A Token-Based Distributed Group Mutual Exclusion Algorithm with Quorums

Hirotugu Kakugawa, Member, IEEE, and Sayaka Kamei, Member, IEEE, and
Toshimitsu Masuzawa, Member, IEEE

Abstract—The group mutual exclusion problem is a generalization of mutual exclusion problem such that a set of processes in the same group can enter critical section simultaneously. In this paper, we propose a distributed algorithm for the group mutual exclusion problem in asynchronous message passing distributed systems. Our algorithm is based on tokens, and a process that obtains a token can enter critical section. For reducing message complexity, it uses coterie as a communication structure when a process sends a request messages. Informally, coterie is a set of quorums, each of which is a subset of the process set, and any two quorums share at least one process. The message complexity of our algorithm is \(O(|Q|)\) in the worst case, where \(|Q|\) is a quorum size that the algorithm adopts. Performance of the proposed algorithm is presented by analysis and discrete event simulation. Especially, the proposed algorithm achieves high concurrency, which is a performance measure for the number of processes that can be in critical section simultaneously.


I. INTRODUCTION

DISTRIBUTED mutual exclusion is one of fundamental problems for conflict resolution in distributed computing. The problem appears in many context such as access to shared object and allocation of shared resource, etc. Because it is a fundamental problem in distributed computing, many generalized problems have been proposed so far. In this paper, we consider the problem of distributed group mutual exclusion[1], which is a generalization of the distributed mutual exclusion problem such that only processes in the same group can enter critical section simultaneously. In other words, no two processes in different groups enter critical section at a time.

As an intuitive example, consider a multicast of a musical live concert on the Internet but network capacity is not enough to transfer more than one concert live. Each user is allowed to make a request to receive a concert live he/she prefers among many concert lives. Although any number of users can receive the same concert live at a time by the multicast technology, no two different concert lives can be multicast simultaneously by limitation of network capacity. The distributed group mutual exclusion problem models such a situation, and it is one of conflict resolution problems in distributed systems.

Concurrency and waiting time are important performance measures for distributed group mutual exclusion algorithm. Concurrency is the number of processes that are in critical section simultaneously, and higher concurrency is better. Waiting time is the time that a process must wait to enter critical section after it issues a request. Design of an algorithm that achieves high concurrency and small waiting time is non-trivial when processes in different groups make request simultaneously. Consider a situation that some processes are in critical section, and a process in the same group makes a request. If the request is granted immediately, concurrency is increased. On the other hand, requests by other processes in different groups must wait to be granted. The difficulty of algorithm design is the trade-off of concurrency and waiting time.

Many distributed group mutual exclusion algorithm have been proposed so far, and they are categorized as follows. (See Section VII for more detailed review.)

- Decentralized permission type [2], [3] — When a process wants to enter critical section, it sends request messages to some processes, and it enters critical section if it obtains permissions from some set of processes defined in advance. Typically, quorums[4] or its variant[3] is used to define a set of processes from which a process must obtain permission. (See Definition 2 for formal definition.)
- Privileged token type [5] — A virtual object called token is maintained by processes, and only a process that obtains a token may enter critical section. In the algorithm proposed in [5], a process sends request messages to all processes so that a request arrives at a process that holds a token.
- Hybrid type [6] — The first process to enter critical section obtains permission from some set of processes (decentralized permission type), and the process grants other processes in the same group to enter critical section by sending a token (privileged token type).

In this paper, we propose a new distributed algorithm TQGmx of privileged token type. Our algorithm also uses two classes of tokens, main-token and sub-token, as [5]. Different from [5], our algorithm uses coterie for communication structure to reduce message complexity. Such an idea is proposed by Mizuno, Neilsen and Rao [7] for (non-group) distributed mutual exclusion – a token is used as a privilege to enter critical section and coterie is used for propagation of requests. (In Section VII, technical description of the algorithm in [7] and ours are presented.)

Outline of our algorithm is as follows. When a process obtains the main-token, it notifies to processes in a quorum that it holds the main-token. When a process makes a request, it sends request messages to processes in a quorum. A request message received is forwarded to the holder of the main-token, eventually received by the holder of the main-token, and enqueued in the main-token.
Finally the request is granted and a token (main- or sub-token) is sent to the requesting process.

Main contribution of our algorithm is twofold:

- Flexible control mechanism of concurrency. Requests are aggregated in the main-token, and the holder of the main-token may control the order to grant requests in the queue. We propose a priority scheme that can control concurrency and waiting time.

- Less message complexity. The message complexity of our algorithm is 0 in the best case and \( O(|Q|) \) in the worst case, where \(|Q|\) is a quorum size that the algorithm adopts. When we adopt a coterie based on finite projective plane[8], we have \(|Q| \approx \sqrt{n}\).

This paper is organized as follows. In Section II, we describe the computational model assumed in this paper, and define the group mutual exclusion problem and coterie. In Section III, we present the proposed algorithm for the group mutual exclusion problem. In Sections IV and V, we show correctness and performance analysis of the proposed algorithm, respectively. In Section VI, we show simulation result of the proposed algorithm. In Section VII, related works are referred to, and performance of the proposed algorithm is compared with other algorithms. Especially, show simulation results of algorithms proposed in [6]. Finally, in Section VIII, we give concluding remarks.

II. DEFINITIONS

A. The computational model

In this paper, we assume that a distributed system consists of a set of \( n \) processes \( V = \{P_0, P_1, ..., P_{n-1}\} \) and a set of communication channels \( E \subseteq V \times V \). Each process has a unique identifier selected from a set of integers \( \{0, 1, ..., n - 1\} \). A distributed system is asynchronous, i.e., there is no common global clock. Information exchange between processes is done by asynchronous message passing. Each communication channel is FIFO, and each message sent is delivered within finite time. We assume that there is no upper bound on message delivery time. We assume that a system is error-free.

B. The group mutual exclusion problem

Let \( G = \{0, 1, ..., m - 1\} \) be a set of groups. Each process selects a group \( g \in G \), and makes a request for critical section entries. We model such a behavior of each process \( P_i \) as shown in Figure 1.

Formally, the problem of group mutual exclusion is defined as follows.

**Definition 1:** The group mutual exclusion problem is a problem to control execution of processes to satisfy the following two conditions.

- Safety: No two processes in different groups are in critical section simultaneously, and
- Liveness: Any requesting process eventually enters its critical section.

Note that this definition does not include any requirement for concurrent access by processes in the same group. We leave such a condition as a goodness measure of algorithms because some algorithms may have limited concurrency to achieve low message complexity (e.g., [9]).

C. Coterie

Coterie is defined formally as follows.

**Definition 2:** (Coterie [4]) Let \( U = \{P_0, P_1, ..., P_{n-1}\} \) be a set. A set \( C \) of subsets of \( U \) is a coterie under \( U \) if and only if the following three conditions are satisfied.

1) **Non-emptiness:** For each \( Q \in C \), \( Q \) is not empty and \( Q \subseteq U \).
2) **Intersection property:** For any \( Q, Q' \in C \), \( Q \cap Q' \) is not empty, and
3) **Minimality:** For any \( Q, Q' \in C \), \( Q \) is not a proper subset of \( Q' \).

An element of \( C \) is called a quorum.

Some examples of coterie are given below.

**Example 1:** (Majority coterie [4]) For any integer \( n \), let \( U = \{P_i : 0 \leq i < n\} \). A majority coterie \( C_{maj} \) under \( U \) is defined as follows.

- When \( n \) is odd, \( C_{maj} = \{Q \subseteq U : |Q| = (n + 1)/2\} \), and
- When \( n \) is even, \( C_{maj} = C_1 \cup C_2 \), where, for some \( P \in U \), \( C_1 = \{Q \subseteq U : P \notin Q \text{ and } |Q| = n/2\} \) and \( C_2 = \{Q \subseteq U : |Q| = (n/2) + 1 \text{ and } Q \not\subseteq P \text{ for any } Q' \in C_1\} \).

**Example 2:** (Grid coterie [8]) For integer \( \ell \), let \( U = \{P_{x,y} : 0 \leq x, y < \ell\} \). A grid coterie \( C_{grid} \) under \( U \) is \( \{Q_{x,y} : 0 \leq x, y < \ell\} \), where \( Q_{x,y} = \bigcup_{0 \leq i < \ell} \{P_{x+y,i}\} \cup \bigcup_{0 \leq j < \ell} \{P_{x,j+y}\} \). Size of each quorum is \( 2\ell - 1 = 2\sqrt{n} - 1 \), where \( n = |U| = \ell^2 \).

As a general concept, **quorum system** is defined as a set of quorums in which intersection and minimality properties may not be satisfied. Coterie is a quorum system in which the two properties are satisfied. Since many variants of quorum systems have been proposed so far[10], [11], [12], [3], [13], a quorum system for coterie is called an ordinary quorum system.

D. Performance measures

We define performance measures of a distributed group mutual exclusion algorithms.

**Definition 3:** Message complexity is the number of messages exchanged per request for critical section. **Maximum concurrency** is the maximum number of processes that can enter critical section simultaneously. **Waiting time** is the time between a process makes a request and it actually enters critical section, by assuming that each message transmission consumes 1 time unit and local computation time is zero. **Synchronization delay** is the time between all the processes in a group exit critical section and a process in another group enters critical section.

Usually, waiting time is measured when the system load is light, and synchronization delay is measured when the system load is heavy in such a way that different groups are requested by processes simultaneously [14], [6], [5].
III. THE PROPOSED ALGORITHM

In this section, we present a distributed algorithm TQGmx for the group mutual exclusion algorithm based on coterie. Let $C$ be a coterie under $V$, and $n$ be the total number of processes, i.e., $n = |V|$. Procedures Request and Release are shown in Figure 2, which are used to request for entry to and exit from critical section.

A. Outline

Our algorithm TQGmx uses two types of tokens, the main-token and sub-tokens. The main-token is maintained so that it is unique in the system. A sub-token is generated by the holder of the main-token, and the number of sub-tokens varies. A process obtains the main-token to enter critical section if there is no process in critical section. A process enters its critical section by receiving a sub-token in case other process in the same group is in critical section. A process that holds the main-token maintains the following two values:

- **Current group**, which is the group of processes that are currently in critical section, and
- **Group size**, which is the number of processes currently in critical section.

The rough sketch of the proposed algorithm is as follows.

1) When process $P_i$ wishes to enter critical section, it sends a request message to each process $P_j$ in a quorum, and it waits for a token (main-token or sub-token) to arrive.

2) When process $P_j$ receives a request message from $P_i$, it forwards the request to the holder, say $P_k$, of the main-token if $P_j$ knows such $P_k$. Otherwise, the request is buffered until $P_j$ knows such $P_k$.

3) When the holder $P_k$ of the main-token receives a request issued by $P_i$ and forwarded by $P_j$, the arrived request is, in any case, put into the queue of the main-token. Each request in the queue is granted according to the following rules.

- If no process is in critical section, the main-token is transferred by a token message to a requesting process.
- If some processes are in critical section already and the requested group is the same as the current group, $P_k$ sends a sub-token message to $P_i$, as long as there is no request in other group with higher priority. See subsection III-E for a priority scheme of the queue to guarantee liveness.
- Otherwise, the request is kept in the queue of the main-token. The main-token may be transferred to other process to grant the request of the process, and this may happen several times. The request of $P_i$ is eventually granted by a process that holds the main-token, say $P_l$, which may not be the same as $P_k$.

B. Local variables at each $P_i$

Each process $P_i$ maintains the following local variables:

- **mode**, represents current status of $P_i$, and its value is IDLE (not interested in critical section), TRYING (making a request for critical section entry), and INCS (process is in critical section).
- **tok_i** is the main-token object if $P_i$ holds it. Otherwise, its value is ⊥.
- **type_i** is the type (MAIN or SUB) of a token that $P_i$ holds when $P_i$ enters critical section. If $P_i$ has no token, its value is ⊥. When $P_i$ is not in critical section, its value is ⊥ even if $P_i$ holds the main-token.
- **ts_i** is the timestamp value for each request, which is incremented by one when $P_i$ makes a new request. Each request is identified by a pair of process name and timestamp value.
- **qs_i** is a quorum that $P_i$ uses.
- **grp_i** is the group name that $P_i$ is currently interested in. If $P_i$ is not interested in critical section, its value is ⊥.
- **holder_i** is process name that holds the main-token to the best knowledge of $P_i$. Its value is ⊥ if $P_i$ does not know.
- **home_i** is the process name to which $P_i$ should return a sub-token. Its value is ⊥ if $P_i$ does not hold a sub-token.
- **tmpQ_i** is a temporary queue for request items\(^1\) that $P_i$ receives to forward to the holder of the main-token.
- **acqs_i** is a set of processes $P_k$ such that $P_i$ sent an acquired message to $P_k$ but has not sent a leave message to $P_k$.
- **acks_i** is a set of processes $P_k$ such that $P_i$ is waiting for an ack message from $P_k$ as a reply of leave message to $P_k$.
- **leaving_i** is true if and only if $P_i$ is waiting for an ack message.

C. Structure of the main-token

The main-token contains the following data.

- **gName** holds the current group. If no process is in critical section, its value is ⊥.
- **gSize** holds the group size.
- **reqQ** is a queue of request items.
- **tsReq** is an array of timestamps (sequence numbers) of size $n$. For each $0 \leq i < n$, the value of $tsReq_i$ is the largest timestamp value of requests by $P_i$ that the main-token enqueued so far.

In case a request message is sent to each process in a quorum, more than one request messages for the same request may be delivered at the holder of the main-token. To enqueue each request only once into the queue of the main-token, $tsReq_i$ is used. The timestamp value of $P_i$ is attached to each request message, and a request of $P_i$ that arrived at the holder of the main-token is taken into consideration only when its timestamp is larger than $tsReq_i$. Otherwise, the request message is simply discarded.

\(^1\)A request item is a triplet of process name, timestamp and group, for a mutual exclusion request.
D. Description of the algorithm

Formal description of the proposed algorithm is shown in Figures 3, 4 and 5. Precondition (labeled “PC”) is attached for each handler and procedure which is satisfied on invocation. When a process, say \( P_i \), sends messages to processes in a quorum, it does not send a message to \( P_i \) itself to reduce message complexity. In description below, “\( P_i \) sends a message to quorum \( q_i \)” means that “\( P_i \) sends a message to each process in quorum \( q_i \) except \( P_i \) itself”.

We assume that each handler is executed atomically. That is, any message arrival and local event never interrupt execution of a handler. Message arrival events and local events are handled in the order of occurrence. That is, after execution of a handler finishes, a next handler is invoked. (Note that each handler does not contain any blocking operation.)

Operations \texttt{enqueue}, \texttt{dequeue} and \texttt{peek} are defined on queues. We assume that \texttt{reqQ} of the main-token is a priority queue based on priority to be discussed in subsection III-E. We assume that a \texttt{dequeue} followed by a \texttt{peek} always returns the same item as the one returned by the \texttt{peek} as long as queue contents are not changed after the \texttt{peek}.

1) System startup (line 1.1 – 1.15): When a system starts, each process selects a quorum\(^2\). Process \( P_0 \) plays a special role. It creates the main-token, initializes it, and sends an \texttt{acquired} message to each process in a quorum. \( P_0 \) maintains a set \texttt{acq} of processes to which \( P_0 \) sent an \texttt{acquired} message.

2) Making a request (line 2.1 – 2.14): Process \( P_i \) invokes procedure \texttt{requestEvent}. \( P_i \) increments its timestamp, and the request of \( P_i \) is sent. Our algorithm makes use of quorums to forward the request to the holder of the main-token with some optimization as follows.

- Case A1 (line 2.2–), When \( P_i \) holds the main-token: \( P_i \) acts as if it receives a request message from itself. First, \( P_i \) updates the \texttt{tsReq}[i] in the main-token, and enqueues the request into the queue of the main-token.

Then, \( P_i \) invokes procedure \texttt{handlePendingRequests} at line 2.5. If the request of \( P_i \) has priority, \( P_i \) enters critical section (line 4.7 or 4.25) in procedure \texttt{handlePendingRequests}. Otherwise, it waits for a message of token or subtoken type.

- Case A2 (line 2.6–), When \( P_i \) does not hold the main-token and \( holder_i = \perp \): Since \( P_i \) knows the holder of the main-token, it sends a request message directly to the holder of the main-token, and it waits for a message of token or subtoken type.

- Case A3 (line 2.8–), When \( P_i \) does not know the holder of the main-token: \( P_i \) sends a request message to quorum \( q_i \). In case \( P_i \in \{ q_i \), the request is enqueued into its local temporary queue \texttt{tmpQ}_i. Out-dated requests of \( P_i \) in \texttt{tmpQ}_i, if any, are deleted since they are no longer necessary. It waits for a message of token or subtoken type.

3) Exit from critical section (line 3.1 – 3.13): Process \( P_i \) invokes procedure \texttt{releaseEvent}. There are two cases depending on type of token it holds.

- Case B1 (line 3.1–), When \( P_i \) holds the main-token: It decrements group size by one. If all processes exit critical section, procedure \texttt{handlePendingRequests} is invoked to switch current group if there is a pending request.

- Case B2 (line 3.8–), When \( P_i \) holds a sub-token: \( P_i \) sends a release message to the holder of the main-token.

4) Procedure \texttt{handlePendingRequests} (line 4.1 – 4.30): In this procedure, pending requests are granted. Note that this procedure is invoked only when \( P_i \) holds the main-token. If \( P_i \) is waiting for any \texttt{ack} message, i.e., \texttt{leaving}_i is true, invocation of this procedure has no effect. Below, we explain when \texttt{leaving}_i is false.

Requests in the queue of the main-token are granted in the order of priority. If no process is in critical section, let \( P_j \) be the process with the highest priority in the queue (line 4.2). If

\begin{verbatim}
Fig. 3. Description of TQQGmx for \( P_i \) (1/3).
\end{verbatim}
procedure handlePendingRequests;
PC: (toki_1 ∉ ⊥)
4.1 if ¬leaving_i ∧ (toki_1 ∉ ⊥) ∧ ¬empty(toki_.reqQ)
4.2 (P_i, t, g) := peek(toki_.reqQ); // peek the top item
4.3 if (P_j = P_i) {
4.4 dequeue(toki_.reqQ); // discard the top item
4.5 toki_.gName := grp_; toki_.gSize := 1;
4.6 type_i := MAIN; mode_i := INC;
4.7 trigger event requestDone_i; // Enter CS
4.8 } else {
4.9 if (acqs_i ≠ 0) {
4.10 call beginTokenTransfer;
4.11 }
4.12 dequeue(toki_.reqQ); // discard the top item
4.13 send (token) to P_j; toki_i := ⊥; 4.14 } 4.15
4.16 4.17 while ¬leaving_i ∧ (toki_1 ∉ ⊥) ∧ ¬empty(toki_.reqQ) {
4.18 (P_i, t, g) := peek(toki_.reqQ);
4.19 if (g ≠ toki_.gName)
4.20 break;
4.21 dequeue(toki_.reqQ);
4.22 if (P_j = P_i) {
4.23 toki_.gSize := toki_.gSize + 1;
4.24 type_i := MAIN; mode_i := INC;
4.25 trigger event requestDone_i; // Enter CS
4.26 } else {
4.27 toki_.gSize := toki_.gSize + 1;
4.28 send (subtoken) to P_j;
4.29 }
4.30 }

on receipt of (token); toki:
PC: (toki_1 = ⊥) ∧ (type_i = ⊥) ∧ (toki_.gSize = 0)
∧ (home_i = ⊥) ∧ ¬leaving_i ∧ (acki_1 = 0) ∧ (acqs_i = 0)
5.1 toki_i := toki;
5.2 acqs_i := q ⊖ {P_i};
5.3 send (acquired_i P_j) to each P_j ∈ acqs_i;
5.4 type_i := MAIN; mode_i := INC;
5.5 toki_.gName := grp_; toki_.gSize := toki_.gSize + 1;
5.6 trigger event requestDone_i; // Enter CS
5.7 while ¬empty(reqQ_i) {
5.8 (P_i, t, g) := dequeue(reqQ_i);
5.9 if (toki_.tsReq_i[j] < t) {
5.10 toki_.tsReq_i[j] := t;
5.11 enqueue(toki_.reqQ, (P_j, t, g));
5.12 }
5.13 5.14 call handlePendingRequests;

procedure beginTokenTransfer;
PC: (toki_1 ∉ ⊥) ∧ (toki_.gSize > 0) ∧ ¬leaving_i
6.1 toki_.gSize := toki_.gSize − 1;
6.2 if (toki_.gSize = 0) {
6.3 toki_.gName := ⊥;
6.4 call handlePendingRequests;
6.5 }

on receipt of (request P_k, t, g);
PC: true
7.1 if (toki_1 ∉ ⊥) { – case C1
7.2 if (toki_.tsReq_i[k] < t) {
7.3 toki_.tsReq_i[k] := t;
7.4 enqueue(toki_.reqQ, (P_k, t, g));
7.5 call handlePendingRequests;
7.6 }
7.7 } else if (hoster_i ∉ ⊥) { – case C2
7.8 send (request P_k, t, g) to hoster_i;
7.9 }
7.10 else { – case C3
7.11 delete (P_k, +, ) from reqQ_i;
7.12 on receipt of (ack from P_k):
PC: (hoster_i = P_k)
8.1 hoster_i := ⊥;
8.2 send (ack) to P_k;

on receipt of (ack from P_k):
PC: (toki_1 ∉ ⊥) ∧ (toki_.gSize = 0) ∧ ¬leaving_i
∧ (acki_1 ≠ 0) ∧ (acqs_i ≠ 0)
9.1 leaving_i := true;
9.2 send (leave) to each P_j ∈ acqs_i (≠ q_i − {P_j});
9.3 acqs_i := 0; acki_i := q_i − {P_j};

on receipt of (leave) from P_k:
PC: (hoster_i = P_k)
10.1 hoster_i := ⊥;
10.2 send (ack) to P_k;

P_j = P_i (line 4.3–), P_i enters critical section by the main-token. Otherwise (line 4.8–), P_i transfers the main-token to P_j. If acqs_i = 0, the main-token is transferred immediately (line 4.13). If acqs_i ≠ 0, procedure beginTokenTransfer is invoked to schedule token transfer (line 4.10). In procedure beginTokenTransfer, the value of leaving_i is changed to true (line 10.1), and it remains true until P_i receives ack message from each process in q_i − {P_j}.

Then (line 4.17–4.30), if the main-token is not transferred (toki_1 ∉ ⊥) and is not scheduled to transfer (leaving_i is false), each request in the queue is granted in the order of priority as long as its group is the same as the current group.

5) Receiving a token message (line 5.1–5.14): First, P_i starts a new current group containing P_i only, and it sends an acquired message to a quorum (line 5.1–5.5), P_i is enabled to enter critical section (line 5.6). Then, P_i moves request items from a temporary

Fig. 4. Description of TQGmx for P_i (2/3).

Fig. 5. Description of TQGmx for P_i (3/3).
queue \( \text{tmpQ} \) into the queue of the main-token (line 5.7–5.13). Finally, \( P_i \) invokes procedure handlePendingRequests in which \( P_i \) sends a subtoken message (line 5.14).

6) Receiving a subtoken message (line 6.1–6.2): First, \( P_i \) sets the value of \( \text{home}_i \) to the sender process of this message, to which \( P_i \) should send a release message on exit of critical section. Then, \( P_i \) enters critical section.

7) Receiving an acquired message (line 7.1–7.5): The value of \( \text{holder}_i \) is set to the sender process of this message (line 7.1). The value of \( \text{holder}_i \) is the process that holds the main-token, and hence to which \( P_i \) should forward request messages (lines 7.2–7.5).

8) Receiving a request message (line 8.1–8.12): There are three cases depending on the value of \( \text{holder}_i \). Let \( P_k \) be the process that issued the request in question. Note that \( P_k \) may not be the sender of the message since the request may be forwarded by some process.

- Case C1 (line 8.1–): When \( P_k \) holds the main-token: If the request has not been enqueued yet, it is enqueued, and \( P_i \) invokes procedure handlePendingRequests. Otherwise, the message is ignored.
- Case C2 (line 8.7–): When \( P_i \) does not hold the main-token but \( \text{holder}_i \neq \bot \) holds: It forwards the request to the holder of the main-token.
- Case C3 (line 8.9–): Otherwise: \( P_i \) enqueues the request into its local temporary queue \( \text{tmpQ} \). Before this action takes place, \( P_i \) deletes the out-dated request (if any) in \( \text{tmpQ} \).

9) Receiving a release message (line 9.1–9.5): Process \( P_i \) decrements the group size by one. If no process is in critical section, \( P_i \) invokes procedure handlePendingRequests to grant pending requests in the queue.

10) Transfer of the main-token: Procedure beginTokenTransfer is invoked to start transfer of the main-token.

In procedure beginTokenTransfer (line 10.1–10.3), \( P_i \) sends a leave message to a quorum, and waits for an ack message for each leave message. To avoid further invocation of this procedure while this action is in progress, local variable \( \text{leaving}_i \) is used. The value of \( \text{leaving}_i \) remains true until \( P_i \) receives an ack message from each process in \( Q_i - \{P_i\} \).

When a process \( P_j \) receives a leave message, it nullifies \( \text{holder}_j \), and sends an ack message back to \( P_i \) (line 11.1–11.2).

When \( P_j \) receives all the ack messages (line 12.1–12.2), it is ready to transfer the main-token. Let \( P_j \) be the process with the highest priority\(^3\).

- If \( P_j = P_i \) (line 12.6), \( P_i \) uses the main-token again. \( P_i \) sends an acquired to each process in a quorum, and invokes procedure handlePendingRequests to send subtoken messages.
- Otherwise (line 12.14), \( P_j \) transfers the main-token to \( P_j \) by a token message.

Note that the case \( P_j = P_i \) occurs in the following scenario. While \( P_j \) is waiting for ack messages, (1) it issues a new request for mutual exclusion, and (2) the priority of its request is the highest.

\(^3\)Note that the queue is always non-empty at line 12.5. Procedure beginTokenTransfer is invoked at line 4.10 when the queue is not empty. (See line 4.1). After procedure beginTokenTransfer is invoked, requests in the queue are not dequeued until all the ack messages are received by the value of \( \text{leaving} \). Thus, the queue is not empty at line 12.5.

11) A note on maintenance of the value of \( \text{holder}_i \): The protocol described above to maintain the value of \( \text{holder}_i \), and current holder of the main-token, is summarized as follows.

1) Before the main-token moves from \( P_k \) to other process, \( P_k \) sends a leave message to each process \( P_j \) in a quorum (line 10.2), and \( P_j \) notifies the value of \( \text{holder}_j \) (line 11.1).
2) As a response to a leave message, each \( P_j \) sends an ack message to \( P_k \) (line 11.2).
3) By receiving an ack message from each process in a quorum (line 12.1–12.2), \( P_k \) is ready to transfer the main-token.

By this protocol, we have the following two properties.

- Suppose that \( P_j \) sends a request message to \( \text{holder}_j \) when \( \text{holder}_j \neq \bot \) holds. By an ack message and FIFO property of a channel, the request message is received by \( P_k \) before \( P_k \) transfers the main-token to other process.
- By intersection property of coterie, at least one process in any quorum (eventually) knows the holder of the main-token, and at least one request message sent to each process in a quorum is successfully forwarded to the holder of the main-token.

E. Priority scheme of the token queue

In the proposed algorithm, each request is put into the queue of the main-token, even if a requesting process holds the main-token that is not in use. Because of this structure, we can define various priority schemes on the queue of the main-token.

A simple priority scheme is the first-come-first-served (FCFS) scheme such that requests are granted in the order of enqueue. Unfortunately, this scheme yields low expected concurrency of critical section entries. Suppose that each process selects one of two groups uniformly at random when it makes a request. Let \( r_1, r_2, \ldots \) be requests such that \( r_i \) is the \( i \)-th request that arrives at the quorum of the main-token. Let \( K \) be a random variable such that groups of each \( r_1, r_2, \ldots, r_K \) are the same and that of \( r_{K+1} \) is different. By simple probabilistic analysis, expected value of \( K \) is at most 2, and hence the FCFS scheme is not enough to yield higher concurrency. On the other hand, priority scheme must be non-starving, i.e., every request must be granted eventually for any execution.

Let \( r_1, r_2, \ldots, r_L \) be requests in the queue, where \( L \) is the number of requests in the queue. We assume that \( r_i \) is the \( i \)-th oldest request in the queue. That is, \( r_1 \) is the oldest request and \( r_L \) is the latest one. Since the requests are centralized at the main-token, we can define various priority schemes. As an example, we define priority \( P(r_i) \) for each \( r_i \) as follows. \( P(r_i) = \alpha G(r_i) + \beta X(r_i) + \gamma S_G(r_i) + \delta A_G(r_i) \), where

- \( \alpha, \beta, \gamma, \delta \) are constant parameters.
- Group: \( G(r_i) \) is 1 if the group of \( r_i \) is the same as the current group, and 0 otherwise.
- Order: \( X(r_i) = (L - i + 1)/L \) is the reversed order of arrival of \( r_i \) at the queue. The oldest request has value 1, and the latest one has the smallest value. \( 0 < X(r_i) \leq 1 \) for each \( r_i \)
- Group Size: \( S_G(r_i) = s_g(r_i)/L \), where \( s_g(r_i) \) is the number of requests that are in the same group as \( r_i \)'s. \( 0 < S_G(r_i) \leq 1 \) for each \( r_i \)
- Group Age: \( A_G(r_i) = \sum r_j a_g(r_j)/n \), where \( a_g(r_j) \) is the age of \( r_j \) which is the number of granted requests after \( r_j \).
that the number of tokens and current group size are correctly maintained invariants are shown in Figure 6. Invariants InvA and InvB assert maintaining the invariants shown in Figure 7, and symbols used in Table 1 are shown in Figure 8. Because of space limitation, we show only outline. The algorithm maintains all the invariants, and each handler is invoked with precondition being satisfied. Suppose that a handler is invoked with precondition being satisfied just before the handler is invoked. Then, execution of a handler maintains all the invariants.

Fig. 6. Symbols used in invariants.

is enqueued. The sum is taken over requests \( r_j \) that are in the same group as \( r_i \)'s. \( A_G(r_i) \geq 0 \) and its value has no upper bound.

Based on this framework, we obtain various priority scheme by setting four parameters \( \alpha, \beta, \gamma, \delta \).
- \( \alpha = 0, \beta = 1, \gamma = \delta = 0 \) : This priority scheme is the same as the FCFS scheme. Although this scheme yields low concurrency, this is a non-starving scheme.
- \( \alpha = 1, \beta = 0, \gamma = \delta = 0 \) : Although this scheme yields high concurrency, this scheme is starving.
- \( \alpha, \delta > 0 \) : This priority scheme yields both concurrency and non-starving.

IV. PROOF OF CORRECTNESS

In this section, we show proof of correctness of TQGmx. Because of space limitation, we show only outline. The algorithm maintains invariants shown in Figure 7, and symbols used in the invariants are shown in Figure 6. Invariants InvA and InvB assert that the number of tokens and current group size are correctly maintained. Invariant InvD asserts group mutual exclusion condition is maintained.

**Lemma 1**: Invariant InvA is maintained for any execution.

For simplicity of description of invariants in Figure 7, the (unique) main-token is denoted by \( tok \), which may be held by a process or may be in a token message in transit.

By induction, we show that invariants are maintained. We omit the proofs of the following lemmas because we can easily check that each action of message handler and event handler maintains the invariants.

**Lemma 2**: (Base step) When the system is initialized, all the invariants are satisfied, and precondition of requestEvent is satisfied at each process.

**Lemma 3**: (Induction step, precondition of message receipt) Suppose that a message arrives at \( P_i \) provided that (1) precondition of a handler that sends the message is satisfied, and (2) all the invariants are satisfied just before the message arrives at \( P_i \). Then, precondition of corresponding handler is satisfied.

**Lemma 4**: (Induction step, precondition of procedure call) Suppose that a handler is invoked with precondition being satisfied when all the invariants are maintained. Then, precondition of any procedure invoked from the handler is also satisfied.

**Lemma 5**: (Induction step, invariants) Suppose that a handler is invoked with precondition being satisfied and all the invariants are satisfied just before the handler is invoked. Then, execution of a handler maintains all the invariants.

**Lemma 6**: Suppose that \( \text{requestDone} \) event is triggered assuming that (1) precondition of message handler or procedure, in which the event is triggered, is satisfied, and (2) all the invariants are satisfied on invocation of message handler or procedure. Then, precondition of releaseEvent handler becomes true, and it remains true until releaseEvent handler is invoked.

**Theorem 1**: For any execution, the proposed algorithm maintains all the invariants, and each handler is invoked with precondition being satisfied.

Now we have safety and liveness properties of the proposed algorithm.

**Theorem 2**: (Safety) For any execution, no two processes in different groups are in critical section simultaneously.

**Proof**: Assume contrary that there exists an execution and two processes \( P_i \) and \( P_j \) such that

\[
(\text{mode}_i = \text{mode}_j = \text{INCS}) \land (\text{grp}_i \neq \text{grp}_j).
\]

By InvB (2), we have \( gSize > 0 \). This implies \( gName \neq \perp \) by InvD (9). By InvD (13), we have \( gName = \text{grp}_i = \text{grp}_j \); a contradiction.

**Theorem 3**: (Liveness) A process that makes a request eventually enters critical section, provided that priority scheme of the queue of the main-token is non-starving.

**Proof**: (Outline) Suppose that \( P_i \) makes a request. The request of \( P_i \) is eventually enqueued into the queue of the main-token by locally, by a direct request message, or by an indirect request message forwarded by a process in a quorum. Then, the request in the queue is eventually granted since the priority scheme is non-starving.

Note that the latest request item in \( \text{tmpQ}_j \) is not deleted for every \( P_j \in q_i \) (at line 2.11 and line 8.10). Assume contrary that, for each \( P_j \), the latest request item of \( P_i \) is deleted from \( \text{tmpQ}_j \), and all the latest request items of \( P_i \) are lost. The only possible scenario is that each \( P_j \) receives an old request of \( P_i \) when the latest request is in \( \text{tmpQ}_j \). By FIFO property of a channel, this is possible only when

- The latest request arrives at \( P_j \), and then
- An old request of \( P_i \) arrives by asynchrony of a system.

Since \( P_k \) forwards the request to \( P_j \), \( P_j \) is the holder of the main-token. Thus, the latest request is enqueued into the queue of the main-token when it arrives at \( P_j \).

Now we have the following theorem.

**Theorem 4**: The proposed algorithm is a distributed group mutual exclusion algorithm.
In this section, we show performance analysis of TQQmx for performance measures defined in II-D.

Lemma 8: Message complexity of the proposed algorithm is 0 in the best case and 5|Q| + 1 in the worst case, where |Q| is the size of quorum used by the algorithm.

Proof: Consider a process P_i holds the main-token when it makes a request and priority of its request is the highest. P_i uses it without any message exchange. Thus, the number of messages in this case 0.

The scenario of the worst case is as follows. (1) A requesting process P_i sends a request message to each process in a quorum, (2) Each request is forwarded to the holder, say P_k, of the main-token, (3) P_k sends a leave message to each process, and (4) An ack message is sent back to P_k from each process in a quorum. Although the main-token may be sent to other party, say P_1, we do not count such messages since they are not caused by P_1. (5) The main-token is transferred to P_i by a token message from some process, and (6) P_i sends an acquire message to each process in a quorum. In total, 5|Q| + 1 messages are exchanged.

Note that the worst case message complexity 5|Q| + 1 only happens for a process that receives the main-token. If x processes enter critical section simultaneously, the number of amortized messages is at most (5|Q| + 1 + (x - 1) · (2|Q| + 1))/x, and this is equal to 2.75|Q| + 1 when x = 4. Although the worst case message complexity is large, the number of amortized messages is small when concurrency is large.

Lemma 9: Maximum concurrency of the proposed algorithm is n.

Proof: When each process makes a request for the same group simultaneously, n−1 sub-tokens are generated by the holder of the main-token. Thus, the maximum concurrency is n.

Lemma 10: Waiting time of the proposed algorithm is at most 5 message hops.

Proof: Let us observe a chain of messages. A requesting process sends a request message to each process in a quorum, and a request message is forwarded to the holder of the main-token. Then, leave and ack messages are exchanged, and finally a token messages is transferred. Thus, 5 message hops are required.
in the worst case.

Lemma 11: Synchronization delay of the proposed algorithm is at most 4 message hops.

Proof: Let us observe the action when groups are switched. The worst case occurs when the last process, say $P_i$, of the current group holds a sub-token. When $P_i$ exits critical section, it sends a release message to the holder of the main-token, say $P_k$, then $P_k$ sends leave messages and waits for ack messages. Then, it sends a token message. Thus, 4 hops are required to start next critical section entries.

The best case is that $P_i$ holds the main-token, and synchronization delay is 3 message hops.

VI. SIMULATION RESULTS

We carried out a simulation to evaluate average performance of TQGmx. Simulation parameters are shown in Table I. We assumed that any local computation and message transmission do not consume local time. The number of trials (simulations) is 100 for each combination of simulation parameters, and each simulation terminates when the total number of critical section entries by all processes reaches 1000$n$. Although the proposed algorithm assumes asynchronous distributed system, our simulation model is a synchronous one with a global clock.

The simulation was carried out on a discrete event simulator we developed in C language. Computing environment is as follows.

- IBM IntelliStation A Pro with dual Opteron 245 (2.8GHz clock) and 10G byte memory
- RedHat Enterprise Linux WS4 for AMD64/EM64T
- C compiler: GCC version 3.4.2

A. The number of messages

Figures 8(a) and (b) show relations between $n$ and the average number of messages per critical section entry. For cases that each process selects the same group (“fixed $g(i) = 0$”), less messages are exchanged because most of processes enters critical section by sub-tokens. On the other hand, for cases that each process selects different group (“fixed $g(i) = i$”), more messages are exchanged because the main-token is transferred for each entry of critical section. When requests are likely to conflict ($T_{idle} = 0.01$, Figure 8(a)), average message complexity is approximately $1.1 \times |Q|$ (resp., $3.3 \times |Q|$) in the best (resp., worst) case. When requests are unlikely to conflict ($T_{idle} = 1000$, Figure 8(b)), it is approximately $4.4 \times |Q|$.

Figure 8(c) shows relations between $T_{idle}$ and the number of messages when $n = 100$. The simulation results match our estimation shown below.

- When $T_{idle}$ is large, every request is unlikely to conflict. Assuming that each request does not conflict, we derive an upper bound of the expected number of messages. Suppose $P_i$ makes a request when $P_j$ holds the main-token. With probability $1/n$, $P_i = P_j$ holds and no message is exchanged. With probability $(|Q| - 1)/n$, $P_i \in q_j$ holds and 3$|Q| + 2$ messages are exchanged. With probability $(n - |Q|)/n$, $P_i \notin q_j$ holds and at most 5$|Q| + 1$ messages are exchanged. Thus, the expected number messages exchanged
B. Concurrency

Figure 9 shows distribution of concurrency when $T_{idle} = 0.1$. In our simulation, distribution of concurrency is measured. For each time a process enters critical section, we measure concurrency (the number of processes) that are in critical section at the same time, and frequency (the number of occurrences) of each concurrency value is counted. Normalized frequency value is computed such that the sum of normalized frequency equals 1.0. For example, Figure 9(a) shows that, in case $\alpha = 0.25$, 39 processes are likely to be in critical section with probability 0.10 when a process enters critical section.

The figures show that the proposed algorithm achieves high concurrency by setting the value of $\alpha$.

C. Waiting time

Figure 10(a) shows waiting time decreases when the processes are likely to select the same group. Figure 10(b) shows waiting time for some values of $\alpha$ when group selection is “random $g(i) = 0.2$”. When the value of $\alpha$ is increased, (1) concurrency is increased as shown in Figure 9(b), (2) waiting time of processes in the same group is decreased, but (3) waiting time of processes in other groups is increased. Figure 10(b) shows that average waiting time decreases by increase of $\alpha$.

Figure 10(c) shows waiting time in message hops when each process always select group 0. As the Figure shows, 5 message hops is the worst case and 3 message hops is the best case and

---

The value is computed by dividing the waiting time measured in simulation time by message delay $D_{msg}(=0.01)$.  

---
VII. RELATED WORKS

A. Quorum-based distributed algorithms

As a communication structure in distributed systems, coterie (or quorums), proposed by Garcia-Molina and Barbara in [4], is widely used for communication structure between processes. A distributed mutual exclusion and concurrency control of database, for example, can be implemented with less message complexity and more reliability by coterie [15], [8]. After Garcia-Molina and Barbara proposed the concept of coterie, many variants of coterie (or quorums) have been proposed for solving various conflict resolution and coordination problems in distributed systems. For example, see [7], [10], [16], [11], [12], [17], [13].

B. Distributed group mutual exclusion algorithms

The problem of group mutual exclusion was first proposed by Joung in [1], in which an algorithm for shared memory parallel computer systems is proposed. Below, we review major distributed algorithm for the problem of group mutual exclusion.

1) Joung’s broadcast-based algorithm: In [2], Joung proposed a distributed algorithm for the problem by extending the Ricart and Agrawala’s distributed mutual exclusion algorithm [18]. He proposed two algorithms: RA1 and RA2. Since these algorithms broadcast request messages, message complexities are both $O(n)$.

In RA1, a process wishing to enter critical section broadcast a request message to every other process. If it obtains an acknowledgment from every other process, it enters critical section. Although the maximum concurrency of RA1 is $n$ in the best case, the expected concurrency is shown to be $O(1)$ even if the number of groups is two. To this end, RA2 is proposed to increase the expected concurrency by giving a priority on requests with the same group as the current group.

Implementation of priority schemes in RA1 and RA2 are distributed in a sense that each process locally decides whether it grants to a request or not. Unfortunately, it is difficult to control priority scheme of the algorithm because implementation of its priority scheme is distributed.

2) Joung’s quorum-based algorithm: In [3], Joung proposed a quorum system, called surficial quorum system, for solving the problem. A requesting process must obtain a permission from each process in a quorum to enter critical section. Implementation of priority scheme is distributed. Two algorithms are proposed based on concurrency of issuing request messages. A concurrent version Maekawa_M sends all request messages in parallel, which requires messages to avoid deadlocks. A serial version named Maekawa_S sequentially obtains permission from each process in a quorum to avoid deadlocks (cf. [19]).

Surficial quorum system has limitation in the maximum concurrency because of its structure. These algorithms can use ordinary quorum systems (i.e., coteries), instead of surficial quorum systems, and the maximum concurrency becomes $n$.

3) Atreya and Mittal’s algorithm: In [6], Atreya and Mittal proposed an algorithm Surrogate based on quorums under a concept of surrogate quorums. The algorithm adopts leader-follower scheme. A process that obtains a permission from each process in a quorum, it becomes the leader. Other process in the same group obtains permission from the leader, and enters critical section as a follower. The algorithm achieves low message complexity $O(|Q|)$ even in the worst case, where $|Q|$ is the size of quorum used.

The leader grants to followers’ requests only when it is about to enter critical section, and no requests are granted when the leader is in critical section. Hence it is difficult to increase concurrency. To improve concurrency they proposed another algorithm in [6] so that the leader can grant to followers’ requests while it is in critical section. We call it SurrogatedEx in this paper. Unfortunately, as we will show its simulation result in subsection VII-C, improvement of concurrency is limited in average situation.

Limited concurrency comes from leader’s protocol for granting followers’ requests. After a leader exits critical section, even if there are many pending requests in the same group, it never grants any requests. Thus, if a leader exits critical section immediately, increase of concurrency is limited.

4) Mittal and Mohan’s algorithm: In [5], Mittal and Mohan proposed an algorithm TokenGMX based on token which is an extension of the distributed mutual exclusion algorithm by Suzuki and Kasami [20]. When a process makes a request, it broadcasts request messages to every other processes, and waits for arrival of a token. Thus its message complexity is $O(|Q|)$. There are two types of tokens, primary and secondary, as in our algorithm. The holder of the primary token generates a new secondary token when it receives a request for the same group. Since the primary token holds a request queue, implementation of priority scheme is centralized.

5) Proposed algorithm TQGmx: Our algorithm is an extension of Mizuno, Neilsen and Rao [7] for (non-group) distributed mutual exclusion. It is based on a token for permissions and quorums for requesting communication. Their algorithm uses three message types, request for request (sent to processes in a quorum), acquired for notification of token acquisition (sent to processes in a quorum), and privileged for token transfer. Each request is forwarded to a token holder, and it is enqueued in a queue of a token. Then, request is granted.

To make it a group mutual exclusion algorithm, a token holder may simply issue a sub-token for requests in the same group based on the leader-follower scheme. Unfortunately, such a straightforward extension does not increase concurrency because, by the property of the algorithm in [7], at most one request is forwarded after receipt of an acquired message from a token holder, and a request message may be forwarded to a process that does not hold a token. To increase concurrency, our algorithm forwards any number of requests to a holder of the main-token. To keep the message complexity small, our algorithm maintains a holder of the main token up-to-date by introducing messages leave and ack and a local variable holder, so that a forwarded request is received by a holder of the main.

In TQGmx, even after a token holder (leader) exits critical section, it may continue granting followers’ requests to increase concurrency as long as there is a request with the highest priority in the current group. When there is a request in the current group, pending requests in different groups can be delayed longer to increase concurrency by setting parameters of the priority scheme proposed in Subsection III-E.

C. Comparison of performance

In Table II, performance of distributed group mutual algorithms are summarized. The proposed algorithm improves in message complexity of previous algorithms (except Maekawa_S which requires large waiting time). Among these algorithms, Surrogate and SurrogateEx have competitive performance to our algorithm.
We carried out simulation to evaluate performance of these two algorithms under the same setting described in Section VI. Results are shown\(^6\) in Figures 11, 12 and 13.

- **Message complexity.** (See Figures 11 and 8(c).) Surrogate and SurrogateEx is efficient when requests are unlikely to conflict (i.e., \(T_{\text{idle}}\) is large) since deadlock avoidance mechanism is not triggered, and group selection has a little effect on message complexity. On the other hand, TQGmx requires small number of messages when requests are likely to conflict, and message complexity is reduced if processes are likely to select the same group.

- **Concurrency.** (See Figures 12 and 9.) Even if every process selects the same group, concurrency of SurrogateEx is low in practice. On the other hand, TQGmx achieves higher concurrency even if each process selects a group randomly from two or three.

- **Waiting time.** (See Figures 13 and 10(a).) When system is busy (i.e., \(T_{\text{idle}}\) is small) and each process selects the same group (i.e., “fixed \(g(i) = 0\)”), waiting time of TQGmx is much smaller than that of SurrogateEx. When each process selects different group (i.e., “fixed \(g(i) = i\)”), waiting time of SurrogateEx is better than that of TQGmx, however, their difference is small.

As a conclusion, TQGmx achieves high concurrency and small message complexity when the system is busy.

\(^6\)Since concurrency and waiting time of Surrogate are worse than those of SurrogateEx, only the results of SurrogateEx are shown.

VIII. CONCLUSION

In this paper, we proposed a distributed group mutual exclusion algorithm based on tokens for permissions and quorums for requesting communication. Although the proposed algorithm is fully distributed, management of pending requests is centralized at the holder of the main-token. The proposed algorithm is implemented on a discrete event simulator and its performance is evaluated. Especially, our algorithm is shown to achieve high concurrency in average which is an important factor for group mutual exclusion and benefit of the proposed algorithm. Development of an algorithm that reduces waiting time and synchronization delay with keeping high concurrency is left as a future work.

ACKNOWLEDGMENT

This work is supported in part by Global COE (Centers of Excellence) Program of MEXT, Grant-in-Aid for Scientific Research ((B)19300017, (B)19700075 and (B)17300020) of JSPS, and Kayamori Foundation of Informational Science Advancement.

REFERENCES

TABLE II

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
<th>Worst Complexity</th>
<th>Maximum Concurrency</th>
<th>Waiting Time</th>
<th>Synchronization Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>TQGmx (this paper)</td>
<td>0</td>
<td>5</td>
<td>Q</td>
<td>+ 1</td>
<td>n</td>
</tr>
<tr>
<td>RAT [2]</td>
<td>2(n – 1)</td>
<td>2(n – 1)</td>
<td>n</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>RA2 [2]</td>
<td>2(n – 1)</td>
<td>3(n – 1)</td>
<td>n</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Maekawa_M w/ S.Q. [3]</td>
<td>3\sqrt{2n(m – 1)} / m</td>
<td>O(nm)</td>
<td>\sqrt{2n/m(m – 1)}</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Maekawa_M w/ O.Q. [3]</td>
<td>3|Q|</td>
<td>O(n</td>
<td>Q</td>
<td>)</td>
<td>n</td>
</tr>
<tr>
<td>Maekawa_S w/ O.Q. [3]</td>
<td>(2</td>
<td>Q</td>
<td>+ 1)</td>
<td>2</td>
<td>Q</td>
</tr>
<tr>
<td>Surrogate [6]</td>
<td>3|Q</td>
<td>+ 2</td>
<td>8</td>
<td>Q</td>
<td>+ 1</td>
</tr>
<tr>
<td>SurrogateEx [6]</td>
<td>3|Q</td>
<td>+ 2</td>
<td>10</td>
<td>Q</td>
<td>+ 1</td>
</tr>
<tr>
<td>TokenGME [5]</td>
<td>0</td>
<td>2n – 1</td>
<td>n</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

"S.Q." denotes surficial quorum systems, "O.Q." denotes ordinary quorum systems, i.e., coterie. n (resp. m) is the number of processes (resp. groups).


Hirotsugu Kakugawa received the B.E. degree in engineering in 1990 from Yamaguchi University, and the M.E. and D.E. degrees in information engineering in 1992, 1995 respectively from Hiroshima University. He is currently an associate professor of Osaka University. He is a member of the IEEE Computer Society and the Information Processing Society of Japan.

Sayaka Kamei received the B.E., M.E. and D.E. degrees in electronics engineering in 2001, 2003 and 2006 respectively from Hiroshima University. She is currently an assistant professor of Tottori University of Environmental Studies. Her research interests include distributed algorithms. She is a member of the Institute of Electronics, Information and Communication Engineers and IEEE Computer Society.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987-1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994-2000. He is now a visiting associate professor of Department of Computer Science, Cornell University between 1993-1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, IEICE and the Information Processing Society of Japan.