A token based k-resilient mutual exclusion algorithm for distributed systems

Dhananjay M. Dhamdhere *   Sandeep S. Kulkarni

Department of Computer Science and Engineering

Indian Institute of Technology, Bombay 400 076, (India).

Abstract

This paper presents a token based mutual exclusion algorithm which is k-resilient in that it can tolerate upto k site/link failures. The algorithm uses many features of the token based algorithm by Raymond [5] and inherits many of its advantages, viz. simplicity of the token message and the local data structure at each site. Correctness proofs of the algorithm and complexity analysis are included.

Keywords: Distributed systems, mutual exclusion, fault tolerance, k-resiliency

1 Introduction

In distributed systems, the mutual exclusion problem arises in various contexts like resource allocation, atomic commitment, access to replicated data, etc.

Suzuki and Kasami [7] presented a token based algorithm for mutual exclusion in distributed systems. A node can enter the critical section (CS) only if it has the *privilege* token. The algorithm requires every node making a request for mutual exclusion to send a message to every other node in the system to determine whether it can obtain the token. The token carries a queue of CS requests, which are serviced strictly in a FIFO order. Singhal [6] proposed a heuristic based on the state information of other sites in the system to reduce the message complexity of the algorithm. This algorithm also addressed the issue of fault-tolerance. Nishio et al [4] report a resilient algorithm based on the token algorithm of Suzuki and Kasami. Another fault tolerant algorithm is reported in [1].

Raymond proposed a tree based algorithm for mutual exclusion which reduced the message complexity from *O(n)* to *O(log n)*, where n is the number of nodes in the system [5]. Each node i in the system maintains a variable called $holder_i$ which indicates the neighbour in whose direction the token is located. Each node also maintains a FIFO queue of requests it receives. When a node i wishes to enter the critical section, it sends a request (if it has not already sent one on behalf of some other

*dmd@cse.iitb.ernet.in

node) to the node $j = holder_i$. Node j similarly passes the request to $holder_j$ unless it itself holds the token. Whenever a node finishes its critical section, it sends the token to the first node in its local queue (if it is not empty). It now updates its *holder* and reverses the direction of the link to the *holder*. A node receiving the token checks whether it is for itself or for some other node from its queue, and propagates the token if necessary.

Raymond's algorithm is communication efficient, does not require the token message to carry the pending requests and does not require a timestamp or a sequence number. However, being tree based, even a single site or link failure partitions the network leading to starvation in the partition(s) not containing the token.

In [3], Raymond's algorithm for mutual exclusion has been extended to incorporate fault tolerance by requiring each node to have a set of edges (called *Out*) on which it can send the request. One of the *Out* edges — the edge to the *near* of a node — is favoured for sending request messages. When a node receives the token, it sends a reverse message along all edges in its *Out*, and then sets *Out* $= \phi$. If the edge to *near* fails, the node sends its requests along some other edge in *Out*.

In order to avoid the deadlock condition, wherein no site is executing its critical section (CS) but no requesting site is in a position to enter CS, the algorithm strives to ensure that directed cycles do not arise in the system. However, the algorithm still suffers from the deadlock problem as illustrated in example 1.

Example 1

Consider the system shown in Fig.1(a). Let node 1 possess the token, and let nodes 2 and 4 make requests for mutual exclusion such that the request of node 2 is the first one to reach node 1 ($near_2 = 1$). The queue at node 1 now contain nodes 2 and 3. When node 1 finishes the critical section, it will send the token to node 2 (with a request to send it back since node 1's request queue contains another entry $-$ that for node 3).

Let the following sequence of events occur in the system :

(a) Node 2 receives the token, sets $Out_2 = \phi$ and sends a Reverse message to node 3. Let the delivery of the Reverse message be delayed.

(b) Node 2 finishes using the CS and sends the token to node 1. Node 1 sends it to node 3 and adds the edge (1,3) to Out_1. Node 3 deletes the edge (3,1) from Out_3. It now forwards the token to node 4 and adds edge (3,4) to Out_3. Node 4 deletes edge (4,3) from Out_4.

(c) The Reverse message from node 2 to node 3 (refer step (a) above) arrives at node 3.

(d) Edge (3,4) fails now, and node 5 sends its request to node 3. Node 4 finishes using the CS.

According to the algorithm in [3], node 3 adds edge (3,2) to Out_3 in step (c). The system is now as shown in Fig.1(b). Nodes 3, 2 and 1 will forward the request made by node 5 in step (d). However,

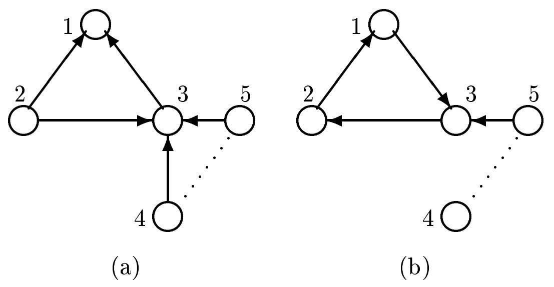

Figure 1: An example of problems due to communication delays (*note :* $\rightarrow$ indicates an *out* edge, while $\cdots$ indicates a communication link)

due to the cycle $3-2-1-3$, the request does not reach node 4. Deadlock now exists in the system.

2 The Algorithm

To counter the problem created by communication delays, a monotonically increasing sequence number is associated with the privilege token as it circulates around the system. When site i receives the token, it notes the sequence number in the local variable $myseq_i$. While passing the token to another site, it does so with a new sequence number. To ensure that no cycles develop, the system maintains the following invariant :

Invariant

$(y, x) \in Out_y \Rightarrow myseq_x > myseq_y$.

The system is initialized in such a manner as to satisfy the invariant. This can be performed by assigning depth first numbers [2] as sequence numbers to the nodes of the graph.

```
User Process :
    Repeat for ever
        Request_CS;
            { Critical Section }
        Release_CS;
            { Rest of the cycle }
end;

Control Process :
    Repeat for ever
        Await a message;
        Call appropriate procedure to
          handle the message;
end;
```

Figure 2: User and Control Processes

2.1 User and Control Processes

Fig.2 shows the generic form of the processes existing at each site in the system. The *user process* implements the application program, while the *control process* performs the control computations required by the mutual exclusion algorithm. When a node fails, it is assumed that both processes cease to function. During node recovery, the processes are created anew.

2.2 Assumptions about the Network

All communication is assumed to be FIFO. When a link (i,j) fails, nodes i and j both become aware of the failure immediately. When a site fails all its neighbours become aware of the failure. It is assumed that the time required for a link/site to be restored is large compared to the time consumed by the fault tolerance actions. A restored site/link is added to the network only after the recovery action has been completed.

The site possessing the privilege token never fails. Also, the token is never lost during transit.

2.3 Data Structures

The privilege token is represented by a single field *sequence*. Every site maintains the following data structures

- Out : Set of out edges $\{out\}$.
- me : ID of the site.
- $near$: $near = me$ if the site possesses the token, else $(me, near) \in Out$ in the absence of failures.
- $requesting$: boolean variable.
- Q : queue of requests.
- $update$: array of
  $site\_in\_trouble$: i such that $Out_i = \phi$
  $firstrequester$: site from which the
  Backmessage was first received
  $timestamp$: timestamp of the request
- $myseq$: Sequence number when the site last received the token.

2.4 Messages

The messages used in the algorithm are

- Request($sender$) : Request for mutual exclusion.
- Token($sender, Q\_size_{sender}, seq$) : Sending the token.
- Add_edge($sender, seq_{sender}$) : Receiving node should add (me, $sender$) to its Out.
- Backmessage($sender, x, timestamp_x$) : Information concerning site x, which faces the situation $Out_x = \phi$. *Sender* is propagating this message.
- Update_graph($sender, Q\_size_{sender}, x,$ $timestamp_x, seq$) : Passing the token to site x, in response to its Backmessage.
- Backmessage_cancel($x, timestamp_x$) : For cleaning the *update* entries (x, –, $timestamp_x$).

2.5 The Basic Algorithm

Fig.3 shows the procedures called by the user process. In the basic algorithm, it is assumed that the conditions 'if (me,$near$) $\in$ Out' and 'if (sizeof(*update*) = 0)' of procedures Request_CS and Release_CS respectively are always satisfied.

A process calling procedure Request_CS can enter the critical section straightaway if the node already possesses the token. This is indicated by the condition $near = me$. If $near \neq me$, Request_CS enters the request in the Q at that node, sends a Request message to *near* (if necessary) and awaits the condition $near = me$ before returning to the user process. After finishing the CS, a user process calls procedure Release_CS which passes the token to the first process in the Q.

Fig.4 shows the procedures called by the control process. In the basic algorithm it is assumed that the condition if (me, $near$) $\in$ Out' in procedure Request_Handler is always satisfied. If node i possesses the token when a request is received from some node x, procedure Request_Handler passes

Procedure Request_CS
    requesting = *true*;
    If $near \neq me$ then
        Enqueue (Q, me);
        if (me, $near$) $\in Out$ then
            if (sizeof(Q) = 1) then
                Send Request(me) to $near$;
        else if ($Out \neq \phi$) then
            $near = x$ for some (me, x) $\in Out$;
            Send Request(me) to $near$;
        else
            Send Backmessage(me, me,
              $timestamp_{me}$) on all live links;
        await ($near = me$);
end;

Procedure Release_CS
    requesting = *false*;
    if (sizeof(*update*) $\neq 0$) then
        Copy ($site\_in\_trouble, firstrequester$,
          $timestamp$) fields of update[1] into
          x, y, t respectively;
        $size$ = sizeof(*update*) + sizeof(Q);
        Send Update_graph(me, $size$, x,
          $timestamp_x$, $myseq + 1$) to y;
        $near = y$;
        $Out = Out \cup \{(me, near)\}$;
    else if (sizeof(Q) $\neq 0$) then
        $near = Q[1]$; Delete $Q[1]$ from Q;
        Send Token(me, sizeof(Q), $myseq + 1$) to
          $near$;
end;

Figure 3: Procedures called by the User Process

the token to x with a new sequence number and sets $near_i$ to x. Else the request is entered in i's Q, and the Request message is forwarded to $near_i$, if necessary.

The Token_Handler procedure takes the following actions to preserve the system invariant : It sets $near = Q[1]$, $Out_i = \phi$ and sends an Add_edge message on each erstwhile *out* edge — except the link to its *near*. It also sends an Add_edge message to node x, from whom it received the token.

Procedure Request_Handler
/* Handles a Request(x) message. */
    Enqueue (Q, x);
    if (($near = me$) and ($\neg Requesting$)) then
        $near = Q[1]$; Delete $Q[1]$ from Q;
        Send a Token (me, sizeof(Q), $myseq + 1$)
          to $near$;
        $Out = Out \cup \{(me, near)\}$;
    else if ($near \neq me$) then
        if (me, $near$) $\in Out$ then
            if (sizeof(Q) = 1) then
                Send Request(me) to $near$;
        else if ($Out \neq \phi$) then
            $near = x$ for some (me, x) $\in Out$;
            Send Request(me) to $near$;
        else
            Send Backmessage(me, me,
              $timestamp_{me}$) on all live links;
end;

Procedure Token_Handler
/* Handles a Token(x, $size$, seq) message. */
    if (size > 0) then Enqueue(Q, x);
    $myseq$ = seq;
    $near = Q[1]$; Delete $Q[1]$ from Q;
    for each (me, y) $\in Out$ do
        $Out = Out - \{(me, y)\}$;
        if $y \neq near$ then
            Send Add_edge(me, $myseq$) to y;
    Send Add_edge(me, $myseq$) to x;
    if ($near \neq me$) then
        Send Token(me, sizeof(Q), $myseq + 1$) to
          $near$;
    /* process enters CS when $near = me$. */
end;

Procedure Add_edge_Handler
/* Handles an Add_edge(x, seq) message. */
    if ($myseq$ > seq) then
        Send Add_edge(me, $myseq$) to x;
        $Out = Out - \{(me, x)\}$;
    else
        if ($Out = \phi$) then $near = x$;
        $Out = Out \cup \{(me, x)\}$;
end;

Figure 4: Procedures called by the Control Process

It now checks if the token is for some other node (indicated by $near_i \neq me$), in which case it sends the token to $near_i$.

On receiving an Add_edge message from node x, node i checks whether this is a delayed Add_edge message, *i.e.* whether it has already received the token by some other path. (This condition is indicated by $myseq_i > seq$ field of the Add_edge message.) For a delayed Add_edge message, it responds by sending an Add_edge$(i, x, myseq_i)$ message to node x. Else it adds the edge (i, x) to Out_i. This maintains the system invariant.

2.6 Resilient algorithm

2.6.1 Handling link failures

The basic fault tolerance capability arises from the k resiliency of the communication network, whereby every node has at least one neighbour despite k failures in the system. The algorithm utilizes this feature to have a set of *out* edges for each node, such that a node can send out Request messages along an alternate link when the link to its *near* fails. However, it is still possible that node i has no active *out* edges when it wishes to make a request. In this case, node i broadcasts a Backmessage to all its neighbours. Every neighbour remembers the identity of the node from which it received this message for the first time in the field $firstrequester$ of the *update* array entry, and broadcasts the message to its neighbours. On receiving this message, the node holding the token gives a favoured treatment to the site which has transmitted the Backmessage. Hence, instead of sending the token to the first site in its Q, it sends the token to the site in $update[1].firstrequester$ using the Update_graph message. The site receiving this message sends an Update_graph message to its $firstrequester$ until the token reaches node i. The Update_graph_handler performs actions concerning the reversal of edges, etc. analogous to the Token_Handler procedure of the basic algorithm.

Procedures Request_CS and Release_CS of Fig.3 perform the resiliency actions under the conditions 'if $(me, near) \notin Out$' and 'if (sizeof($update$) $\neq 0$)'. If the process waiting to enter the CS is informed of the failure of the link on which its request was transmitted, it simply repeats the call on the procedure Request_CS.

Fig.5 shows the procedures called by the control process in the resilient version of the algorithm. Note that the procedures Request_Handler, Add_edge_Handler and Token_Handler remain the same as in the basic version of the algorithm (Fig.4).

Procedure Request_Handler performs resiliency actions on failure of the link to *near*. These are analogous to the actions in procedure Request_CS, viz. pick some other edge from *Out* (if one exists) to send the request, else broadcast a Backmessage. Functioning of the Backmessage_Handler and Update_Handler is as outlined at the beginning of this section. Procedure Backmessagecancel_Handler (not shown in Fig.5) cleans the information concerning a Backmessage from *update* and broadcasts the message to its neighbours.

Other resiliency actions required are as follows : If node i wishes to pass the token to node j – which is at the head of Q_i – and finds that link (i, j) has

Procedure Backmessage_Handler

```
/* Handles Backmessage(y, x, timestamp_x). */
   return if this is a duplicate message;
   if (near = me) then
      if (Requesting = False) then
         size = sizeof(Q) + sizeof(update);
         Send updategraph(me, size, x,
           timestamp_x, myseq + 1) to y;
         near = y;
         Out = {(me, y)};
      else Copy this entry in the array update;
   else
      Send Backmessage(me, x, timestamp_x)
        to all neighbours;
      Copy this entry in the array update;
end;
```

Procedure Update_graph_Handler

```
/* Handles Updategraph(y, size, x,
    timestamp_x, seq) */
   if (size > 0) then add y at the head of Q;
   myseq = seq;
   if (x ≠ me) then
      Delete the entry (x, firstrequester,
       timestamp_x) from update after
        setting near = firstrequester;
      ∀ g such that (me, g) ∈ Out do
         Out = Out − {(me, g)};
         Send Add_edge(me, myseq) to g;
      size = sizeof(Q) + sizeof(update);
      Send Update_graph(me, size, x,
        timestamp_x, myseq + 1)
        to firstrequester;
      Send Add_edge(me, myseq) to y;
   else
      Send Backmessagecancel(x,
        timestamp_x) to all neighbours;
      near = Q[1]; Delete Q[1] from Q;
      ∀ g such that (me, g) ∈ Out do
         Out = Out − {(me, g)};
         if ( g ≠ near ) then
            Send Add_edge(me, myseq) to g;
      if (near ≠ me) then
         Send Token(me, sizeof(Q), myseq + 1)
           to near;
end;
```

Figure 5: Procedures called by the Control Process in the Resilient version of the algorithm

failed, it simply removes j from Q_i and proceeds to pass the token to the next process in Q_i(if any). If a link fails while transmitting the update_graph message, the node sends Backmessagecancel messages on all links and transmits the token to the next element in *update* or Q (if one exists). If a link fails while sending a Backmessagecancel message, the message is retransmitted after the link is restored. A site sending a Request message or a Backmessage retransmits the message if a timeout occurs.

2.6.2 Node Failure

Failure of a node i is modelled as the failure of all links connected to it.

2.6.3 Recovery

When a node is restored, it obtains mutual exclusion and sends an Add_edge message on all links connected to it. When a link connecting the nodes i and j comes up, the node with the higher sequence number obtains mutual exclusion and sends an Add_edge message on this link.

2.7 Example

Consider Fig.1 and step (c) of the sequence of events mentioned in section 2. When the Add_edge message from node 2 arrives at node 3, node 3 simply rejects the message since $myseq_3 > seq$ mentioned in the message, which is the value of $myseq_2$ when node 2 sent the message. Hence edge (3,2) is never added to Out_3. When the request from node 5 arrives at node 3, it finds $Out_3 = \phi$, and broadcasts a Backmessage. The Backmessage reaches node 4 along the path 3-5-4. The token is sent to node 3 along this path, which will send it to the first node in its Q, i.e. to node 5.

3 Correctness Proofs

The system invariant holds when the system is initialized. The graph constructed by *out* edges changes only when some node i receives a Token, Update_graph or Add_edge message from node j. In the case of Token or Update_graph messages, i deletes all edges in out_i before instructing node j to add the edge $(j,\ i)$ to out_j. In the case of an Add_edge message i adds (i,j) to Out_i only if $myseq_i <$ sequence-number in the Add_edge message, which is $\leq myseq_j$ since sequence numbers of all nodes are monotonically increasing. These actions maintain the system invariant.

3.1 Freedom From Deadlocks

Definition 6.1 : A *request path* is a path $x = \alpha_0 - \alpha_1 \ \ldots \ \alpha_n = y$ (denoted as $x \stackrel{r}{\to} y$) such that $\forall i = 0 \ldots n-1 \ near_{\alpha_i} = \alpha_{i+1}$ and α_i is in $Q_{\alpha_{i+1}}$.

Theorem 1 :

Despite k failures, a request path $x \stackrel{r}{\to} y$ will eventually be established for every requesting node x, such that node y holds the privilege token.

Proof :

Let node x make its request after some n failures have occurred in the system. Let a request path $x \stackrel{r}{\to} l$ be established. If a link (α_i, α_{i+1}) along the request path fails, node α_i will attempt to establish a request path $\alpha_i \stackrel{r}{\to} y$, where y is the current holder of the privilege token, by sending its request to some node $g \in Out_{\alpha_i}$ or by broadcasting a Backmessage. This situation is analogous to node α_i attempting to build $\alpha_i \stackrel{r}{\to} y$ for its own request after $n+1$ failures have occurred in the system. Hence to prove the theorem, it is adequate to prove that a request path will eventually be established for a node making its request after k failures have occurred in the system.

If a directed path $x \to y$ exists in the system after k failures, eventually $x \stackrel{r}{\to} y$. If not, let a directed path $x \to g$ exist in the system wherein either $x = g$ or the first edge in the path is (x, h) for some $h \in Out_x$ (possibly $h = near_x$). From k-resiliency of the communication graph, undirected path(s) must exist between g and y. One such path will be used in the transmission of Backmessage and Update_graph messages such that node g becomes the holder of the privilege token. Hence, $x \stackrel{r}{\to} g$ and g holds the privilege token.

3.2 Freedom from starvation

Theorem 2 :

Despite k failures, every node wishing to enter the critical section will do so in finite time.

Proof :

By theorem 1, for every requesting node x, a request path $x = \alpha_0, \alpha_1, \alpha_2, \cdots, \alpha_n = y$ is eventually established, where y holds the privilege token. The requests are entered in a Q in FIFO order, except in the event of an Update_graph message. However, since the number of failures in the system is bounded, the number of rquests processed out of order is also bounded. Hence, in bounded time the request of α_i will be first in the queue of α_{i+1} $\forall i = 0 \dots n-1$. The token will be sent to x on this path in the reverse direction.

4 Complexity

In the absence of failures, the number of Request messages is bounded by d, where d is the maximum length of any path in the system. The number of Token messages is similarly bounded by d. Reverse messages are sent by every node on the path along which the Token message is propagated. Assuming each node to have $O(k)$ neighbours, the total number of Reverse messages generated is $O(d \cdot k)$. Thus the total number of messages generated per CS entry is $O(d \cdot k)$ in the absence of failures.

A Backmessage used in the case of a link failure is propagated at most once along each edge (similarly for Backmessagecancel). Hence, the number of messages generated per failure is $O(e)$ where e is the number of edges in the communication graph. Thus, the worst case message traffic is $O(d \cdot k^2 + e \cdot k)$, which is $O(e)$ when k is bounded by a constant.

5 Concluding remarks

This algorithm only addresses the k-resiliency aspects. Other aspects of fault tolerance, viz. regeneration of a lost token, can be superimposed on it.

References

[1] Agrawal, D. and Abbadi, E. L. : An efficient and fault-tolerant solution for distributed mutual exclusion, *ACM Transactions on Computer Systems*, **9**, 1 (1991), 1-20.

[2] Aho, A. V., J. E. Hopcroft and J. D. Ullman, *Design and analysis of algorithms*, Addison Wesley, 1976.

[3] Chang, Y. I., M. Singhal and M. T. Liu : A Fault Tolerant Algorithm for Distributed Mutual Exclusion, *Proceedings 9th symposium on Reliable Distributed Systems*, IEEE Computer Society, 1990, 146-154.

[4] Nishio, S., K. F. Li and E. G. Manning : A resilient mutual exclusion algorithm for computer networks, *IEEE Transaction on Parallel and Distributed Systems*, **1**, 3, (1990), 344-355.

[5] Raymond, K. : A Tree Based Algorithm for Mutual Exclusion, *ACM Transactions on Computer Systems*, **7**, 1 (1989), 61-77.

[6] Singhal, M. : A heuristically-aided algorithm for mutual exclusion in distributed systems, *IEEE Transactions on Computers*, **38**, 5 (1989), 651-662.

[7] Suzuki, I. and T. Kasami : A distributed mutual exclusion algorithm, *ACM Transactions on Computer Systems*, **3**, 4 (1985), 344-349.