete j from i's copy of the information structure.
(2) Check to see that the information structure still satisfies the sufficient conditions to guarantee mutual exclusion. If not, it will be because there was some k such that before the failure, $I_i \cap I_k = \{j\}$.
(3) If the information structure, according to the local copy, no longer satisfies the sufficient conditions, modify the local copy so that $k \in R_i$ and $i \in R_k$.
(4) If i was waiting for a GRANT from the failed node j, then REQUEST messages should then be sent to all new additions to R_i. If i was waiting for a RELEASE from j, then i should act as if a RELEASE had actually been received: clearing CSSTAT and sending a GRANT to the next waiting process.

Although the topological structure of the algorithm is not preserved under this failure recovery mechanism, it has the advantage of requiring absolutely no coordination or communication between various processes to maintain correctness. The local copies of the information structures may be inconsistent during the evolution of the algorithm, for example, when the failure has not yet been detected by all relevant processes, but the algorithm will never violate mutual exclusion.

8. DYNAMIC INFORMATION STRUCTURES: AN EXAMPLE

The work presented above has assumed that the information structure of an algorithm is static. The only time that an information structure changed was during failure recovery. In this section an example is given of an algorithm with an information structure that changes as the algorithm evolves. This algorithm was proposed in [2] as an improvement to the completely decentralized algorithm of [7]. The idea is that, initially, a process wishing to enter the critical section will send a REQUEST to and wait for a GRANT from all processes before entering the critical section. This step is the same as in [7]. The improvement comes when one notices that if the same process wants to enter the critical section again, and no other process is attempting to enter, then there actually is no need to request permission from all other processes. More generally, a process will only ask permission from processes that it has never received permission from, or that have been granted permission by it since it has been granted permission by them.

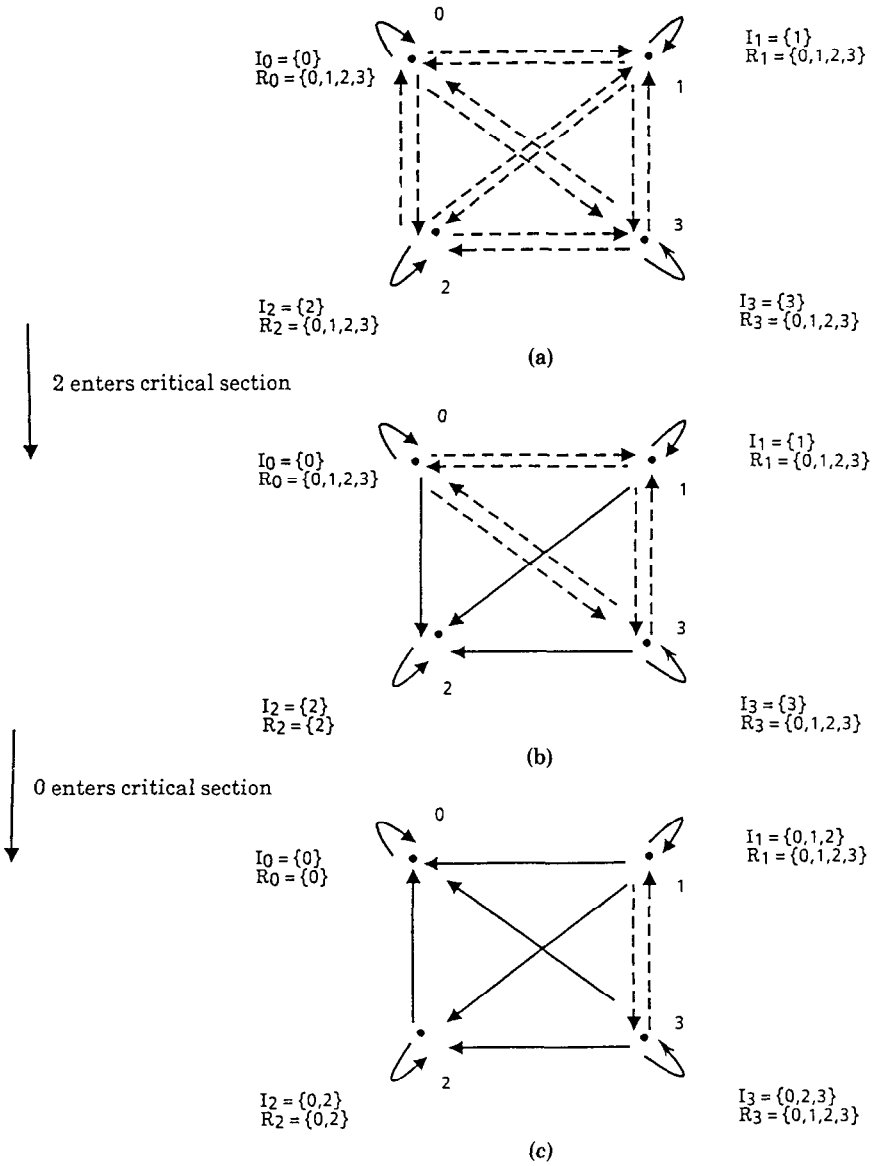

Figure 5

Casting this in our framework, we see that the information structure changes during the normal evolution of the algorithm. The information structure of process i changes according to the following rules:

(1) When a GRANT is sent to a process j, j is added to I_i and R_i and removed if present from S_i.

(2) When a GRANT from process i is received by j, i is removed from R_j and added to S_j.

This is illustrated in the example shown in Figure 5. The important consequence of dynamic information structures is that the complete information structure is not known to any process. Each node knows the initial information structure, the rules for changing the information structure, and the messages sent and received by itself. The process does not know the message traffic at the other process and the resulting changes in the global information structure. An algorithm with dynamic information structures will be satisfied if two conditions are satisfied. First, the union of the local views must satisfy the conditions of the theorem, where i's local view is just I_i, R_i, and S_i. Second, the local view must be consistent in the sense that if $j \in I_i$, then $i \in S_j$. In this example, for any pair of processes i and j, either $j \in R_i$ and $i \in R_j$, or one process is in the inform set of the other. The local views of the information structure are indicated on the figure. Note that in Figure 5c, process 3 would need to communicate only with processes 0 and 2 to maintain correctness of the algorithm. Because 3 does not have knowledge of the actual inform sets of 0 and 1, it cannot tell that it could be obtaining information indirectly through these processes.

9. CONCLUSION

In this report, the concept of an information structure of a distributed mutual exclusion algorithm has been introduced. The information structure captures the distinguishing features of several algorithms that have appeared in the literature, as well as facilitating the discovery of new ones. Future research will include further exploration of dynamic information structures and applications of these ideas to other distributed decision problems.

ACKNOWLEDGMENT

I would like to thank Ashok Agrawala for many stimulating discussions and also for his comments on an earlier draft of the paper. The detailed comments by the referees were also helpful.

REFERENCES

1. BUCKLEY, G., AND SILBERSCHATZ, A. A failure tolerant centralized mutual exclusion algorithm. In *Proceedings of the 4th International Conference on Distributed Computing Systems* (May 1984), pp. 347–356.
2. CARVALHO, S. F., AND ROUCAIROL, G. On mutual exclusion in computer networks. *Commun. ACM 26*, 2 (Feb. 1983), 146–147.
3. LAKSHMAN, T. V., AND AGRAWALA, A. K. Efficient decentralized consensus protocols. *IEEE Trans. Softw. Eng.* (May 1986), 600–607.
4. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* (July 1978), 558–564.
5. MAEKAWA, M. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst. 3*, 2 (May 1985), 145–159.
6. MOHAN, C., AND SILBERSCHATZ, A. Distributed control—Is it always desirable? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems* (July 1981).
7. RICART, G., AND AGRAWALA, A. K. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM 24*, 1 (Jan. 1981), 9–17.
8. SANDERS, B. The information structure of distributed mutual exclusion algorithms. University of Maryland, Dept. of Computer Science, Tech. Rep. CS-TR-1644, (June 1986).

Received June 1986; revised December 1986; accepted March 1987