Operating Systems

R. Stockton Gaines
Editor

An Optimal Algorithm for Mutual Exclusion in Computer Networks

Glenn Ricart
National Institutes of Health

Ashok K. Agrawala
University of Maryland

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. The algorithm sends only 2*(N − 1) messages, where N is the number of nodes in the network per critical section invocation. This number of messages is at a minimum if parallel, distributed, symmetric control is used; hence, the algorithm is optimal in this respect. The time needed to achieve mutual exclusion is also minimal under some general assumptions.

As in Lamport's "bakery algorithm," unbounded sequence numbers are used to provide first-come first-served priority into the critical section. It is shown that the number can be contained in a fixed amount of memory by storing it as the residue of a modulus. The number of messages required to implement the exclusion can be reduced by using sequential node-by-node processing, by using broadcast message techniques, or by sending information through timing channels. The "readers and writers" problem is solved by a simple modification of the algorithm and the modifications necessary to make the algorithm robust are described.

Key Words and Phrases: concurrent programming, critical section, distributed algorithm, mutual exclusion, network, synchronization

CR Categories: 4.32, 4.33, 4.35

Authors' addresses: G. Ricart, Division of Computer Research and Technology, National Institutes of Health, Bethesda, MD 20205; A.K. Agrawala, Department of Computer Science, University of Maryland, College Park, MD 20742.

This research was supported in part by the Air Force Office of Scientific Research under grant AFOSR 78-3654A.

1. Introduction

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. It is assumed that there is an error-free underlying communications network in which transit times may vary and messages may not be delivered in the order sent. Nodes are assumed to operate correctly; the consequences of node failure are discussed later. The algorithm is symmetrical, exhibits fully distributed control, and is insensitive to the relative speeds of nodes and communication links.

The algorithm uses only 2* (N − 1) messages between nodes, where N is the number of nodes and is optimal in the sense that a symmetrical, distributed algorithm cannot use fewer messages if requests are processed by each node concurrently. In addition, the time required to obtain the mutual exclusion is minimal if it is assumed that the nodes do not have access to timing-derived information and that they act symmetrically.

While many writers have considered implementation of mutual exclusion [2,3,4,5,6,7,8,9], the only earlier algorithm for mutual exclusion in a computer network was proposed by Lamport [10,11]. It requires approximately 3* (N − 1) messages to be exchanged per critical section invocation. The algorithm presented here requires fewer messages (2* (N − 1)).

2. Algorithm

2.1 Description

A node enters its critical section after all other nodes have been notified of the request and have sent a reply granting their permission. A node making an attempt to invoke mutual exclusion sends a REQUEST message to all other nodes. Upon receipt of the REQUEST message, the other node either sends a REPLY immediately or defers a response until after it leaves its own critical section.

The algorithm is based on the fact that a node receiving a REQUEST message can immediately determine whether the requesting node or itself should be allowed to enter its critical section first. The node originating the REQUEST message is never told the result of the comparison. A REPLY message is returned immediately if the originator of the REQUEST message has priority; otherwise, the REPLY is delayed.

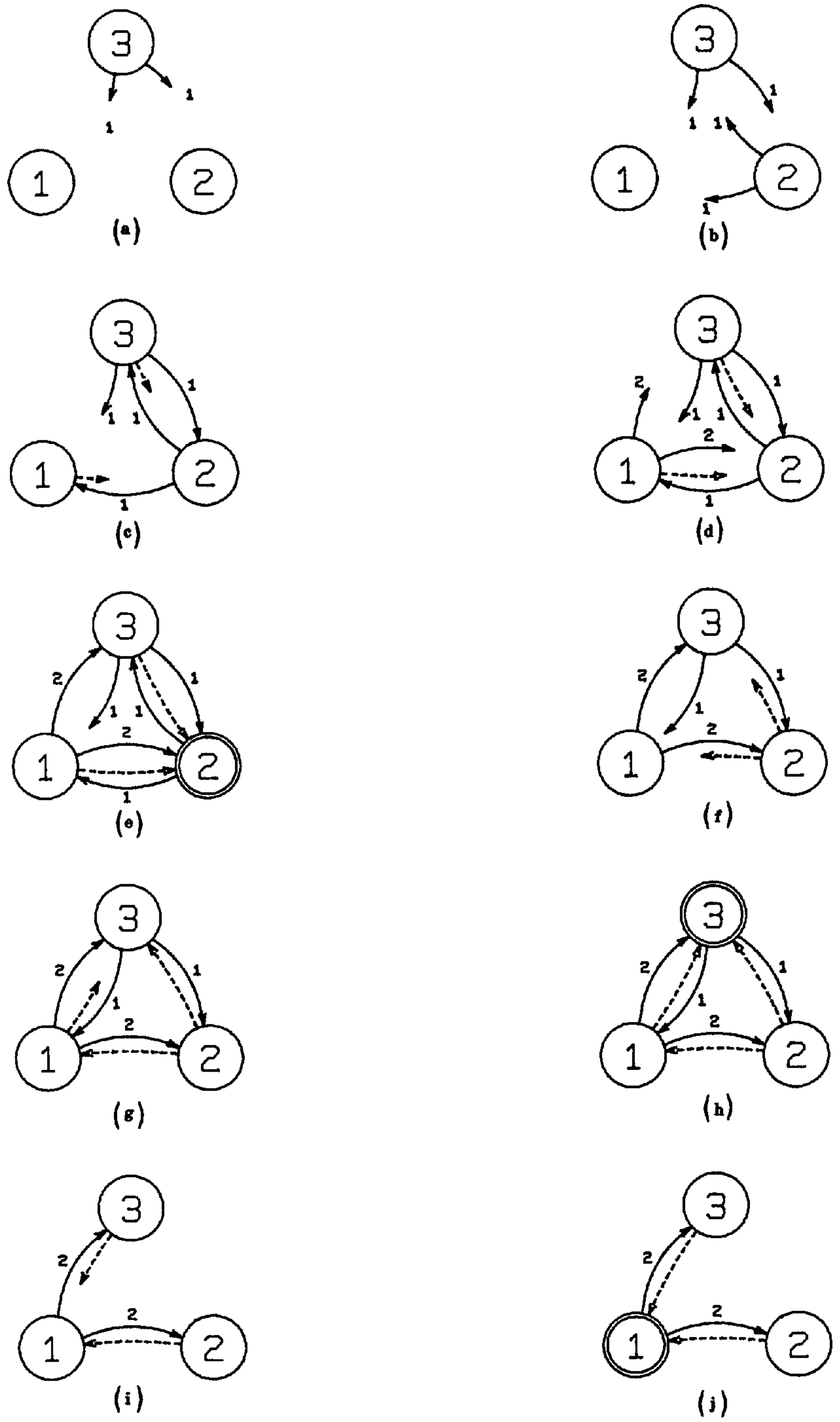
The priority order decision is made by comparing a sequence number present in each REQUEST message. If the sequence numbers are equal, the node numbers are compared to determine which will enter first.

2.2 Specification

The network consists of N nodes. Each node executes an identical algorithm but refers to its own unique node number as ME.[1]

[1] ME is a pun on "mutual exclusion."

Fig. 1. Three-node mutual exclusion example. (See Sect. 2.3.)



Each node has three processes to implement the mutual exclusion:

(1) One is awakened when mutual exclusion is invoked on behalf of this node.
(2) Another receives and processes REQUEST messages.
(3) The last receives and processes REPLY messages.

The three processes run asynchronously but operate on a set of common variables. A semaphore is used to serialize access to the common variables when necessary. If a node can generate multiple internal requests for mutual exclusion, it must have a method for serializing those requests.

The algorithm is expressed below in an Algol-like language.

```
SHARED DATABASE
CONSTANT
                 me, ! This node's unique number
                 N;  ! The number of nodes in the network
INTEGER          Our_Sequence_Number,
                    ! The sequence number chosen by a request
                    ! originating at this node
                 Highest_Sequence_Number initial (0),
                    ! The highest sequence number seen in any
                    ! REQUEST message sent or received
                 Outstanding_Reply_Count;
                    ! The number of REPLY messages still
                    ! expected
BOOLEAN          Requesting_Critical_Section initial (FALSE),
                    ! TRUE when this node is requesting access
                    ! to its critical section
                 Reply_Deferred [1:N] initial (FALSE);
                    ! Reply_Deferred [j] is TRUE when this node
                    ! is deferring a REPLY to j's REQUEST message
BINARY SEMAPHORE
                 Shared_vars initial (1);
                    ! Interlock access to the above shared
                    ! variables when necessary

PROCESS WHICH INVOKES MUTUAL EXCLUSION FOR
THIS NODE
Comment Request Entry to our Critical Section;
  P (Shared_vars)
     Comment Choose a sequence number;
     Requesting_Critical_Section := TRUE;
     Our_Sequence_Number := Highest_Sequence_Number + 1;
  V (Shared_vars);
  Outstanding_Reply_Count := N - 1;
  FOR j := 1 STEP 1 UNTIL N DO IF j ≠ me THEN
       Send_Message(REQUEST(Our_Sequence_Number,me),j);
  Comment sent a REQUEST message containing our sequence num-
  ber and our node number to all other nodes;
Comment Now wait for a REPLY from each of the other nodes;
  WAITFOR (Outstanding_Reply_Count = 0);
Comment Critical Section Processing can be performed at this point;
Comment Release the Critical Section;
  Requesting_Critical_Section := FALSE;
  FOR j := 1 STEP 1 UNTIL N DO
    IF Reply_Deferred[j] THEN
    BEGIN
      Reply_Deferred[j] := FALSE;
      Send_Message (REPLY, j);
      Comment send a REPLY to node j;
    END;

PROCESS WHICH RECEIVES REQUEST (k, j) MESSAGES
Comment  k is the sequence number begin requested,
         j is the node number making the request;
BOOLEAN Defer_it   ;
                   ! TRUE when we cannot reply immediately
Highest_Sequence_Number :=
   Maximum (Highest_Sequence_Number, k);
P (Shared_vars);
   Defer_it :=
        Requesting_Critical_Section
        AND ((k > Our_sequence_Number)
                OR (k = Our_Sequence_Number AND j > me));
V (Shared_vars);
Comment Defer_it will be TRUE if we have priority over
  node j's request;
IF Defer_it THEN Reply_Deferred[j] := TRUE ELSE
   Send_Message (REPLY, j);

PROCESS WHICH RECEIVES REPLY MESSAGES
Outstanding_Reply_Count := Outstanding_Reply_Count - 1;
```

The REPLY processing can be represented by a decision table:

Condition and action entries	Rule number				
	1	2	3	4	5
Receiving node is also requesting the resource	*N*	*Y*	*Y*	*Y*	*Y*
Received message's sequence number compared to ours	–	<	>	=	=
Received message's node number compared to ours	–	–	–	<	>
Send REPLY back	*X*	*X*		*X*	
Defer the REQUEST			*X*		*X*

2.3 Example

Imagine a three-node network using this algorithm. Initially the highest sequence number at each node is zero. Solid lines show REQUEST messages; the number is the sequence number of the request. The dashed lines show REPLY messages.

In Figure 1(a), node 3 is the first to attempt to invoke mutual exclusion. It chooses sequence number 1 and sends REQUEST messages to nodes 1 and 2.

Before either message can arrive, node 2 wishes to enter its critical section. It also chooses sequence number 1 and sends REQUEST messages to the other nodes (Figure 1(b)).

In Figure 1(c) node 2's messages have arrived. At node 1, which has not yet made a request itself, a REPLY is immediately generated. At node 3, 2's request is found to have an identical sequence number to 3's request; node 2 wins on the node number tie-breaking rule. A REPLY is sent. But at node 2, 3's request is found to have an identical sequence number but loses the tie-breaker. A reply is deferred.

Figure 1(d) shows node 1 making a request to enter its critical section. It uses sequence number 2 since it has received a REQUEST message with a sequence number of 1 (from node 2). Owing to an anomaly in the communications system, the REQUEST message to node 2 overtakes the REPLY that is on its way there. No reply message is sent since the message's sequence number is higher than node 2's sequence number.

In Figure 1(e), node 2 can now enter its critical section since it has received both of the necessary replies. Node 1's REQUEST has also arrived at node 3 but has been deferred since the request's sequence number is higher than that selected by node 3.

When node 2 has finished its critical section processing, it sends REPLY messages back to both nodes 1 and 3 (Figure 1(f)).

In Figure 1(g), nodes 1 and 3 have received their REPLY messages from node 2 but not yet from each other. Node 3's request has arrived at node 1. Since it bears a smaller sequence number, a REPLY is immediately generated.

Figure 1(h) shows node 3 entering its critical section after it received both replies.

In Figure 1(i), node 3 has finished its critical section processing and is returning the deferred REPLY message to node 1.

Finally in Figure 1(j), node 1 begins critical section processing. At the conclusion of its critical section, node 1 does nothing since it knows of no other node wishing to invoke mutual exclusion.

2.4 Discussion

The sequence numbers are similar to the numbers used by Lamport's "bakery algorithm." [9] The node with the lowest number is the next one to enter the critical section. Ties are broken by comparing node numbers. A REPLY is generated when its sender agrees to allow the node sending a REQUEST to enter its critical section first.

The sequence numbers prevent high numbered nodes from being "shut-out" by lower numbered nodes. Once node *A*'s REQUEST messages have been processed by all other nodes, no other node may enter its critical section twice before node *A* has entered its critical section.

The sequence numbers and node numbers form a virtual ordering among requesting nodes. No one of the nodes has any more information than a list of some or all of the other nodes following it in the virtual order. Yet the system as a whole defines a unique virtual ordering based on a first-come-first-served discipline.

3. Assertions

3.1 Mutual Exclusion

Mutual exclusion is achieved when no pair of nodes is ever simultaneously in its critical section. For any pair of nodes, one must leave its critical section before the other may enter.

ASSERTION. *Mutual exclusion is achieved.*

PROOF. Assume the contrary, that at some time two nodes (*A* and *B*) are both in their critical sections at the same time. Examine the message traffic associated with the current cycle of the algorithm that occurred in each node just prior to this condition. Each node sent a REQUEST to the other and received a REPLY.

CASE 1: Node *A* sent a REPLY to Node *B*'s REQUEST before choosing its own sequence number. Therefore *A* will choose a sequence number higher than *B*'s sequence number. When *B* received *A*'s REQUEST with a higher number, it must have found its own Requesting_Critical_Section = **TRUE** since this is set to be **TRUE** before sending REQUEST and *A* had received this request before sending its own REQUEST. The algorithm then directs *B* to defer the REQUEST and not reply until it has left its critical section. Then node *A* could not yet be in its critical section contrary to assumption.

CASE 2: Node *B* sent a REPLY to *A*'s REQUEST before choosing its own sequence number. This is the mirror image of Case 1.

CASE 3: Both nodes sent a REPLY to the other's REQUEST after choosing their own sequence numbers. Both nodes must have found their own Requesting_Critical_Section to be **TRUE** when receiving the other's REQUEST message. Both nodes will compare the sequence number and node number in the REQUEST message to their own sequence and node numbers. The comparisons will develop opposite senses at each node and exactly one will defer the REQUEST until it has left its own critical section contradicting the assumption.

Therefore, in all cases the algorithm will prevent both nodes from entering their critical sections simultaneously and mutual exclusion is achieved.

3.2 Deadlock

The system of nodes is said to be deadlocked when no node is in its critical section and no requesting node can ever proceed to its own critical section.

ASSERTION. *Deadlock is impossible.*

PROOF. Assume the contrary, that deadlock is possible. Then all requesting nodes must be unable to proceed to their critical sections because one or more REPLYs are outstanding. After a sufficient period of time, the only reason that the REPLY could not have been received is that the REQUEST is deferred by another node which itself is waiting for REPLYs and cannot proceed. Therefore, there must exist a circuit of nodes, each of which has sent a REQUEST to its successor but has not received a REPLY.

Since each node in the loop has deferred the REQUEST sent to it, it must be requesting the critical section itself and have found that the sequence number/node number pair in that REQUEST was greater than its own. However, this cannot hold for all nodes in the supposed circuit, and thus the assertion must be true.

3.3 Starvation

Starvation occurs when one node must wait indefinitely to enter its critical section even though other nodes are entering and exiting their own critical sections.

ASSERTION. *Starvation is impossible.*

PROOF. Assume the contrary, that starvation is possible. Nodes receiving REQUEST messages will process them within finite time since the process which handles them does not block. After processing the REQUEST sent by the starving node, a receiving node cannot issue any new requests of its own with the same or lower sequence number. After some period of time the sequence number of the starving node will be the lowest of any requesting node. Any REQUESTs received by the starving node will be deferred, preventing any other node from entering its critical section. By the previous assertion, deadlock cannot occur and some process must be able to enter its critical section. Since it cannot be any other process, the starving process must be the one to enter its critical section.

4. Message Traffic

This algorithm requires one message to (REQUEST) and one message from (REPLY) each other node for each entry to a critical section. If the network consists of N nodes, $2^*(N-1)$ messages are exchanged. It will be shown that this number is the minimum required when nodes act independently and concurrently. Hence, the algorithm is optimal with regard to the number of messages exchanged.

4.1 Concurrent Processing

For a symmetrical, fully distributed algorithm there must be at least one message into and one message out of each node. If no message enters/leaves some node, that node must not have been necessary to the algorithm; then the algorithm is not symmetrical or is not fully distributed. Furthermore, to allow the algorithm to operate concurrently at all nodes, the messages entering nodes must not wait for the message generated at the conclusion of processing at other nodes. This would indicate that two separate messages per node are required. The requesting node does not need to send and receive messages to itself, however, and so a total of $2^*(N-1)$ messages are needed. This number must be a minimum for any parallel, symmetric, distributed algorithm.

4.2 Serial Processing

If the nodes do not act independently of each other, it is possible to reduce the number of messages by using serial node-by-node processing. The first condition discussed earlier (one message into and out of each node) still holds so a minimum of N messages are required. No parallelism can exist in such a structure since a message out of a node must double as the message into some other node.

If the algorithm presented here is modified so that messages are sent from node to node sequentially, it achieves the theoretical minimum number of messages in this case also. Parallel operation is necessarily sacrificed. The modifications required are considered in Section 6.3.

5. Delay in Granting Critical Sections

The algorithm also grants mutual exclusion with minimum delay if some general assumptions are made.

5.1 Definition of Delay

The delay involved in granting the critical section resource is the stretch of time beginning with the requesting node asking for the critical section and ending when that node enters its critical section. The execution time of the instructions in the algorithm is assumed to be negligible compared to the message transmission times.

5.2 Assumptions

The following assumptions prevent the use of central control or extra information derived from timing:

Assumption 1. No information is available bounding transmission time delays or giving actual transit times. Because of this assumption, it takes one round-trip time to determine the state of another node. By adopting this assumption, sending information through timing channels becomes impossible.

Assumption 2. No node possesses the critical section resource when it has not been requested. This assumption prevents a node or series of nodes from acting as a central control because it retained the critical section resource.

Assumption 3. Nodes do not anticipate requests.

5.3 Bounds

Three conditions that put a lower bound on delay times are developed and the mutual exclusion algorithm is shown to achieve these bounds.

5.3.1 Bound 1: Minimum delay time per request. Before a node enters its critical section, it must make sure that no other node is entering. To do this it must determine the current status of any other node that could take precedence if there is a time overlap and both nodes are said to be requesting concurrently [10]. By assumption 1 this will take at least one round-trip transmission time. By assumptions 2 and 3 this process cannot start before the request arrives. Therefore, no request can be serviced in less than one round-trip time.

5.3.2 Bound 2: Minimum delay time with conflict. When two nodes are requesting concurrently, they do not know which of them made their request first because of the absence of timing information. A tie-breaking scheme, representing a total ordering among requesting nodes, must be used. Since the tie-breaking rule does not know which node actually made the earlier request, half of the time a critical section grant cannot be made until after the node making the later request has received its round-trip replies. Conflict may also occur with more than two nodes. One of them must be selected by the tie-breaker to be granted access to its critical section first.

5.3.3 Bound 3: System throughput. Once a node has released the critical section resource, no other node can enter its critical section in less than a one-way trip transmission time. This is the minimum amount of time needed to notify other nodes that critical section processing has been completed and to transmit the new values of network-wide information.

5.4 Compliance

The algorithm achieves these bounds:

CASE A: If when a critical section is released at least one node is eligible to enter its critical section based on Bounds 1 and 2 within a one-way trip time in the future, the algorithm will achieve the more ambitious Bound 3.

If the next node to enter its critical section is eligible under Bounds 1 and 2 within a one-way trip time in the future, then at least one one-way trip time has elapsed already since that node made its request. Since it is next, only the node currently releasing the critical section could be delaying a REPLY message and this REPLY will be triggered by the release of the critical section. This final reply will reach the next node in a one-way trip time satisfying Bound 3.

CASE B: Case A does not hold. The algorithm achieves Bound 1 or Bound 2 depending upon interference. The node with lowest sequence number/node number pair among requesting nodes will have none of its requests queued by other nodes and, hence, will enter its critical section in the minimum amount of time given by Bounds 1 and 2.

In short, the algorithm achieves Bound B whenever it can do so without violating Bounds 1 and 2. The algorithm therefore has minimal delay times under assumptions 1, 2, and 3.

The delay time envelope when plotted against arrival rate is discussed further in [12].

When a particular network has closely bounded message delay times and either synchronized clocks or knowledge of transit times, this timing information can be used to reduce delay times still further [13].

6. Modifications

Several interesting modifications can be made to the algorithm to take advantage of different environments.

6.1 Implicit Reply

The REPLY message carries only a single bit of information. When the message transmission time between nodes has an upper bound, the sense of the response can be changed so that no reply within that time period indicates an implicit reply. An explicit message, called "DEFERRED", is sent when REPLY would ordinarily not be sent.

The number of messages required by the implicit reply scheme varies between $1*(N-1)$ and $3*(N-1)$ depending on the number of DEFERRED messages sent. When there is little contention for the critical section resource, the number of messages approaches $1*(N-1)$.

Since a requesting node must usually wait for the maximum round-trip time before entering its critical section, the usefulness of this modification depends on an upper bound for transmission time which is not much larger than the average.

6.2 Broadcast Messages

When the communications structure between nodes permits broadcast messages, the initial REQUEST message can be sent using that mechanism. The message traffic is reduced to N messages, one broadcast REQUEST and $(N-1)$ REPLYs. If combined with the implicit reply modification discussed above, the message count can be as low as one.

6.2.1 Communications medium sequencing. Broadcast REQUEST messages need not contain the usual sequence number if their time of successful transmission can be monitored. The broadcast medium enforces serialization of the REQUESTs and a queueing order equivalent to the sequence numbers may be obtained by observing the order of REQUEST messages appearing on the broadcast medium.

The REPLY messages can also be broadcast, and only two messages per critical section invocation are required. REPLYs are only needed from those other nodes which have themselves successfully broadcast a prior REQUEST but received no corresponding REPLY.

6.2.2 No communications medium sequencing. Even if the order of successful REQUEST broadcasts cannot be monitored, it is useful to broadcast the REPLY messages following critical section processing. The size of the audience depends on the degree of contention. A broadcast REPLY message must contain a list of intended recipients because it is not sufficient for nodes waiting for a REPLY to assume it applies to them.[2]

[2] Example: While node 1 is performing critical section processing related to its request with sequence number 1, node 2 decides to issue a REQUEST message with sequence number 2. Before the REQUEST message arrives at node 1, node 1 completes its critical section processing and broadcasts the REPLY it owes some other node(s). Without a list of intended recipients, node 2 might think that the REPLY applies to its REQUEST message and continue. In fact, node 1 may make a new request with sequence number 2 and be entitled to enter its critical section first due to the tie-breaking rule.

6.3 Ring Structure

The number of messages can be cut to N by processing the requests serially through a logical circuit consisting of all nodes instead of allowing processing to proceed concurrently. N is the minimum number of messages required for any distributed symmetric algorithm when broadcasting is not available and information is not sent via timing channels.[3]

The algorithm must be modified by replacing the REPLY message with an echo of the REQUEST message. As the REQUEST message travels around the circuit of nodes, it may be deferred at several stops. When it is received at the initiating node, mutual exclusion has been achieved and critical section processing may begin.

A further possible modification sends the REQUEST message from node to node around the circuit without pause but the notation "DEFERRED by node j" is added by each node j that is copying and deferring the request. The Outstanding_Reply_Count is then set according to the notations when it arrives back at the initiating node. The nodes which have marked the REQUEST as deferred generate individual REPLYs in the usual way. This technique comes close to N messages while eliminating the cumulative delays at each stop.

6.4 Bounding Sequence Numbers

The sequence numbers in the algorithm increase at each critical section invocation and are theoretically unbounded. The ticket numbers of the "bakery algorithm" [9] suffer from the same problem.

A technique for limiting the amount of storage necessary to hold these unbounded numbers can be borrowed from computer communications protocols. Although the numbers themselves are unbounded, their range is bounded. The sequence numbers increase by no more than one each time a node requests entry to its critical section. That request cannot be granted as long as a lower sequence number request is outstanding. Therefore the numbers must fall within the range from x to $x + N - 1$.

The sequence numbers can be stored modulo M where $M \geq 2N - 1$. When making a comparison, the smaller number should be increased by M if the difference is N or more. Thus only $log_2(2N - 1)$ bits of storage are needed regardless of the number of times the critical section is entered.

6.5 Sequence Number Incrementation

Aside from this method for limiting the storage required to hold sequence numbers, there is no reason for incrementing sequence numbers in unit steps. Two situations make larger increments attractive:

(1) The algorithm tends to favor lower numbered nodes slightly, owing to the tie-breaking rule. This favoritism can be reduced by incrementing the sequence number by a random integer. The tie-breaking node number is still required in case the random integers used were equal.

(2) Deliberate priority can be introduced by instructing high priority nodes to use small increments and low priority nodes to use large increments. In addition, high priority nodes may be allowed to monopolize critical section processing until forced to increment their sequence numbers past the one chosen by a lower priority node. In doing so, the process at a high priority node which receives and handles messages may choose to delay acting on those received from low priority nodes in order to keep the Highest_Sequence_Number from being prematurely incremented past the one chosen by the low priority node.

[3] To involve all nodes, at least one message must be received and one sent per node. The minimum number of messages that meet this requirement is N.

6.6 Readers and Writers

The algorithm is easily modified to solve the "Readers and Writers" problem [1] where writers are given priority. The modification is simply that "readers" never defer a REQUEST for another "reader"; instead they always REPLY immediately. "Writers" follow the original algorithm.

7. Considerations for Practical Networks

7.1 Node Numbers

It is more convenient to draw node numbers from a larger range than 1 . . . N. The algorithm may be changed to map the integers 1 . . . N into the actual node numbers by indexing a table NAMES [1 . . . N]. The comparison of node numbers should then be performed by comparing the values contained in NAMES.

7.2 Insertion of New Nodes

New nodes may be added to the group participating in the mutual exclusion algorithm. They must be assigned unique node numbers, obtain a list of participating nodes, be placed on every other node's list of participants, and acquire an appropriate value for their Highest_Sequence_Number variable.

7.2.1 Restart interval. If the node could have been previously operational in the group (e.g., it failed and is now restarting), it should first notify other nodes that it failed and then wait long enough to be sure its old messages were delivered and the network processed its removal. Usually the network will already be aware of the node's failure, but this cannot be assumed. If this step was not followed, the failure may be detected at approximately the same time as the node rejoins the group. This would result in conflicting bookkeeping at different nodes.

7.2.2 Reconcile participant lists. A new node must obtain a list of other participating nodes and have itself added to the others' lists. A new node should contact a "sponsor" node which is already participating in the group. The sponsor should then invoke mutual exclusion, initialize the new node's participant list from its own, and broadcast the new node's identity before releasing mutual exclusion. Each node receiving this notification adds the new node number to its NAMES array and increments N, the number of active nodes.

An alternative is possible if the communications network can deliver a message to all other nodes without the sender naming all the other nodes in the network. In this case a new node obtains a list of participants from a nearby node and then sends a broadcast message asking all other nodes to include it on their list of participating nodes.

7.2.3 Set highest sequence number. The Highest_Sequence_Number variable of a new node must not be set to any value lower than the sequence number of any REQUEST message which would already have been received had the new node been continuously active. Until an appropriate value of Highest_Sequence_Number is obtained, mutual exclusion cannot be requested and incoming REQUEST messages are processed normally.

A new node can determine that its Highest_Sequence_Number is high enough by several methods.

(1) Ask all other nodes for their Highest_Sequence_Number and use the largest.
(2) Wait until one REQUEST message has been received from every other node.
(3) Wait until the sequence numbers on REQUEST messages have increased by $N - 1$.
(4) Wait until all $(N - 1)$ nodes would have time to enter and leave their critical sections even if they all had outstanding requests. This requires the ability to bound message transmission times and critical section times. If no REQUEST message is received during this time, the value of Highest_Sequence_Number from any nearby node can be used.
(5) Wait until the fourth REQUEST message is received from a single node. This method requires that messages are sent and delivered in the same order. [See Appendix.]

The new node may request access to its critical section after any of the above methods has been used to verify that its Highest_Sequence_Number variable is sufficiently high.

7.3 Removal of Nodes

A node wishing to leave the group may do so by notifying all other nodes of its intention. The other nodes should acknowledge this message. While waiting for acknowledgement, the departing node may not request mutual exclusion and must continue to send REPLY messages to any REQUEST messages it receives. Each node checks to see if the departing node is listed in its NAMES array, and if so, removes it and decrements the value of N, the number of active nodes, by one. If messages may be delivered out of order, a node awaiting a REPLY message from a departing node should pretend the REPLY was received.

7.4 Node Failures

In practice some nodes fail and will not respond to messages directed at them. To prevent this situation from stopping the proposed mutual exclusion algorithm, a timeout–recovery mechanism may be added. The timeout detection of a failed node relies on knowledge of an upper bound on the time which may elapse before a working node responds to a message and an estimate of the maximum processing time within a critical section. The only message in the original algorithm which demands a response is the REQUEST message.

A requesting node should start a timer when the REQUEST messages are sent. The timer should be restarted when a REPLY is received and cancelled when the critical section processing begins.

A bit map, Awaiting_Reply [1 . . . N], can be used to identify which nodes have not yet sent a REPLY message. The Awaiting_Reply array is set to all **TRUE** values before a REQUEST message is issued. Individual bits are turned off when REPLY messages are received.

If the timer expires,[4] all nodes for which Awaiting_Reply is **TRUE** are suspected of having failed. A probing message, ARE_YOU_THERE(me), should be sent to each suspect node. If no answer is received during a second timeout period,[5] the suspect node has failed.

When an ARE_YOU_THERE(j) message is received, Reply_Deferred[j] should be examined. If it is **FALSE**, it must be that the REQUEST was not received, the REPLY was lost, or the node has restarted; the correct response is REPLY(me). If Reply_Deferred[j] is **TRUE**, a YES_I_AM_HERE message should be sent to confirm that the node is alive.

The timeout does not impose an upper limit on the duration of a critical section. If critical section processing exceeds the timeout, all nodes will respond with YES_I_AM_HERE messages and a new timeout period may begin.

When it has been determined that node j has failed, this can be broadcast by the node detecting the failure. Any node which is awaiting a REPLY message from the failed node should pretend that a REPLY was received. In addition the node should be erased from the NAMES array if present and N, the number of active nodes, decremented by one.

If the failed node recognizes that it has failed and has been restarted, it may return to the group through

[4] The appropriate value is worst-case round-trip message transmission time plus worst-case processing time at the distant node plus a reasonable estimate of maximum critical section time.

[5] In this case just round-trip message time plus worst-case processing time at the distant node.

the mechanism for adding a new node. If it does not know that it has failed and issues new REQUEST messages, any node which receives the REQUEST message and does not find the node's name in its NAMES array may return a special message notifying the node that it should restart itself and use the insertion protocol.

8. Conclusion

An algorithm is presented that implements mutual exclusion in a computer network. No algorithm uses fewer messages, operates faster, and exhibits concurrent, symmetric, and distributed control. The algorithm is safe and live and mechanisms exist to handle node insertion, removal, and failure.

Modifications can be made to reduce the number of messages by taking advantage of serial processing, broadcast messages, and transmitting information through omitted responses. The sequence numbers can be stored in limited memory by keeping them as residues of a modulus that is at least twice as large as the number of nodes. The readers and writers problem is solved by the same algorithm with a simple modification.

Appendix. The Effect of Message Ordering

The algorithm presented in this paper does not depend on messages being delivered or acted upon in the order in which they are sent. If such a condition does exist, there is a stronger limit to the number of times other nodes can enter their critical sections before a requesting node A can.

Without delivery in order of transmission, the worst case analysis shows that $N(N+1)/2 - 1$ nodes can enter their critical section before Node A may.

To determine this bound, assume that A has the highest node number and therefore the least priority in breaking ties. A's sequence number may be $(N - 1)$ higher than the lowest outstanding sequence number. (See Section 6.4.) It is possible, by judiciously ordering the delivery of messages, for each other node to enter its critical section with its sequence number taking on each value between its current value and A's value. To get the worst case, assume that all nodes have chosen a distinct sequence number with A's number the highest. Therefore, one node can enter its critical section N times before A may, another $(N - 1)$, another $(N - 2)$ and so on down to the node whose REQUEST message caused A's sequence number selection. This takes two critical section entries at most. This sum, $N + (N - 1) + (N - 2) + \ldots + 3 + 2$, is the number of times other nodes may enter their critical section after A has made a request in the worst case.

If delivery is guaranteed to be in the order of transmission, no other node may enter its critical section more than twice between the time that A selects a sequence number and A is permitted to enter its critical section. No more than $2^*(N - 1)$ critical sections are possible before A may enter.

To get this bound observe that after node A has done its "Node Requests Critical Section" processing, it cannot receive more than one REQUEST from another node (j) which contains a lower or equal sequence number. By the time it gets the REPLY from this REQUEST, it must also have received A's REQUEST; it cannot thereafter select a lower or equal sequence number. Each other node j can enter its critical section at most once because of an already approved REQUEST and once with the one REQUEST which contains a lower or equal sequence number. If every other node follows this worst case pattern, at most $2^*(N - 1)$ critical section entries may preceed A's

When delivery in order is used, a new node may assume its Highest_Sequence_Number is synchronized when it has heard the fourth REQUEST message from the same node.

Assume that a node j sent its REQUEST messages before the new node came on-line. The new node is not synchronized until it holds a higher number in Highest_Sequence_Number than the sequence number used by j. The reference node B (which is generating the four requests) can enter its critical section at most twice before node j enters its critical section. Therefore, by the time B enters its critical section the third time, no nodes like j exist which did not know about the new node when they made their requests. Reference node B may have issued three REQUEST messages seen by the new node before entering its critical section for the third time. The fourth REQUEST message guarantees that the critical section was entered for the third time.

Acknowledgment. The authors wish to thank R. Stockton Gaines for his detailed and helpful comments on the presentation of this material.

Received 3/78; revised 3/80; accepted 9/80

References
1. Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent control with "readers" and "writers." *Comm. ACM 14,* 10 (Oct. 1971), 667–668.
2. deBruijn, N.G. Additional comments on a problem in concurrent programming and control. *Comm. ACM 10,* 3 (March 1967), 137–138.
3. Dijkstra, E.W. Hierarchical ordering of sequential processes. *Acta Informatica 1,* 2 (1971), 115–138.
4. Dijkstra, E.W. Solution of a problem in concurrent programming control *Comm. ACM 8,* 9 (Sept. 1965), 569.
5. Dijkstra, E.W. The structure of the THE multiprogramming system. *Comm. ACM 11,* 5 (May 1968), 341–346.
6. Eisenberg, M.A., and McGuire, M.R. Further comments on Dijkstra's concurrent programming control problem. *Comm. ACM 15,* 11 (Nov. 1972), 999.
7. Hill, J. Carver. Synchronizing processors with memory-contents-generated interrupts. *Comm. ACM 16,* 6 (June 1973), 350–351.
8. Knuth, D.E. Additional comments on a problem in concurrent programming control. *Comm. ACM 9,* 5 (May 1966), 321–322.
9. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM 17,* 8 (Aug. 1974), 453–455.
10. Lamport, L. Time, clocks and the ordering of events in a distributed system. *Comm. ACM 21,* 7 (July 1978), 558–565.
11. Lamport, L. Time, clocks and the ordering of events in a distributed system. Rep. CA-7603-2911, Mass. Comptr. Assoc., Wakefield, Mass. March 1976.
12. Ricart, G., and Agrawala, A.K. Performance of a distributed network mutual exclusion algorithm. Tech. Rept. TR-774, Dept. Comptr. Sci., Univ. of Maryland, College Park, Md., March 1979.
13. Ricart, G., and Agrawala, A.K. Using exact timing to implement mutual exclusion in a distributed network. Tech. Rept. TR-742, Dept. Comptr. Sci., Univ. of Maryland, College Park, Md. March 1979.