A Generalized Mutual Exclusion Problem and Its Algorithm

Aoxue Luo, Weigang Wu
Department of Computer Science
Sun Yat-sen University
Guangzhou 510006, China
luoaoxue,wuweig@mail.sysu.edu.cn

Jiannong Cao
Department of Computing
Hong Kong Polytechnic University
Kowloon, Hong Kong
csjeao@comp.polyu.edu.hk

Michel Raynal
IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
raynal@irisa.fr

Abstract—Mutual exclusion (ME) is a fundamental problem for resource allocation in distributed systems. It is concerned with how the various processes access shared resources in a mutually exclusive way. Besides the classic ME problem, several variant problems have been proposed and studied. In this paper, drawing inspiration from the scenario of controlling autonomous vehicles at intersections, we have defined a new ME problem, called Local Group Mutual Exclusion (LGME), where mutual exclusion is necessary only among the processes requesting overlap but not the same set of resources. In comparison with other variant problems of ME, LGME is more general but also more challenging. To solve the LGME problem, we propose a novel notion called strong coterie, which can handle the complex process relationship in LGME. Based on strong coterie, we have designed an ME algorithm, which can handle concurrent CS execution and message asynchrony. The correctness of our algorithm is rigorously proved.

Keywords—Mutual exclusion; quorum; coterie; resource allocation; distributed algorithm

I. INTRODUCTION

Mutual exclusion (ME) is one of the key problems in distributed systems to allocate resources shared by processes [18][19][27]. Originally, ME is defined as that[21][27][31], a set of processes intermittently requires access to an identical resource called the critical section (CS) and no more than one process can be in CS at any given time. Later, with the consideration of different application scenarios, several variants [2][5][6][8][15] have been proposed and studied. In general, these variants extend ME by specifying different modes of resource sharing. That is, the conflict in resource access may be necessary among only part of all processes, and some of them may even access the shared resource simultaneously. Recently, mutual exclusion has been studied under new environments, i.e. wireless and ad hoc networks [1][31].

In this paper, we consider ME in Vehicular Ad hoc Network (VANET) [13], which enables a vehicle to communicate with others and exchange messages via wireless links. With the development of autonomous vehicle (AV) [32], traffic control at intersections can be conducted via distributed coordination among vehicles rather than pre-deployed traffic lights, which turns out to be more flexible and efficient.

Such a coordination problem among vehicles can be viewed as an ME problem. The lanes at intersection can be viewed as resources shared by vehicles. Correspondingly, vehicles need to send requests to and get permission from other vehicles before they can proceed and pass the intersection. As we know, vehicles with crossing paths are conflicting with each other, and they must pass the intersection mutually exclusively. On the other hand, vehicles with non-crossing paths can pass concurrently without affecting each other. Moreover, AVs on the same lane can pass the intersection together (Although they need to proceed in the order as they arrive in practice, the resources can be allocated to them simultaneously).

An example scenario is shown in Fig. 1. Vehicles at Lane 0 and Lane 2 are conflicting and they must pass the intersection mutually exclusively. Vehicles V_{40} and V_{41} are at the same lane, so V_{41} should be able to pass following V_{40}, i.e. they can execute CS simultaneously. V_{00} is non-conflicting with vehicles at Lane 4, so it should not be blocked by V_{40}/V_{41}.

Based on the observation above, we define a new variant of ME, called Local Group ME (LGME), which means the mutual exclusion relationship is not global (i.e. among all processes in the system), and processes are assigned into different groups. Processes with the same group can share the access privilege to resource and processes with non-conflicting groups can execute CS simultaneously. V_{00} is non-conflicting with vehicles at Lane 4, so it should not be blocked by V_{40}/V_{41}.
passing the intersection, and the exact position and movement of vehicles have not been considered.

Compared with other ME variants, LGME is more general and more challenging. In fact, it is the generalization of the dining philosopher problem with the concept of group. Since two groups may not be conflicting with each other and a process may join different groups at different CS executions, the competition of CS in LGME is difficult to handle.

To solve the LGME problem, we propose a novel notion called strong coterie, which has intersection properties different from traditional quorum systems [16][21]. With strong coterie, we design a mutual exclusion algorithm for LGME. Two major issues have been addressed in the algorithm design. namely, 1) Achieving concurrency within the same group and between non-conflicting groups simultaneously. These two types of concurrency are similar but different operations must be considered. 2) Handling message asynchrony due to message delay. Message delivery order will affect the behavior of quorums significantly, and must be handled carefully.

In summary, the major contributions of this paper include:
1) Defining LGME, a new variant of mutual exclusion, and discuss the relationship among different variants. LGME is the generalization of several other ME problems.
2) Proposing strong coterie, a novel notion, which can be used to handle the complex relationship among processes in LGME. We also present methods to construct such coteries.
3) Designing a strong coterie based algorithm to solve the LGME problem. The correctness of the algorithm is also proofed.

The rest of this paper is organized as follows. In Section 2, we briefly review existing variants of ME, compare them with ours. Section 3 first specifies the system model and then formally defines the LGME problem. Especially, different concurrency properties are defined and discussed in this section. Section 4 presents the design of strong coterie and discusses some key properties. We describe the strong coterie based LGME algorithm in Section 5, prove its correctness in Section 6. Finally, Section 7 concludes the paper with scope for future work.

II. RELATED WORK

ME is originally defined for resource allocation problem, which requires that no more than one process can be in CS at any time. Several ME variant problems have been proposed, which considers different scenarios of resource sharing.

The k-ME problem [5][9][10][26] assumes that there are multiple copies (or units) of resources, each of which should be accessed mutually exclusively. Each process can request one unit of the shared resource at the same time and therefore at most k processes can be in CS simultaneously. The h-out-of-k-ME problem [22][24][25] is a generalization of k-ME, it allows every process to request h(1 ≤ h ≤ k) units of the shared resource at the same time.

Another variant of ME is the dining philosophers problem [7][8][19][28][29]. In the original version of dining problem[8], philosophers (processes) are arranged in a ring with one fork (resource) between each pair of neighbors. In order to eat (do work), a philosopher must have exclusive access to both of its adjacent forks. A more general version of the problem allows arbitrary number of neighbors for each philosopher. The drinking problem [6][12][23][30] is a dynamic variant of the dining problem, where there is a maximum set of resources that a process can request and each CS execution may request an arbitrary subset. A recent work [1] extends dining problem by considering ad hoc networks, where neighborhood is defined by the physical location and it is changed due to node movements.

In the group ME problem [4][15][17], the concept of “group” is introduced. The shared resource can be accessed in different sessions. Processes may request different sessions to complete their tasks. The processes of the same session constitute a group, and they are allowed to access a shared resource simultaneously. On the other hand, processes in different groups have to compete for the resource and access it mutually exclusively.

Although different variants have different requirements on mutual exclusion, they all in fact try to generalize the relationship among processes. Especially, LGME is the generalization of the Dining problem and group ME problem. The Dining problem is an LGME with only one process in each group. The GME problem is a special case of LGME with global conflicts among groups. Hence, LGME is more general than others and consequently more difficult to solve.

III. PROBLEM DEFINITION

A. System Model and Definition of LGME Problem

It is assumed that the system contains n processes, denoted by \{p_1, p_2, ..., p_n\}, and the processes will not fail. Processes communicate with each other by exchanging messages. There is one communication channel between each pair of processes and it has the FIFO property, i.e. messages via the same channel are delivered in the same order as they are sent. The timestamp of any event (including sending, receiving, or internal step) is globally unique and comparable with others.

There are totally number m of tasks to be intermittently carried out by processes and each task is associated with a certain subset of all r resources. A process may be requested to execute different tasks at different times, and it will not get any new request if the previous one has not been executed.

The processes that execute the same task constitute a group. There can be totally m groups, denoted by \(G=\{g_1, g_2, ..., g_m\}\), and a process’s group is dynamically determined by its current task.

![Fig. 2. Execution states of a process](image-url)
Any resource must be accessed mutually exclusively by processes from different groups, but processes in the same group can access the same resource simultaneously. Then, processes with overlapping resource sets have to access CS mutually exclusively while processes of the same group can execute CS simultaneously. A process knows the existence of $m$ groups and can determine which group it belongs to upon task assignment by users or upper layer programs. Then, the execution of one process follows a cycle of three phases as shown in Fig. 2. Initially, all processes in the system are all in the Idle state. Upon request generation, a process will be assigned to some group. It switches its state to Wait and execute the trying section to get permission of entering CS.

Based on the sets of resources associated with different groups, the following relationships can be defined:

**Conflicting groups/processes.** Groups/Processes with different but overlapping resource sets are said to be conflicting with each other. Processes from conflicting groups cannot be in CS simultaneously.

**Mate process:** processes in the same group are mates of each other.

**Rival process:** processes in conflicting groups are rivals of each other.

Moreover, the overall conflicting relationship among different groups can be represented by a group conflict graph, which is defined below.

**Group conflict graph:** a graph $(G, E)$, where $G$ is the set of groups and $E$ is the set of edges. An edge refers to two adjacent groups that are conflicting with each other. Conflict graph reflects the competitive relationship among different groups of processes.

The picture on the right in Fig. 1 shows the group conflict graph of the intersection scenario shown on the left.

More precisely, the following properties must hold so as to guarantee that LGME is correctly solved.

- **Mutual Exclusion (safety):** No two conflicting processes can be in CS at the same time.
- **Deadlock Free (liveness):** If a process is waiting for CS, then in a finite time some process enters CS;
- **Starvation Free (fairness):** If a process requests to enter CS, a finite number of CS requests can be met before it can enter CS.
- **Bounded Exit:** Each process in CS eventually exits within a bounded number of its own steps.

### B. The Concurrency Properties

Besides above properties for correctness, the concurrency among processes in CS execution should also be considered. Concurrency refers to what degree processes can execute CS concurrently, so that the resources can be used more efficiently and tasks can be completed more quickly. Concurrency is generally complementary to the mutual exclusion property. Concurrency among processes in the same group (intra-group) has been discussed in [15][34].

In LGME, however, besides mate processes and rival processes as in group ME problem, there are also non-conflicting processes. That is, concurrent CS executions are possible among both mate processes and non-conflicting processes. Correspondingly, we define both intra-group concurrency and inter-group concurrency below.

**Intra-concurrent occupy:** if a process $p_i$ requests to enter CS and no processes in a different group request CS, then $p_i$ will eventually enter CS.

**Intra-concurrent entering:** if a process $p_i$ requests to enter CS and no processes in a different group request CS, then $p_i$ will enter CS within a bounded number of its own steps.

As opposed to intra-concurrent occupy, intra-concurrent entering requires that the processes of the same group should be able to enter CS without unnecessary synchronization if no processes of other group request CS.

**Inter-concurrent occupy:** if a process $p_i$ requests to enter CS and no processes in a conflicting group request CS, then $p_i$ will eventually enter CS.

**Inter-concurrent entering:** if a process $p_i$ requests to enter CS and no processes in a conflicting group request CS, then $p_i$ will enter CS within a bounded number of its own steps. The Inter-concurrent entering requires processes from non-conflict groups to not block each other.

From the definitions, we can see that concurrency is increased from concurrent occupy to concurrent entering. That is, concurrent entering requires higher concurrency than concurrent occupy. Concurrent occupy is in fact implied by the correctness properties of liveness and fairness.

### IV. The Strong Coterie System

#### A. Preliminaries on Quorum

A quorum is a subset of nodes or processes. Although nodes and processes are usually used interchangeably, following the convention in [16], we use the term “node” specifically to refer to a process as a quorum member.

A quorum system is a collection of sets $S = \{S_1, \ldots, S_m\}$, which has the intersection property. Moreover, a quorum system with the minimality property is called a *coterie* [2][3][11][14].

- **Intersection:** $\forall P, Q \in C :: P \cap Q \neq \emptyset$
- **Minimality:** $\forall P, Q \in C, P \neq Q :: P \preceq Q$

Quorum has been studied and applied in various problems, such as mutual exclusion, data replication protocols, name servers, selective information dissemination, and distributed access control. In mutual exclusion algorithms, when a process wants to enter CS, it first needs to get permission from all nodes in a quorum. On the other hand, a node in coterie can grant the permission to CS to no more than one process at any time. Then the intersection property ensures that no two processes can execute CS simultaneously. The minimality property ensures that no process is required to lock more nodes than necessary to achieve mutual exclusion.
Some existing algorithms for GME use a special type of quorum system called group quorum [16], where any two quorums belonging to the same group need not be intersecting, whereas quorums belonging to different groups must be intersecting. The $k$-coterie, proposed for $k$-mutual exclusion, requires any $(k+1)$ quorums exist pairwise disjoint quorums [10]. The $k$-arbiter [22], proposed for $h$-out-of-$k$ problem, is also a quorum system which requires that any $k+1$ quorums must have a non-empty common intersection. Existing quorum systems, including group quorum and $k$-arbiter, can handle none or only intra-group concurrency and cannot be used to solve LGME.

B. Definition of Strong Coterie

To solve the LGME problem, we propose a new quorum system called strong coterie which only concerns the processes from a conflicting group pair. That is, each coterie is responsible for the processes of a pair of conflicting groups in the group conflict graph.

Then, when a process $p_i$ requests to enter CS, it needs to send CS request to quorum associated with $p_i$ in each coterie according to the group conflict graph. After permission is granted by all these quorums, $p_i$ can enter CS (Please notice that quorums in different coteries are operated independently and isolatedly).

The coterie $C$ needs to satisfy the following properties so that it can be used to solve the LGME problem.

(P1) 3-intersection: for any three distinct quorums $Q_i, Q_j, Q_k$ in $C$: $Q_i \cap Q_j \cap Q_k \neq \emptyset$

(P2) 4-nonintersection: for any four distinct quorums $Q_i, Q_j, Q_k, Q_l$ in $C$: $Q_i \cap Q_j \cap Q_k \cap Q_l = \emptyset$

(P3) Equal-intersection: for any four quorums $Q_i, Q_j, Q_k, Q_l$ in $C$ and $Q_i \neq Q_j \neq Q_k$: $|Q_i \cap Q_j| = |Q_k \cap Q_l|$

With process groups, the permission to access CS should be granted according to the highest priority in each group, where priority is determined based on the timestamp of requests. P1 is used to guarantee that, for each process $p_i$, at least one node is simultaneously associated with $p_i$ and two other special processes $p_x$ and $p_y$, i.e., the processes with the smallest timestamp in $p_i$’s own group and the one from $p_i$’s rival group, respectively. Then, this node can get enough knowledge about the highest priority in different process groups and accordingly determine which group should be granted permission.

P2 and P3 are supplementary to P1. With P2, the node associated with $p_x, p_y$, and $p_i$ will not serve any other processes, so that it will not be misled by other processes. P3 is used to check the validity of a process or request, as defined below.

Valid request process: for two processes $p_i$ and $p_j$ requesting CS, and their corresponding quorums $Q_i$ and $Q_j$, $p_i$ or $p_j$’s request is valid for $p_i$, if and only if $p_j$’s request information (timestamp) is included in the messages from all nodes in $Q_i \cap Q_j$ to $p_i$.

With P3, the validity of another process or request is checked by counting the number of messages carrying the request information. To make such counting workable, each process should be assigned one quorum and the size of the intersection of any two quorums should be deterministic.

Please notice that P1 and P2 are concerned with the intersection of multiple quorums, which is different from the (non-) intersection property in quorum for $k$-mutual exclusion, where only pairwise quorum intersection is considered. Moreover, P2 makes our coterie different from $k$-arbiter.

C. Construction of Strong Coterie

A coterie $C$ with $n$ quorums, denoted by $C(n) = \{Q_1, Q_2, \ldots, Q_n\}$, that satisfies properties $P_1$, $P_2$, and $P_3$ can be easily constructed below.

1) Initially, each quorum is set to be empty.
2) For each triple of quorums, add one unique node as their member.
3) The resultant quorums constitute a correct coterie.

| TABLE I. AN EXAMPLE COTERIE WITH FOUR QUORUMS |
|------------------|------------------|------------------|------------------|
| $Q_1$            | $n_1$            | $n_2$            | $n_3$            |
| $Q_2$            | $n_1$            | $n_2$            | $n_3$            |
| $Q_3$            | $n_1$            | $n_2$            | $n_3$            |
| $Q_4$            | $n_3$            | $n_2$            | $n_3$            |

The coterie $C$ constructed above satisfies all the three properties. An example coterie with four quorums is shown in Table I.

Now, let us introduce two interesting properties of $C$, which are crucial to our algorithm design later.

Lemma 1. The total number of nodes in $C(n)$ is $C_n^m$; for any quorum $Q$ in $C$, $|Q| = C_{n-1}^1$.

Proof. The lemma trivially holds.

Lemma 2. For any two quorums $Q_i, Q_j$ in $C(n)$, we have $N = |Q_i \cap Q_j| = |C|-n-2$.

Proof. The proof can be done by induction on $|C|$. Due to the limit in space, we present only the proof sketch.

Basis: if $|C|=4$, the lemma holds as shown in Table I.

Inductive step: If the lemma holds for $C(k)$, then it holds for $C(k+1)$. To construct $C(k+1)$ from $C(k)$, we add one new node for each pair of quorums $C(k)$ and also add the node to the new quorum $Q_{k+1}$. The lemma also holds.

Corollary 1. With a coterie $C$ constructed above, a request or process is valid to another process $p_i$ if the request is received by $p_i$ from $|C|-2$ nodes.

Proof. Combine the definition of valid process and Lemma 2.

D. Virtualization of the Strong Coterie

With the strong coterie defined and constructed above, in a system with $n$ processes, a coterie $C(n)$ should be constructed...
for each conflict group pair, and \( C_n \) nodes are needed. In most cases, this is quite a large number compared to the number of processes to be served. To reduce the number of quorum nodes needed, we can virtualize a strong coterie to multiple logical instances.

Only one coterie is really constructed and deployed. The coterie plays as multiple coteries logically, each of which is assigned to one edge of the conflict graph, i.e. a pair of conflicting groups. Each instance is a virtual coterie and acts independently and isolatedly. With such virtual coteries, no matter how many groups are there in the system and how they conflict with each other, only one real coterie is needed. Of course, the real coterie should know the group conflict graph so that it can fork enough virtual coteries.

V. THE LGME ALGORITHM

A. Overview of the Algorithm.

When a process needs to enter CS, it will first send a request to one quorum in each coterie associated, according to the group it belongs to. The group of a process is determined by the task it needs to execute, and accordingly the coteries to contact can be determined by the conflict graph. The timestamp of the request is used as the priority of the process and is carried by the request message.

The quorum nodes receive requests from processes and determine which one should be granted the permission of CS according to the conflicting relationship and the timestamp of requests. To achieve intra-group concurrency, permission is granted to the process with the least timestamp and its mate processes. A process can enter CS after it gets permission from all coteries associated. The permission is released after a process exits from CS.

Different from existing quorum system for ME, a quorum in our algorithm replies requesting processes with timestamps collected rather than the explicit permission of entering CS. Such a change is mainly to address the following two issues.

1) To achieve both starvation-free and intra-group concurrency. Inter-group can be achieved by the design of strong coterie, because only conflicting group pair is assigned with a coterie. Intra-group concurrency has to be handled by the LGME algorithm itself. To do so, we allow a node obtain privilege by following its mate process with a higher priority. However, this may cause starvation if any process is allowed to enter CS following its mate. Then, special mechanism is designed to determine what processes can follow its mate.

2) To handle message asynchrony caused by varied message delay. Due to asynchrony in message delivery, a request with smaller timestamp may be delivered later than one with larger timestamp. Existing quorum systems handle request disorder by taking its permission back from the process whose request arrives earlier but with lower priority. However, in LGME, permission call back will involve a group of processes rather than one, which is much more complex.

B. Variables and Message Types

1) Variables at process \( p_i \)

\( pstate; \) the state of \( p_i \); \( WAIT/EXE/IDLE \), which means waiting for CS, executing CS, and no request, respectively.

\( ts_i; \) the timestamp of the current CS request of \( p_i \). The smaller a timestamp is, the higher priority a process has.

\( ta/n_i; \) the smallest timestamp of valid processes of \( p_i \)’s mate/rival processes to the best of its knowledge.

\( nodelist[]; \) the set of quorum nodes which \( p_i \) needs to send request to and get acknowledgement from for entering CS.

\( matelist[]/rivalist[]; \) the list to record the processes in the same/conflicting group requesting to enter CS. Each element is a triple as \((p_j, ts_j, nlst)_j\), where \( ts_j \) is the timestamp of \( p_j \) while \( nlst \) is the list of nodes from which \( p_j \) has obtained \( p_i \)’s request information. The list is dynamically changed based on messages received by \( p_i \).

2) Variables at quorum node \( n_i \)

\( nstate; \) the state of \( n_i \); CERTAIN/UNCERTAIN. Normally, \( n_i \) is in the CERTAIN state. When a request with higher priority arrives later than one with lower priority, \( n_i \) needs to update processes with new request information and its state is set to be UNCERTAIN during the update.

\( locklist[]; \) the list of processes to which \( n_i \) has granted permission. As we know, these processes are in the same group. Each element is a triple as \((p_j, ts_j, grants)_j\), where \( ts_j \) is the timestamp of \( p_j \) while \( grants \) is the timestamp of used to grant permission to \( p_j \). If \( grants \) is not equal to \( ts_j \), it should be a timestamp of some mate with higher priority.

\( lockedgrp, ts, grants; \) the variable storing the information of permission granted, where \( grp \) is the group id of the processes in locklist, \( ts/grants \) is the least value of locklist[j].ts/locklist[j].grants.

\( pendlist[]; \) the list of processes whose requests are pended. Each element is a triple as \((p_j, ts_j, ts_i)_j\), where \( ts_j \) is the timestamp of \( p_j \) and \( ts_i \) is the least timestamp in rival group. \( tsupd; \) the timestamp of the request that causes \( n_i \) switches to UNCERTAIN and sends \( UPD \) messages.

\( updlist; \) the list of processes to which \( UPD \) message is sent.

3) Message types

\( REQ(p_i, g_i, ts_i); \) the message from a process \( p_i \) to its quorum nodes to request permission of entering CS, where \( g_i \) and \( ts_i \) are group id and timestamp request respectively.

\( ACK(flg, locklistn); \) the acknowledgement message from quorum nodes to process corresponding to a request message. The parameter \( flg \) is a Boolean value that indicates it is a positive/negative acknowledgement, and \( locklistn \) is the locklist at the sender node.

\( REL(ts_i); \) the message from a process to quorum nodes to release the permission.

\( ERE(flg, plist); \) The message from a quorum node to pended processes to release a previous permission and grant a new one. The parameter \( plist \) is the list of processes newly granted while \( flg \) indicates whether the receiver is granted.

\( UPD(p_i, ts_i); \) the update message from a quorum node to update processes with a smaller timestamp upon request disorder.

\( UPA(ts_i); \) the message from a process to acknowledge \( UPD \).
--/Code for a process p/*)--

P1: On generating CS request
101) pstate←WAIT;
102) send REQ(p, g, ts) to nodes in nodelist;
103) for eachp, in locklist {for each
104) if(matelist[p].ts = NULL)
105) to,min(to,locklist[p].ts);
106) }

111) if(matelist[p].nlst = N)
112) add (matelist[p].ts, ts);
113) }
114) pstate←EXEC;
115) enter CS;

P2: On receiving ACK(fig, locklist) from node n
120) nstate←CERTAIN;
121) sendREL(ts)
122) for eachp, in locklist {
123) update matelist using (p, locklist[p].nlst);
124) if (matelist[p].nlst = N)
125) to,min(to,locklist[p].ts);
126) }
127) nstate←UNCERTAIN; tsendp( ts);
128) updlst←locklist;
129) send UPD(ts, ts) to updlst;
130) }

P3: On receiving UPD(p, n) from node n
131) if(pstate=EXEC){
132) update rivalist with (p, ts);
133) nstate←NULL; pstate←IDLE;
134) }

P4: On receivingERE(fig, pilist) from nodes
135) delete from matelist[.nlst and rivalist[.nlst;
136) if(fig= true) add records in pilist to matelist;
137) else add records in pilist to rivalist;
138) to,min(to,matelist[.ts];
139) }
140) updlst←m
141) send ERE( true, locklist)
142) enter CS;

P5: On exiting CS
143) empty the list matelist;
144) empty the list rivalist;
145) sendRE[ts] to nodes in nodelist;
146) }
147) if(matelist[p].ts = NULL)
148) to,min(to,locklist[p].ts);
149) }
150) }
151) nstate←CERTAIN; pstate←NULL;
152) grantACK(p, ts);
153) }
154) }

--/Code for primitives/*)--

N1: On receiving REL(p, g, ts) from process p
203) if(nstate = CERTAIN ^ g = locked grp)
204) grantACK(p, ts);
205) }
206) }
207) nstate←CERTAIN; tsendp[ ts];
208) updlst←m
209) send UPD[ts, ts) to updlst;
210) }
211) grantACK(p, ts, locked.ts);

213) for eachp in locklist;
214) nstate←CERTAIN
215) }
216) add (p, ts, NULL) to pendlist;
217) send REL(p, ts, ts) to pendlist;
218) send ERE(true, locklist)
219) send ACK(false, locklist)
220) send ACK(true, locklist)
221) send ACK(false, locklist)

Fig. 3. Pseudo code of our algorithm

---/Code for a process p/*)--

P1: On generating CS request
101) pstate←WAIT;
102) send REQ(p, g, ts) to nodes in nodelist;
103) for eachp, in locklist {
104) if(matelist[p].nlst = N)
105) to,min(to,locklist[p].ts);
106) }

111) if(fig = false)
112) for eachp, in locklist {
113) update rivalist using (p, locklist[p].nlst);
114) }
115) if(matelist[p].nlst = N)
116) pstate←EXEC;
117) enter CS;

P2: On receiving ACK(fig, locklist) from node n
120) nstate←CERTAIN;
121) sendREL(ts)
122) for eachp, in locklist {
123) update matelist using (p, locklist[p].nlst);
124) if (matelist[p].nlst = N)
125) to,min(to,locklist[p].ts);
126) }
127) nstate←UNCERTAIN; tsendp( ts);
128) updlst←locklist;
129) send UPD(ts, ts) to updlst;
130) }

P3: On receiving UPD(p, n) from node n
131) if(pstate=EXEC){
132) update rivalist with (p, ts);
133) }
134) nstate←CERTAIN;
135) pstate←null;

P4: On receivingERE(fig, pilist) from nodes
136) delete from matelist[.nlst and rivalist[.nlst;
137) if(fig= true) add records in pilist to matelist;
138) else add records in pilist to rivalist;
139) to,min(to,matelist[.ts];
140) }
141) nstate←CERTAIN
142) sendREL(p, ts, ts) to pendlist;
143) send ERE(true, locklist)
144) send ACK(false, locklist)

P5: On exiting CS
145) empty the list matelist;
146) empty the list rivalist;
147) sendRE[ts] to nodes in nodelist;
148) }
149) if(matelist[p].ts = NULL)
150) to,min(to,locklist[p].ts);
151) }
152) }
153) nstate←CERTAIN; pstate←NULL;
154) grantACK(p, ts);
155) }
156) }
157) nstate←CERTAIN; pstate←NULL;
158) grantACK(p, ts, locked.ts);
159) }
160) }
161) nstate←CERTAIN
162) sendREL(ts)
163) for eachp, in locklist {
164) if(matelist[p].nlst = N)
165) to,min(to,locklist[p].ts);
166) }
167) nstate←CERTAIN; tsendp( ts);
168) updlst←m
169) send UPD[ts, ts) to updlst;
170) }
171) grantACK(p, ts, locked.ts);

213) for eachp in locklist;
214) nstate←CERTAIN
215) }
216) add (p, ts, NULL) to pendlist;
217) send REL(p, ts, ts) to pendlist;
218) send ERE(true, locklist)
219) send ACK(false, locklist)
220) send ACK(true, locklist)
221) send ACK(false, locklist)

Fig. 3. Pseudo code of our algorithm

---/Code for primitives/*)--

F1: grantsACK(p, tgrant)
301) if(nstate is not in locklist) add (p, tgrant, tgrant) to locklist;
302) locked.grp[ p's group;
303) locked.ts←locked.grants[ min(t ,locked.ts);
304) locklist[.grants[ min(t ,locklist[.grants[;
305) send ACK(true, locklist) to p;

F2: newGrant()
306) choose pV, from pendlist whose ts is the least;
307) move mate processes of pV, from pendlist to locklist;
308) for each pV in locklist:
309) if(locklist[ pV].grants ts is the least)
310) else move ERE(true, locklist)
311) for eachp in pendlist:
312) if(locklist[ pk].grants ts is the least)
313) else move ERE(false, locklist)
314) locked.ts←min(locked.ts[.ts);[.
315) locked.grants←locked.ts;
316) locklist[.grants[ locked.ts;
317) pendlist[.grants[ locked.ts;

Fig. 3. Pseudo code of our algorithm
C. Operations of Our Algorithm

The algorithm consists of two parts: operations at a process and operations at a quorum node. Fig. 5 shows the pseudo code of our algorithm. Because request disorder, i.e. a request with a smaller timestamp arrives at a node later than one with a larger timestamp, will cause additional operations, we describe the scenarios without request disorder and then present how to handle request disorder.

A process can enter CS only after it gets permission from all strong coteries associated. Since each coterie is treated isolatedly at a process, for the simplicity of presentation, here we describe the operations on one coterie associated with a process, and “CS entering” or “Enter CS” just means that the process is granted permission.

1) Request-Acknowledgement: the base case

The algorithm starts from a request to enter CS at a process \( p_i \) (Code P1). Based on the task to be carried out, it determines which group it belongs to and accordingly which coteries should be contacted. The request message \( \text{REQ} \) will be sent to the quorums associated with \( p_i \). Then, \( p_i \) waits for replies from quorum nodes.

When the request message \( \text{REQ}(p_i, g, ts) \) arrives at a quorum node \( n_i \), if \( n_i \) is in UNCERTAIN state (Line 201), which means that request disorder has occurred and \( n_i \) is in the procedure of updating grant information, \( p_i \)'s request will be put into \( n_i \)'s pendlist. This request will be handled when \( n_i \) switches to CERTAIN state (Line 224 or Line 244). Otherwise, if \( n_i \) is in CERTAIN state, two cases may occur.

Case 1) \( n_i \) has granted permission to processes in \( p_i \)'s rival group (Line 213). Then, \( p_i \) needs to wait until the permission is released by its rival group. So, \( n_i \) will put \( p_i \)'s request into pendlist, and send negative \( \text{ACK} \) to \( p_i \).

Case 2) The permission of \( n_i \) has been granted to \( p_i \)'s mate processes (Line 205) or none process (Line 202). Then, \( n_i \) will grant permission to \( p_i \) by putting \( p_i \) into locklist and sending positive \( \text{ACK} \) to \( p_i \). By carrying information from quorum nodes, \( p_i \) can get permission following its mate processes and intra-group concurrency is achieved. However, if one process can obtain permission by simply following any mate process granted, processes in one group may alternately enter CS and block the processes in their rival group forever.

To avoid such problem, one process can only follow the mate processes that is granted permission and with smaller timestamp than any rival processes being pended. That is, when the processes with higher priority than rival processes exit from CS, new request from a mate process cannot be granted even if other mate processes have been granted. Such handling is realized based on the data structure \( \text{locklist} \). Before granting permission to \( p_i \), \( n_i \) will check whether the smallest timestamp of processes in \( \text{locklist} \) is smaller than all rival processes in the pendlist (Line 204). If the result is “yes”, \( p_i \) can be granted permission by following its mate processes. Otherwise, \( p_i \) is blocked and its request will be put into the pendlist (Line 213).

On receiving \( \text{ACK}(\text{flg, locklistn}) \) which piggybacks the information of \( \text{locklistn} \) from node \( n_i \), it will first update its matelist (if flg equals to “true”) or rivallist (if flg equals to “false”) with \( \text{locklistn} \) carried by \( \text{ACK} \). Also, \( ta \) or \( tm \) will be updated. Please notice that, \( ta \) accepts only timestamp from valid process/request (Line 109). Once \( p_i \) gets \( \text{ACK} \) from each node in the quorum associated, \( p_i \) compares its \( ta \) and \( tm \). If \( tm \) is NULL or smaller than \( ta \) (Line 115), \( p_i \) perceives that it has been granted with permission from the coterie.

2) Update-Ack: the case of disordered request

Due to the asynchrony in message transmission, request messages may be delivered to different quorum nodes in different orders. Such message disorder may cause nodes in the same quorum have different views of request information at a given time. This will affect the operations of quorum nodes inconsistently. With our analysis of various scenarios of message disorder, two of them need to be specially handled.

Scenario 1). The late request message at \( n_i \) is from a process \( p_i \) in the group holding permission granted by \( n_i \) (Line 206). \( n_i \) switches to UNCERTAIN and conducts timestamp update to let other processes know the new information, and \( n_i \) now becomes UNCERTAIN. The rivals are informed via \( \text{UDP} \) messages (Line 211), while mate processes will be informed via \( \text{ACK} \) messages (Line 236). Moreover, rivals should be updated first and then the mates, which is necessary to guarantee the safety property as shown in correctness proof section.

When a process \( p_i \) receives the \( \text{UDP} \) message from node \( n_i \), if \( p_i \) is not in CS, it will update its rivalist and \( tm_i \) and acknowledge the \( \text{UDP} \) message via a \( \text{UP4} \) message (Code P3). If \( p_i \) is already in CS, it will ignore the \( \text{UDP} \) message.

When all \( \text{UDP} \) messages are acknowledged, update to rivals is completed and \( n_i \) switches to CERTAIN (Line 226). Then, \( n_i \) will try to grant permission to \( p_m \) the process with the smallest timestamp (Line 226), and its mate processes (Line 232-237).

An exception may occur if \( p_m \)'s timestamp is even smaller than that of the process which has caused the previous update, i.e. a new request disorder occurs, and \( p_m \) is in the same group as that process (Line 227). A new round of update will be conducted (Line 228-230) before \( p_m \) and its mate are granted. If \( p_m \) is in the rival group of the process which has caused the previous update, request disorder of \( p_m \) needs no special handling, because \( p_m \)'s rivals must have not received \( \text{ACK} \) from \( n_i \) and they cannot enter CS before they receive the \( \text{ACK} \) with \( p_m \)'s timestamp sent at Line 237. Please notice that consecutive updates will not occur endlessly, because for any given request, the number of possible disordered requests is finite.

Scenario 2) The late message is from a process in the group currently blocked by \( n_i \) (Line 215). In such scenario, permission has been granted to processes with lower priority. The new request will be pended (Line 216), and the processes already granted will be updated with the new request from rival group (Line 217-219). Then, the \( \text{UDP} \) message will be handled the same way as in Scenario 1).
3) Exit and Release

When process \( p_i \) finishes CS execution (Code P5), it will reset its variables and send REL message to associated quorum nodes, i.e. nodes in \( \text{nodeList} \). Upon receiving a REL message (Code N3), \( n_i \) will delete \( p_i \) from locklist or pendlist.

If the REL message is from a process \( p_i \) in updlist and \( n_i \) is in UNCERTAIN state (Line 239), \( p_i \) must have received UPD from \( n_i \) when it is in CS. Such REL message will be handled the same way as an UPD (Line 240).

If \( p_i \) is in locklist and the locklist becomes empty after \( p_i \) is removed (Line 241), new grant will be issued (Line 245). If \( n_i \) is UNCERTAIN, it needs to remove the update record for \( p_i \) (Line 242-244) before the new grant of permission.

The operations of issuing new grant by a node \( n_j \) are shown in Code F2. Permission will be granted to \( p_{ij} \), the process with the smallest timestamp, and its mate processes. Node \( n_i \) will send different messages to nodes in locklist. It \( p_i \) has never been acknowledged before (Line 309), \( n_i \) sends an ACK message; otherwise, an ERE message will be sent to \( p_i \) (Line 310) so as to erase the information of a negative ACK previously sent. Besides processes in locklist, node \( n_i \) will also send message to each process in pendlist similarly, except that the message is negative (Line 312-313).

When a process \( p_i \) receives the message ERE(\( flg, plist \)) from \( n_i \) (Code P4), it will first remove the request information ever received from \( n_i \). Then, if \( flg \) is true, \( p_i \) is granted permission by \( n_i \) and nodes in \( plist \) will be added to its mateList (Line 123). Otherwise, \( p_i \) is pended by \( n_i \) and rivalist is updated using information in \( plist \) (Line 124). Finally, \( p_i \) will enter CS after it gets new information with ERE (Line 127-129).

VI. CORRECTNESS PROOF

Bounded exit is trivial and we only prove the other three correctness properties.

\( C_q \): the coterie associated with an edge between group \( i \) and group \( j \) in conflict graph. Obviously, \( C_q \) is equal to \( C_{ij} \).

Complete process: a process \( p_i \) is called a complete process for a coterie \( C_q \) if \( p_i \)'s request has been received by all nodes in a quorum of \( C_q \).

Grant of permission: a process \( p_i \) is said to be granted permission from a coterie \( C_q \) if \( t_a-t_m \) or \( t_n = \text{NULL} \) after \( p_i \) receives ACK from all nodes in a quorum of \( C_q \).

A. Proof of mutual exclusion

Lemma 3. If process \( p_i \) in group \( i \) is granted by a coterie \( C_q \) with \( t_n=\text{NULL} \), then no process in group \( j \) can get permission from \( C_q \) before the permission is released or taken back via UPD messages.

Proof. The proof is by contradiction. Assume there is some process \( p_j \) in group \( j \) that is granted by a quorum \( Q_j \) in \( C_q \) before \( p_i \) releases its permission. By the 3-intersection property, any two quorums must have a non-empty intersection, each node in \( Q_i \cap Q_j \) must have received requests from \( p_i \) and \( p_j \). If \( p_i, p_j \) must have \( t_a-t_m \) when it receives ACK from all nodes in \( Q_j \), because \( p_i \) should not receive positive ACK from \( Q_j \).

Without loss of generality, we assume the timestamp \( t_{a_i} \) and \( t_m \) are from process \( p_i \) and \( p_j \), respectively, then:

\[ t_a < t_{a_j} \]

Assume \( t_{a_j} \) is the timestamp with which some node in \( S \) has granted permission to \( p_j \). The request with \( t_{a_j} \) must arrived at all nodes in \( Q_i \cap Q_j \) (Line 109), so the nodes in \( Q_i \cap Q_j \cap Q_k = S \) receives both \( t_{a_j} \) and \( t_{a_i} \). And the ACK message arrives at \( p_j \) has the information of \( t_{a_j} \). Since the least negative timestamp received by \( p_i \) is \( t_{a_i} \) (Line 214), we have

\[ t_{a_i} < t_{a_j} \]

Together (1) with (2), conclude that: \( t_{a_i} < t_{a_j} \). Since \( p_i \) has got positive ACK from nodes in \( S \), with \( t_{a_i} < t_{a_j} \), we can conclude that, \( p_i \)'s request must arrive earlier than \( p_j \)'s at nodes in \( S \) (Line 215). Then, these nodes should send UPD to \( p_i \). Then two possible cases may occur. 1) \( p_i \) received no acknowledgement before (Line 309), \( n_i \) sends an ACK message; otherwise, an ERE message will be sent to \( p_i \) (Line 310) so as to erase the information of a negative ACK previously sent. Besides processes in locklist, node \( n_i \) will also send message to each process in pendlist similarly, except that the message is negative (Line 312-313).

Lemma 4. If process \( p_i \) is granted by a coterie \( C_q \) with \( t_a-t_m \), then no process in group \( j \) can get permission from \( C_q \) before the permission is released or taken back via UPD messages.

Proof. The proof is by contradiction. Assume there is some process \( p_j \) in group \( j \) that is granted by a quorum \( Q_j \) in \( C_q \) before \( p_i \) releases the permission.

Without loss of generality, let \( p_{i_1}, p_{i_2}, p_{j_1}, p_{j_2} \) denote the processes from which \( p_i \) and \( p_j \) get the timestamps \( t_{a_i}, t_m, t_{a_j}, t_m \), respectively. Then, \( p_i \) and \( p_j \) are valid for \( p_i \) and \( p_j \), respectively. Also,

\[ i) t_{a_i} < t_{a_j} \]

\[ ii) t_{a_i} < t_{a_j} \]

By the 3-intersection property, i.e.

\[ Q_i \cap Q_j \cap Q_k = \emptyset \neq \emptyset \]

Since \( p_i \) is valid for all nodes in \( S \) and \( p_j \) is in locklist. Considering \( t_{a_j} \) is the least timestamp got by \( p_j \), we have \( t_{a_j} \leq t_{a_i} \) (Line 214). Combing inequality i) we have:

\[ iii) t_{a_i} \leq t_{a_j} \]

Similarly, from \( Q_i \cap Q_j \cap Q_k = \emptyset \neq \emptyset \) we can get \( t_{a_j} \leq t_{a_i} \). Combing inequality ii), we have:

\[ iv) t_{a_j} \leq t_{a_i} \]

Since two inequalities iii) and iv) are contradictory with each other, the lemma holds.

Theorem 1. No two conflicting processes can be in CS at the same time.

Proof. By Lemma 3 and Lemma 4, one coterie grants permission to at most one group at any given time. Therefore, for each pair of conflicting groups, at each give time, only
processes from the same group can be in CS. Since each process cannot enter CS unless it has permission granted from the coterie at each edge in group conflicting graph, at most one group in a conflicting group pair can get permission granted at any time. The theorem holds.

\[ \Box \]

B. Proof of deadlock freedom

**Theorem 2.** If a process is waiting for the CS, then in a finite time some process enters the CS.

**Proof.** If no waiting processes can enter CS forever, a deadlock must occur. The proof is by contradictions.

Assume a deadlock occurs. A deadlock means that, there is a cycle of processes and each process is waiting for its predecessor to release the permission, no process is currently in CS and all messages sent out have been delivered.

Now, let us examine the process \( p_w \), which is with the least timestamp among all processes in the waiting circle. Assume \( p_w \)’s predecessor is \( p_x \), i.e. \( p_w \) is waiting for \( p_x \) to release the permission. It is easy to derive that, \( p_x \) and \( p_i \) are in different groups. Also, by the assumption of \( p_w \)’s \( ts_w \leq ts_x \).

By the 3-intersection property, the intersection of any two quorums is also not empty. Then, we have \( Q_w \cap Q_x \neq \emptyset \). From the assumption of deadlock, both \( p_w \)’s and \( p_x \)’s request has been delivered to each node \( n \) in \( T \). The following proof is divided into two cases according to the order of the delivery of \( p_w \)’s and \( p_x \)’s requests to any node \( n \) in \( T \).

Case 1). The request of \( p_w \) is delivered earlier than that of \( p_x \). Then, \( n \) must sends positive ACK to \( p_w \) upon receiving \( p_w \)’s request. When \( p_x \)’s request arrives later, since \( ts_w < ts_x \), \( n \) should send negative ACK to \( p_w \), with \( p_w \)’s timestamp included.

Case 2). The request of \( p_x \) is delivered earlier than that of \( p_w \). When \( p_w \)’s request is delivered, \( n \) should have sent positive ACK to \( p_w \). Since \( ts_w < ts_x \), request disorder occurs and the update procedure will be carried out. UPD messages should be sent to all rival processes of \( p_w \), including \( p_x \). Since no process is currently in CS, the UPD message should be acknowledged by all receivers immediately. Then, \( n \) should send positive ACK to \( p_w \).

In either of the two cases, \( p_w \) will receive positive ACK from \( n \). Considering \( n \) is any node in \( T \), the coterie between \( p_w \) and \( p_x \) should grant permission to \( p_w \) instead of \( p_x \), which contradicts with the assumption that \( p_w \) is waiting for \( p_x \). The theorem holds.

\[ \Box \]

C. Proof of starvation freedom

**Theorem 3.** If a process requests to enter CS, a finite number of CS requests will be met before it can enter CS.

**Proof.** With our algorithm, a process \( p_i \) can get permission following its mate process \( p_j \) if \( p_j \’s \) priority is higher than \( p_i \’s \) and \( p_j \) has finished CS execution. In such case, \( p_i \) will get permission earlier than the situation without the help of \( p_j \). Therefore, we only need to prove that a process \( p_i \) can eventually enter CS even with its own priority.

Let us examine what requests can be met before \( p_i \) gets permission. As we know, all the requests with timestamp smaller than \( ts_i \) may be met earlier than \( p_i \’s \). The number of such requests is finite and static. A process \( p_i \) with timestamp greater than \( ts_i \) may also enter CS earlier than \( p_i \) in two cases. Case 1) \( p_i \) has a mate process \( p_j \) whose timestamp is less than \( ts_i \) and \( p_j \’s \) request arrives at some quorum node in \( Q \cap Q_j \) before \( p_i \’s \) release message arrives. Since the arrival time of \( p_j \’s \) release message is specific, only a finite number of processes can enter CS in such a way. Case 2) \( p_i \’s \) requests arrive at all nodes in \( Q \cap Q_j \) earlier than \( p_i \’s \). Since the arrival time of \( p_i \’s \) request at each node is specific, only a finite number of requests can be met in such a way. In summary, only a finite number of requests can be met earlier than \( p_i \’s \). The theorem holds.

\[ \Box \]

VII. CONCLUSION AND FUTURE WORK

Mutual exclusion is a key problem in distributed computing. Inspired by autonomous vehicle control at intersections, we define LGME, a new variant of ME, which is the generalization of the Dining problem and group ME problem. To solve LGME, we propose the novel quorum system, called strong coterie, based on which an algorithm is designed for LGME. Our algorithm achieves high concurrency in CS executions.

Since this is the first work on the LGME problem, a lot of effort needs to be put in further study. The current LGME algorithm is not so efficient with the repeated coterie construction for each conflict group pair and the message cost due to the communication with multiple coteries for one CS request. It is possible to merge messages to or from multiple quorum nodes that in fact played by one real node. However, this is not a trivial task because it in fact necessary to simultaneously handle different types of relationship among processes. Another interesting issue deals with handling process failures and message losses.

ACKNOWLEDGMENT

This research is partially supported by Guangdong Natural Science Foundation (No. S201201001670), Pearl River Nova Program of Guangzhou (No. 2011J2200088).

References


308


