A Fault Tolerant Token-based Algorithm for Group Mutual Exclusion in Distributed Systems

Abhishek Swaroop, and Awadhesh Kumar Singh

Abstract—The group mutual exclusion (GME) problem is a variant of the mutual exclusion problem. In the present paper a token-based group mutual exclusion algorithm, capable of handling transient faults, is proposed. The algorithm uses the concept of dynamic request sets. A time out mechanism is used to detect the token loss; also, a distributed scheme is used to regenerate the token. The worst case message complexity of the algorithm is n+1. The maximum concurrency and forum switch complexity of the algorithm are n and min (n, m) respectively, where n is the number of processes and m is the number of groups. The algorithm also satisfies another desirable property called smooth admission. The scheme can also be adapted to handle the extended group mutual exclusion problem.

Keywords—Dynamic request sets, Fault tolerance, Smooth admission, Transient faults.

I. INTRODUCTION

The mutual exclusion is a classical problem of distributed systems. Joung [1] proposed the group mutual exclusion (GME) problem, a generalization of the mutual exclusion problem, and modeled it as the congenital talking philosophers (CTP) problem. In group mutual exclusion, a process requests a resource type (group) before entering its critical section (CS). Processes requesting the same group are allowed to be in their CS simultaneously. However, processes requesting different groups, must execute their CS in mutually exclusive way. The time interval in which all critical sections executed are of the same type is called a ‘session’. An interesting application of GME is, when several users share large data objects stored in some secondary storage (such as CD’s), and only one data object can be loaded in the buffer at a time. A solution of group mutual exclusion problem must satisfy the following requirements:

Safety: No two processes, requesting different groups, can be in their critical sections concurrently.

Starvation Freedom: A process attempting to attend a session will eventually succeed.

Concurrent Occupancy: If some process P, has requested a group X, and no philosopher is currently attending or requesting a different group, then P can attend X, without waiting for any other process to leave the CS. The term ‘concurrent occupancy’ was first used by Kean and Moir in [2].

Joung solved the GME problem for shared memory systems in [1]. Later on, a number of solutions were proposed using different approaches like, permission-based algorithm [3], token-based algorithms [4-10], and non token-based solutions [11-13]. Out of them, there are only three token-based algorithms for fully connected networks: Mittal-Mohan’s TokenGME [8], Mamun-Nazakato algorithm [6] and Swaroop-Singh algorithm [10]. Mittal-Mohan’s TokenGME, which is based upon Suzuki-Kasmi algorithm [14], uses two types of tokens, primary token and secondary tokens. The algorithm uses static request sets and its message complexity is 2*(n-1). In Mamun-Nazakato algorithm, a session is opened for a predefined time and processes are made aware about it, through broadcast. The processes interested in the currently open session, may join it without incurring any message overhead. However, the algorithm needs that the processes maintain synchronized logical clocks. In [10] Swaroop and Singh presented a token-based algorithm in which each process announces a priority level along with its request. The worst case message complexity of the algorithm is n+1. The algorithm favors the request with higher priority levels. This feature makes the algorithm suitable for soft real time distributed systems. The concept of aging is used to remove the possibility of starvation. The algorithm presented in [10] assumes that all channels and processes are reliable.

The token-based algorithms are susceptible to token loss and token has to be regenerated in case token is lost in transit or the site holding the token fails. In the present paper, we propose a token-based algorithm, called DRS_GME henceforth, to solve the group mutual exclusion problem. Our algorithm uses the concept of dynamic request sets. Chang, Singhal, and Liu [15] used the dynamic request sets in their algorithm to solve the classical mutual exclusion problem. The proposed algorithm is capable of handling transient faults [16]. The algorithm also satisfies a desirable property called smooth admission [17], which ensures that when captain is in its critical section, a process requesting for the same group is
allowed to enter in its CS immediately by the captain. The use of dynamic request sets reduces the number of messages per CS request considerably, when the system is lightly loaded. The reason is that the cardinality of request sets will be far less than \( n-1 \) in that case. In the proposed scheme, a captain process is responsible for the session initiation and sending ‘start’ message to other processes requesting the same resource type as requested by the captain, in order to allow them to enter in CS as follower. The algorithm uses a distributed scheme, adapted from Manivannan and Singhal [18], for the token regeneration. A timeout mechanism detects message losses due to site failure and (or) communication link failure.

II. SYSTEM MODEL

We assume that the system has \( n \) sites, numbered as 1,2,3,...,\( n \). The only way of communication between sites, is through message passing. The system is fully logically connected. We assume that, at each site \( i \), there exists exactly one process \( P_i \). Hence, we can use site and process interchangeably. The maximum message delay and the time for which a process can be in its CS are bounded. We also assume that only transient faults occur in the system, and for which a process can be in its CS are bounded. We assume that the system has \( 2n \) processes, and may work as captain. Initially the process \( P_1 \) holds the token.

A process \( P_i \) requesting a resource sends its request to all members in its request set, if it is not holding the ‘valid’ token. However, if it is holding the valid token in state \( HI \), it immediately enters in its CS as captain. If \( P_i \) is in state \( HS \), it enters in its CS only if the requested resource type is the same as the token.type and the token.queue is empty; otherwise, the request is added in token.queue.

When a process \( P_i \) holding the valid token receives a request from \( P_j \) for the resource type \( X \), it transfers the token to \( P_j \) immediately, if state is \( HI \). If \( P_i \) is in state \( N, R, \) or \( EF \) it adds \( P_j \) in its request set if \( P_j \) is not already there . Furthermore, \( P_i \) also sends a request message to \( P_j \), if \( P_j \) is not in \( RS \); and state is \( R \). When the request of a process reaches the captain which is executing in its CS, It issues a ‘start’ message to \( P_j \), if \( X \) is the same resource as token.type. However, if the request is conflicting, it is added in the token.queue. Furthermore, in order to remove the possibility of starvation, if the captain is in state \( HS \), it issues a ‘start’ message only if the request is of the same type and there are no conflicting pending requests.

A process upon receiving a ‘start’ message enters in its CS as follower and sends a ‘complete’ message to its captain upon exiting from its CS. When a captain process comes out of its CS, it waits till all its followers have come out of CS and only then it selects next captain from the front of the token.queue and passes the token to the next captain if any; otherwise, it holds the token in state \( HI \). When a captain process transfers the token to new captain the copy of the token is stored in a local variable old_token, which is used in token regeneration process. Furthermore, the id of processes, which can work as future captain, are added in the request set of current captain before transferring the token to the new captain. As soon as a process \( P_i \) receives a valid token (old_token,session<token.session) it empties its request set, delete entry at the front of the token.queue, sends an start message to all of its followers, and enters in its CS.

In our algorithm, the token is a message, which contains an FCFS queue, namely token.queue, in order to store all pending requests. The token stores the number of the last completed session in token.session. The token contains two more variables (a) token.type that stores the type of current session and (b) token.followers that stores the number of follower processes. The requests for the same resource are grouped together and treated as one entry in the token.queue.

Each process may be in any one of the following six states: \( N \) (Not requesting), \( R \) (Requesting), \( EC \) (Executing in CS as captain), \( EF \) (Executing in CS as follower), \( HS \) (Captain but not in CS), and \( HI \) (Holding token because no request is pending).

The process \( P_i \) at site \( i \), has the following local variables:

<table>
<thead>
<tr>
<th>state, - the current state of ( P_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>captain,- stores the id of its captain.</td>
</tr>
<tr>
<td>SN, - is an array of sequence numbers.</td>
</tr>
<tr>
<td>RS, - request set of site ( i ).</td>
</tr>
<tr>
<td>old_token, - a copy of the token is stored in it.</td>
</tr>
<tr>
<td>TGII,- indicates whether the token regeneration process has been initiated by site ( i ).</td>
</tr>
<tr>
<td>follower, - is the set of follower</td>
</tr>
</tbody>
</table>

IV. THE ALGORITHM

The pseudo code of the algorithm is given in Appendix A; however, for reader’s convenience a high level description of the algorithm is presented in this section. Our algorithm uses the dynamic request sets technique and each site stores in its request set the process identifiers, called id henceforth, of sites which are possibly holding the token. This request set changes dynamically as the execution progresses. The sequence numbers are used to differentiate between old delayed requests and new requests. There exists a unique valid token and the process, holding this token only, may initiate a session and may work as captain. Initially the process \( P_i \) holds the token. A process \( P_i \) requesting a resource sends its request to all members in its request set, if it is not holding the ‘valid’ token. However, if it is holding the valid token in state \( HI \), it immediately enters in its CS as captain. If \( P_i \) is in state \( HS \), it enters in its CS only if the requested resource type is the same as the token.type and the token.queue is empty; otherwise, the request is added in token.queue.

When a process \( P_i \) holding the valid token receives a request from \( P_j \) for the resource type \( X \), it transfers the token to \( P_j \) immediately, if state is \( HI \). If \( P_i \) is in state \( N, R, \) or \( EF \) it adds \( P_j \) in its request set if \( P_j \) is not already there . Furthermore, \( P_i \) also sends a request message to \( P_j \), if \( P_j \) is not in \( RS \); and state is \( R \). When the request of a process reaches the captain which is executing in its CS, It issues a ‘start’ message to \( P_j \), if \( X \) is the same resource as token.type. However, if the request is conflicting, it is added in the token.queue. Furthermore, in order to remove the possibility of starvation, if the captain is in state \( HS \), it issues a ‘start’ message only if the request is of the same type and there are no conflicting pending requests.

A process upon receiving a ‘start’ message enters in its CS as follower and sends a ‘complete’ message to its captain upon exiting from its CS. When a captain process comes out of its CS, it waits till all its followers have come out of CS and only then it selects next captain from the front of the token.queue and passes the token to the next captain if any; otherwise, it holds the token in state \( HI \). When a captain process transfers the token to new captain the copy of the token is stored in a local variable old_token, which is used in token regeneration process. Furthermore, the id of processes, which can work as future captain, are added in the request set of current captain before transferring the token to the new captain. As soon as a process \( P_i \) receives a valid token (old_token,session<token.session) it empties its request set, delete entry at the front of the token.queue, sends an start message to all of its followers, and enters in its CS.
once after $P_1$ has executed in its CS as captain then $P_1$ generates a new token with the help of old_token, $P_1$ selects new captain from old_token_queue and sends the newly generated token to it. When a process $P_2$ receives a token, it checks whether the session number of new token is greater then that of older one. If so, it accepts the newly received token as valid token; otherwise, the token is deleted.

When timer $T2$ of the captain node expires, it suspects the loss of message ‘complete’ or ‘start’ and sends the message is_complete($j,X$) to all its follower sites from where it has not yet received the message ‘complete’. Upon reception of ‘is_complete’ message, there could be three possibilities: (a) if $P_j$ is in state $EF$ then $P_j$ ignores it, (b) if $P_j$ is requesting for session $X$, it enters in its CS, and (c) otherwise, process $P_j$ sends a ‘complete’ message to the captain.

The value of $T_{req}$ should be chosen carefully so that a token loss is detected well in time and the false token losses go undetected. Let $t_{m}=$ maximum message delay and $t_{c}=$ the maximum time period a process will be executing inside its CS. A reasonable value of $T_{req}$ would be $(n+1)*t_{m} + (n-1)*t_{c}$. The suggested value for $T_{req}$ is $2*t_{m}+t_{c}$ because in $2*t_{m}+t_{c}$ time the captain must have received a complete message from a follower.

**Example:** In order to provide convenience to the reader, we consider the following example. Let $P_1$, $P_2$, $P_3$ and $P_4$ be the four processes in the system and let $g_1$, $g_2$ and $g_3$ be the three groups. Initially token_queue is empty, $RS_1=$ $\Phi$, $RS_2=$ {1,3,4}, $RS_3=$ {1,2,4} and $RS_4=$ {1,2,3}. We consider the following sequence of events:

- (a) $P_2$ sends request for group $g_2$ to {1,3,4}. $P_2$’s request for $g_2$ reaches at sites {1,3,4}. $P_1$ transfers token to $P_2$ and $P_2$ enters in its CS as captain.
- (b) $P_1$’s request for $g_3$ reaches at $P_2$ which is in CS.
- (c) $P_3$ sends request for $g_1$ to {1,2,4}. The request reaches at sites {1,2,4}, however, $P_2$ still in CS.
- (d) $P_4$ sends request for $g_3$ to {1,2,3} The request reaches at sites {1,2,4}, however, $P_2$ still in CS.
- (e) $P_2$ comes out of CS; $P_2$ adds 1 and 3 in its request set, selects $P_1$ as new captain, sends token to $P_1$. $P_1$ on receiving token sends start message to $P_4$.

Table I describes the changes in token_queue and Request sets with the occurrence of above mentioned events.

<table>
<thead>
<tr>
<th>Table I: CHANGES IN TOKEN_QUEUE AND REQUEST SETS</th>
<th>event</th>
<th>token_queue</th>
<th>Request sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>After event (a)</td>
<td></td>
<td></td>
<td>$RS_1={2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$RS_2=\Phi$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$RS_3={1,2,4}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$RS_4={1,2,3}$</td>
</tr>
<tr>
<td>After event (b)</td>
<td>$g_3$</td>
<td>1</td>
<td>$RS_1={2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$RS_2=\Phi$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$RS_3={1,2,4}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$RS_4={1,2,3}$</td>
</tr>
</tbody>
</table>

V. PROOF OF CORRECTNESS

In this section, we prove that DRS.GME satisfies the requirements of GME problem namely safety, Starvation freedom, and concurrent occupancy. Let, type(i) = type of resource being used by $P_i$ and session(i)=latest session number executed or being executed by $P_i$.

Following invariants hold in the system:

- $session(captain) = session(i)$  
- $type(captain) = type(i)$

We take the help of the following boolean functions to prove the properties of our algorithm.

- $in\_CS(i) = true$, if $P_i$ is in state $EC$ or $EF$, false, otherwise.
- $holds\_valid\_token(i) = true$, if $P_i$ is in state $EC$, $HS$, or $HI$, false, otherwise.
- $captain(i) = true$, if $P_i$ is in state $EC$ or $HS$, false, otherwise.

**Lemma 1.** There exists at most one valid token in the system.

**Proof.** We assume, initially $P_1$ has the token, hence, $holds\_valid\_token(1) = true$ and $holds\_valid\_token(i) = false$ for sites $i\neq 1$. This token is transferred from one captain to another captain as the algorithm progresses. In response to $gen\_token(i,X,SN/session)$, a process $P_i$ generates a token, only if the condition $(session = old\_token/session)$ is satisfied. $P_i$ transfers this newly generated token to the site to which it has sent the token most recently. A token received by a process $P_i$ is valid only if $old\_token/session < token\_session$. All the invalid tokens are deleted immediately. The condition $old\_token/session < token\_session$ can be true for at most one token, which is generated by the site that executed in its CS as a captain most recently. Only one such site may exist in the system. Therefore, only one valid token will be retained.

**Safety:** If two processes are executing in their CS simultaneously then both the CS is of the same type.

**Proof:** Let us assume the contrary. The two processes $P_i$ and $P_j$ are in CS having their type as $t_i$ and $t_j$ ($t_i\neq t_j$) respectively. Since $P_i$ and $P_j$ are in CS, $state_i=EC$ or $EF$ and similarly $state_j=EC$ or $EF$. Now, four cases are possible:

- **Case1:** $state_i=state_j=EC$: This implies that both captain(i) and captain(j) are true. From lemma 1 we conclude that this is not feasible.
Case 2: state=state=EF: From lemma 1 we know that there exists only one captain. Therefore, captain= captain. On applying type function, we get type(captain)=type(captain). Now, from invariant (2) we can write type(i)=type(j). This contradicts the assumption.

Case 3: state=EC and state=EF
From state=EC, we observe that captain(i) true.
Now, from lemma 1, only one captain exists in the system. Therefore, captain=i
On applying type function, we get type(captain)=type(i)
Now, from (2) we get type(j)=type(i), which contradicts the assumption.

Case 4: state=EF and state=EC: The proof is similar to case 3.
Therefore, it is proved that if two processes are executing in their CS then both the CS are of the same type.

Starvation Freedom: To show that a request will eventually be serviced following three conditions must be satisfied.
A request will eventually reach the site holding the valid token.
A request that reach the token holding site will be issued a ‘start’ or ‘token’ message to enter CS as follower and will be added in token.queue.
A request that is added in token.queue will eventually be served.

Lemma 2: ∀i, j : i ∈ RS_i or j ∈ RS_j
Proof: Initially RS_i=Φ and for all i= 2 to n, RS_i contains all other sites except itself. Hence, for any two sites i and j, the condition is satisfied. Now, the request set of a site i changes in following conditions:
(i) Site i receives valid token: RS_i is emptied.
(ii) Site i’s request or message gen_token reaches site i and site i is not holding token: j is added in RS_j if j is not in RS_i.
(iii) Site i transfers token to new captain j: In this case Site i adds node j and other possible token holders in RS_j, whose requests are in token.queue.

The entries from a request set are deleted only when condition (i) holds. However, at this time the site i will be in the request set of all other sites. Therefore, in all above three cases, the request sets changes in such a manner, that the condition i ∈ RS_i or j ∈ RS_j remains true for any two nodes i and j.

Lemma 3: A request will reach the valid token holding site, if ∀i, j : i ∈ RS_i or j ∈ RS_j
Proof: Suppose when site i makes a request, the valid token is held at site j. Now, RS_i=Φ because site j is holding the valid token; therefore, site j should be an element of RS_j. Hence, site i’s request will reach at site j. When node i’s request reach site j, there are two possibilities: (i) j still holds the token and (ii) j has transferred the token to next captain, say k. In case (i) the request of site i will reach the site holding valid token. In case (ii) if k is in RS_i, site i’s request will reach site k. However, if site i is in RS_i, site k must have sent a request message to site i. Consequently, site i will add site k in RS_i, if site k is not in RS_i. Furthermore, site i will send a request message to k. Subsequently, site i’s request will reach the valid token holding site k.

From lemma 2 and lemma 3 the first part of the starvation freedom is proved.

Now, we prove the second part of the starvation freedom. When a request (j, SN, X) reaches the token holding site i, which may be in any one of the three states HI, EC or HS.
Case 1: P_i is in state HI: when P_i receives request from site j it will send a token message to j.
Case 2: P_i is in state EC: P_i will issue a ‘start’ message to j if token.type=X; otherwise, the request is added in token.queue.
Case 3: P_i is in state HS: If token.type=X and token.queue=Φ, a ‘start’ message is issued to j by i; otherwise, the request is added in token.queue.

Now, we prove part three of the starvation freedom. In our algorithm token.queue is an FCFS queue and when a session terminates the token is transferred to the process at the front of the token.queue. Therefore, a request that is added in token.queue will eventually be served. Thus, part three of the starvation freedom will always be satisfied.

Hence, the algorithm ensures starvation freedom.

Concurrent Occupancy and Smooth Admission: In our algorithm, when a process starts execution in its CS as a captain, it sends ‘start’ message to the processes whose requests are stored in token.queue and requesting the same resource. When a captain process executes in its CS receives a request of the same type it issues a ‘start’ message to allow the requesting process to enter in its CS as follower and hence the algorithm satisfies ‘smooth admission’ property. However, if captain is in state HS, it sends a ‘start’ message in response of a request of the same type only if token.queue=Φ (no conflicting pending requests). A requesting process immediately enters in its CS as follower upon receiving a start message. Further if the token holding process is in state HI, it will transfer the token as soon as it receives a request message. This implies that, the algorithm satisfies concurrent occupancy property.

VI. PERFORMANCE OF THE ALGORITHM
In this section we discuss the performance of our algorithm based upon following parameters: message complexity per CS request, average message size, forum switch complexity, maximum concurrency, synchronization delay. First, we analyze the performance of our algorithm in fault free scenario.

In the worst case n+1 messages needs to be exchanged (n-1 ‘request’, one ‘start’ and one ‘complete’ message) per CS entry. However in the best case no message needs to be exchanged. Among the messages used in the algorithm, only
the token has the size $O(n)$. Therefore, in the best case (all processes requesting for the same session), the average message size will be $O(1)$, because one token, $n-1$ ‘start’, $n-1$ ‘complete’ and some ‘request’ messages (depending upon the cardinality of the request sets at each site), will be exchanged. However, in the worst case (all processes requesting for a different session); $n$ token messages will be exchanged, besides the ‘request’ messages. Therefore, in this case the average message size will be $O(n)$.

In our algorithm all $n$ processes could be executing in their CS concurrently, if the system does not have any conflicting request pending. Hence, maximum concurrency of our algorithm is $n$. The requests for the same session are grouped together and treated as one entry in the token.queue. Therefore, at any point of time there can be at most $\min(n,n,m)$ entries in token.queue. Therefore, the forum switch complexity of the algorithm is $\min(n,n,m)$.

The synchronization delay of a distributed algorithm generally considered when the system is heavily loaded. Under heavy load conditions, there will always be some request pending, in token.queue. Hence, immediately, after captain comes out of its CS and no follower is in its CS, the request pending, in token.queue. Therefore, at any point of time there can be at most $\min(n,n,m)$ entries in token.queue. Therefore, the forum switch complexity of the algorithm is $\min(n,n,m)$.

Performance in case of message loss: If a token regeneration process has been initiated by site $i$, it sends a ‘gen_token’ message to all sites including itself. If site $j$ satisfies the condition $\text{session} \cap \text{old_token}_{i} \neq \emptyset$, it would send a ‘complete’ message to the captain that, in turn, terminates the session passing the token to next captain. Therefore, the synchronization delay would be $2\tau$.

If a token regeneration process has been initiated by site $i$, it sends a ‘gen_token’ message to all sites including itself. If site $j$ satisfies the condition $\text{session} \cap \text{old_token}_{i} \neq \emptyset$, it would send a ‘complete’ message to the captain that, in turn, terminates the session passing the token to the next captain. Therefore, the synchronization delay would be $2\tau$.

In case of message loss, if a token regeneration process has been initiated by site $i$, it sends a ‘gen_token’ message to all sites including itself. If site $j$ satisfies the condition $\text{session} \cap \text{old_token}_{i} \neq \emptyset$, it would send a ‘complete’ message to the captain that, in turn, terminates the session passing the token to the next captain. Therefore, the synchronization delay would be $2\tau$.

VIII. CONCLUSION

The proposed scheme satisfies the strongest fairness requirement, i.e. FCFS, in addition to the properties like safety and concurrent occupancy. The algorithm satisfies another desirable property called smooth admission. The maximum concurrency of the algorithm is $n$ and the forum switch complexity is $\min(n,n,m)$. Due to its fault tolerant feature, the scheme is of practical significance rather than being only of theoretical interest. More importantly, the scheme can be applied to another, more complex problem, that is, the extended GME problem. The concept of dynamic request sets has appeared earlier in the literature, nevertheless, its application to handle GME and extended GME problem, is the novelty of the present work. Due to the use of dynamic request sets the algorithm performs better than the algorithms using static request sets when the system is lightly loaded. The comparative performance analysis of the proposed algorithm with other existing schemes is being postponed for the full paper.

APPENDIX

A. The Pseudo Code of the Algorithm DRS_GME

Initialization:

For $i = 1$ to $n$

- $\text{state} = N$; $\text{captain} = \text{NULL}$
- $\text{RS}_i = \{\text{ids of all processes except } P_i\}$
- $\text{follower} = \emptyset$; $\text{TGI} = \text{false}$
- $\text{old_token}_{i,\text{session}} = \emptyset$
- $\text{old_token}_{i,\text{queue}} = \emptyset$
- $\text{old_token}_{i,\text{type}} = \text{NULL}$
- $\text{old_token}_{i,\text{followers}} = 0$

For $j = 1$ to $n$

- $\text{SV}_{j} = 0$
- $\text{state} = \text{HI}$; $\text{RS}_j = \emptyset$
- $\text{token}_{i,\text{type}} = \text{NULL}$; $\text{token}_{i,\text{queue}} = \emptyset$
- $\text{token}_{i,\text{followers}} = 0$; $\text{token}_{i,\text{session}} = 0$

VII. EXTENSION OF THE ALGORITHM TO SOLVE EXTENDED GME PROBLEM

Manabe and Park [10] suggested a modification of the GME problem and named it the Extended GME problem, in which a process is allowed to specify more than one resource type, while making a request. The request made by a process is serviced if the process can be allowed to join any one of the requested sessions. The Extended GME problem removes the possibility of unnecessary blocking.

The proposed algorithm can be modified to solve the Extended GME problem. The ‘request’ message is modified, and a process $P_i$ specifies a set of resource types $X$ in its ‘request’ message instead of specifying only one type. The process sends such ‘request’ message to all processes whose id is in its request set. When ‘request’ message reaches at the token possessing process $P_i$, $P_i$ checks whether the current session $X$ is in $X$, and token.queue is empty. If so, $P_i$ sends start $(j,X)$ message to $P_j$. Otherwise, $P_i$ creates multiple entries of $P_j$ in token.queue, one for each member of $X$. When a process receives token, it deletes all entries of $P_i$ in the token.queue. Similarly, when a process $P_i$ sends start $(i,X)$ message to process $P_j$, $P_i$ deletes all entries related to process $P_j$ from token.queue.

The synchronization delay of a distributed algorithm generally considered when the system is heavily loaded. Under heavy load conditions, there will always be some request pending, in token.queue. Hence, immediately, after captain comes out of its CS and no follower is in its CS, the request pending, in token.queue. Therefore, at any point of time there can be at most $\min(n,n,m)$ entries in token.queue. Therefore, the forum switch complexity of the algorithm is $\min(n,n,m)$.

In case of message loss, if a token regeneration process has been initiated by site $i$, it sends a ‘gen_token’ message to all sites including itself. If site $j$ satisfies the condition $\text{session} \cap \text{old_token}_{i} \neq \emptyset$, it would send a ‘complete’ message to the captain that, in turn, terminates the session passing the token to the next captain. Therefore, the synchronization delay would be $2\tau$.

The synchronization delay of a distributed algorithm generally considered when the system is heavily loaded. Under heavy load conditions, there will always be some request pending, in token.queue. Hence, immediately, after captain comes out of its CS and no follower is in its CS, the request pending, in token.queue. Therefore, at any point of time there can be at most $\min(n,n,m)$ entries in token.queue. Therefore, the forum switch complexity of the algorithm is $\min(n,n,m)$.

In case of message loss, if a token regeneration process has been initiated by site $i$, it sends a ‘gen_token’ message to all sites including itself. If site $j$ satisfies the condition $\text{session} \cap \text{old_token}_{i} \neq \emptyset$, it would send a ‘complete’ message to the captain that, in turn, terminates the session passing the token to the next captain. Therefore, the synchronization delay would be $2\tau$.

The synchronization delay of a distributed algorithm generally considered when the system is heavily loaded. Under heavy load conditions, there will always be some request pending, in token.queue. Hence, immediately, after captain comes out of its CS and no follower is in its CS, the request pending, in token.queue. Therefore, at any point of time there can be at most $\min(n,n,m)$ entries in token.queue. Therefore, the forum switch complexity of the algorithm is $\min(n,n,m)$.

In case of message loss, if a token regeneration process has been initiated by site $i$, it sends a ‘gen_token’ message to all sites including itself. If site $j$ satisfies the condition $\text{session} \cap \text{old_token}_{i} \neq \emptyset$, it would send a ‘complete’ message to the captain that, in turn, terminates the session passing the token to the next captain. Therefore, the synchronization delay would be $2\tau$. 
Event 1: \( P_i \) request for a forum \( X \)

\[ ++SN_i[i] \]

**Switch(state):**

**Case HI:**

- \( token.type=; state_i=EC \)
- \( ++token.session \)
- \( token.followers=0; \) Enter CS

**Case HS:**

- \( (token.queue=\emptyset) \) \&\& \( (token.type=X) \)
  - \( state_i=EC; \)
  - Enter CS

Else:

- Add request \((i,SN_i[i],X)\) to token.queue
- Start timer \( T1 \)

Default:

- \( state_i=R; \) Start timer \( T1 \)
- Send request \((i,SN_i[i],X)\) to all members of \( RS_i \)

Event 2: \( P_j \) receives request \((j,SN_j,X)\)

- If \( SN>SN[j]\) /* otherwise old request
  - \( SN[j]=SN \)
- **Switch(state):**

  **Case Ri:**

  - \( (j \not\in RS_i) \)
  - Add \( j \) to \( RS_i \)
  - Send request \((j,SN[j],X)\) to \( P_j \)

  **Case EC:**

  - \( ++\) \(token.followers\)
  - \( reset.timer T2 \)
  - Add \( j \) to \( follower_i \)
  - Send start \((i)\) to \( P_j \)

  **Else:**

  - Add request \((j,SN_j,X)\) to token.queue

**Case HI:**

- Add \( j \) to \( RS_i \)
- Add request \((j,SN_j,X)\) to token.queue
- \( old_token_i=; token.type=; state_i=EC; \)
- Send token \((token.queue,token.type,token.followers,token.session)\) to \( P_j \)

**Case HS:**

- \( (token.type=X) \) \&\& \( (token.queue=\emptyset) \)
  - \( ++token.followers; \)
  - \( Reset.timer T2 \)
  - Add \( j \) to \( follower_i \)
  - Send start \((i)\) to \( P_j \)
  - Else Add request \((j,SN_j,X)\) to token.queue

Default:

- If \( (j \not\in RS_i) \) Add \( j \) to \( RS_i \)

Event 3: \( P_j \) receives start \((j)\)

- If \( TG_i=; TG_i=\) /* otherwise invalid
  - \( TG_i=; \) reset timer \( T1 \)
- Close timer \( T1 \)
  - \( captain_i=j; state_i=EF; \) Enter CS

Event 4: \( P_j \) exits from CS:

- If \( state_i=EF \)
  - Send complete \((i)\) to \( captain_i \)
  - \( captain_i=NULL; \)
  - \( state_i=N \)

Else:

- If \( (token.followers=0) \) \&\& \( (token.queue=\emptyset) \)
  - Close timer \( T2 \)
  - \( state_i=HI; \)
  - \( token.type=NULL; \)
  - \( old_token_i=token \)
  - \( (token.followers=0) \) \&\& \( (token.queue=\emptyset) \)

Close timer \( T2 \)

- \( state_i=N; \)
- \( old_token_i=token \)
- \( RS_i=\{id’s \ of \ all \ processes \ which \ are \ in \ token.queue \ and \ will \ work \ as \ captain \ in \ future\} \)
- \( P_j \) at the front of \( token.queue \) is selected as captain
- Send token \((token.queue,token.type,token.followers,token.session)\) to \( P_j \)

Event 5: \( P_j \) receives complete \((j)\)

- If \( (j \ in \ follower_i) \)
  - \( --\) \( token.followers \)
  - Remove \( j \) from \( follower_i \)
  - If \( follower_i=\) close timer \( T2 \)

Else:

- If \( (i’s \ request \ in \ token.queue) \) \( state_i=R \)
  - \( else \) \( state_i=N; \)
  - \( RS_i=\{id’s \ of \ all \ processes \ which \ are \ in \ token.queue \ and \ will \ work \ as \ captain \ in \ future\} \)
  - \( P_j \) at front of \( token.queue \) is selected as captain
  - \( old_token_i=token \)
  - Send token \((token.queue,token.type,token.followers,token.session)\) to \( P_j \)

Event 6: \( P_j \) receives token

- If \( (old_token_i.session<token.session) \) /* otherwise invalid
  - \( (TG_i=) \) \( TG_i=\) false
  - Close timer \( T1 \)
  - delete \((token.queue) \) /* delete \( P_j \) and its followers
  - \( token.type=X \) /* \( X \) is the type of deleted entry
  - \( token.followers=\) number of followers of \( P_j \)
  - Add all followers of \( P_j \) to \( follower_i \)
  - Send start \((j)\) to \( followers_i \)
  - \( state_i=EC; \) enter CS; \( RS_i=\emptyset \)

Event 7: Timer \( T1 \) at \( P_j \) exceeds the value \( T_{req} \)

- Reset timer \( T1 \)
  - \( TG_i=\) true
  - Send \((gen_token,(j,SN[j],old_token,session))\) to all sites

Event 8: \( P_j \) receives \((j,SN,session)\)

- If \( (SN>SN[j]) \)
  - \( SN[j]=SN \)
  - **If** \( (state_i=N/R) \)
    - **If** \( (j \not\in RS_i) \) Add \( j \) to \( RS_i \)
    - **If** \( state_i=R \) Send \( i’s \ request \) message to \( P_j \)
    - **If** \( (session<old_token,session) \)
      - \( P_j \) process at the front of \( old_token,queue \)
      - Send token \((token.queue,token.type,token.followers,token.session)\) to \( P_i \)
  - Else **If** \( state_i=EC \)
    - **If** \( (token.type=X) \)
      - Reset timer \( T2 \)
      - Send start \((i)\) to \( P_j \)
    - **If** \( (j \ not \ in \ follower_i) \)
      - \( ++\) \( token.followers \)
      - Add \( j \) to \( follower_i \)

Else Add request \((j,SN,X)\) to token.queue
Else If\((\text{state}=\text{H})\)
 Add \(j\) to \(RS\)
 Add request \((j,SN,X)\) to old_token.queue
 old_token\(_i\),=token
 Send token \((\text{token.queue, token.type, token.followers, token.session})\) to \(P_j\)
Else If\((\text{state}=\text{HS})\)
 If\((\text{token.type}=\text{X})\) && \((\text{token.queue}=\emptyset)\)
 Reset timer \(T_2\);
 Send start\((i)\) to \(P_j\)
 If\((j\) not in follower\)
 Add \(j\) to follower,
 ++token.followers
 Else Add request \((j,SN,X)\) to token.queue

Event 9: Timer \(T_2\) exceeds value \(2^\ast t_a+t_t\) at \(P_i\)
Send is_complete\((i,X)\) to all processes in follower,

Event 10: \(P_i\) receives is_complete\((i,X)\)
If \((P_i\) is requesting for session \(X\)
 If \((TGI_i=\text{True})\text{ and }TGI_j=\text{False}\)
 Close timer \(T_1\); \(\text{captain}=j\)
 is_complete\((i,EF)\) to all processes in \(P_i\)

REFERENCES