Chapter 4

Inserting Priority Serialization in Token Based Algorithms

4.1 Introduction

The performance of distributed algorithms for mutual exclusion has been generally measured by the number of messages exchanged per CS entry, the delay between two successive executions of the CS and the attitude of the algorithm to be resilient to failures. However, no attention has been paid to the serialization of requests. In particular, such algorithms use an FCFS discipline to serialize the accesses to the CS. This discipline is based on either request timestamp or the request's physical reception time at the node where a token is located. Real time systems, or systems that have critical tasks that must execute quickly for good performance, need prioritized serialization. Moreover, the insertion of the priority discipline, such as Shortest Job First (SJF), Head of the Line (HOL), Shortest Remaining Job First (SRJF) etc. could be useful in several applications to optimize some performance indices like average response time. In priority serialization, every request for the token has a priority attached. When the token holder releases the token, it should be given to the processor with the highest priority request. However, priority based disciplines may exhibit starvation phenomenon. That is, the higher priority
requests continue to supercede the lower priority requests indefinitely. Therefore, a solution to the starvation problem has to be found.

Baldoni [BC95] inserted priority serialization discipline in Maekawa type algorithms and showed how it can be used to improve the average response time compared to FCFS discipline. Moreover their scheme had an additional advantage of being inherently deadlock free and had lower message complexity. To avoid starvation they used Gated Batch priority serialization discipline. In such a discipline requests are collected in batches, and batches are served serially. The Gated Batches of the Gated Batch priority serialization discipline are formed as follows: The first request that is generated by a process, say i, forms the 0th batch. All requests that arrive when i is in its CS form the 1st batch. When i exits from its CS, the requests of the 1st batch are served according to a non preemptive priority serialization discipline. Then, all the requests that arrive during the service time of the mth batch form the (m+1)th batch. Therefore, requests memorized in the (m+1)th batch can never supercede requests memorized during previous batches even though they may have a higher priority value. When the size of the successive batch is 0 (i.e. no request is found), the first request forms the 0th batch. This way starvation freedom is achieved. Johnson and Newman have presented three priority based mutual exclusion algorithms [JNW96]. Two of the algorithms are based on Li and Hudak’s path compression techniques [LH89] and the third algorithm uses Raymond’s fixed tree structure [Ray89b], but they have not taken into account the starvation issue.

This chapter gives a general scheme for inserting priority based serialization discipline in token and tree based algorithms avoiding starvation. As a particular case, we also show how priority based serialization is inserted in Raymond’s [Ray89b] and Naimi et al’s
[NTA96] mutual exclusion algorithms. This algorithm can be used to produce particular priority based mutual exclusion algorithms better fitted to a particular situation. Further, a new broadcast based, token based algorithm is presented which uses priority serialization.

4.2 A Prioritized General Scheme for Token and Tree based Algorithms

Helary et al [HMR94] presented a general scheme for the class of token based algorithms using a rooted tree to move the requests. The information structure includes a dynamic rooted tree structure logically connecting the nodes involved in the system and a behavior attribute (Transit or Proxy), dynamically assigned to each node. The behavior of the nodes can change dynamically. The generality of the algorithm lies in the ability of the algorithm to take into account these behavior modifications. By modifying the behavior of the nodes, particular algorithms can be deduced from this general algorithm. For example, if we consider all nodes to have transit behavior, then we get a variant of Naimi-Trehel [NTA96] algorithm. If the behavior of each node is transit when it has the token & proxy otherwise, then Raymond’s algorithm[Ray89b] is obtained. Moreover the behavior of each node can be defined dynamically in order to fit the topology of the underlying network. For example, on receiving a request message from node j, node i can be defined as transit if there exists a physical link between node j and fatheri, otherwise it behaves as proxy. Such a rule to define the behavior of a node allows the token to use shorter paths towards the requesting node.
We propose to insert priority based serialization discipline in Helary's algorithm because this algorithm is a generic model for a class of mutual exclusion algorithms based on the use of a token for safety and on a rooted tree structure carrying the requests, for liveness. The serialization of concurrent accesses to a CS has to be carried out by following a starvation free discipline. Helary et al [HMR94] uses a waiting queue data structure associated with each node to deal with the multiplicity of requests. The requests wait in this queue when the node is busy and are served on FCFS basis. Therefore no starvation occurs. The straight implementation of a priority based discipline may cause an indefinite delay of a process. This is true since processes with a high priority may continue to enter and exit from their own CS causing starvation of processes with lower priority. A solution to starvation avoidance is to serialize the incoming mutual exclusion requests by a discipline based on Batch Priority Queues (BPQ). Each site maintains a BPQ to store the incoming requests. Let each batch be of size ‘x’. So a request is stored in the BPQ with its priority value and batch number. Batches are served serially and in each batch the requests are ordered on the basis of their priority. The value of ‘x’ is adjustable depending upon the application. If x=1, then the serialization is on FCFS basis. If x \to \infty, the serialization is priority based, but it leads to starvation. Therefore, depending upon how much waiting is tolerable for the lower priority processes, the value of ‘x’ can be adjusted. Here we take this value to be equal to ‘N’ i.e. equal to the number of the sites in the system. This ensures that the lowest priority request would be served atmost after N requests. Hence starvation is avoided.

4.2.1 Priority based General Algorithm
4.2.1.1 **Data structure used in the algorithm**

In this algorithm, we abstract the nodes, the request messages and the token message as three different types of objects. Each of them is associated with a set of information structure. Each node performs actions depending upon the type of message received.

For each node $i \ [i=1-N]$, the following data structures are constructed to record necessary state information, where $N$ is the total number of nodes in the network.

1. $\text{tokenhere}_i$: It is a boolean variable indicating the presence of token, its value is true if and only if node $i$ has the token.

2. $\text{asked}_i$: It is a boolean variable and has the value true if and only if node $i$ is currently waiting for the token or executing a CS.

3. $\text{father}_i$: Its value lies in the domain $\{1-N\} \cup \text{nil}$. When node $i$ wants to obtain the token, it sends a request message to its qualified neighbor $\text{father}_i$ and then waits for the token arrival. Initially all father variables are set in such a way that they define a rooted tree structure over the nodes, with the token located at the root.

4. $\text{lender}_i$: Its value lies in the domain $\{1-N\} \cup \text{nil}$. Its value indicates the node to which $i$ will have to give back the token when leaving the CS.

5. $\text{mandator}_i$: The value of this variable is a node identity and is meaningful only when node $i$ has requested the token for satisfying a request.

6. $\text{behavior}_i$: Each node can have a Proxy behavior or a Transit behavior. This behavior can be dynamically changed. The possibility of such a modification is a fundamental characteristic of the general algorithm. Suppose a node $i$ receives a request message from node $j$. Node $i$ reacts to this message depending upon its behavior:
Transit: Node i forwards the request message to father_i and sets father_i = j.

Proxy: Node i considers j its mandator and requests the token (to father_i) for itself, thus becoming an asking node i.e. asked_i = true. When i receives the token, the mandator’s request will be satisfied. Node i sends the token to j and considers its mandate for j as completed.

7. Req_no_i: This variable keeps a count of the number of requests received by node i. After the receipt of N requests, the batch number is incremented and req_no is reset to zero. This is done to avoid starvation.

8. Batch_no_i: It is incremented after every N requests.

9. Batch Priority Queue (BPQ): Each node maintains a batch priority queue in which batches are served serially and in each batch, requests are ordered on priority basis. The following operations are defined on these nodes:
   - Push_BPQ_i(id, pri, batch): Pushes a request of node id with priority pri and batch number equal to batch in the BPQ associated with node i. If a request from node id already exists in the queue, then it is replaced by this new request.
   - high_pri_i(): Returns the node id of the highest priority request and its priority value. Returns -1 if the BPQ_i is empty.
   - Remove_req_i(id): Removes the request of node id from the BPQ_i.
   - BPQ_empty_i(): Returns true if the BPQ_i is empty.

Request Message: It carries the requester’s id and priority value. The format of request message is ReqMsg(requester, pri).
Token Message: The token message carries the identity i of the node to which it must be returned (lender), if any, and nil otherwise. It is denoted by token(i) or token(nil), respectively.

4.2.1.2 Initialization

Initially, the system selects one of the nodes, denoted by node r, as the token owner and initializes the information structures of all nodes as follows:

For each node i \( i \neq r \):

- father\(_i\) = r; tokenhere\(_i\) = false;
- asked\(_i\) = false; lender\(_i\) = nil;
- mandator\(_i\) = nil; req\_no\(_i\) = 0;
- batch\_no\(_i\) = 0;

for node r:

- father\(_r\) = nil; tokenhere\(_r\) = true; asked\(_r\) = false; lender\(_r\) = r;
- mandator\(_r\) = nil; req\_no\(_r\) = 0; batch\_no\(_r\) = 0;

Behavior of each node is initialized depending upon the particular algorithm.

4.2.1.3 Execution rules of the algorithm

Each node \( i \{i=1\ldots N\} \) in the network is driven by the following events:

1. Node i produces a request for accessing the CS:

The request is pushed in the queue with its priority value and the batch number. If node i owns the token, It immediately accesses the CS. Otherwise it checks if asked\(_i\) = false i.e.
node is not busy serving some other request or if node is busy but this request has the highest priority among all waiting requests, then mandator \( i \) is set equal to this requester's id and a request message is sent to father \( i \). If the node \( i \) is busy and this request has lower priority among the waiting requests, then it remains in the \( BPQ_i \) till other higher priority requests are served. The detailed description of the procedure follows:

```c
/* This procedure is executed when node \( i \) wants to access the CS */

Make_Request()
{
    req_no_i++;
    if(req_no_i == N) {
        batch_no_i++; /* increments the batch number if required */
        req_no_i = 0;
    }
    Push_BPQ_i(i,pri,batch_no_i); /* pushes the request in BPQ_i */
    if(tokenhere_\( i \)) { /* if node \( i \) owns the token, it accesses the CS */
        asked_i = true;
        enter_cs();
        exit_cs();
    }
    else { /* sends a request only if node \( i \) is not busy or the current request is of highest priority */
        if(((asked_i == true) && i == high_pri_i()) || asked_i == false) {
            asked_i = false;
            mandator_i = i;
            send ReqMsg(i,pri) to father_i;
        }
    }
}
```

2. **Node \( i \) receives a request message ReqMsg(j,pri)**

The request is pushed in the \( BPQ_i \) with its priority value and batch number. If node \( i \) is not busy then depending upon its current behavior it acts as follows:

- **Proxy behavior**: If node \( i \) has the token, it sends it to node \( j \), with lender as itself. It also removes the request from its \( BPQ_i \). If \( i \) does not have the token, it sets its
mandator_i is equal to j, and sends a request message to father_i with itself as requester and priority value of j. Node i also sets asked_i = true implying that it is busy serving a request.

- Transit behavior: If node i has the token it sends it to node j, otherwise it forwards the request to father_i. Node i sets father_i = j, so that future requests are sent to node j. Node i removes the request from the BPQ_i.

If node i is busy when the request message arrives then if it is the highest priority request in BPQ_i then the following action is taken:

If node i is proxy, then mandator_i is set to j and a request on node i's behalf with priority value of node j is sent to father_i. If node i is transit, then the request is forwarded to father_i and father_i is set to j. Also the request is removed from the BPQ_i. Note that mandator_i is equal to the highest priority requester at node i. The detailed description of the procedure follows:

/* This procedure is executed when node i receives a request from node j with priority equal to pri */

ReqRecv(j,pri)
{
    req_no_i++;
    if(req_no_i == N) {
        batch_no_i++; /* batch number is updated */
        req_no_i = 0;
    }
    Push_BPQ_i(j,pri,batch_no_i); /* The received request is pushed in BPQ_i */
    if(asked_i = false) { /* no pending request */
        case (behavior_i) {
            proxy :
                asked_i = true; /* node i sets itself to busy */
                if(tokenhere_i) {
                    send token(i) to j; /* If i has the token it sends it to j */
                    remove_req_i(j); /* Removes the request from its BPQ */
                    tokenhere_i = false;
                } else 

mandator_i = j;
send ReqMsg(i,pri) to father_i; /* sends request to father_i with priority pri */
}
break;

transit:
if(tokenhere_i) { /* If i has the token, it sends it to j, otherwise sends the request of j to father_i */
lender_i = nil;
send token(nil) to j;
tokenhere_i = false;
}
else
send request(j,pri) to father_i;
remove_req_i(j);
father_i = j;
break;
}
else { /* node is busy serving some request */
case( behavior_i) {
proxy:
if(j == high_pri_i) { /* request is further propagated only if it has the highest priority among the waiting requests at node i */
mandator_i = j;
send ReqMsg(i,pri) to father_i;
}
transit:
if(j == high_pri_i) {
send ReqMsg(j, pri) to father_i;
father_i = j;
remove_req_i(j);
}
}
}

3. Node i receives the token message token(j) from node k:

On receipt of token three cases are to be considered.
• i is the lender and the token is given back to i after a loan. At this time, \( \text{mandator}_i = \text{nil} \). On receipt of the token, the node i sets \( \text{asked}_i = \text{false} \) and processes the next request from the \( \text{BPQ}_i \), if any.

• \( \text{mandator}_i = i \) implies that node i wants to enter the CS. If the token is not to be returned to the lender i.e. \( j = \text{nil} \), then i becomes its own lender and \( \text{father}_i \) is set to \( \text{nil} \). Otherwise \( \text{lender}_i \) is set to j and node k becomes the \( \text{father}_i \). \( \text{mandator}_i \) is set to \( \text{nil} \) and node i enters the CS.

• If \( \text{mandator}_i \) is not equal to \( \text{nil} \) or i, it implies that a request was made by i on account of another node j. The request is removed from the \( \text{BPQ}_i \) and node i redefines its position in the tree, sets in the token a value, depending on its current behavior and on the value brought up by the token, and sends the token to its \( \text{mandator} \). If \( \text{asked}_i \) becomes equal to false, node i processes the next request in the \( \text{BPQ}_i \), if any. The detailed description of the procedure is as follows:

/* This procedure is executed by node i when it receives the token */

\[
\text{token\_recv}(j)
\]

{  
  \( \text{tokenhere}_i = \text{true}; \)
  \text{case}(\text{mandator}_i) \{
    \text{nil} : /* Token is returned after a loan */
      \text{asked}_i = \text{false};
      \text{process\_next\_request\_i}(); /*as node i is now free, it can service the pending requests*/
      \text{break};
    \text{i} : /* node i wants to access the CS */
      \text{if}(j == \text{nil}) \{
        \text{lender}_i = i; /* The token has no lender, so i becomes the lender */
        \text{father}_i = \text{nil};
      \}
      \text{else} \{
        \text{lender}_i = j; /* i has to return the token to j */
        \text{father}_i = k;
  \}
mandator _i != i.nil : /* i honors the request of its mandator */
  asked _i = false;
  remove _req _i (mandator _i);
  case (behavior _j) {
    proxy :
      if (j == nil) { /* Token has no lender */
        lender _i = i;
        father _i = nil;
        send token (i) to mandator _i;
        asked = true;
      }
      else { /* j is the lender of the token */
        father _i = k;
        send token (j) to mandator _i;
      }
    break;
  }
  transit :
    if (j == nil) { /* Token has no lender */
      lender _i = nil;
      father _i = mandator _i;
      send token (nil) to mandator _i;
    }
    else { /* j is the lender of the token */
      father _i = k;
      send token (j) to mandator _i;
    }
    break;
  }
  mandator _i = nil;
  tokenhere _i = false;
}
if (asked == false) /* If node is free and has some waiting requests, then it processes the next highest priority request */
  process _next _request _i();
}

4. **Node i exits from the CS**
After Node i accesses its CS, it removes its request from $BPQ_i$. If the token was lent by some other node, then it is returned. After execution of the CS, node i processes the next request from $BPQ_i$, if any. The description of the procedure follows:

/* This procedure is executed when i exits its CS */

exit_cs ( )
{
    remove_req_i(i), /* removes the request of i from BPQ_i */
    if(lender_i != i) {
        send token(nil) to lender_i; /* returns token to lender */
        tokenhere_i = false;
    }
    asked = false;
    process_next_request_i( ); /* Services the next request in the BPQ_i, if any */
}

Processing next request from $BPQ_i$

Whenever a node i becomes free, it checks in $BPQ_i$ for any waiting requests. If so, it picks up the highest priority request from the $BPQ_i$ and processes it depending upon its behavior:

- **Proxy behavior**: If the highest priority request is of the node itself and it owns the token, it enters the CS. Otherwise it sends a request message to its father. If the highest priority request is of node $j, j!=i$, then if i has the token, it sends it to j, else considers j as its mandator and sends a request message for itself with priority of j’s request to father_i. asked_i is set to true.
• *Transit behavior*: If the node is in transit mode, it sends the waiting requests to $father_i$. If the waiting request is of node $i$ itself, then it sets $asked_i$ to true and services the request. Once $asked_i$ is set to true, no further requests are processed till it becomes false or a new higher priority request message is received. If node $i$ owns the token, it sends it to the highest priority requester and sets $father_i$ equal to the requester, so that future requests are sent there. Till the $BPQ_i$ is not empty, or the highest priority request is of node $i$ itself, it keeps on sending the request messages to $father_i$ and updates $father_i$.

This way prioritized processing of requests is done. The detailed procedure follows:

```c
Process_next_request_i( )
{
  if(!BPQ_empty_i( )) { /* Requests are waiting in the BPQ_i */
    requester = high_pri_i;
    case(behavior_i) {
      proxy :
        asked_i = true;
        if(requester.id == i) { /* Request of self */
          if(tokenhere_i) {
            enter_cs();
            exit_cs();
          }
        } else {
          mandator_i = i;
          send ReqMsg(requester.id, requester.pri) to father_i;
        }
      }
    }
    else {
      if(tokenhere_i) {
        send token(i) to requester;
        remove_req_i(requester);
        tokenhere_i = false;
      }
    }
  }
  else {
    mandator_i = j;
    send ReqMsg(i, requester.pri) to father_i;
  }
}
```
break;

transit:
  while(!BPQ_empty_i()) {/* Requests are waiting in the BPQ_i */
    if(requester.id == i) {/* If request is of self then other requests wait in queue till it is */
      asked_i = true;
      if( tokenhere_i) {
        enter_cs();
        exit_cs();
      } else {
        mandator_i = i;
        send ReqMsg(requester.id, requester.pri) to father_i,
      }
      break;
    }
    if(tokenhere_i) {
      lender_i = nil;
      send token(nil) to requester.id;
      tokenhere_i = false;
    } else {
      send ReqMsg(requester.id, requester.pri) to father_i;
      father_i = requester.id; /* father is set to requester */
      remove_req_i(requester.id); /* request is removed */
      requester = high_pri_i>(); /* next highest priority request is serviced */
    }
    break;
  }

4.2.2 Correctness of the General Algorithm

1. Mutual Exclusion: As there is a unique token in the system, and a node is allowed to access the CS if and only if it holds the token, mutual exclusion is trivially guaranteed. At any point of time there exists atmost one node i such that tokenhere_i is true. By construction, it holds in the initial state. Later every sending of the token is
possible only by a node $x$, such that $token_{here \ x} = true$ and the sending of the token sets $token_{here \ x}$ to false. Only on receipt of token message by node $i$, it sets $token_{here \ i} = true$. Hence mutual exclusion is achieved.

2. **Starvation freedom**: Starvation occurs when a node wishing to access CS is forever unable to do so while other nodes enter and exit. Starvation freedom is proved in [HMR94] when FCFS discipline is used. When priority based serialization is used, the lower priority nodes may starve if the frequency of higher priority requests is very high. In this case the higher priority requests would keep on accessing the CS and the lower priority requests would not get a chance to enter the CS. To avoid this we have used the Batched Priority Queue concept. The requests are collected in batches and the batches are served serially, hence avoiding starvation.

3. **Deadlock avoidance**: The system is deadlocked when no node is in the CS and all nodes wishing to enter the CS are not able to do so. Deadlock avoidance for FCFS serialization is proved in [HMR94]. After inserting priority also, the liveness (i.e. deadlock freedom & starvation freedom) is retained. Liveness is violated if for some node $i$, $asked_i$ remains continuously and indefinitely true. $asked_i$ is set to true if a node wants a token for itself or for its mandator (Proxy behavior). $asked_i$ is set to false on receiving the token. When a request for token is received by a node $i$ then if the node has the token, it sends it to the requester thereby setting the requester’s asked variable to false. If it does not have the token and the node $i$ is also not currently busy i.e. $asked_i=false$, then depending upon its behavior:

- **Proxy behavior**: Node $i$ sets its $asked_i = true$ and sends a request on its behalf with requester’s priority value to its father and waits for the token.
• Transit behavior: Node i forwards the request to its father and sets $father_i = requester$.

If on receiving the request a node i is busy, then the request is processed only if it has the highest priority among the waiting requests and $mandator_i$ is updated to this highest priority requester's id. Therefore $mandator_i$ always stores the id of the highest priority request a node has received while waiting for the token. Therefore on receipt of the token, the highest priority request is serviced. After servicing the highest priority request, node i removes this request from the $BPQ_i$ and picks up the next highest priority request if any, for servicing. Now, depending upon the behavior:

• Proxy behavior: Node i sets $asked_i$ equal to true and services this request. Only if $BPQ_i$ is empty, $asked_i$ is set to false.

• Transit behavior: Till the $BPQ_i$ is not empty or the highest priority requester is not self, the node i forwards the pending requests to $father_i$ and updates $father_i$ accordingly. The request is removed from the $BPQ_i$. If the request is for self then $asked_i$ is set to true and other pending requests would be serviced after node i satisfies its own request. $asked_i$ is set to false when all waiting requests have been serviced.

This way deadlock freedom is acheived.

4.3 Priority based Raymond's Algorithm

Here we give the algorithm of priority based variant of Raymond 's algorithm deduced from the general priority based algorithm developed in the previous section. All the nodes have a transit behavior when they own the token, and proxy behavior otherwise.
Hence, variables \textit{behavior} \_i and \textit{lender} \_i are removed, and the token does not carry any value. Initialization is same as in the general algorithm. Each node [1—N] in the network is driven by the following events:

1. Upon a call to \texttt{enter_cs} by node \textit{i}, the following module is executed. It is same as the general algorithm. The detailed description of the procedure follows:

/* This procedure is executed when node \textit{i} wants to access the CS */

\begin{verbatim}
Make_Reqst( )
{
    req_no_i++; 
    if(req_no_i == N) {
        batch_no_i++; 
        req_no_i = 0;
    }
    Push_BPQ_i(i,pri,batch_no_i);
    if(tokenhere_i) {
        asked_i = true;
        enter_cs();
        exit_cs();
    } else {
        if(( asked_i == true) && i == high_pri_i()) || asked_i == false) {
            asked_i = true;
            mandator_i = i;
            send ReqMsg(i,pri) to father_i;
        }
    }
}
\end{verbatim}

2. Node \textit{i} receives a request message \texttt{ReqMsg(j,pri)}

On receipt of a request message node \textit{i} increments the batch number, if required and pushes the request in \texttt{BPQ}_i. If node \textit{i} owns the token, it sends it to the requester and removes the request from the \texttt{BPQ}_i and sets it's father variable to the requester's id. If
node \( i \) does not hold the token then if it is free or the received request is of highest priority among the waiting requests, node \( i \) sends a request message with requester as itself and priority value of node \( j \) to its father and waits for the token. \textit{mandator} \_i is set to \( j \). If the received request is of lower priority value then the pending requests at node \( i \), then it is pushed in the \( BPQ \_i \) and serviced only after the higher priority requests have been serviced. The detailed description of the procedure follows:

/* This procedure is executed when node \( i \) receives a request from node \( j \) with priority equal to \( pri \) */

\textbf{ReqRecv}( j, pri)
\{
  \textbf{req\_no\_i}++;
  \textbf{if(req\_no\_i} == \textbf{N}) \{  
    \textbf{batch\_no\_i}++; /* increment batch number, if necessary */
    \textbf{req\_no\_i} = 0;
  \}
  \textbf{Push\_BPQ\_i(j, pri, batch\_no\_i)}; /* Push the request in \( BPQ\_i \) */
  \textbf{if(tokenhere\_i)} \{ /* if token is at node \( i \), it is sent to node \( j \) */
    \textbf{send token}() to \( j \);
    \textbf{tokenhere\_i} = false;
    \textbf{father\_i} = \( j \); /* \( j \) becomes the father of \( i \) */
    \textbf{remove\_req\_i(j)}; /* Request of \( j \) is removed from the \( BPQ\_i \) */
  \}
  \textbf{else} \{
    \textbf{if(( asked\_i} == \textbf{true}) \&\& \( j \) == \textbf{high\_pri\_i(J)} || \textbf{asked\_i} == \textbf{false}) \{ /* If \( i \) is free or the request of \( j \) is of highest priority among the waiting requests at node \( i \) */
      \textbf{asked\_i} = \textbf{true};
      \textbf{mandator\_i} = \( j \); /* \( i \) sets itself as mandator of \( j \) */
      \textbf{send Req\_Msg(i, pri) to father\_i};
    \}
  \}
\}

3. Node \( i \) receives a token message token()

On receipt of the token node \( i \) checks whether the \textit{mandator} \_i is set for itself or some other node. If \textit{mandator} \_i is equal to \( i \), then node \( i \) sets the father variable to nil and
enters the CS. Otherwise node i sends the token to mandator_i and removes the request of mandator_i from the BPQ_i. The algorithm ensures that mandator_i is equal to the highest priority request received so far. After sending the token to mandator_i, node i sets mandator_i to nil and services the next request from the BPQ_i, if any. The detailed description of the procedure is as follows:

/* This procedure is executed by node i when it receives the token */

token_recv( )
{
    tokenhere_i = true;
    case(mandator_i) {
        i: /* i accesses the CS */
            father_i = nil;
            mandator_i = nil;
            enter_cs();
            exit_cs();
            break;
        mandator_i != i: /* i had requested the token for mandator_i */
            asked_i = false;
            remove_req(mandator_i);
            father_i = mandator_i;
            send token() to mandator_i;
            tokenhere_i = false;
            mandator_i = nil;
            process_next_request_i(); /* next highest priority request is serviced from the queue */
            break;
    }
}

4. On exiting the CS

After executing the CS, node i removes its request from the BPQ_i, sets asked_i to false (i.e. indicates that it has serviced the current request) and then services the next highest priority request from the BPQ_i, if any. The description of the procedure follows:

/* This procedure is executed when i exits its CS */

exit_cs ( )
{
remove_req_i(i),
asked = false;
process_next_request_i(); /* next highest priority request is serviced from the queue */

**Processing next request from BPQ_i**

Whenever node i becomes free, it checks the BPQ_i for any waiting requests. If BPQ_i is not empty, it picks up the highest priority request for servicing. If the request is for self, then asked_i is set to true and other waiting requests are processed only after i services its own request. If the request is for some other node, then if i has the token it sends it to the requester and removes the requester from the queue. If the BPQ_i is still not empty, i sends a request to father_i on behalf of the next highest priority requester. If node i does not hold the token, it sends a request to father_i on behalf of the requester. This way prioritized processing of requests is done. The detailed procedure follows:

Process_next_request_i( )
{
    if(!BPQ_empty_i( )) {
        requester = high_pri_i();
        if(requester.id == i) { /* request is for self*/
            asked_i = true;
            if(tokenhere_i) {
                enter_cs();
                exit_cs();
            }
        } else {
            mandator_i = i;
            send ReqMsg(requester.id, requester.pri) to father_i;
        }
    } else { /* request of node j, j!=i */
        if(tokenhere_i) {
            send token() to requester;
            remove_req_i(requester);
        }
    }
}
tokenhere_i = false;
father_i = requester;
if(!BPQ_empty()) { /* send the request for the next highest priority node */
    requester = high_pri_i();
    asked_i = true;
    mandator_i = j;
    send ReqMsg(i, requester.pri) to father_i;
}
else { /* send a request message on requester’s behalf to father_i */
    asked_i = true;
    mandator_i = j;
    send ReqMsg(i, requester.pri) to father_i;
}
}

4.4 Priority based Naimi-Trehel’s Algorithm

Here we give the algorithm of priority based variant of Naimi-Trehel algorithm deduced from the general priority based algorithm developed in the previous section. All the nodes have transit behavior. Hence, variables behaviour_i, mandator_i and lender_i are removed, and the token does not carry any value. Initialization is same as in the general algorithm. Each node [1—N] in the network is driven by the following events:

1. Upon a call to enter_cs by node i, the following module is executed. It is same as the general algorithm. The detailed description of the procedure follows:

/* This procedure is executed when node i wants to access the CS */

Make_Reqst( )
{
    req_no_i++;
    if(req_no_i == N) {
        batch_no_i++;
    }
}
2. **Node i receives a request message ReqMsg(j,pri)**

On receipt of a request message from node j, node i increments the batch number, if required and pushes the request in $BPQ_i$. If i is not busy and holds the token, it sends the token to j and sets its father to j and removes the request from the $BPQ_i$. If node i is busy but j has the highest priority among the waiting requests, the request is forwarded to $father_i$ and $father_i$ is set to j. Request is removed from the $BPQ_i$. The request remains in the queue only if node i is busy and the request is of lower priority than node i’s request. The detailed description of the procedure follows:

/* This procedure is executed when node i receives a request from node j with priority equal to pri */

```c
ReqRecv(j, pri) {
    req_no_i++;
    if(req_no_i == N) {
        batch_no_i++;
        req_no_i = 0;
    }
    Push_BPQ_i(i, pri, batch_no_i);
    if(asked_i == false) {
        if(tokenhere_i) {
            asked_i = true;
            enter_cs();
            exit_cs();
        } else {
            if(((asked_i == true) && i == high_pri_i()) || asked_i == false) {
                asked_i = true;
                send ReqMsg(i, pri) to father_i;
            }
        }
    }
    if(tokenhere_i) {
        send token() to j;
    }
}
```
3. **Node i receives a token message token()**

On receipt of token node i accesses the CS. The detailed description of the procedure is as follows:

/* This procedure is executed by node i when it receives the token */

token_recv()
{
    tokenhere_i = true;
    father_i = nil;
    enter_cs();
    exit_cs();
}

4. **On exiting the CS**

On exiting the CS, node i removes its request from the BPQ_i, sets asked_i to false and services the waiting requests, if any. The description of the protocol follows:

/* This procedure is executed when i exits its CS */

exit_cs()
{
    remove_req_i(i);
    asked = false;
    process_next_request_i();
}
Processing next request from $BPQ_i$

When node $i$ becomes free it services the waiting requests in the $BPQ_i$, if any. The detailed procedure follows:

```plaintext
Process_next_request_i() {
    while(!BPQ_empty) {
        requester = high_pri_i();
        if(tokenhere_i) {
            send token() to requester.id;
            tokenhere_i = false;
        } else {
            send ReqMsg(requester.id, requester.pri) to father_i;
            father_i = requester;
            remove_req_i(requester);
        }
    }
}
```

4.5 A Prioritized Broadcast based – Token based Algorithm

4.5.1 Basic Idea

The new mutual exclusion algorithm, proposed here is token based and uses broadcast based approach to achieve mutual exclusion and supports priority serialization discipline. There is only one token message in the system at any given moment in time. The token is passed from node to node, and only the token holder is permitted to enter its CS. No assumptions are made with respect to the network topology and the communication
medium. The token contains an ordered list or queue (in priority order) also referred to as the Q-list of all the nodes that have been scheduled to execute their CS. The token is passed from node to node in order specified by the Q-list. The node which executes the CS is the current node at the head of the Q-list. The Q-list is created by a node currently designed as the arbiter of the system. Initially a specific node (say node 1) is assigned to be the arbiter by the system. Every node keeps track of the arbiter by setting a parameter called ARBITER. An arbiter node executes the following two phases: Request collection phase and Request forwarding phase.

- **Request Collection Phase**: In this phase the arbiter node collects all the requests from the nodes that are seeking access to their CS's and creates the priority wise ordered list of requests, Q-list. At the end of the request collection phase, when arbiter gets the token in possession, the token is updated with the newly constructed Q-list and transmitted to the node that is at the head of the Q-list, along with the contents of the Q-list. Further, the arbiter node declares the node of the last request in the Q-list as the new arbiter, by sending a broadcast message to all the nodes in the system. A node, on receipt of the message electing it as the new arbiter, enters the request collection phase. The token, which is passed from node to node according to the Q-list, finally reaches the new arbiter node. Nodes requesting access to the CS send their requests to the arbiter. It is possible, however that a node sends its request to the previous arbiter before a message informing it of the current arbiter arrives. In this case, the request message needs to be forwarded to the current arbiter.
• **Request forwarding phase**: An arbiter enters the request forwarding phase after the request collection phase in order to forward the requests that did not arrive during the request collection phase, but were transmitted before the identity of the current arbiter was received at requesting nodes. Any request arriving after the end of the forwarding phase are dropped.

The durations of the collection and forwarding phases are parameters that can be tuned for the best performance. It is observed that with a longer request collection phase, the average number of messages incurred is lower, but the average delay per CS is higher. This because if the request collection phase is longer, then most of the requests will register at a single arbiter and NEW-ARBITER broadcast messages would be less. But as request servicing starts only after request collection phase is over, longer the request collection phase, higher would be the average delay per CS. If request collection phase is smaller, then the arbiter nodes change quickly and therefore message traffic of NEW-ARBITER broadcasting is increased.

**4.5.2 Example**

Consider a distributed system with 5 nodes, numbered 1 through 5 respectively and assume that the message transmission time, request collection and forwarding durations, and the execution time per CS, are each equal to 1 unit of time.

Initially, node 1 is assigned to be the arbiter and enters the request collection phase immediately, whereas nodes 2, 4 and 5 require to be in their CS, and send their requests to node 1.
Let us assume further that the request from node 2, REQUEST(2, pri_1), and from node 5, REQUEST(5, pri_2) arrive at node one within the request collection phase (see fig. 4.1).

**Fig. 4.1 Illustrative Example**

![Diagram showing request collection and forwarding phases]
Let \( \text{pri}_1 > \text{pri}_2 \). At the end of the request collection phase node 1 assigns the node at the tail position of the Q-list - \( \text{Tail}(Q) \), viz., node 5 as the current arbiter, and broadcasts a NEW-ARBITER (5) message to all the nodes in the system. At the same time, the Token(Q) is transmitted to node \(2 - \text{Head}(Q)\), where Q is the priority ordered list : \{ 2,5\}. After this, node 1 enters the request forwarding phase. REQUEST (4,\text{pri}_3) arrives during the request forwarding phase, and node 1 forwards it to the current arbiter (node 5).

On receipt of the NEW-ARBITER (5) message, node 5 starts collecting requests from other nodes to access their CS. On receiving the Token(Q) message, node 2 enters its CS. After executing its CS, node 2 removes itself from the head position of \(Q\), and sends the Token(Q) message to the node currently at the head ( \(Q\) ), viz., node 5. When node 5 subsequently receives the token, it executes its CS, while still collecting requests in the background. After executing its CS node 5 removes itself from the head position, and enters its request collection phase, during which a request from node 3 (REQUEST(3, \text{pri}_4)) arrives. Let \text{pri}_3 be greater than \text{pri}_4.

At the end of the request collection phase, \(Q\) is the priority wise ordered list : \{4,3\}. Node 5 declares node 3 as the current arbiter by broadcasting the NEW-ARBITER(3) message, and the entire process is repeated.

4.5.3 Description of the Algorithm

The following three kinds of messages are used in this algorithm:
• **Token message**: This message is of the form Token(Q) where Q is the priority order list of the nodes which will be granted access to the CS one after the other. The last node in the Q is always the next arbiter node.

• **Request message**: It is of the form REQUEST (j,pri) where node j is requesting access to the CS with priority pri.

• **New Arbiter message**: This message is of the form NEW-ARBITER(j) , where node j becomes the new arbiter.

Each node i \(i = 1—N\) in the network is driven by the following events:

1. **Node i produces a request for accessing the CS**

   When a node i produces a request for accessing the CS, it sends a request message REQUEST(i,pri) with priority value attached with it to the ARBITER node and waits for the token. On receiving the token, the node sets the tokenhere variable to true indicating that it owns the token and executes the CS. Then it removes the highest priority request from the queue (token's Q-list) and sends the token to it. Node i sets tokenhere to false after sending the token to next requester.

   /* This procedure is executed when node i wants to access the CS */

   ```
   Make_Rqst( )
   {
   send REQUEST (i,pri) to ARBITER;
   wait for Token;
   tokenhere = true;
   enter_CS;
   Q = Remove(Q, Head(Q));
   send Token(Q) to Head(Q);
   tokenhere = false;
   }
   ```
2. **Node i enters the Request Collection Phase**

If node i is the current arbiter, it enters the request collection phase. Any incoming request is placed in the queue (Q) in priority order. The duration of the request collection phase (REQ_Collection_Time) is a user defined parameter which can be tuned to achieve desired performance.

/* This procedure is executed when node i enters Request Collection Phase */

```
Re<collection ( )
{
    t = 0;
    while ( t <= REQ_Collection_Time) {
        Add incoming REQUEST (id, pri) to Q;
        t = t+1;
    }
}
```

3. **Node i enters Request Forwarding Phase**

An arbiter enters the request forwarding phase after the request collection phase. It sends the incoming requests to the new arbiter. The duration of the request forwarding phase (REQ_Foward_Time) is an external parameter which can be tuned to achieve desired performance.

/* This procedure is executed when node i enters Request Forwarding Phase */

```
Re<forward ( )
{
    t = 0;
    while ( t <= REQ_Foward_Time) {
        send incoming REQUEST (id, pri) to ARBITER;
        t = t + 1;
    }
}
```
4. **Node i becomes the arbiter node**

When a node receives a NEW-ARBITER message for itself, it becomes the new arbiter. Then it starts collecting requests till it receives the token message. On receipt of the Token message, it checks if the highest priority request in the Q is for itself. If so, node i accesses the CS. Then it enters the request collection phase. After completion of request collection phase, node i checks if the queue (Q) is empty. If so, it continues to be in the request collection phase otherwise it sends the token with the prioritized queue (Q) to the highest priority requester. Then node i broadcasts the NEW-ARBITER message declaring the last node of the Q as the new arbiter. Then node i enters the request forwarding phase.

/* This procedure is executed by an arbiter node */

```c
arbiter_node ( )
{
    Q = empty;
    while (! tokenhere) {
        add incoming REQUEST(id,pri) to Q,
    }
    if(Head(Q) == self) {
        enter_cs;
    }
    Req_collection();
    if( Q is not empty) {
        send Token(Q) to Head(Q);
        Broadcast NEW-ARBITER(Tail(Q));
        Req_forward ();
    }
    else {
        Req_collection ( );
    }
}
```

5. **Node i receives a NEW-ARBITER (id) message**
When a node receives a NEW-ARBITER(id) message, it first checks if id is equal to i, if so, node i becomes the new arbiter, otherwise it updates ARBITER equal to id, so that future requests generated by node i are sent to id.

/* This procedure executes when node i receives a NEW-ARBITER(id) message */

Recv_new_arbiter(id)
{
    ARBITER = id;
    if(ARBITER === i) {
        arbiter_node();
    }
}

4.5.4 Algorithm Correctness

1. Mutual exclusion: Assuming that the token is not lost, or replicated by the network, there is only one token in the system. The main point is that at any given time, only one node possesses the token, and hence is in its CS, and only the same node can pass the token when it exits the CS. In the proposed algorithm, the token is passed to the node at head position of the ordered list of requesting nodes. Since the head position can be occupied by only one node at any given time, (and only the current head can send the token to the new Head(Q)) the token is never sent to more than one node, ensuring that no two nodes can be in the CS simultaneously.

2. Starvation freedom: In the context of mutual exclusion algorithms, starvation is said to occur if one or more nodes do not receive permission to enter the CS for excessively long periods of time, and sometimes never at all. In the proposed
algorithm, starvation may occur only if request messages continuously get forwarded, without the request being registered at the arbiter, or if request messages are continuously dropped because of their arrival at the arbiter after the request collection and forwarding phases are over. This problem may be termed as indefinite forwarding problem. To resolve this problem one node is assigned to be a monitor node whose identity is known to all other nodes. The NEW ARBITER message also broadcasts the Q-list. Request messages that have been forwarded more number of times than a given threshold T, are dropped by an arbiter even if they arrive within the request forwarding phase. A request node resubmits the request to the monitor after failing to see its request in the Q-list in T consecutive NEW ARBITER messages, which is an indication that its request has been dropped. The problem of starvation is eliminated by passing the token to the monitor node periodically with a specific period. The monitor does not forward any request messages, but stores them until the token arrives and the requests can be appended to the Q-list. Thus, the monitor maintains a set of requests that could potentially have become victims of the indefinite forwarding phenomenon.