A Hybrid Algorithm to Solve Group Mutual Exclusion Problem in Message passing Distributed Systems

Abhishek Swaroop
Dept. of CSE, Sharda University
Knowledge park II, Greater Noida, U.P., INDIA

Awadhesh Kumar Singh
Dpt. of Computer Engg., NIT Kurukshetra
Kurukshetra, Haryana, India

ABSTRACT

In the present paper, we propose a hierarchical algorithm to solve the group mutual exclusion (GME) problem in cluster-based systems. We consider a two-level hierarchy in which the nodes are divided in to clusters and a node in each cluster is designated as coordinator which is essentially the cluster head. The number of global messages per critical section entry in our algorithm depends upon the number of clusters in the system unlike most of the existing GME algorithms where it depends upon the total number of nodes in the system. Performance of the algorithm directly depends on the coherent behavior of nodes inside clusters. The results have been substantiated with extensive simulation. A fault tolerant extension of the algorithm has also been proposed in the present exposition.

General Terms

Distributed algorithms, resource allocation, hierarchical systems.

Keywords

Concurrency, Group Mutual Exclusion, Hybrid, Token.

1. INTRODUCTION

The group mutual exclusion (GME) problem proposed by Joung [1], is a widely studied extension of the mutual exclusion problem. The GME deals with two fundamentally contrary issues of mutual exclusion and concurrency. In GME, the processes requesting the same resource (group) can execute their critical section (CS) simultaneously. However, the processes requesting different groups, must execute their CS in mutually exclusive way. An interesting application of the GME problem is a situation in which large amount of data stored in some secondary storage device (such as a CD juke box), is being shared by several processes. The processes are interested in accessing some data stored on the CD's. However, due to the limited amount of buffer space available, 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 in a variety of areas e.g. in controlling the access to a secure database [2], improving the quality of internet servers [3], digital media files with multiple audio and caption streams [4], implementing concurrent data structures [5], in wireless applications [6], and in mobile ad hoc networks [7].
In clustered distributed systems, nodes are arranged in a hierarchical manner. In a two-level hierarchical system, the nodes are grouped into clusters and one node in each cluster is designated as cluster head. Although, two-level hierarchy is the simplest among hierarchical systems, there are some well known practical examples of two-level hierarchical systems, e.g. cellular wireless networks and *kaaza* (an example of P2P networks). The nodes may be grouped into clusters based upon interest-based locality and (or) geographical location-based locality. For example, the nodes inside an institute intranet may be arranged in department wise clusters. Similarly, all the branch offices, of some multi national company, within a particular country may form a cluster, choosing the country head office as the cluster head. If the nodes are grouped in to clusters according to their interest, it is highly likely that the nodes inside a cluster would behave coherently. For example, in an institute's network it is likely that the nodes from the same department, will request the same resource (e.g. same e-book from a collection of e-books). Further, incase the nodes are grouped in to clusters according to their geographical locations, the inter cluster message propagation delay will be significantly greater than the intra cluster message propagation delay. Therefore, the performance can be improved by aggregating requests leading to reduction in the number of inter cluster messages. Hence, a hierarchical approach to handle GME problem in a clustered environment is bound to exhibit better performance.

The first solution of the GME problem was proposed by Joung [1] for shared memory systems. Later on, few more solutions were also proposed in [5, 8-9]. However, the solutions of the GME problem, for message passing systems, fall in either of the two categories, permission-based and token-based. The permission-based algorithms for the GME problem are given in [10-12]. The token-based algorithms for GME have been proposed in [3, 13-15].
Several hybrid algorithms exist in the literature for the classical mutual exclusion problem e.g. [16-19]. Chang-Singhal-Liu [16] used Singhal's algorithm [20] at the intra cluster level and Maekawa's algorithm [21] at the inter cluster level. Bertier et al. [17] proposed two algorithms based on Naimi-Trehel's algorithm [22], taking into account the latency gaps between local and remote cluster machines. The first algorithm gives higher priority to requests issued from the nodes of the same cluster and the second algorithm uses a router layer and views each cluster as a single node. Madhuram-Kumar [18] proposed a flexible hierarchical structure, which can be modified to achieve different performance criteria. They used a different algorithm at each level of hierarchy. Erciyes [19] proposed an architecture that consists of a ring of clusters. Erciyes [19] used Ricart-Agrawala's [23] algorithm at the local level and a token passing algorithm at the global level.
The first hierarchical solution for the GME problem was proposed by Swaroop-Singh [24], in which authors used a centralized algorithm at the intra cluster level and a token-based algorithm at the inter-cluster level. In their algorithm, when a coordinator receives a secondary token, it allows nodes having pending requests for the current session, however, if a node of its cluster requests for the current session later, the requesting node is not allowed to join the

current session even if there are no conflicting requests. Therefore, Swaroop-Singh [24] algorithm does not satisfy the concurrent occupancy property [13] desirable for the algorithms solving the GME problem. Furthermore, once a coordinator has sent a global request for primary/secondary token for any group, it does not send a global request again till it receives a primary/secondary token even if the requests for different groups are pending in its local queue. This reduces the concurrency, which is a vital performance parameter for any algorithm solving the GME problem.

In the present paper, we propose a cluster-based hybrid algorithm (called CGME, henceforth) to solve the GME problem. Similar to Swaroop-Singh algorithm [24], our algorithm uses a centralized algorithm at the intra cluster level and a token-based algorithm at the inter cluster level. The proposed algorithm satisfies the concurrent occupancy property. Further, in CGME, when a coordinator (not possessing primary/secondary token) receives a request for a group, it sends a global request message provided that it has not already sent the global request message for this group. In order to handle coordinator failure, an extension of the algorithm has also been suggested. The effect of coherence and the cluster size on the performance has been studied through extensive simulation experiments.

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 of the algorithm has been discussed in section 4 and section 5, respectively. The simulation results have been presented in section 6 and a fault-tolerant extension of the algorithm has been suggested in section 7. Finally, section 8 concludes the paper.

2. SYSTEM MODEL

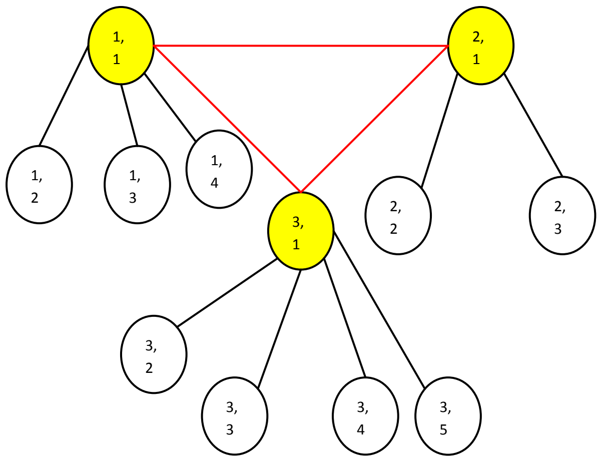

Fig. 1. A two-level hierarchical system

We assume a message passing asynchronous distributed system. The system has n nodes divided in to p clusters. Each cluster i contains n_i nodes such that $\sum_{i=1}^{p} n_i = n$. A node can be identified by a cluster identifier and a node identifier, for example node $n_{i,j}$ means j^{th} node of i^{th} cluster. In each cluster i, node $n_{i,1}$ works as cluster head or coordinator. An example of two-level hierarchical system consisting of three clusters is given in figure 1. Each communication channel is assumed to be FIFO and each message is delivered in finite time. The maximum message propagation delay between the nodes of a cluster is T_l and between the nodes of different clusters is T_r ($T_l < T_r$). Each node may remain inside CS for a finite time.

3. WORKING OF CGME

The data structures required and the complete pseudo code of CGME are given in appendix A and appendix B respectively. However, for the convenience of the readers, we present the high level description of CGME in this section. CGME uses a centralized approach at the intra cluster level and a token-based approach at the inter cluster level. In CGME there exists two type of tokens namely primary token and secondary tokens. The coordinator holding primary token can issue secondary tokens to other coordinators, while the coordinator holding secondary token is not allowed to do so.

A node $n_{i,j}$ requesting a group g, sends its request to the cluster coordinator $n_{i,1}$ and waits for its permission. Upon receiving permission from its coordinator, the requesting node enters CS and informs its coordinator when it comes out of CS.

When a coordinator $n_{i,1}$ receives a request for group g from a node $n_{i,j}$, following three case are possible.

(i) $n_{i,1}$ possesses idle primary token: $n_{i,1}$ immediately sends permission to the requesting node, changes its state, changes the current group to g, and adds the requesting node in the list of nodes to which permission has been granted.

(ii) $n_{i,1}$ possess primary (non idle)/secondary token: In this case, if the current group is the same as being requested by $n_{i,j}$ and $n_{i,1}$ is not aware of any conflicting request then $n_{i,1}$ sends the permission to the requesting node and adds the requesting process in the list of allowed nodes. Otherwise, the request is added in the local queue. Further, if the group requested is not the same as the current group and $n_{i,1}$ holds the primary token, it informs all secondary token holders about the existence of a conflicting request, if it has not already done so. However, if the group requested is not the same as the current group and $n_{i,1}$ holds a secondary token, it informs the primary token holder about the existence of a conflicting request, only if it was not previously aware of any conflicting request.

(iii) $n_{i,1}$ does not possess primary/secondary token: $n_{i,1}$ adds the request in its local queue. Further, it sends a global request to all the coordinators in its request set, if it has not already sent a global request for 'g'.

When a coordinator $n_{i,1}$ holding idle primary token, receives a request from some other coordinator $n_{j,1}$, it immediately sends the primary token to $n_{j,1}$ and adds $n_{j,1}$ in its request set. However, if $n_{i,1}$ is holding non idle primary token, it sends a secondary token to $n_{j,1}$ only if the group requested by $n_{j,1}$ is the current group and there are no conflicting requests known to $n_{i,1}$. Otherwise the request of $n_{j,1}$ is added in the cumulative queue. Besides that, when a primary token holder first time become aware of any conflicting request (current group != requested group) , it informs all the secondary token holders about the existence of a conflicting request with the help of a CR_NOTIFY message. Similarly, when a secondary token holder receives a conflicting request it informs the primary token holder about the conflicting request via CR_NOTIFY message only if it was not aware of any conflicting request before this. However, if node $n_{i,1}$ does not possess primary/secondary token, it adds $n_{j,1}$ in its request set if it is not already there.

Upon receiving secondary token, the coordinator $n_{j,1}$ sends permission to all local nodes requesting the current group g, remove these nodes from its local queue, and waits for the exit of these nodes from CS. Once $n_{j,1}$ has information that all the local nodes to which permission was sent have come out of CS, it returns the secondary token to the primary token holder. Along with this return message, it piggybacks a vector containing the groups for which there are pending requests in its local queue.

The primary token holder $n_{i,1}$ maintains a list of nodes to which it has issued secondary tokens. When a coordinator returns a secondary token, $n_{i,1}$ removes its from this list. If $n_{i,1}$ has received back all the secondary tokens and $n_{i,1}$ has received the information that all the local nodes to which it has sent the permission have come out of CS, it announces the termination of the session and selects the next group *g* and the next primary token holder. Further, it sends the primary token (along with the list of coordinators to which the secondary tokens have to be issued) to the newly selected primary token holder. $n_{i,1}$ removes those entries from the cumulative queue which are for group *g*. Besides that, it adds all the coordinators to which the primary token can be transferred in its request set.

When the newly selected primary token holder receives the primary token, it empties its request set, sets its current group as *g*, and sends permission to all local nodes requesting group *g*. Further, it removes all local nodes requesting group *g* from its local queue and sends the secondary token to all the coordinators which are in the list received along with the primary token.

4. CORRECTNESS PROOF

4.1 Safety

Lemma 1: There exists only one primary token in the system at any time.

Proof: It is assumed that, initially, the system has only one primary token. Since there does not exist any primary token generation procedure, there would exist only one primary token in the system.

Lemma 2: If $n_{i,1}$ is the coordinator having primary token and $n_{j,1}$ is a coordinator possessing secondary token then $mygroup_{i,1} = mygroup_{j,1}$.

Proof: The variable *mygroup* is a local variable; hence, it can be modified only by its owner node. A node modifies its local variable *mygroup* on receiving either primary token or secondary one. On receiving secondary token, a node assigns the value contained in the secondary token, just received, to its local variable *mygroup*. However, the value contained in the secondary token is the value passed by the primary token holder node. Since there exist only one primary token holder, as proved in lemma 1, the variable *mygroup* contains the same value in every token holder node irrespective of primary or secondary. Therefore, lemma 2 holds.

Theorem 1: If two nodes are in CS simultaneously, they must be using the same group.

Proof: A node is allowed to enter CS, if any of the following conditions are met (i) its coordinator has idle primary token (ii) its coordinator has non idle primary/secondary token, the group requested by the node is the same as the currently open group and the coordinator is not aware of any conflicting requests. By lemma 2, in both the cases variable *mygroup* has same value on each coordinator node having a primary/secondary token; therefore, the nodes concurrently in CS use the same group.

4.2 Liveness

In the present section we will prove that CGME fulfils the liveness requirement.

Lemma 3: If $n_{i,1}$ and $n_{j,1}$ are two coordinators then either $n_{j,1} \in RS_{i,1}$ *or* $n_{i,1} \in RS_{j,1}$.

Proof: Initially, The request set of $n_{1,1}$ is empty and the request set of any other coordinator $n_{i,1}$ contains all other coordinators. Therefore, the condition is true for any two coordinators $n_{i,1}$ and $n_{j,1}$. Now, the request set of any node $n_{i,1}$ is changed only if any of the following three events occurs:

(a) The G_REQUEST message of $n_{j,1}$ is received by $n_{i,1}$ and $n_{j,1} \notin RS_{i,1}$.

(b) The current session terminates and the previous token holder $n_{i,1}$ transfers primary token to a new coordinator.

(c) The node $n_{i,1}$ receives the primary token.

We will prove that if the condition is satisfied before the above mentioned events; it will remain true after the occurrence of one of these events. Incase of occurrence of event (a) $P_{j,1}$ is added in $RS_{i,1}$ and in case of occurrence of event (b) $P_{j,1}$ and other nodes having pending requests and which can hold primary token in future are added in $RS_{i,1}$. However, no node is deleted from the request set of any node. Therefore, if the condition is true before event (a) or event (b) occurs, it will remain true even after the occurrence of these events.

The request set of $n_{i,1}$ is emptied when it receives the primary token from $n_{j,1}$ (event(c)). However, we assume that before the occurrence of event (c) $\forall i, j\ (i \neq j)\ n_{j,1} \in RS_{i,1}$ *or* $n_{i,1} \in RS_{j,1}$. Therefore, before receiving the primary token the nodes of the system, except $n_{i,1}$, can be divided in to two sets *S*1 and *S*2. The first set *S*1 contains the nodes, which are in the request set of $n_{i,1}$ and the second set *S*2 contains the nodes, which contain $n_{i,1}$ in their request sets. After the occurrence of the event (c), the $n_{i,1}$ will remain in the request sets of the nodes in set *S*2 and the set *S*1 will be empty set because the request set of $n_{i,1}$ will be emptied. However, before event (c), $n_{i,1}$ must have sent a request to all members in its request set, which would be same as *S*1, before receiving the primary token. Thus, all the nodes in *S*1 will add $n_{i,1}$ in their request set, if it is not already there. Therefore, the nodes previously in set *S*1, will be added in set *S*2 and $n_{i,1}$ will be in the request set of all other nodes except itself. Thus, the condition remains true after the event. Hence, lemma 3 holds.

Lemma 4: G_REQUEST message sent by a node $n_{i,1}$ will eventually reach the primary token holder node.

Proof: $n_{i,1}$ sends G_REQUEST message to all members in its request set $RS_{i,1}$. Let $n_{j,1}$ ($j \neq i$) is the current primary token holder. Now, from lemma 3, For any two coordinators $n_{i,1}$ and $n_{j,1}$, either $n_{j,1} \in RS_{i,1}$ *or* $n_{i,1} \in RS_{j,1}$ and $n_{j,1} = \emptyset$ (because $n_{j,1}$ holds the primary token); therefore, $n_{j,1}$ must be in $RS_{i,1}$ and G_REQUEST message will reach $n_{j,1}$. However, $n_{j,1}$ might have transferred the primary token to some other node, say $n_{k,1}$. Now, before $n_{k,1}$ receives primary token, either $n_{k,1}$ would have been in $RS_{i,1}$ or $n_{i,1}$ would have been in $RS_{k,1}$. If $n_{k,1}$ was in $RS_{i,1}$, $n_{i,1}$ must have sent a G_REQUEST message that would eventually reach $n_{k,1}$. On the other hand, if $n_{i,1}$ was in $RS_{k,1}$, $n_{k,1}$ must have sent a G_REQUEST message to $n_{i,1}$. In this case, $n_{i,1}$ will add $n_{k,1}$ in its request set and will send a G_REQUEST message to $n_{k,1}$. Thus, the G_REQUEST message will eventually reach the primary token holder node.

Lemma 5: The group that is added in the cumulative queue will eventually be selected.

Proof: As soon as a primary token holder node knows about the existence of conflicting request, it stops allowing the nodes of its cluster from joining the current session and informs all secondary token holders about the existence of conflicting request via CR_NOTIFY message. Similarly, when a secondary token holder becomes aware of a conflicting request, it informs the primary token holder about it, if it has not already done so. Therefore, all token holders will know about the existence of a conflicting request in finite time and will stop allowing the nodes of their respective clusters to join the current session. The nodes to which the permission has been granted earlier will come out of CS and the current session will eventually terminate. Now, since the group at the head of the token queue is selected for the next session, any group added in the cumulative queue will reach at the head of the token queue and will be selected for a session in finite time.

Theorem 2: The request of a node $n_{i,j}$ for group g will eventually be serviced.

Proof: The requesting node $n_{i,j}$ will send its request to its coordinator $n_{i,1}$. Now, if $n_{i,1}$ is having primary/secondary token and is able to grant permission to $n_{i,j}$, it sends an ALLOW message to $n_{i,j}$ and $n_{i,j}$ enters CS. If $n_{i,1}$ is possessing primary token and is not able to allow $n_{i,j}$, the request of $n_{i,j}$ is added in the local queue of $n_{i,1}$. Further, the request is added in the cumulative queue only if the request for g from cluster i, is not already in the cumulative queue. However, if $n_{i,1}$ is not having the primary token, the request of $n_{i,j}$ is added in local queue of $n_{i,1}$. Further, the node $n_{i,1}$ forwards the request for group g either through G_REQUEST or through RET_SEC message. When the request for g is forwarded through RET_SEC message, it reaches the primary token holder and it is added in the cumulative queue. If the request for g is forwarded through G_REQUEST, it will eventually reach the primary token holder (lemma 4) and will be added in the cumulative queue. Once the request for group g is added in the cumulative queue, the group g will eventually be selected (lemma 5) and the coordinator $n_{i,1}$ will receive either a primary or secondary token with $mygroup_{i,1}=g$. In both the cases, $n_{i,1}$ will allow $n_{i,j}$ to enter CS using group g and the request of $n_{i,j}$ will be serviced.

4.3 Concurrent Occupancy

In CGME, once a coordinator has received a primary/secondary token, it keeps on allowing the nodes of its cluster to join the currently open session until it becomes aware of a conflicting request. Hence, if there are not any requests for any other group except the group currently being accessed, a node may join the current session without waiting for any other node to come out of CS. Therefore, it is proved that CGME satisfies the concurrent occupancy.

5. PERFORMANCE ANALYSIS

We analyze the performance of CGME algorithms using following performance metrics: message complexity, waiting time, synchronization delay, message size, and maximum concurrency. We also propose a metric to measure the level of coherence shown by the nodes of a cluster.

5.1 Message complexity

In CGME each request requires three intra cluster (local) messages/CS namely REQUEST, ALLOW, and COMPLETE. As far as the inter cluster (global) messages are concerned, the worst case occurs when there is only one node requesting a group in a particular cluster and the coordinator of that cluster contains all other p-1 coordinators in its request set. In this case, if the coordinator receives a primary token, p global messages will be required (p-1 G_REQUEST messages, and one P_TOKEN). On the other hand, if the coordinator receives a secondary token, p+1 global message (p-1 G_REQUEST messages, one S_TOKEN, and one RET_SEC message) will be required. However, in the best case (when the coordinator of the requesting node is holding an idle token) no global message is needed. In average case, the number of global messages required per CS entry is reduced due to following two factors (i) The size of the request set will vary between 0 and p-1 (ii) The requests for the same group within a cluster will be aggregated. The first factor is prominent under the light load conditions and the second factor dominates under heavy load conditions.

5.2 Waiting time and Synchronization delay

The waiting time of a ME/GME algorithm is usually measured when the system is lightly loaded. We have already assumed that the maximum intra cluster message propagation delay is T_l while the inter cluster message propagation delay is T_r. As far as the waiting time is concerned, the worst case occurs when the primary token is not being held by the coordinator of the requesting node. In that case the waiting time will be $2T_l$+$2T_r$ (REQUEST→ G_REQUEST→ S_TOKEN /P_TOKEN→ ALLOW). However, if the coordinator of the requesting node holds a primary token/ secondary token and there are no conflicting requests, the waiting time will be $2T_l$ (REQUEST→ALLOW) only.

The synchronization delay is generally measured when the system is heavily loaded. The worst case for synchronization delay occurs when the last node to come out of CS is from the cluster not having the primary token. In that case the synchronization delay will be $2T_l$+$2T_r$ (COMPLETE→ RET_SEC → P_TOKEN→ ALLOW). However, the best case occurs when the last node to come out of CS to finish the session is from the cluster possessing the primary token and the same node is again selected to hold the primary token for the next session. In that case the synchronization delay will be $2T_l$ (COMPLETE→ ALLOW).

5.3 Message size

The local messages used by our algorithm are ALLOW, REQUEST, and COMPLETE. All local messages are of constant size. Our algorithm uses five inter cluster message namely G_REQUEST, P_TOKEN, S_TOKEN, CR_NOTIFY and RET_SEC. The size of G_REQUEST, S_TOKEN, and CR_NOTIFY messages is O(1). However, the size of P_TOKEN message is O(p) (cumulative queue is passed along with the message) and the size of RET_SEC message is O(m) (A vector containing groups for which there are pending requests is passed along with the message). Here p is the number of clusters in the system and m is the total number of groups in the system. However, only one P_TOKEN message per session is required to be exchanged. Further, since we piggyback the groups (say k groups) having pending requests on RET_SEC message, it saves k*$|RS_{i,1}|$ (where $|RS_{i,1}|$ denotes the cardinality of the request set of the coordinator $n_{i,1}$ which sent the RET_SEC message) G_REQUEST messages to be exchanged.

5.4 Maximum Concurrency

The maximum concurrency of our algorithm is n. If all the nodes of all the clusters are requesting the same group, all these nodes can enter CS simultaneously. In the absence of any conflicting requests, the coordinator possessing the primary token will permit all nodes in its cluster to enter CS. However, the coordinators not possessing the primary token will eventually receive the secondary token and will permit the nodes in their cluster to enter CS (since there are not any conflicting requests). Hence, all the nodes can enter in CS simultaneously provided that all of them are requesting the same group.

5.5 Level of Coherence

Usually, in a cluster-based environment, the nodes in a cluster behave in a coherent manner (many of them interested in the same group). Hence, the requests of such nodes will be aggregated that would lead to the improved performance. In particular, when the level of coherence increases, the concurrency is expected to increase and the waiting time as well as the number of global messages/CS are expected to decrease.

If *nrg*$_i$ is the number of groups requested by the nodes of cluster *i* during the interval under consideration and *ng* is the total number of groups then the level of coherence of cluster *i*, say *coherence*$_i$, can be calculated as follows: *coherence*$_i$ =*ng*/*nrg*$_i$. The value of *coherence*$_i$ lies between 1 and *ng*. The higher is the value of *coherence*$_i$ higher is the coherence among the nodes of the cluster *i*. The value *coherence*$_i$=1 signifies that the nodes of cluster *i* have requested for all the groups during the interval under consideration. On the other hand, the value *coherence*$_i$=*ng*, signifies that all the requests made by the nodes of cluster *i* are for the same single group in that interval.

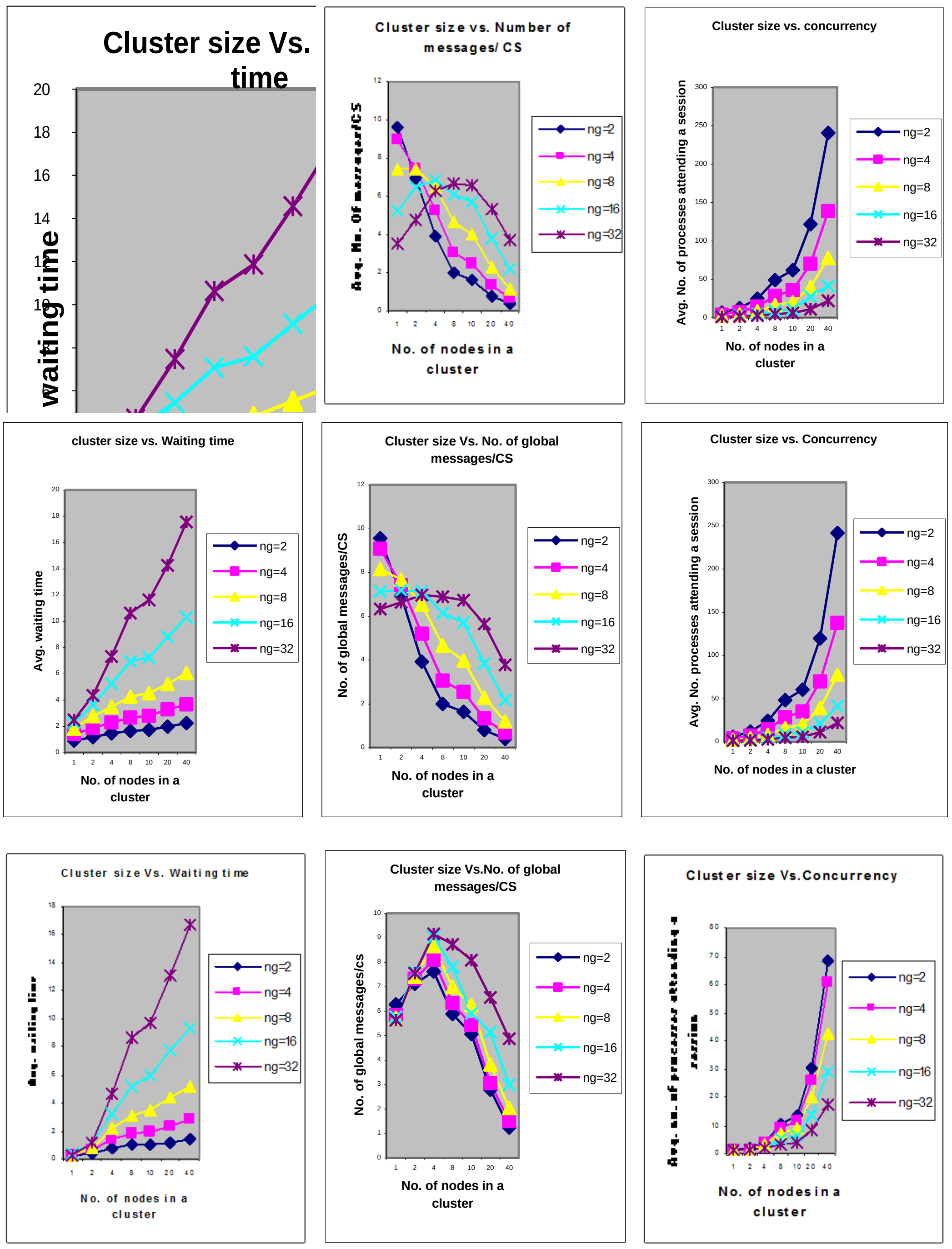

Fig. 2. Effect of cluster size on the performance under heavy load (upper 3 charts), medium load (middle 3 charts), and light load (lower 3 charts)

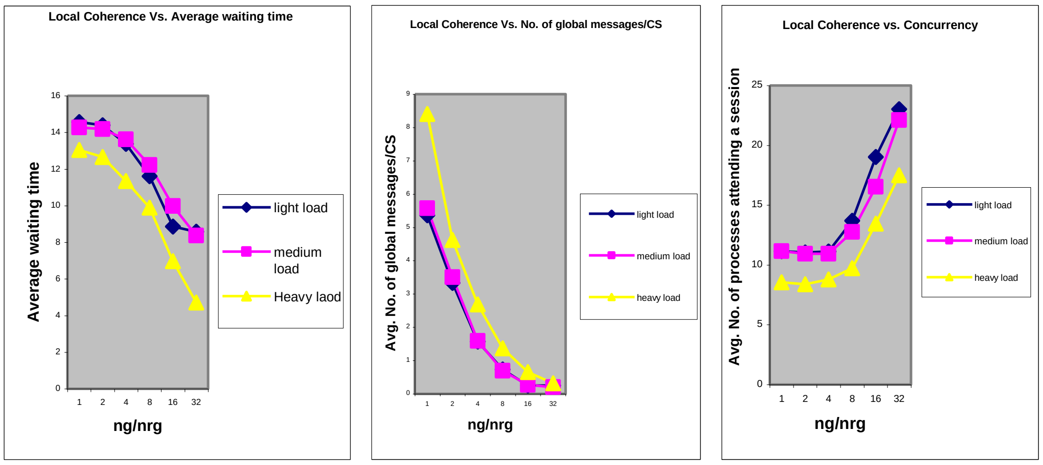

Fig. 3. Effect of local coherence on the performance

6. SIMULATION RESULTS

In order to evaluate the performance of the proposed algorithm, we performed the simulation experiments by varying different parameters such as number of groups, number of processes, contention level, number of nodes in a cluster, and level of coherence among the nodes. The cluster size and the level of coherence among the nodes of a cluster are the two most important parameters; therefore, we divided the simulation experiments in to two parts. The first experiment has been carried out to observe the effect of the cluster size on the performance of the algorithm (results shown in figure 2) while the second experiment has been carried out to observe the effect of coherence (results shown in figure 3).

6.1 Simulation setup

We used a discrete event driven network simulator, namely NS2, to perform the simulations. We measured the average waiting time, the average number of global message/CS and the average number of processes attending a session. We assumed ten clusters in the system. The intra cluster delay and inter cluster delay were assumed to be exponentially distributed with mean value of 2 milliseconds and 50 milliseconds respectively. The time for which a process may remain inside its CS was assumed to be exponentially distributed with mean value of 250 milliseconds (μcs) and the idle time of a node was also assumed to be exponentially distributed with mean value (μncs). Joung [10] proposed that the level of contention can be calculated as follows: contention level = (μcs / (μcs+ μncs))*100. In our experiment, we varied the value of μncs to get the desired level of contention. Both the experiments were conducted under heavy load (contention level 100%), medium load (contention level 50%), and light load (contention level 5%).

We varied the number of nodes in a cluster (1, 2, 4, 8, 10, 20 and 40) and the number of groups in the system (2, 4, 8, 16 and 32) in the first experiment. However, in the second experiment we assumed that there are 20 nodes in each cluster and 32 groups in the system. The $coherence_i$ (the level of local coherence shown by the nodes of a cluster) is varied from 1 to 32 (values considered are 1, 2, 4, 8, 16, and 32). Each node makes 100 requests in each run and each point is plotted after taking the average of ten such runs.

6.2 Discussion on Simulation results

6.2.1 Effect of cluster size

As per expectations, for all contention levels, the average waiting time and concurrency increases as we increase the number of nodes in a cluster for all values of *ng* (the total number of groups in the system). Further, the average waiting time increases more rapidly as *ng* is increased. However, the concurrency increases more rapidly for smaller values of *ng*. As far as global messages are concerned, two factors are effecting the number of global messages required/CS (i) due to the use of dynamic request sets, at the inter cluster level, the size of the request set; and hence, the number of global request messages/CS reduces at low level of contention. (ii) Due to the aggregation of requests the number of global messages reduces specially under heavy contention. Now, when the cluster size increases the number of nodes requesting, during same interval, increases; and hence, the size of the request sets will also increase. Therefore, when the cluster size increases the number of G_REQUEST messages per global request is expected to increase; however, the aggregation effect reduces the number of global requests. The aggregation effect becomes more prominent, particularly, when the total number of groups in the system is less. The simulation results suggest that under low level of contention, when we increase the cluster size up to a limit the number of global messages/CS increases (for all values of *ng*). It signifies that the size of the request set is more dominating factor and there is not much scope for aggregation of requests. However, beyond that limit the aggregation of requests becomes the dominating factor and the number of global messages/CS start reducing with an increase in cluster size.

Under medium load and heavy load conditions for *ng*=16 and *ng*=32 the number of global message/CS increases initially on increasing the cluster size up to a limit and start decreasing thereafter. However, for smaller values of *ng* the number of global messages/CS always decreases on increasing the cluster size. Because, for smaller values of *ng*, more requests will be aggregated in a global request message as there are fewer groups.

7. FAULT TOLERANT EXTENSION OF CGME

There are two inherent disadvantages of using a centralized approach at the intra cluster level:

(i) If the coordinator of a cluster fails, no node of the cluster will be able to enter CS.

(ii) The coordinator of a cluster may be overloaded, specially, under heavy load.

Both of the above mentioned problems can be solved, if we use a technique similar to one used by Yang-Molina [25], although, for super peer networks. Instead of having one coordinator per cluster, we can have *k* coordinators per cluster (where *k* is a configurable parameter). All the nodes of a cluster are connected with all *k* coordinators. Further, each of the *k* coordinators of a cluster is connected with all other coordinators of its own cluster as well as the coordinators of other clusters. On increasing the number of coordinators, the number of communication links will increase. It would result in increase in the average number of global messages/CS and may lead to consistency problem. These problems would further aggregate for the large values of *k*, therefore, smaller values of k (e.g. *k*=2 or *k*=3) are practically preferable.

Besides providing the fault tolerance, the existence of more than one coordinator per cluster is also useful for load sharing. For example, if there are two coordinators per cluster then the nodes with odd ids may send their requests to coordinator 1 and those with even ids may send their requests to coordinator 2. The two coordinators of a cluster periodically communicate with each other to check whether the other coordinator is in-order. Once a coordinator (say $n_{i,1}$) detects that the other coordinator (say $n_{i,2}$) is out of order, $n_{i,1}$ sends a message to all the nodes which were previously connected to $n_{i,2}$. The coordinator $n_{i,1}$ collects the responses from these nodes, updates its data structures, and piggybacks requests (if any) on the message which it sends to all other coordinators of other clusters to make them aware of the fact that now $n_{i,1}$ is the sole head of cluster *i*.

If $n_{i,2}$ recovers, it will start the periodic communication with $n_{i,1}$. As soon as $n_{i,1}$ hears from $n_{i,2}$, it forwards the updates to $n_{i,2}$, about the nodes earlier related to $n_{i,2}$, and also directs them to contact $n_{i,2}$ from now onwards. The coordinator $n_{i,1}$ informs coordinators of other clusters about the recovery of the coordinator $n_{i,2}$. Later on, due to some reason, if $n_{i,1}$ receives some message about a node which is now related to $n_{i,2}$, $n_{i,1}$ forwards the message to $n_{i,2}$.

8. CONCLUSION AND FUTURE WORK

The proposed hierarchical algorithm CGME satisfies all the requirements in order to handle the GME problem. As evident from the simulation results, due to the use of dynamic request sets and the aggregation of requests for the same group, CGME exhibits remarkable performance at heavy, medium and light load conditions. The performance metrics, particularly the number of global messages/CS and the concurrency, further improves as we increase the level of coherence or the cluster size.

CGME handles only two-level hierarchical systems not the general (*k*-level) hierarchical distributed systems, therefore, as a future work we plan to design an approach which can handle the GME problem in *k*-level hierarchical distributed systems. The fault tolerant extension of CGME is capable of handling the coordinator failure. Besides that it provides load sharing among the *k* coordinators of each cluster. However, details of the fault tolerant version are beyond the scope of present exposition.

9. REFERENCES

[1] Joung, Y.J., 2000,, Asynchronous Group Mutual Exclusion, Distrib. Comput. 13(4), 51 –60.

[2] Park, J., Gang, S., Kim, K., 2003, Group Mutual Exclusion Based Secured Distributed Protocol, Joho Shori Gakkai Shinpojiumu Ronbushu. 15, 445-450.

[3] Thiare, O., Gueroui,M., Naimi, M., 2006, Distributed Group Mutual Exclusion Based on Clients/ Server Model, In Proc. 7th Int. Conf. on Parallel and Distribut. Comput. Appl. and Technol. 67-73.

[4] Advance System Format (ASF) specifications for digital media files available at URL: http://www.msdn.microsoft.com/en-us/library/bb643323.aspx.

[5] Kean, P., Moir, M., 1999, A Simple Local Spin Group Mutual Exclusion Algorithm, In Proc. 18th Annu. ACM Symp. on Princ. of Distrib. Comput, 23-32.

[6] IEEE 802.11 Working Group for Wireless Local Area Networks, available at URL: http://ieee802.org/11/,2003

[7] Jiang, J.R., 2002, A Group Mutual Exclusion Algorithm for Ad Hoc Mobile Networks, In Proc. 6th Int. Conf. on Comput. Sci. and Inf. 266-272.

[8] Hadzilacos, V., 2001, A Note on Group Mutual Exclusion. In Proc. 20th ACM Symp. on Princ. of Distrib. Comput., 100-106.

[9] Jayanti, P., Petrovic, S., Tan, K., 2003, Fair Group Mutual Exclusion, In Proc. 22nd Conf. on Princ. of Distrib. Comput., 275-284.

[10] Joung, Y.J., 2002, The Congenial Taking Philosopher Problem in Computer Networks, Distrib. Comput., 15(3), 189-206.

[11] Attreya, R., Mittal, N., 2005, A Dynamic Group Mutual Exclusion Algorithm using Surrogate Quorums, In Proc. 25th IEEE Conf. on Distrib. Comput. Syst., 251-260.

[12] Joung, Y.J., 2003, Quorum-Based Algorithms for Group Mutual Exclusion, IEEE Trans. on Parallel and Distrib. Comput. Syst., 14(5), 463-476.

[13] Mamun, Q.E.K., Nakazato, H., 2006, A New Token Based Algorithm for Group Mutual Exclusion in Distributed Systems, In Proc. 5th Int. Symp. on Parallel and Distrib. Comput., 34-41.

[14] Mittal, N., Mohan, P.K., 2007, A Priority Based Distributed Group Mutual Exclusion Algorithm when Group Access is Non-Uniform., J. of Parallel and Distrib. Comput. 67(7), 795-815.

[15] Swaroop, A., Singh, A. K., 2007, A Token-Based Fair Algorithm for Group Mutual Exclusion in Distributed systems, J. of Comput. Sci., 3(10), 829-835.

[16] Chang, Y. I., Singhal, M., Liu, M. T., 1990, A Hybrid Approach to Mutual Exclusion for Distributed Systems, In Proc. 14th Intl. Comput. Softw. and Appl. Conf., 289-294.

[17] Bertier, M., Arantes, L., Sens, P., 2006, Distributed Mutual Exclusion Algorithms for Grid Applcations: A Hierarchical approach, J. of Parallel and Distrib. Comput., 66, 128-144.

[18] Madhuram, S., Kumar, A., 1994, A Hybrid Approach for Mutual Exclusion in Distributed Computing Systems, In Proc. 6th IEEE Symp. on Parallel and Distrib. Process., 18-25.

[19] Erciyes, K., 2004, Distributed Mutual Exclusion Algorithms on a ring of Clusters, In Proc. Int. Conf. on Comput. Sci. and Its Appl., 518-527 .

[20] Singhal, M., 1989, A Dynamic Information Structure Distributed Mutual Exclusion Algorithm for Distributed Systems. In Proc. 9th Int. Conf. on Distrib. Comput. and Syst., 70-78.

[21] Maekawa, M., 1985, An Algorithm for Mutual Exclusion in Decentralized Systems, ACM Trans. on Comput. Syst. 3(2) (1985) 145–159.

[22] Naimi, M., Trehel, M., Arnold, A., 1993, A Log(N) Distributed Mutual Exclusion Algorithm based on Path Reversal. J. of Parallel and Distrib. Comput., 34, 1-13.

[23] Ricart, G., Agarwala, A. K., 1981, An Optimal Algorithm for Mutual Exclusion in Computer Networks, Commun. Of the ACM, 24(1), 9-17.

[24] Swaroop, A., Singh, A. K., 2009, A Hierarchical Approach to Handle Group Mutual Exclusion in Distributed Systems, In Proc. Int. Conf. On Dist. Comput. And Networking, 462-467.

[25] Yang, B., Molina, H. G., 2003, Designing a super peer networks, In Proc 19th Intl. Conf. on Data Engg., 49-60.

Appendix A: Data Structure and Messages used in CGME

Each node except the cluster head may remain in any one of the flowing three states R: requesting, N: not requesting, CS: executing in CS
Each coordinator maintains following data structure:
$state\_c_{i,1}$ - Stores the state of a coordinator possible states are : W -Waiting for primary/secondary token, NW -Not waiting HST -Holding secondary token HIPT -Holding idle primary token HPT -Holding primary token.
$local\_q_{i,1}$ - A queue which is used to store local requests.
$allow\_l_{i,1}$ - Set of local nodes to which the coordinator has sent permission.
$allow\_r_{i,1}$ - Set of coordinators to which the primary token holder has sent secondary tokens.
$mygroup_{i,1}$ - The group which is currently being accessed by the nodes of cluster *i*.
$session\_no_{i,1}$ - It stores latest session number known to $P_{i,1}$.
$myprimary_{i,1}$ - The id of the primary token holder node to which secondary token has to be returned.
$RS_{i,1}$ - Request set containing the id's of the nodes to which the global request has to be sent.
gr_req_i - It stores the groups for which the coordinator has sent the global request.
$cr\_flag_{i,1}$ - TRUE if coordinator knows about any conflicting pending request FALSE otherwise.

The primary token contains following arrays:
cum_q - It stores all the global requests.
session_pt - An array of size *p*, it stores the session number of each cluster known to primary token.
allow_sec - It contains the ids of the coordinators to which secondary tokens have been sent.

The intra cluster messages exchanged among the nodes of a cluster are as following:
REQUEST - A requesting node sends this message to its coordinator.
ALLOW - The coordinator sends ALLOW to a node permitting it to enter in CS.
COMPLETE - The node sends COMPLETE message to its coordinator to inform that it has come out of CS.

The inter cluster messages exchanged between coordinators are as following:
G_REQUEST- A coordinator sends G_REQUEST message to all the coordinators in its request set. It contains the group requested and id of the requesting coordinator.
P_TOKEN -This message contains the group selected, the set of coordinators to which secondary tokens have to be sent, cumulative queue and an array of session numbers.
S_TOKEN - It contains the group selected, the id of the primary token holder and the session number.
RET_SEC - It is used to return secondary token. It also piggybacks vector containing the groups for which there are pending requests in its local queue.
CR_NOTIFY - Used to notify other coordinators about existence of a conflicting pending request.

Appendix B: The Pseudo Code of CGME
Initialization:

```
For $i$=1 to $p$
   $RS_{i,1}$=id's of all other coordinators
      $local\_q_{i,1}$=Ø; $cum\_q_{i,1}$=Ø ; $state\_c_{i,1}$=NW;
   $myprimary_{i,1}$=NULL
   $allow\_l_{i,1}$=Ø;  $allow\_r_{i,1}$= Ø;  $session\_no_{i,1}$=0 ;
   $cr\_flag_{i,1}$=0
   $session\_pt[i]$=0;  $mygroup_{i,1}$=NULL; $gr\_req_{i,1}$= Ø
   For $j$ = 1 to $n_i$
      $state\_m_{i,j}$=NR
$state\_c_{1,1}$=HIPT; $RS_{1,1}$= Ø ;$allow\_sec$= Ø; $cum\_q$= Ø
```

$n_{i,j}$ Requesting for group g

```
Send REQUEST ($n_{i,j}$ , $g$) to $n_{i,1}$;$state\_m_{i,j}$=R
```

$n_{i,j}$ receives ALLOW (g) from $n_{i,1}$

```
Enter CS; $state\_m_{i,j}$=CS;……executing in CS…..
Exit CS; $state\_m_{i,j}$=N; Send COMPLETE ($n_{i,j}$) to $n_{i,1}$
```

$n_{i,1}$ receives REQUEST ($n_{i,j}$, g)

```
If ($state\_c_{i,1}$== HIPT) :
   Send ALLOW ($g$) to $n_{i,j}$
   $session\_no_{i,1}$++;  $mygroup_{i,1}$=$g$; $session\_pt[i]$++;
$state\_c_{i,1}$=HPT
    $allow\_l_{i,1}$=$n_{i,j}$; $allow\_r_{i,1}$= Ø
Elseif ($state\_c_{i,1}$== HPT)
    If (($mygroup_{i,1}$=$g$) &&( $cr\_flag_{i,1}$==0)
          Send ALLOW ($g$) to $n_{i,j}$;  $allow\_l_{i,1}$=$allow\_l_{i,1} \cup n_{i,j}$
    Else
         If ($cr\_flag_{i,1}$==0)
              $cr\_flag_{i,1}$=1; send CR_NOTIFY ($n_{i,1}$) to all nodes in
$allow\_r_{i,1}$
        Add REQUEST ($n_{i,j}$ , $g$) in $local\_q_{i,1}$
Elseif ($state\_c_{i,1}$== HST)
         If (($mygroup_{i,1}$=$g$) &&( $cr\_flag_{i,1}$==0)
            Send ALLOW (g) to $n_{i,j}$ ; $allow\_l_{i,1}$=$allow\_l_{i,1} \cup n_{i,j}$
         Else
              If ($cr\_flag_{i,1}$==0)
                 $cr\_flag_{i,1}$=1;
                 send CR_NOTIFY($n_{i,1}$) to $myprimaryi_{,1}$
        Add REQUEST ($n_{i,j}$ , $g$) in $local\_q_{i,1}$
 Else
      If ($g$ not in $gr\_req_{i,1}$)
         Send G_REQUEST($n_{i,1}$,$g$,$session\_no_i$) to nodes in $RS_{i,1}$
         append $g$ in  $gr\_req_{i,1}$
    Add request  in  $local\_q_{i,1}$
```

$n_{i,1}$ receives COMPLETE ($n_{i,j}$)

```
Remove $n_{i,j}$ from $allow\_l_{i,1}$
```

```
If ($state\_c_{i,1}$=HPT)
   If ($allow\_l_{i,1}$=Ø) && ($allow\_r_{i,1}$=Ø)
       Call Sel_next_coord ( )
Else
    If ($allow\_l_{i,1}$=Ø)
         $mygroup_{i,1}$=NULL; $myprimary_{i,1}$=NULL
         If ($local\_q_{i,1}$≠Ø)
            $state\_c_{i,1}$=W ;
           append groups having requests in vect and in $gr\_req_{i,1}$
     Else
         $state\_c_{i,1}$=NW; vect= Ø
  Send RET_SEC ($n_{i,1}$, vect) to $myprimary_{i,1}$
```

$n_{i,1}$ receives P_TOKEN (*g, allow_sec, session_pt, cum_q,cr_f*)

```
 Remove g from $gr\_req_{i,1}$
$state\_c_{i,1}$=HPT; $RS_{i,1}$=Ø;$mygroup_{i,1}$=g;
$allow\_r_{i,1}$=allow_sec; $session\_no_{i,1}$=session_pt[i]
send ALLOW (g) to  nodes requesting  for g ,
 Add these nodes in $allow\_l_{i,1}$ and Remove these  nodes from
$local\_q_{i,1}$
If (cum_q= Ø) && ($local\_q_{i,1}$= Ø)
       $cr\_flag_{i,1}$=0
Else
        $cr\_flag_{i,1}$=1
send S_TOKEN to all processes in $allow\_r_{i,1}$;
increment their session no. in session_pt
```

$n_{i,1}$ receives S_TOKEN (*$n_{j,1}$, g,session,cr_f*)

```
Remove g from $gr\_req_{i,1}$
$session\_no_{i,1}$=session; $mygroup_{i,1}$=g;$state\_c_{i,1}$=HST;
$myprimary_{i,1}$=$n_{j,1}$
If (cr_f==1)
    $cr\_flag_{i,1}$=1
 Send ALLOW (g) to nodes requesting for g
Remove these nodes from $local\_q_{i,1}$
If ($local\_q_{i,1}$!= Ø) && ($cr\_flag_{i,1}$==0)
    $cr\_flag_{i,1}$=1
    Send CR_NOTIFY($n_{i,1}$) to $myprimary_{i,1}$
```

$n_{i,1}$ receives RET_SEC (*$n_{j,1}$, vect*)

```
Remove $n_{j,1}$ from $allow\_r_{i,1}$;Add requests in vect in cum_q
If ($allow\_r_{i,1}$=Ø) && ($allow\_l_{i,1}$=Ø)
    $mygroup_{i,1}$=NULL; $myprimary_{i,1}$=NULL
       Call Sel_next_cord ( )
Else
      If ($cr\_flag_{i,1}$==0) && (cum_q!= Ø)
           $cr\_flag_{i,1}$=1
           Send CR_NOTIFY($n_{i,1}$) to all processes  in $allow\_r_{i,1}$
```

$n_{i,1}$ receives G_REQUEST (*$n_{j,1}$, g, session*)

```
If ($state\_c_{i,1}$=HIPT)
   If (session=session_pt[j] )
     $RS_{i,1}$=$n_{j,1}$ ;    session_pt[j] ++;     cum_q= Ø;   allow_sec=
Ø
    Send P_TOKEN (g, allow_sec, session_pt, cum_q)    to $n_{j,1}$
Else if ($state_{i,1}$=HPT)
    If (session=session_pt[j])
      If ($mygroup_{i,1}$==g) && ($cr\_flag_{i,1}$=0)
         $allow\_r_{i,1}$=$allow\_r_{i,1}$ ∪ $P_{j,1}$;  session_pt[j]++
         Send S_TOKEN ($n_{i,1}$, g ,session_pt[j], $crflag_{i,1}$)    to $n_{j,1}$
      Else
         Add request of $n_{j,1}$ in cum_q if not already there
         If  ($cr\_flag_{i,1}$==0)
             $cr\_flag_{i,1}$=1; Send CR_NOTIFY($n_{i,1}$) to all
processes  in $allow\_r_{i,1}$
Else if ($state\_c_{i,1}$=HST)
    If ($mygroupi_{,1}$!=g) && ($cr\_flag_{i,1}$==0)
       $cr\_flag_{i,1}$=1;  Send CR_NOTIFY to $myprimaryi_{,1}$
Else
    If    $(n_j \notin RS_{i,1})$
       $RS_{i,1}$=$RS_{i,1}$ ∪  $n_{j,1}$
If ($local\_q_{i,1}$!= Ø)
          if g’ is the first group in $local\_q_{i,1}$   Send
G_REQUEST ($n_{i,1}$, g) to $n_{j,1}$
```

<u>$n_{i,1}$ receives CR_NOTIFY (nj,1)</u>

```
 If ($state\_c_{i,1}$==HPT)
    If ($cr\_flag_{i,1}$=0)
       $cr\_flag_{i,1}$=1; Send CR_NOTIFY to all secondary token
holder other than $n_{j,1}$
If ($state\_c_{i,1}$==HST)
    If ($cr\_flag_{i,1}$=0)
       $cr\_flag_{i,1}$=1
```

<u>Proc Sel_next_coord ($n_{i,1}$)</u>

```
Add requests pending in local_q in cum_q  and add these
groups in $gr\_req_{i,1}$
If (cum_q ≠Ø)
    Let  $n_{j,1}$ is the coordinator whose request is at the front of
the queue and
   g is the group being requested by $n_{j,1}$.
    $n_{j,1}$ will be the new primary token holder and   g the group
selected for next session  .
    session_pt[j]++.;   allow_sec= Ø
   Append    all the   coordinators $n_{k,1}$ requesting for g ;
   add  these  in  allow_sec, and remove these requests from
cum_q
   Send P_TOKEN (g,cum_q, allow_sec, session_pt[j]) to $n_{j,1}$ .
   Add $n_{j,1}$ & other coordinators which can hold primary token
in future to $RS_{i,}$
Else  $state\_c_{i,1}$=HIPT;  $mygroup_{i,1}$=NULL
```