A Fault Tolerant Distributed Scheduling Algorithm

FILIZ UCAR
Virginia Tech

This paper presents a fault tolerant distributed mutual exclusion algorithm using only $c\sqrt{N}$ messages to have mutual exclusion in a distributed system, where $N$ is the number of nodes and $c$ is a number between 3 and 5. In this algorithm, each node asks for request to a critical region to only a set of nodes, not all the nodes. These nodes are called arbitrators. First of all, assuming that the channels are fault free, the request resolution algorithm will be explained. Then, how the list of arbitrators for each node is selected will be presented. Then, the whole algorithm with the assumption of having a fault fee channel will be elucidated. Finally, the fault-tolerance concept is introduced and the algorithm is extended.

1. INTRODUCTION

Having mutual exclusion in a computer network under distributed control is a significant issue. Since there is no shared memory, the only way for the nodes to maintain mutual exclusion is message passing. There have been so many algorithms so far. The algorithm proposed by Ricart and Agrawala [1] uses $2(N - 1)$ messages: $(N - 1)$ messages to ask for a request to all other nodes and $(N - 1)$ messages to get permissions from them. If all nodes agree upon giving permission then node asking for the request can enter the critical section. Each node always has an equal amount of responsibility to control mutual exclusion and is required to perform an equal amount of work to get mutual exclusion.

Another technique used was voting technique, introduced by Thomas [1], is based on getting permissions from majority of the other nodes, not necessarily from all. This reduces the number of permission messages required to obtain mutual exclusion is reduced to a half, $N/2$, in the best case. This technique was extended by Gifford [1] and Skeen [1] to allow nodes to give weighted votes, in other words, to give more than one vote to a node. So, a node only needs to get a majority number of votes not necessarily from majority number of nodes. Garcia-Molina and Barbara then introduced the relationship between weighted voting and sets of nodes with pairwise nonnull intersections [1], which means each pair of node should ask for permission to at least one common node.

The algorithm presented in this paper is a distributed mutual exclusion algorithm and requires only $3\sqrt{N}$ messages per mutual exclusion: $\sqrt{N}$ messages to ask for a request, $\sqrt{N}$ messages to receive permissions, and $\sqrt{N}$ messages to release mutual exclusion. This algorithm also uses the voting technique used in Thomas, deferral, the technique used in Ricart and Agrawala, and relinquishment to avoid deadlocks.
2. REQUEST RESOLUTION

In order to choose between two requests, any pair of two requests must be known to at least one of the arbitrators. If they do not have a common arbitrator, one gets permission from its half of arbitrators and the other gets permissions from its half of arbitrators, which result in deadlock or two nodes in critical section (CS) at the same time. Thus, to be able to obtain mutual exclusion then there must exist at least one common node between a pair of $S_i$ and $S_j$, where $S_i$ and $S_j$ are the set of arbitrators for nodes i and j respectively, so that the common node can serve as an arbitrator. Therefore, the $S_i$’s must satisfy the \textit{pairwise nonnull intersection} property, which is defined as follows:

(a) For any combination of $i$ and $j$, $1 \leq i, j \leq N$, $S_i \cap S_j$ is not equal to 0.

The request resolution rule then requires that when node $i$ wants to invoke mutual exclusion, it send a REQUEST message to every member of $S_i$ and get a permission from all of them. Since each member of $S_i$ serves as an arbitrator, the requesting node knows that it is the only node that has been granted mutual exclusion, when every member of $S_i$ returns a permission message. It means that no other node has been permitted due to nonnull intersection property.

The following properties are also required for truly distributed algorithms:

(b) $S_i$, $1 \leq i \leq N$, always contains $i$, which means that $i$ is also an arbitrator of itself. This helps to reduce the number of messages to be sent and received by a node

(c) The size of $S_i$, $|S_i|$, is the same for any $i$, implying that each node needs to send and receive the same number of messages to obtain mutual exclusion.

(d) Any $j$, $1 \leq j \leq N$, is contained in the $D S_i$’s, $1 \leq i \leq N$. This means that each node is arbitrator fro the same number of nodes resulting in equal responsibility in maintaining mutual exclusion.

3. CHOICE OF ARBITRATORS

Number of different sets of arbitrators, satisfying the properties (a), (b), (c), and (d), mentioned in the previous section, can be selected. From properties (b) and (d), each member of $S_i$ can be contained in $(D - 1)$ other subsets. “Therefore, the maximum number of subsets that satisfy property (a) is given by

$$(D - 1)K + 1,$$

where $K$ is size of each $S_i$.

Since $N$ is desired to be set to this maximum number so that $K$ is minimized for a given $N$, we have

$$N = (D - 1)K + 1. \ [1]$$

If we say $D=K$, then $N=K^2-K+1<=K^2$

So, $K >= \sqrt{N} \Rightarrow K = \sqrt{N}$ with a fractional error.
Examples:

(1) N=3 \rightarrow K=2
   S_1 = \{1,2\}
   S_2 = \{2,3\}
   S_3 = \{3,1\}

(2) N=7 \rightarrow K=3
   S_1 = \{1,2,3\}
   S_2 = \{2,4,6\}
   S_3 = \{3,5,6\}
   S_4 = \{4,5,1\}
   S_5 = \{5,7,2\}
   S_6 = \{6,7,1\}
   S_7 = \{7,3,4\}

4. ALGORITHM

The algorithm, which is used in all nodes, is now explained below:

(1) Node i sends a REQUEST message to each arbitrator of its when it wants to invoke mutual exclusion. Since i is also arbitrator of itself it pretends to have received a REQUEST. A sequence number greater than any REQUEST message sent, received, or observed at this node is assigned to theis REQUEST message.

(2) When a node receives a REQUEST from another node, it can do any of the following:
   (a) Marks itself locked for the REQUEST if it is not currently locked for another REQUEST, and then returns a LOCKED message to the requesting node i.
   (b) If the node is locked for a REQUEST from another node, the REQUEST from node i is placed in the WAITING QUEUE of the node. So, REQUEST by I becomes an outstanding REQUEST. It is then tested to determine whether the current locking REQUEST or any other outstanding REQUEST has a smaller sequence number than it has (or if sequence numbers are equal if it has a smaller node ID than it has), which means that another node has sent a REQUEST to invoke mutual exclusion before it did. If so, a FAILED message is returned to node i. Otherwise, an INQUIRE message is sent to the owner of the current locking REQUEST to inquire whether this originating node has succeeded in locking all its members.

When a node receives an INQUIRE message, it returns a RELINQUISH message if it knows that it will not succeed in locking all its members; that is, it has received a FAILED message from some of its members. When the arbitrator node, who has sent the INQUIRE message, receives the RELINQUISH message, it relieves itself of the current locking REQUEST placing the current locking REQUEST in the WAITING QUEUE and then locks itself for the most preceding REQUEST in the WAITING QUEUE. Thus, Relinquishing prevents deadlocks. The node cancels the LOCKED message previously received from the arbitrator, who has sent the INQUIRE message. If an INQUIRE message is received before it is known whether the node will succeed or fail to lock all its members, a reply is deferred until this
becomes known. When the node has succeeded in locking all its members and is in its critical section, it returns a RELEASE message, but only after it has completed its critical section. If an INQUIRE message has arrived after the node has sent a RELEASE message, it is simply ignored.

(3) If all arbitrators of node i have returned a LOCKED message, it enters its critical section.

(4) Upon completing the critical section, node i sends a RELEASE message to all of its arbitrators.

(5) When a node receives a RELEASE message, it relieves itself from the current locking REQUEST. It deletes this locking REQUEST and then relocks itself for the most preceding REQUEST in the WAITING QUEUE if the queue is not empty. A LOCKED message is returned to the node originating the new locking REQUEST. If the WAITING QUEUE is empty, the node marks itself unlocked.

(6) The steps (1)-(5) are repeated every time mutual exclusion is invoked.

5. AN EXAMPLE

\[
\begin{align*}
S_1 & = \{1, 2, 3, 4\}  \\
S_2 & = \{2, 5, 8, 11\}  \\
S_3 & = \{3, 6, 8, 13\}  \\
S_4 & = \{4, 6, 10, 11\}  \\
S_5 & = \{1, 5, 6, 7\}  \\
S_6 & = \{2, 6, 9, 12\}  \\
S_7 & = \{2, 7, 10, 13\}  \\
S_8 & = \{1, 8, 9, 10\}  \\
S_9 & = \{1, 2, 3, 4\}  \\
S_{10} & = \{3, 5, 10, 12\}  \\
S_{11} & = \{1, 11, 12, 13\}  \\
S_{12} & = \{4, 7, 8, 12\}  \\
S_{13} & = \{4, 5, 9, 13\}
\end{align*}
\]

Initially all sequence numbers are set to 0.

Figure 1a. (1) Node 11 is the first to attempt mutual exclusion. Its REQUESTs have arrived at nodes 12 and 13 and have locked them, but its REQUEST to node 1 is still on its way.

(2) Node 7 then invokes mutual exclusion. Its REQUESTs have arrived at nodes 2 and 10 and have locked them but its REQUEST to node 13 is still on its way.
Node 8 then invokes mutual exclusion. It locks itself and sends a REQUEST to nodes 1, 9, and 10 but fails to lock node 10 because node 10 has already been locked by a preceding REQUEST from node 7.

The REQUEST message originating at node 11 has finally arrived at node 1, while the REQUEST message from node 7 arrives at node 13. Node 1 then returns a FAILED, whereas node 13 sends an INQUIRE message to node 11 (sequence number of REQUEST by 11 and REQUEST by 7 are the same but node ID of 7< node ID of 11).

In this situation, nodes 7, 8 and 11 circularly lock each other. 8 and 11 knows that they can not get mutual exclusion since each has already received a FAILED message. But node 7 doesn’t know it yet. The INQUIRE message from node 13 reaches node 11. Node 11 , knowing that it won’t be able to get mutual exclusion, sends a RELINQUISH message to node 13. It deletes node 13’s LOCKED message. Node 13 puts node 11’s REQUEST into WAITING QUEUE and locks itself with node 7’s REQUEST. Then, it sends LOCKED message to node 7.

6. FAULT TOLERANCE

6.1. Extended Algorithm

Before beginning explaining about fault tolerant version of distributed mutual exclusion algorithm, let’s again give the definitions of the some key variables we have used so far:

- \( N \) = Number of nodes in the system.
- \( S_i \) = Set of all nodes which constitute the arbitrator set of node i.
- \( D_i \) = Set of all nodes that have node i in their arbitrator sets.
- \( K = \sqrt{N} \)

[2] In the fault tolerant algorithm we will use some new key words:
• MAKER(i) = Each node i in the system has a designated node which simulates it if it fails. This node is called the MAKER node of i and denoted by MAKER(i).
• \( \lambda \) = Maximum roundtrip communication delay of the network.
• \( T_{\text{MAX}} = \lambda + \sqrt{N} \) - 1 times the maximum time a node can spend in a critical section.
• \( T_{\text{INQ}} = \lambda \) + Maximum time required by a node to respond to an INQUIRE message.
• \( T_{\text{MAKE}} = \lambda \) + Minimum time required by a node to respond to a MAKEUP message.

Each node in the system runs three processes:
  a) MAIN_PROCESS that does the normal user job at this node and makes requests for the critical section whenever necessary,
  b) MESSAGE_INTERPRETER which acts as a message interpreter to handle different messages other than the INQUIRE message received by the node.
  c) INQUIRE_WAIT that handle the INQUIRE message.

The MAIN-PROCESS runs at the node itself. MESSAGE_INTERPRETER and INQUIRE_WAIT are executed in an interrupt-driven fashion. MESSAGE_INTERPRETER is called whenever a message, other than INQUIRE message, is received. INQUIRE_WAIT is called when an INQUIRE message is received and will execute till there are no more INQUIRE messages for which a reply has not yet been received.

It has been mentioned that each node had a MAKER node simulating it when it is down. There are rules in selecting a MAKER node for each node in the system (called Maker Selection Criteria), which is explained below:
  a) MAKER of each node is distinct.
  b) The sequence of nodes i, MAKER(i), MAKER(MAKER(i)), ... will form a loop containing all the nodes in the network.

If a node, after sending a REQUEST or INQUIRE message does not receive a reply within TMAX time units it sends a probe message to see if that node is down. TMAX is a system-dependent parameter varying according to the usage and need in the given distributed system. The probe message has the highest priority to get a response. If a response for a probe message is not received within the minimum time required for a round trip in the network, then the corresponding node is assumed to have failed. Whenever the failure of node i is detected, the node MAKER (i) is informed and requested to simulate the failed node i so as to maintain the pairwise non-null property required for the arbitrator sets of the algorithm. The MAKER(i) starts simulating the failed node i and informs all other nodes notifying them that it is simulating a particular node. Then all the nodes, which have i in their arbitrator set, send their status with respect to i to MAKER(i) so that MAKER(i) can have the same status (LOCKED or UNLOCKED) and the same WAITING_QUEUE as i before i has failed. These status messages carry the time-stamp of the original request so that they can be put in the
WAITING QUEUE properly, and the locking request could be selected properly. Then MAKER node starts simulating the failed node. Other nodes having the failed node in their arbitrator node sends REQUEST and RELINQUISH messages to it. Thus, system starts working normally again.

The informal description of the algorithm for each specific event is given below:

(1) **Request to enter critical section:** Node i requests entry to critical section by sending a time stamped REQUEST message to every member of $S_i$. Node i pretends to have received the request just like in Maekawa’s algorithm but it also waits for TMAX time units for a response.

(2) **Processing a REQUEST message:** When receiving a REQUEST, receiving node enters into a locked state for the REQUEST (and sends out a LOCKED message) if it is not currently locked for any other REQUEST; else the current REQUEST is placed in the WAITING QUEUE. It then tests if the current locking REQUEST or any outstanding REQUEST at this node precedes the current REQUEST (i.e., the REQUEST just now received); if so, a FAILED message is returned. Otherwise an INQUIRE message is sent to the locking request to see if it has succeeded in locking all other nodes and the node waits to receive a RELEASE or a RELINQUISH message. The fault tolerant algorithm is extended at this point because this time the node waits for only TINQ time units to get a RELEASE or a RELINQUISH message.

(3) **Detecting a node failure:** A node waits for TMAX time units after sending a REQUEST message. It waits for TINQ time units after sending a INQUIRE message. In either case when the waiting period expires and the node doesn’t receive any message from any of the nodes in its arbitrator set, it sends ARE_YOU_THERE(j) message to that node j. If j is up it sends I_AM_FINE(j). If node I doesn’t get I_AM_FINE message within the minimum time that a message spends for a roundtrip in the network, it assumes that the corresponding node has failed and sends MAKEUP(j) to MAKER of the failed node and waits for TMAKE time units. If it does not receive any message by this time it starts node failure detection process for MAKER(j) this time to find out if it is also down.

(4) **Processing a MAKEUP message:** After receiving a MAKEUP message, a MAKER node sends MAKING(j) to all the nodes to inform them that j is down it is compensating for it. If a MAKER node receives more than one MAKEUP messages it will simply ignore the rest. After receiving MAKING(j) all the rest of the nodes having j in their arbitrator set state their status with respect to j as a STATUS message to MAKER, delete j from all the remaining $S_i$‘s and merge the D set of j with its own.
(5) Processing a INQUIRE message, a RELINQUISH message, or a RELEASE message and entering the critical section are done in the same way as in the algorithm without fault tolerance.

6.2 Number of Messages Exchanged

The number of messages exchanged can be slightly more in this algorithm depending on whether any node fails and on whether contention occurs.

**No contention and no node failure:** If there is no contention and no node fails, the critical section access requires (K-1) REQUEST, (K-1) LOCKED and (K-1) RELEASE messages as in the algorithm with no fault tolerance.

**No contention (single node failure):** If a single node fails, since we have a unique MAKER and only those nodes having failed node in their arbitrator set (that is, the nodes in D set of the failed node) are affected, we have (K-1) ARE-YOU-THERE, (K-1) MAKEUP, (N-2) MAKING and (K-1) STATUS messages in addition to the regular messages. It is very good enhancement in decreasing number of messages when compared to other algorithms literature where any node can simulate any other node.

**Contention occurs:** If contention occurs and no node fails, the critical section access requires in the worst case (K-1) REQUEST, (K-1) INQUIRE, (K-1) RELINQUISH (K-1) LOCKED and (K-1) RELEASE messages. If a single node fails we have additionally at the most (K-1) ARE-YOU-THERE, (K-1) MAKEUP, (N-2) MAKING and (K-1) STATUS messages.

7. **CORRECTNESS PROOFS**

a) Mutual exclusion

**Proof:**

Assume that more than one node are in the critical section at the same time.

(1) All the nodes in the critical section should have received a LOCKED message from all nodes in their arbitrator set.

(2) Since these nodes have one common arbitrator to get a LOCKED message, then the mutual arbitrator of the nodes in the critical section should send LOCKED message to more than one node.

(3) An arbitrator doesn’t send a LOCKED message before it gets a RELEASE message from the current locking process in the critical section or before getting a RELINQUISH message if it is NOT in the critical section.

(4) Since a node in the critical section never sends a RELEASE message to its arbitrators before it gets out of the critical section or never enters the critical section after it sends a RELINQUISH message to its arbitrator, it is impossible to have more than one node in critical section at the same time.
With the addition of the fault tolerance, the mutual exclusion property is not destroyed. The MAKER node, which simulates it, eventually replaces the failed node and this node becomes part of the arbitrator set of the nodes in D set of the failed node. All the nodes which formed the D set of the failed node are now merged with D set of the MAKER and require its permission to enter the critical section. So all the nodes, which should have received a LOCKED message from the failed node to enter the critical section now, need to receive a LOCKED message from the MAKER of the failed node. Hence mutual exclusion is preserved.

b) No deadlock

Proof:

Deadlock is possible when we have a circular wait among the nodes requesting mutual exclusion. But this is not possible in our algorithm because all the requests are ordered by their request sequence numbers and the requests are resolved according to this sequence numbers. If a node with a lesser priority locks a particular node for critical section access and another node with a higher priority arrives at that node then the node with a lower priority has to come out of the critical section (RELEASE) or is forced out of it (RELINQUISH). Since in our algorithm the nodes sending their status with respect to a failed node to its MAKER with the original request sequence numbers the ordering of the requests is preserved is same as the ordering before the node failed. So there is no possibility of introducing a deadlock.

Example:

![Figure 2. A circular waiting. 1 waits for 2; 2 waits for 4 and 6; 4 waits for 1; 6 waits for 1.](image-url)

Let us assume that all REQUESTS in Figure 2 have the same sequence number. Then the REQUEST from node 1 is the most preceding (its node ID is the smallest). The circular waiting is broken because the REQUESTS from node 4 and 6 are preceded by the REQUESTS from node 1 and thus their members are relinquished. This will allow node 2 to succeed in locking all its members and allow it to enter its critical section.
c) No starvation

Proof :

The starvation of node i occurs when other preceding REQUESTS are continuously locking or waiting at a member of Si. All the requests are granted according to the request sequence numbers and every node updates sequence number after it receives any request with a higher sequence number. So after a request is received by all the nodes there is no possibility of a request coming with a lower sequence number than this request. So every request will become the request with the highest priority after a finite amount of time and hence should be granted. So there is no way for a node to wait forever to receive a LOCKED message from any of its arbitrators. The only way it may happen is that their arbitrator fails while they were waiting but since our fault tolerant algorithm introduces the solution of simulating it with another node, this is not a problem anymore. Thus, starvation is not possible.

d) Every critical section request is serviced within a finite amount of time if the Maker Selection Criteria is satisfied.

Proof : We choose Makers such that each node i has an unique MAKER and the loop of MAKER’s is of length n.

Example:

Considering a system with 7 nodes we can have the following MAKER node chain:

![Diagram of Maker nodes]

Figure 3. Chain of the MAKER nodes.

In the Figure 3, 4 is a MAKER of 1 and so on. We can see that the MAKER nodes form a single loop of size 7. Since there is a single loop of MAKER’s, no matter how the nodes fail there will still be a single loop of MAKER’s. After 6 nodes fail in the final picture there is only one working node being MAKER of itself and there is no possibility of any other node trying to detect its failure and keep waiting.

8. CONCLUSION

A distributed algorithm that creates mutual exclusion using $c\sqrt{N}$ messages, where c is a constant between 3 and 5, and its variation with fault tolerance has been presented. The algorithm is symmetric and allows fully parallel operation. It also allows a node removal. The algorithm is optimal in terms of the number of messages used to create mutual exclusion among fully distributed algorithms, where the term distributed is used here to
mean that each node serves as an arbitrator for the same number of nodes. Besides, with the introduction of MAKER nodes and timeout values for reply messages, the algorithm can tolerate failure of any number of nodes and can generate all the information lost due to the failure of a node by exchanging some extra messages. The algorithm also prevents deadlock and starvation, and lets a node get a reply to its messages within a finite amount of time as proved in Section 7.

REFERENCES
