An Improved Quorum-Based Algorithm for Extended GME Problem in Distributed Systems

ABHISHEK SWAROOP* AND AWADHESH KUMAR SINGH+

*Department of Computer Science and Engineering
G.P.M. College of Engineering
Delhi, 110036 India
E-mail: asa1971@gmail.com

+Department of Computer Engineering
National Institute of Technology
Kurukshetra, Haryana, 136119 India

The extended GME (group mutual exclusion) problem is a natural extension of the GME problem. In extended GME problem, processes are allowed to request more than one resource at a time, in order that the processes that can proceed by having access to any one of the requested resource can be allowed to do so. Manabe-Park suggested a quorum based solution for the extended GME problem. However, the worst case message complexity of the Manabe-Park algorithm is $9q$, where $q$ is the quorum size. Further, the synchronization delay of Manabe-Park algorithm is $4T$, where $T$ is the maximum message propagation delay. In the present paper, we propose a quorum based solution for the extended GME problem. The worst case message complexity of our algorithm is $7q$ and synchronization delay is $3T$. Moreover, in the best case, the synchronization delay and message complexity come down to $2T$ and $3q$ respectively.

Keywords: concurrency, quorum, distributed systems, unnecessary blocking, group mutual exclusion

1. INTRODUCTION

1.1 Preliminaries

The mutual exclusion (ME) problem is a widely studied problem of distributed systems. The group mutual exclusion (GME) problem was first proposed by Joung [1] as an extension of the classical ME problem. Joung [1] modeled the GME problem as Congenial Talking Philosophers (CTP) problem. Later on, Joung [2] presented a message passing based solution for GME problem. In group mutual exclusion, a process requests a resource type (also termed, group) before entering its critical section (CS). The processes requesting the same group are allowed to be in their CS simultaneously. However, the processes requesting different groups, must execute their CS in mutually exclusive way. An example of the group mutual exclusion problem is a situation in which a CD jukebox storing large amount of data, is being shared by several processes. The processes are interested in reading some data stored on the CD’s. However, due to the limitation of the buffer space, only one CD can be loaded in the buffer at a time. Therefore, only the processes, interested in the data stored in the currently loaded CD, may access the required...
data concurrently. The concept of GME can be applied to a variety of areas, e.g., in controlling the access to a database [3], improving the quality of internet servers [4], digital media files [5], implementing concurrent data structures [6], in wireless applications [7], and in mobile ad hoc networks [8].

Joung [9] generalized the GME problem and allowed a process to become the member of multiple groups; however, the process must select a unique group at the time of making request to enter CS. Manabe-Park [10] further modified the GME problem and allowed a process to make request for a subset of groups instead of a unique group. Manabe-Park [10] named the problem as extended GME problem.

If we consider the CD jukebox example, the data required by process \( P_i \) may be stored in CD A as well as CD E. Therefore, the request of process \( P_i \) may be satisfied, if either CD A or CD E is currently loaded in the buffer. Another example of extended GME can be a situation where a multimedia file containing a video stream and audio streams in many different languages e.g., English, French, German, and Russian is available for transmission. Out of the available audio streams, only one audio stream can be selected for transmission at a time. If we apply group mutual exclusion in this situation, only the users interested in current language will be able to receive the transmission. However, there may be users who may be interested in more than one language, e.g., English and French. The extended GME allows such users to request for both the languages and to receive the transmission, if any one of these languages is currently selected.

1.2 Definition of Extended GME Problem

Let \( P = \{P_1, P_2, \ldots, P_n\} \) is the set of processes and \( G = \{g_1, g_2, \ldots, g_m\} \) the set of groups in the system. A process \( P_i \) selects a group set \( GS(i) \subseteq G \), while making a request for CS. \( P_i \) may enter its CS, if any group ‘\( g \)’ from its group set \( GS(i) \) is currently open or a session using ‘\( g \)’ can be initiated at this time.

Unnecessary blocking: The existing GME algorithms can not be directly applied to the extended GME problem because these algorithms force a requesting process to select a unique group while making a request. The requesting process can enter in its CS only if the requested group is the current group or no group is currently in use. This will lead to unnecessary blocking. Assume that a process \( P_i \) is interested either in group \( g_i \) or in group \( g_j \). If a GME algorithm is being used, \( P_i \) has to select one of these groups while making a request. Suppose \( P_i \) selects \( g_i \); however, the currently open group is \( g_j \). In this case, \( P_i \) is not allowed to enter CS and unnecessarily blocked to enter CS.

Unnecessary blocking freedom: Manabe-Park [10] gave the following definition of unnecessary blocking freedom “When a pivot process \( P_i \) is entering CS using \( g_i \), any process \( P_j \), such that \( g_i \in GS(P_j) \) can enter at the same time.” However, this definition does not allow a process \( P_j (g_j \in GS(P_j)) \) to enter CS using current group \( g_i \), if the pivot process \( P_i \) has already entered in CS using \( g_i \). In this situation, \( P_j \) is unnecessary blocked. However, if we always allow a process \( P_j \) fulfilling the condition \( g_i \in GS(P_j) \) to enter CS (where \( g_i \) is the group currently being used), starvation will occur. Therefore, we propose the following modified definition of the unnecessary blocking freedom:
If $P_i$ is the pivot process for the currently open session (current group is $g$) then a process $P_j$ such that $g \in GS(P_j)$ can enter in its CS, if any of the following condition is satisfied: (1) $P_i$ is aware of the request of $P_j$ at the time of initiating the session (2) $P_i$ is still inside CS, when it becomes aware about $P_j$’s request.

1.3 Related Work

The first solution of the group mutual exclusion problem was proposed by Joung [1] for shared memory systems. Later on, few more solutions were also proposed e.g., [6, 11, 12]. However, the solutions of the GME problem, for message passing systems, fall in either of two categories, token-based and permission based. The token-based algorithms for ring networks are given in [13, 14]. Beauquier et al. [15] proposed a token-based algorithm for the GME problem for tree based networks. However, the algorithms for the ring and tree networks suffer from the problem of high synchronization delay and high waiting time. The token-based algorithms for completely connected networks have been proposed in [16-18]. The permission-based algorithms are given in [2, 9, 19, 20].

All the above mentioned algorithms are for the GME problem and suffer from unnecessary blocking [10] when applied to solve the extended GME problem. Unnecessary blocking is a situation in which two processes are prevented from entering CS simultaneously even when they are capable of doing so. Manabe-Park [10] presented a quorum based algorithm for solving extended GME problem. They have assumed two types of processes, application processes (requesting resources) and manager processes (granting permissions). The algorithm uses a two phase locking scheme, similar to [21]. The best case message complexity of the algorithm is $3q$; however, the worst case message complexity is $9q$. The synchronization delay of the algorithm is $4T$.

1.4 Our Contribution

In the present paper, we propose a quorum based algorithm for solving the extended GME problem. Any ordinary quorum systems that can be used to solve the classical mutual exclusion problem can be used in our algorithm. The proposed algorithm does not suffer from unnecessary blocking. The algorithm uses a leader-follower approach introduced by Joung in [2]. In the leader-follower approach, the process which first enters in CS using a group initiates a session and becomes the leader (also termed as captain). The captain allows other processes to enter CS as follower provided that these are requesting the same group. In our algorithm, the follower processes reports the captain upon exiting from CS. The worst case message complexity of our algorithm is $7q$; moreover, the best case message complexity is $3q$. The heavy load synchronization delay of the algorithm is $2T$, if the last process to come out of its CS is a leader. Otherwise, the synchronization delay is $3T$. Further, our algorithm allows a process to select its group set from all the available groups in the system. Moreover, while initiating a session, the captain instead of arbitrarily selecting a group from its group set, selects a group from its group set for which the maximum number of processes are requesting according to its information.

The rest of the paper is organized as follows. The system model is given in section 2. The description of the algorithm is provided in section 3. The correctness and performance analysis are presented in sections 4 and 5 respectively. We present our experimental results in section 6. Finally, section 7 concludes the paper.
2. SYSTEM MODEL

We assume a message passing asynchronous distributed system. The system has \( n \) sites, numbered as 1, 2, 3, \ldots, \( n \). We assume a fully connected network. The maximum message propagation delay and the time for which a process can remain in CS, are bounded but unpredictable. We also assume reliable FIFO (First In First Out) channels. Once a process has made a request to enter CS, it will not make another request till the old request is satisfied.

Each process maintains following variables:

- \( \text{state} \): Stores the current state of the process. The possible states are:
  - N: not requesting,
  - R: requesting,
  - CSC: executing in CS as captain,
  - CSF: executing in CS as follower,
  - NCSC: captain but not in CS

- \( \text{mystatus} \): Stores the status of the process \( P_i \) as a quorum member.

- \( \text{pivot} \): id of the captain of \( P_i \).

- \( \text{mygroup} \): The group which is currently being accessed, according to the information of \( P_i \).

- \( G_i \): Group set of \( P_i \).

- \( \text{selected} \): The quorum selected by \( P_i \).

- \( \text{requesteq} \): The request queue of process \( P_i \).

- \( \text{seq_no} \): It stores the latest sequence no. of requests of other processes known to \( P_i \).

- \( \text{enter_arr} \): The set of processes, which are follower of \( P_i \).

- \( \text{req-info} \): It is an \( m \times n \) array which stores the request information gathered by a process, which it piggybacks on the OK message. \( m \) is the total number of groups and \( n \) is the number of processes in the system.

- \( \text{cum_req} \): It is also an \( m \times n \) array used by \( P_i \) to accumulate the request information received with OK messages.

- \( \text{permitted} \): It stores the id of the process to which \( P_i \) has sent permission.

- \( \text{grant} \): It stores the id’s of the processes from which \( P_i \) has received permission.

The messages used during the execution of the algorithm are as following:

- \( \text{REQUEST} \): A requesting process sends this message to all the processes in its selected quorum.

- \( \text{OK} \): Sent by a quorum member as permission.

- \( \text{LOCK} \): The captain process sends Lock message to all the processes in its selected quorum informing about session initiation.

- \( \text{ENTER} \): The captain process invites the follower process to join the current session by sending ENTER message.

- \( \text{COMPLETE} \): The follower process sends COMPLETE message to captain on exiting from CS.

- \( \text{RELEASE} \): The captain sends RELEASE message to all members in its selected quorum, when the current session terminates.

- \( \text{NONEED} \): The follower process sends NONEED message to the members of its selected quorum after exiting from CS.

- \( \text{CANCEL} \): The quorum member tries to preempt the permission for a higher priority request.

- \( \text{CANCELLED} \): The requesting process allows the quorum member to grant permission to another process.

Fig. 1. Data structure and messages used.
3. WORKING OF THE ALGORITHM

The data structures used and the messages exchanged by the processes during the execution of the algorithm and the pseudo code have been presented in Figs. 1 and 2 respectively. The high level description of the algorithm is presented in this section. The proposed algorithm uses logical clocks [22] to decide priorities among competing requests.

A process \( P_i \) wishing to enter CS, selects an arbitrary quorum, sends a request to the processes in its selected quorum, and waits for their permission. When a process \( P_j \) receives a request from \( P_i \), \( P_j \) appends the request in its request queue according to the priority. If \( P_i \) is executing in CS as captain, it invites the requesting process to join the session provided that the group currently being used is in the group set of \( P_j \). However, if \( P_i \) is not executing in CS as captain, it takes action according to its status. If \( P_i \) has not granted permission to any process till now, it grants permission along with the information about other requests to \( P_j \). If \( P_i \) has already granted permission to some process (say \( P_k \)) and yet received a confirmation from \( P_k \), it compares the priorities of \( P_j \) and \( P_k \). If the priority of \( P_j \) is higher, \( P_i \) tries to preempt the granted permission. Upon receiving the permission back from \( P_k \), \( P_i \) grants permission to the process at the front of its request queue. Incase, \( P_i \) has been locked by the current captain and the group currently being used is in the group set of \( P_j \), \( P_i \) forwards request of \( P_j \) to the captain.

Upon receiving permission from all members of the quorum, a process \( P_i \) locks all those processes, selects a group from its group set for which the maximum requests are pending, invites the processes requesting the selected group, adds these in list of followers, and enters CS as captain. When the captain comes out of CS, it terminates the session by informing the processes in its selected quorum, only if no follower is in CS at this time. However, if some followers of \( P_i \) are still in CS, it waits for their exit. The captain stops inviting other processes after it has come out of CS. Upon receiving information about the session termination, it grants the permission to the process at the front of its request queue and piggy backs pending requests along with permission.

When a requesting process \( P_i \) receives an invitation from the captain to join the session, it enters in CS as follower. Upon exiting from CS, \( P_i \) informs about its successful completion of CS to its captain and to all the processes in its selected quorum. When a quorum member who has granted permission to \( P_i \), receives information about the successful execution of \( P_i \)'s request, it removes \( P_i \)'s request from its request queue and grants permission to the process at the front of the request queue provided that the request queue is not empty at this time.

4. CORRECTNESS PROOF

The algorithm satisfies various safety and progress requirements. The proof of the same is presented below.

4.1 Safety

**Lemma 1**  No two processes may execute inside CS, as captain, at any point of time.
Initialization:
gran, = Ø; pivot, = NULL; mygroup, = NULL
SNi, = 0; statei, = Ni; req_infoi, = Ø
Set all entries of cum_req, = 0
For j = 1 to n seq_no[j] = 0
requestq, = Ø; selectedq, = Ø
perm, = NULL; mystatus, = available
P, request CS, group set is G:
Select a quorum selectedq, arbitrarily
SNj, = SN, + 1; statei, = R; seq_no[j] = SN,
Send REQUEST (P, i,j, SNj) to every process in
selectedq,
P, receives REQUEST (j, Gj, SNj):
SN, = max (SN, SNj)
Add request in requestq, according to its priority
Update req_info,
If (SN, > seq_no[j])
seq_no[j] = SN,
If (state, = CSC) && (mygroup, ∈ Gj)
enter_arr, = enter_arr, ∪ j
Send ENTER (P, i, mygroup, SNj) to Pj
Else
If (mystatus, = available)
mystatus, = wait for lock; permitted, = j
Send OK (P, i, SNj, req_info, ) to Pj
Elseif (mystatus, = wait for lock)
If Pj, is at the front of the requestq,
Send CANCEL (Pj, SNj) to permitted,
mystatus, = wait for cancel
Else (mystatus, = locked)
If (mygroup, ∈ Gj)
Forward REQUEST (j, Gj, SNj) to pivot,
P, receives OK (j, SNj, req_info,):
SN, = max (SN, SNj)
If (state, = R)
gran, = gran, ∪ j; Update cum_req,
If (gran, = selectedq,)
pivot, = i
Select a group gi, from Gj, for which the max. no.
of requests are there in cum_req,
Add processes requesting gi, to enter_arr,
mygroup, = gi; state, = CSC
Send LOCK (i, mygroup, SNj) to all processes
in selectedq,
Set all entries of cum_req, to 0
Send ENTER (i, mygroup, SNj) to all processes
in enter_arr,
P, enters in its CS as captain
..... Executing CS. . . . . . . . . . .
P, exits from CS
If (enter_arr, = Ø)
state, = N; Send RELEASE (i, SNj) to all
processes in selectedq,
P, receives LOCK (j, g, SNj):
SNj, = max (SN, SNj); mygroup, = g; pivot, = j
Remove request of Pj, from requestq,
Reset corresponding entry in req_info,
Forward request of all Pj, in requestq, satisfying the
condition (g ∈ Gi) to pivot,
mystatus, = locked
P, receives ENTER (j, g, SNj):
SNj, = max (SN, SNj)
If (state, = R) && (g ∈ Gi)
state, = CSF; pivot, = j; mygroup, = g
P, enter CS as follower
…..executing in CS…..
Exits from CS
Send NONEED (i, SNj) to all processes in selectedq,
Send COMPLETE (i, SNj) to pivot,
pivot, = NULL; mygroup, = NULL; state, = N
Else
Send COMPLETE (i, SNj) to Pj,
P, Receives COMPLETE (j, SNj):
SN, = max (SN, SNj)
If (requestq, = Ø)
If (state, = NCSC)
state, = N; Send RELEASE (i, SNj) to all processes
in selectedq
Else
Let Pj, be at the front of the requestq,
Send OK (i, SNj) to Pj; permitted, = r
mystatus, = wait for lock
P, receives CANCEL (j, SNj):
SN, = max (SN, SNj)
If (state, = CSC) && (state, != NCSC)
gran, = gran, − j
Send CANCELLED (i, SNj) to Pj,
P, Receives CANCELLED (j, SNj):
SN, = max (SN, SNj)
If (requestq, ≠ Ø) && (mystatus, != locked)
Let Pj, be the process at the front of the requestq,
Send OK (i, SNj, req_info, ) to Pj,
mystatus, = wait for lock; permitted, = Pj,
P, Receives NONEED (j, SNj):
SNj, = max (SNj, SNj); requestq, = requestq, − j
Reset corresponding entry in req_info,
If (requestq, ≠ Ø) && (mystatus, != locked)
Let Pj, be at the front of the requestq,
Send OK (i, SNj, req_info, ) to Pj,
mystatus, = wait for lock; permitted, = r

Fig. 2. The pseudo code of the algorithm.
**Proof:** We assume the contrary that $P_1$ and $P_2$ are inside their CS as captain simultaneously. Let $Q_1$ and $Q_2$ be the quorum selected by $P_1$ and $P_2$ respectively. In order to be inside their CS as captain, a process must acquire permission from all the processes in its selected quorum. Now, a process can grant permission to only one process at a time. Therefore, $P_1$ and $P_2$ can be inside their CS simultaneously, only if $Q_1 \cap Q_2 = \emptyset$. However, this violates the necessary condition that any two quorums must intersect. Hence our initial assumption is wrong and Lemma 1 holds.

**Theorem 1** $P_1$ and $P_2$ can never be inside their CS simultaneously, if they are not using the same group.

**Proof:** In order to prove the theorem, we assume the contrary. Suppose $P_1$ and $P_2$ are in their CS simultaneously, the groups being used by them are $mygroup_1$ and $mygroup_2$ respectively. Further, $mygroup_1$ is not the same as $mygroup_2$. Now, there are four possibilities:

1. Both $P_1$ and $P_2$ enter their CS as captain: From Lemma 1, it is clear that no two processes can be inside their CS as captain. Therefore, case 1 is not a feasible system state.
2. $P_1$ enters its CS as captain and $P_2$ as follower: A captain process sends the group selected by it along with the ENTER message. Upon receiving an ENTER message, a process can enter inside its CS only using the group selected by the captain. This implies that $mygroup_2 = mygroup_1$, which contradicts our initial assumption.
3. $P_2$ enters as captain and $P_1$ as follower: Case 3 is similar to case 2.
4. $P_1$ and $P_2$ enter their CS as follower: Let $P_3$ be the current captain and $g$ the group selected by $P_3$. As there can be only one captain (from Lemma 1) and it can select only one group, therefore, both $P_1$ and $P_2$ must have entered in their CS, receiving an ENTER message from $P_3$. This implies that $mygroup_1 = mygroup_2 = g$, which is a contradiction.

Our initial assumption is invalid for all the possible cases. Hence, Theorem 1 is proved.

**4.2 Starvation Freedom**

**Lemma 2** If no session is currently open, the highest priority process among the requesting processes will succeed in receiving permission from all the processes in its selected quorum.

**Proof:** Let $P_1$ is the process with the highest priority request and $Q$ the quorum selected by it. Any process in $Q$ will grant the permission to $P_1$, if the status of this process in $Q$ is available at the time of receiving the request of $P_1$. However, if any of the processes in $Q$ (say $P_j$) has already granted permission to some other process (say $P_k$) and waiting for lock then $P_j$ tries to preempt the permission by sending a CANCEL message to $P_k$. $P_j$ successfully preempts the permission given to $P_k$ and sends permission to $P_j$ provided that $P_j$ is not the current captain. Further, the captain will inform its quorum member about the termination of the session. Hence, $P_j$ will grant permission to $P_j$ after knowing
about the termination of the current session, provided that \( P_i \) is still the highest priority process in \( P_j \)'s request queue. Therefore, the highest priority request will eventually receive permission from all processes in its selected quorum. Hence, Lemma 2 holds.

**Lemma 3** An open session will terminate in finite time.

**Proof:** The captain stops inviting other processes to join the currently open session once it has come out of its CS. Therefore, after the captain has come out of its CS, only existing followers can be inside CS. These followers will also come out of CS in finite time and will inform the captain by sending a COMPLETE message. Upon receiving COMPLETE message from all of its followers, the captain announces the termination of the session. If \( T_c \) is the maximum time for which a process can remain in CS, the session will terminate in at most \( 2(T_c + T) \) time.

**Lemma 4** The request having the highest priority will eventually be served.

**Proof:** If no session is currently open, the process \( P_i \) having highest priority request will be able to receive permission from all processes of its selected quorum \( (Q_i) \) (from Lemma 2) and hence will be successful in entering CS as captain. However, if a session is already open and \( P_i \)'s request is compatible with it, \( P_i \) will enter CS as follower. Otherwise, \( P_i \) will have to wait for the termination of the session. The session will terminate in finite time (from Lemma 3) and the captain \( P_j \) will send RELEASE messages to all the processes in its selected quorum \( (Q_j) \). Now, process \( P_i \) must have received permission from all processes except processes in \( Q_i \cap Q_j \) and must be at the front of the request queue of all the processes in \( Q_i \cap Q_j \). Therefore, upon receiving a RELEASE message from \( P_j \), the processes in \( Q_i \cap Q_j \) will send permission to \( P_i \) and \( P_i \) will be successful in entering CS as captain.

**Lemma 5** The process having a pending request will either be invited by some other process to join the session or it will attain the highest priority in finite time.

**Proof:** Let \( P_i \) is requesting group 'g' and \( Q_i \) the quorum selected by \( P_i \). If some process \( P_j \) \((\neq P_i)\) executing in CS as captain using group g, knows about the request of \( P_i \), it will invite \( P_i \) to join the session as follower of \( P_j \). Therefore, the request of \( P_i \) will be served. The worst case occurs when \( P_i \) is added at the rear of the request queue of all the quorum members and \( P_i \) does not receive invitation from any captain to join a session. Each quorum member will grant permission to a process (say \( P_j \)) at the front of its request queue. However, the quorum members will send the value of sequence number of the request also with the permission. Therefore, when \( P_j \) will make another request, the value of its sequence number will be greater then the value of \( P_i \) and hence the priority of \( P_j \)'s new request will be less than the priority of \( P_i \)'s request. Thus, we can say that the request of \( P_i \) will attain the highest priority after at most \( n - 1 \) sessions and each session will terminate in finite time (from Lemma 3), therefore, \( P_i \)'s request will attain highest priority in finite time.

**Theorem 2** Every pending request for CS is eventually serviced.
Proof: Follows from Lemmas 4 and 5.

4.3 Freedom from Unnecessary Blocking

Theorem 3 The algorithm does not suffer from unnecessary blocking.

Proof: The captain process selects a group from its group set, for which the maximum number of processes are requesting according to its information. The captain invites all the processes requesting for the selected group. Further, upon receiving a LOCK message from the captain, a quorum member forwards all compatible requests to the captain. When a quorum member who has been locked by the current captain, receives some compatible requests (requesting the group currently being used), it forwards these requests to the captain. From the above discussion, it is clear that our algorithm successfully avoids unnecessary blocking.

5. PERFORMANCE ANALYSIS

We analyze the performance of our algorithm using following performance metrics: message complexity, waiting time, synchronization delay, message size, and concurrency.

5.1 Message Complexity

First we discuss the case in which requests arrive in order of their priorities. We assume that the size of quorum is \( q \). If there is only one requesting process then the messages exchanged for the session initiation, would be as follows: \( q \) number of REQUEST messages, \( q \) OK messages, \( q \) LOCK messages, and \( q \) RELEASE messages. Therefore, the message complexity of the algorithm would be \( 4q \). Now, suppose \( k \) processes enter in CS as follower along with the captain. The number of messages required for the captain will remain \( 4q \) in the absence of CANCEL and CANCELLED messages. Further, the number of messages required by the follower processes will be at most \( 3q + 1 \) (\( q \) REQUEST messages, at most \( q - 1 \) OK messages, one ENTER message, one COMPLETE message, and \( q \) NONEED messages). Thus, If we assume that in a session the captain has invited \( k \) followers, the message complexity per CS request turns out to be \( \frac{4q + k(3q + 1)}{k+1} \).

The worst case occurs when a high priority request arrives after the permission has been granted to some other requests and the session has not been initiated yet. In this case, at most \( 7q - 3 \) messages (\( q \) REQUEST messages, \( q - 1 \) OK messages to the previous request, \( q - 1 \) CANCEL messages, \( q - 1 \) CANCELLED messages, \( q \) OK messages to the new request, \( q \) LOCK, and \( q \) RELEASE messages) will be required.

5.2 Waiting Time and Synchronization Delay

The waiting time of a ME/GME algorithm is usually measured when the system is lightly loaded. The light load waiting time of our algorithm is \( 2T \) (REQUEST followed
by OK). The synchronization delay is more relevant, when the system is heavily loaded, i.e., when a session finishes, new requests are always available. If the captain is the last process to come out of CS upon termination of a session, the heavy load synchronization delay of our algorithm will be $2T$ (RELEASE followed by OK). Obviously, this is the best case. However, the synchronization delay is $3T$ (COMPLETE followed by RELEASE followed by OK) in the worst case.

5.3 Message Size

The messages used by our algorithm are REQUEST, OK, LOCK, RELEASE, ENTER, COMPLETE, CANCEL, CANCELLED, and NONEED. Except the REQUEST and OK, all other messages are of $O(1)$ size. The size of REQUEST message is $O(m)$. Also, the size of OK message is $O(m \times n)$, as the request array $req\_info$ is piggybacked with it. Nevertheless, each entry in the group set and $req\_info$ array requires only one bit. Hence, several entries can be packed inside a word. Therefore, the size of REQUEST message and OK message will be less than $m$ and $m \times n$ respectively.

6. SIMULATION RESULTS

We compare the performance of our algorithm with Manabe-Park [10] algorithm using following performance metrics: number of message/CS requests, number of bytes/CS requests, average waiting time, and average number of processes attending a session.

6.1 Simulation Set Up

We used a discrete event simulator to perform the simulation with ten processes and eight groups (resources). We assumed that a process remains inside its CS for 500 milliseconds. We used a fully connected network so that each process may directly send message to other processes. Each reading is recorded by taking average of 1000 requests. While making a request for CS every process selects three groups randomly (size of the group set = 3), out of the total available groups in the system.

6.2 Discussion on Simulation Results

As the simulation results shown in Fig. 3 suggests the average waiting time and the number of messages/CS in our algorithm are significantly lower than Manabe-Park algorithm [10] in all type of load conditions. However, the difference in the number of messages required/CS reduces in heavy load conditions. The size of OK message is $O(m \times n)$ in our algorithm because we piggy back pending requests along with OK message. This reduces the advantage gained in terms of number of messages when we compare the number of bytes transferred/CS. However, under light load and medium load conditions our algorithm outperforms Manabe-Park algorithm [10] as far as the number of bytes transferred/CS is concerned. The concurrency (number of processes attending a session) of both the algorithm is almost same. Nevertheless, our algorithm shows slightly better concurrency in medium load and heavy load conditions.
7. CONCLUSION

The paper presents a quorum based algorithm for extended GME problem. It is obvious from the static analysis and experimental results that the algorithm performs better than Manabe-Park [10] algorithm with reference to many widely used performance metrics, namely message complexity, waiting time, and concurrency.

REFERENCES

4. O. Thiare, M. Guerou, and M. Naimi, “Distributed group mutual exclusion based on client/server model,” in Proceedings of the 7th International Conference on Parallel
5. Advance System Specifications (ASF), “An extensible file format to store digital
6. P. Kean and M. Moir, “A simple local spin group mutual exclusion algorithm,” in
Proceedings of the 18th Annual ACM Symposium on Principles of Distributed
8. J. R. Jiang, “A group mutual exclusion algorithm for ad hoc mobile networks,” in
Proceedings of the 6th International Conference on Computer Science and
9. Y. J. Joung, “Quorum-based algorithms for group mutual exclusion,” IEEE Transac-
10. Y. Manabe and J. Park, “A quorum based extended group mutual exclusion algo-
rithm without unnecessary blocking,” in Proceedings of the 10th International Con-
539-543.
group mutual exclusion in token passing networks,” in Proceedings of ACM Symposi-
15. J. Beauquier, et al., “Group mutual exclusion in tree networks,” Journal of Informa-
exclusion in distributed systems,” in Proceedings of the 5th International Symposium
on Parallel and Distributed Computing, 2006, pp. 34-41.
17. N. Mittal and P. K. Mohan, “A priority based distributed group mutual exclusion
algorithm when group access is non-uniform,” Journal of Parallel and Distributed
18. A. Swaroop and A. K. Singh, “A token-based fair algorithm for group mutual exclu-
group mutual exclusion,” in Proceedings of the 4th International Conference on Pa-
rallel and Distributed Computing Application and Technologies, 2003, pp. 742-746.
20. R. Attreya and N. Mittal, “A dynamic group mutual exclusion algorithm using sur-
rrogate quorums,” in Proceedings of the 25th IEEE Conference on Distributed Com-
21. H. Kakaguwa and M. Yamashita, “Local coteries and a distributed resource alloca-
tion algorithm,” Transactions on Information Processing Society of Japan, Vol. 37,
22. L. Lamport, “Time, clocks, and ordering of events in a distributed system,” Commu-
Abhishek Swaroop received B.Tech. in Computer Science and Engineering from G.B. Pant University, Pantnagar, India in 1992 and M.Tech. in Information Technology in 2004 from Punjabi University Patiala, India. He is Assistant Professor in Computer Science and Engineering department in GPM College of Engineering Delhi, India. His research interests are distributed algorithms, fault tolerance, and group mutual exclusion.

Awadhesh Kumar Singh received B.E. degree in Computer Science and Engineering from Gorakhpur, India in 1988. He received M.E. and Ph.D. (Engg.) in the same area from Jadavpur University, Kolkata, India. He is Assistant Professor in Computer Engineering department, National Institute of Technology, Kurukshetra, India. His current research interest includes distributed algorithms, mobile computing, and fault tolerance.