Distrib Comput (1995) 8: 171–180

DISTRIBUTED COMPUTING

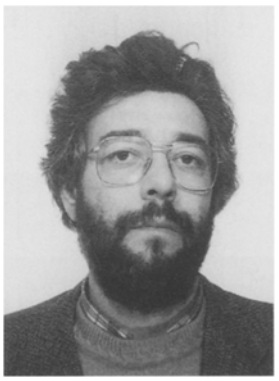
A class of high performance Maekawa-type algorithms for distributed systems under heavy demand*

Roberto Baldoni, Bruno Ciciani**

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italia
e-mail {baldoni, ciciani}@dis.uniromal.it

Received: August 1993 / Accepted: November 1994

Roberto Baldoni was born in Rome on February 1, 1965. He received the Laurea degree in electronic engineering in 1990 from the University of Rome "La Sapienza" and the Ph.D. degree in Computer Science from the University of Rome "La Sapienza" in 1994. Currently, he is a researcher in computer science at IRISA, Rennes (France). His research interests include operating systems, distributed algorithms, network protocols and real-time multimedia applications.



Bruno Ciciani received the Laurea degree in electronic engineering in 1980 from the University of Rome "La Sapienza". From 1983 to 1991 he has been a researcher at the University of Rome "Tor Vergata". He is currently full professor in Computer Science at the University of Rome "La Sapienza". His research activities include distributed computer systems, fault-tolerant computing, languages for parallel processing, and computer system performance and reliability evaluation. He has published in IEEE Trans. on Computers, IEEE Trans. on Knowledge and Data Engineering, IEEE Trans. on Software Engineering and IEEE Trans. on Reliability. He is the author of a book titled "Manufactoring Yield Evaluation of VLSI/WSI Systems" to be published by IEEE Computer Society Press.

Summary. Distributed Mutual Exclusion algorithms have been mainly compared using the number of messages exchanged per critical section execution. In such algorithms, no attention has been paid to the serialization order of the requests. Indeed, they adopt FCFS discipline. Conversely, the insertion of priority serialization disciplines, such as Short-Job-First, Head-Of-Line, Shortest-Remaining-Job-First etc., can be useful in many applications to optimize some performance indices. However, such priority disciplines are prone to starvation. The goal of this paper is to investigate and evaluate the impact of the insertion of a priority discipline in Maekawa-type algorithms. Priority serialization disciplines will be inserted by means of a *gated batch* mechanism which avoids starvation. In a distributed algorithm, such a mechanism needs synchronizations among the processes. In order to highlight the usefulness of the priority based serialization discipline, we show how it can be used to improve the *average response time* compared to the FCFS discipline. The gated batch approach exhibits other advantages: algorithms are inherently deadlock-free and messages do not need to piggyback timestamps. We also show that, under heavy demand, algorithms using gated batch exchange less messages than Maekawa-type algorithms per critical section excution.

Key words: Distributed mutual exclusion – Priority – Performance – Distributed synchronization

1 Introduction

Over the last years, many distributed mutual exclusion (dmutex) algorithms have appeared in the literature [12]. In order to reach a common terminology and ground to evaluate such algorithms, a taxonomy has been recently proposed by Singhal [17]. He divides dmutex algorithms in two categories: "token-based" and "non-token-based". Such algorithms have generally been evaluated according to the number of messages exchanged per CS execution (NM) and the synchronization delay (D), which is the

* This research was supported in part by the Consiglio Nazionale delle Ricerche under grant 93.02294.CT12

** *Current address*: IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France. This author is also supported by a grant of the Human Capital and Mobility project of the European Community under contract No. 3702 "CABERNET"

Correspondence to: R. Baldoni

number of sequential messages between two successive executions of the critical section (CS). In particular, Singhal shows that a reduction of NM comes always at a cost of higher D.

Maekawa-type algorithms [10] are a subclass of the "non-token-based" ones. As thoroughly investigated in [16], such a subclass is particularly important since it achieves a very good compromise between NM and D. Indeed, D is at most $2T$ (where T is the expected message propagation delay between two processes) and the minimal value of NM is $3\lceil\sqrt{N-1}\rceil$ (where N is the number of processes). A weakness of Maekawa-type algorithms is the proneness to deadlocks. Each time there is a suspicious of a deadlock, additional rounds of messages need to be exchanged among the processes to resolve it; obviously, deadlocks are especially frequent under heavy demand.

Singhal extracts a class of deadlock-free algorithms from Maekawa-type algorithms [16]. Such a class is deadlock-free in the sense that an algorithm does not need extra rounds of messages to recover from a deadlock. The detection and resolution of the deadlock is done locally by each process. In this class, D is always T, but the minimal value of NM is $\frac{3}{2}(N-1)$.

In "non-token-based" algorithms [1, 8, 10, 13, 14, 16], no attention has been ever paid to the serialization order of the requests. Indeed, they use an FCFS discipline to serialize the accesses to the critical section, and in such a discipline, requests are served in timestamp order. Conversely, the insertion of a priority discipline, such as Short-Job-First (SJF), Head-Of-Line (HOL), Shortest-Remaining-Job-First (SRJF) etc. [7, 9] could be useful in several applications to optimize some performance indices [7, 9, 18]. However, priority-based disciplines may exhibit starvation phenomenon. Therefore, a solution to the starvation problem has to be found.

The goal of this paper is to investigate and evaluate the impact of insertion of a priority serialization discipline in Maekawa-type algorithms. To avoid starvation we use the *Gated Batch Priority serialization discipline* [18]. In such a discipline requests are collected in batches, and batches are served serially [5]. In a distributed algorithm a synchronization mechanism [15] is necessary to determine the end of a batch. Therefore, the execution of each process can be seen as a sequence of phases; when a phase ceases, a synchronization mechanism among a set of processes is employed to enter the next phase.

In the following we call the Gated Batch Priority serialization discipline as P/GB discipline and the algorithms based on it P/GB algorithms, where the priority P may be of any type (SJF, HOL, SRJF, etc.).

The use of the gated batch serialization discipline exhibits other advantages: (i) A P/GB algorithm is intrinsically deadlock free, (ii) messages do not need to piggyback timestamps nor processes need logical clocks [8, 6]. Conversely, Maekawa-type algorithms need mechanisms for timestamp management [8, 6] and deadlock detection and resolution.

To estimate the algorithms' performance, first we compare the FCFS discipline with the SJF/GB one and then, we compare the values of NM and D for P/GB algorithms with those of Maekawa and Singhal. We found that P/GB algorithms need a greater number of messages per CS execution only when the request rate is low. Under heavy demand, there is a benefit in message traffic irrespective of the serialization discipline. Regarding the synchronization delay, it is equal to the one of Maekawa-type algorithms in the best case, and is $3T$ in the worst one.

The paper is organized as follows: In Sect. 2 we briefly introduce the system model and basic assumptions and definitions. Section 3 gives a short description of Maekawa and Singhal-type algorithms. In Sect. 4, we introduce the Gated Batch Priority (P/GB) serialization discipline and compare it with the FCFS discipline using an SJF priority as P. In Sect. 5, we show how to modify Maekawa-type algorithms in order to obtain the P/GB algorithms class and we prove its correctness. In Sect. 6, we compare the performance indices of P/GB algorithms with Maekawa and Singhal-type algorithms. Section 7 contains conclusions of our work.

2 System model and basic definitions

A distributed system is a set U of N processes $\{1,\ldots N\}$ that communicate solely by exchanging messages. Processes are connected by asynchronous and buffered channels. For each channel we assume that: (A1) *it is reliable*, (A2) *the transmission times are unpredictable but finite*, (A3) *it is FIFO* (messages are delivered in the order they were sent). Processes do not share either a common memory or a common clock, and no bound exists to the relative speed of the processes. Finally, we assume that (A4) *no process can remain indefinitely in its CS*.

Before introducing the Maekawa-type algorithms, we give some basic definitions:

request set: Let i be a process, its request set, R_i, is the set of processes to which i sends a request message when trying to execute its CS;

grant set: Let i be a process, its grant set, Q_i, is the set of processes having i in their request set, i.e., Q_i is the set of processes from which i may receive requests;

arbiter: Let R_i and R_j be the request sets of processes i and j, respectively. Any process $k \in R_i \cap R_j$ is an arbiter of the pair (i, j).

While the performance of a dmutex algorithm are measured by means of NM and D, we compare the serialization disciplines using the *average response time* [7, 9, 18]:

average response time: the average response time of a request of a process is the average interval from the request transmission time to the time when the process ends executing its CS; it consists of the sum of the average interval from the generation time to the time when the process starts executing its CS – the *average waiting time* – and the time spent by the process in its CS – the *average execution time*.

3 Related work

3.1 Maekawa-type algorithms

In Maekawa-type algorithms a *request* set and *a grant* set are associated to each process, and each process may both

request a CS execution and act as an arbiter. Request sets satisfy the following conditions:

(i) $(\forall i \in U :: i \in R_i)$
(ii) $(\forall i, j \in U :: R_i \cap R_j \neq \emptyset)$

A process, say i, contains the following data structures: a local queue and a logical clock which is updated according to Lamport's rules [6, 8]. The timestamp of i's message is equal to i's logical clock value at the time the message was sent. The skeleton code executed by i to enter its CS is the following (for more details see [10] and [14]):

> i sends a REQUEST message with a timestamp to each member of R_i. The process executes its CS only when it has received a GRANT message from each member of R_i. Upon exiting from the CS, i sends a RELEASE message to each member of R_i.
> Upon receiving a REQUEST message from a member of Q_i, an arbiter i sends a GRANT message to the requesting process only if it has not currently granted permission to another process. Otherwise the grant is postponed and the request is put on a local queue ordered by timestamp. Upon receiving a RELEASE message from a member of Q_i, process i sends a GRANT message to the process whose request is on the top of the local queue.

Each arbiter can grant only one request at a time; thus receiving a GRANT message is like locking the arbiter in "exclusive" mode [16]. Such algorithms are prone to deadlock and hence, they need mechanisms (extra messages and procedures) for deadlock detection and resolution that increase the message traffic. In particular, an arbiter k suspects a deadlock when it has already sent a GRANT message to a process i whose request timestamp is greater than the request of the just arrived process j. In such a case, a recovery action will be undertaken by k to recover a correct serialization order. i.e., if i has not received all the GRANT messages, it prematurely releases the lock on k so that j may lock k. A recovery action then needs additional rounds of messages to resolve a deadlock even though there is actually no deadlock. Therefore, NM_i is at most $5|R_i|$. In the best case (i.e., no suspect of deadlock) NM_i is equal to $3|R_i|$.

As far as D is concerned, it is either T or $2T$ depending on the request sets of the processes of two sucessive requests. Let i and j be the processes of two successive requests, if the arbiter of the pair (i, j) is one of them, then D is T, otherwise, D is $2T$.

3.2 Maekawa's symmetric algorithm

Among Maekawa-type algorithms, the symmetric solution, based on "equal efforts" and "equal distribution of responsibility", is particularly significant since it gets the minimal size of the request sets (i.e., optimal request sets). In Maekawa's symmetric algorithm, request sets satisfy the following two additional conditions:

(iii) $(\forall i \in U :: |R_i| = K)$
(iv) $(\forall i \in U :: |\{j \mid i \in R_j\}| = K)$

The problem of computing the size of request sets in the symmetric algorithm is solved by considering a finite projective plane of N points whose order is $K - 1$. In that plane, lines are equivalent to request sets and points to processes. It is proved that such a projective plane exists if $K - 1$ is a positive power of a prime. Using the properties of finite projective planes, the size of symmetric request sets is $O(\sqrt{N})$. Hence, we get:

$$NM = 5\lceil \sqrt{N-1} \rceil \quad \text{(worst case)} \tag{3.1}$$

$$NM = 3\lceil \sqrt{N-1} \rceil \quad \text{(best case)} \tag{3.2}$$

3.3 Singhal-type algorithms

Singhal [16] extracts a class of deadlock-free algorithms from that of Maekawa. Such a class deadlock-free in the sense that an algorithm does not need extra messages to recover from deadlocks. The recovery action is undertaken locally by each arbiter (for more details see [16]).

As opposed to Maekawa, each arbiter may grant more than one request at a time. This means that an arbiter can be locked by several processes concurrently. In particular, an arbiter i works as follows:

> Upon receiving a REQUEST message from a member of Q_i, i sends back a GRANT message if the timestamp of the request is lower than the timestamps of all the requests of processes that currently lock the arbiter. Otherwise the grant is postponed and the request is put on a local queue ordered by timestamp.
> Upon receiving a RELEASE message from a member of Q_i, process i sends a GRANT message to the process whose request is on the top of the local queue.

The request sets in Singhal's algorithm must satisfy the following condition (Condition (ii) is not strong enough):

(v) $(\forall i, j \in U \text{ such that } i \neq j :: (i \in R_j) \vee (j \in R_i))$

Condition (v) implies an increase in the size of request sets that, in the optimal case, becomes $O(N/2)$. Singhal also gives a method to construct a set of optimal request sets. The average number of messages exchanged per CS execution is $3(\sum_{i=1}^{N} |R_i|)/N$ and in the optimal case it is:

$$\frac{3(N-1)}{2} \tag{3.3}$$

Condition (v) also implies that for each pair of processes (i, j), the relative arbiter is always either i or j, so D is always T.

4 Serialization discipline characteristics

The serialization of concurrent accesses to a CS has to be carried out by following a starvation-free discipline. The algorithm that implements it has to preserve this characteristic; for instance, the FCFS discipline is starvation-free and its implementation in a centralized system can be done by means of a simple queue data structure. On the other

hand, in a distributed computing system, the data structure storing requests cannot be a single one, but it must be replicated or distributed in each process. Given the presence of the communication media, the message latency time cannot be zero and hence, the status of these data structures cannot be identical at any given time. To get a consistent and unique serialization order of requests, timestamps are used in [1, 2, 8, 10, 13, 14, 16].

In general, when one wants to implement a serialization discipline in a distributed environment, one must:

- ensure that the discipline is starvation-free;
- evaluate the complexity of its implementation in a distributed environment and the communication overhead.

4.1 Gated Batch Priority (P/GB) serialization discipline

The straight implementation of a priority-based discipline may cause indefinite delay of a process. This is true since processes with a high priority may continue to enter and exit from their own CS's starving processes with low priority. A solution to starvation avoidance is to serialize the incoming requests by a discipline based on Gated Batch Priority Queue [18].

Gated batches of the P/GB discipline are formed as follows [18]. The first request generated by a process, say i, forms the 0-th batch. All requests that arrive when i is in its CS form the 1-st batch. When i exits from its CS, the requests of the 1-st batch are served according to a non-preemptive priority serialization discipline. All requests that arrive during the service time of the m-th batch, form the $m+1$-st batch. Therefore, requests arrived in the $m+1$-st batch can never supercede requests arrived during previous batches even though they have higher priority. When the size of the next batch is zero (i.e., no request is found), the first request forms the 0-th batch.

To clarify the usefulness of the P/GB serialization discipline, we show how it improves the average response time (see Section. 3) compared to the FCFS serialization discipline. In particular, to optimize the average response time, we use a Short-Job-First discipline [7, 9] in each batch (SJF/GB). For an SJF, the priority class is assigned in proportion to the expected CS execution time.

4.2 SJF/GB serialization discipline analysis

In this section we compare the average response time of the SJF/GB discipline with the one of FCFS in the case that the queueing system can be modelled as an M/G/1 system [18]. The following notation will be used:

$E[T]$	mean response time of a request;
$E[W]$	mean waiting time of a request;
$E[W(x)]$	mean waiting time of a request whose CS execution time is x;
b_p	execution time of class p;
b	mean execution time;
$b^{(2)}$	second moment of the execution time;
$B(x)$	the distribution function of the execution time x;
λ_p	request arrival rate of class p;
λ	request arrival rate;
ρ_p	traffic intensity of requests of class p (i.e., $\rho_p = \lambda_p b_p$);
ρ_p^+	traffic intensity of requests of class $1, \ldots, p$ (i.e., $\rho_p^+ = \sum_{i=1}^{p} \rho_i$);
ρ	traffic intensity, or offered load (i.e., $\rho = \sum_{p=1}^{n} \rho_p$);

The average response time is given by the following equation:

$$E[T] = E[W] + b$$

Since b is independent of the serialization discipline, we can analyse $E[W]$ in order to compare $E[T]_{\text{SJF/GB}}$ with $E[T]_{\text{FCFS}}$.

For an SJF/GB serialization discipline, where the priority class is assigned in proportion to the execution time, $E[W(x)]$ of a request of CS execution time x is [18]:

$$E[W[x]]_{\text{SJF/GB}} = \frac{E[W]_{\text{FCFS}}}{1+\rho}\left(1 + 2\lambda \int_0^x t dB(t)\right)$$

where:

$$E[W]_{\text{FCFS}} = \frac{\lambda b^{(2)}}{2(1-\rho)}$$

and

$$b^{(2)} = \int_0^\infty x^2 dB(x)$$

Then, to evaluate $E[W]_{\text{SJF/GB}}$, the knowledge of $B(x)$ is necessary.

To compare the SJF/GB discipline with the FCFS, let us discuss an example. Assume having two classes of requests whose mean execution times are equal to b_1 and b_2; moreover, let us assume that b_2 is greater than b_1. We thus obtain for each class of request [18]:

$$E[W[b_1]]_{\text{SJF/GB}} = \frac{1+\rho_1^+}{1+\rho} E[W]_{\text{FCFS}}$$

$$E[W[b_2]]_{\text{SJF/GB}} = \frac{1+\rho_1^+ + \rho_2^+}{1+\rho} E[W]_{\text{FCFS}}$$

Therefore, the *average waiting time* is

$$E[W]_{\text{SJF/GB}} = \frac{\lambda_1}{\lambda_1+\lambda_2}\left(\frac{1+\rho_1}{1+\rho} E[W]_{\text{FCFS}}\right) + \frac{\lambda_2}{\lambda_1+\lambda_2}\left(\frac{1+\rho_1+\rho}{1+\rho} E[W]_{\text{FCFS}}\right)$$

i.e.,

$$E[W]_{\text{SJF/GB}} = \frac{E[W]_{\text{FCFS}}}{1+\rho}\left(1 + \rho_1 + \frac{\lambda_2}{\lambda_1+\lambda_2}\rho\right)$$

then $E[W]_{\text{SJF/GB}}$ is less than $E[W]_{\text{FCFS}}$ if the following inequality holds

$$\frac{1 + \rho_1 + \frac{\lambda_2}{\lambda_1+\lambda_2}\rho}{1+\rho} < 1$$

i.e.,

$$\frac{\lambda_1 b_1}{\lambda_1 b_1 + \lambda_2 b_2} + \frac{\lambda_2}{\lambda_1 + \lambda_2} < 1$$

Without loss of generality let us assume $\lambda_1 = k\lambda_2$ and $h = \frac{b_2}{b_1}$. We have:

$$\frac{k}{k+h} + \frac{1}{k+1} < 1$$

The latter inequality holds iff:

$h > 1$ i.e., $b_2 > b_1$.

Hence, the SJF/GB discipline is better (i.e., it has smaller average response time) than the FCFS in the case of two classes of requests. The result can easily be extended to the general case of an arbitrary number of classes of requests [3].

5 P/GB algorithms

In the case of a centralized algorithm, the data structure implementing the P/GB serialization discipline could be a *gated queue*. The gated queue data structure is characterized by a queue Q_c and an array Q_w.

The requests of the current batch, say m, are stored in Q_c and are being served. Requests that are received during the m-th batch are stored in Q_w and will form the $m + 1$-st batch. When Q_c is empty, Q_w's requests are appended to Q_c according to the priority discipline, and the system enters the $m + 1$-st batch.

To implement the gated batch mechanism in Maekawa-type algorithms, we use a gated queue at each process. Given variable communication delays, we need a synchronization [15] among the processes to determine the end of a batch and to define which requests will form the next batch. Each process execution proceeds in phases. During each phase, requests of the current batch are granted following the tasks of Maekawa-type algorithms. A process starts the synchronization only if it has no request of the current batch and enters the next phase only when it receives the next batch to be served. Hence, P/GB algorithms are obtained from those of Maekawa in the following way:

(i) *forcing each process to make a request in each phase,*
(ii) *delaying the transmission of the request until the process enters the synchronization, and*
(iii) *forcing each process to start granting requests only when it receives a request from each process of its grant set.*

Obviously, a request may be authentic for the CS access or it may be used only for synchronization (dummy request). Each batch will contain only authentic requests.

Let us spend some words on deadlock. In Maekawa-type algorithms [10], the deadlock occurs when distinct arbiters take a conflicting decision about the serialization of two or more requests. Indeed, each arbiter can receive a set of requests in any order due to communication delays. As we will prove in Sect. 5.3, our algorithm is inherently deadlock-free. In fact, in each phase, it serves a totally ordered and stable set of requests (batch), since the batch's service can never be preempted by other requests, no conflicting decision about the CS execution can ever be taken by arbiters.

Phase-based techniques have been used in many distributed applications. For example, in [11], it was used to deliver multicast messages in causal order in a distributed system with overlapping groups. Moreover, it was also used in the distributed event simulations based on conservative method [5].

In the following, we say that a process is in the m-th phase if it is processing the m-th batch as shown in Fig. 5.1. For the sake of clarity, we will explain the data structure and the phase transition protocol used in P/GB algorithms separately. The P/GB algorithm code of a generic process, i, is shown in Fig. 5.2, where each message handling code is executed atomically.

5.1 Data structures

Local data structures can be split into two groups. The first includes the data structures used to serve requests in a batch (GRANT[ ], Q_c), whereas the second includes data structures of the synchronization (Q_w[ ]). The Boolean variable *synch-phase* indicates if a process is in a synchronization. GRANT[ ] is an array of Boolean that keeps track of the arrival of the GRANT messages from the members of R_i. Its size is $|R_i|$ and its components are initially set to "false". To execute its CS, a process must have all the GRANT components set to "true". Q_w and Q_c have been explained in the previous section; Q_c is a FIFO (First-In-First-Out) queue, Q_w is an array of integers (in process i its size is $|Q_i|$) that registers the arrival of REQUEST messages to determine the end of a synchronization and to store the priority ($P > 0$) of the requests (Q_w's components are initially set to " -1"). In the following we assume that dummy requests have P set to zero and that in a batch P ties are broken using process identifiers. Such a rule induces a total order, called P-order, on requests in a batch.

However, as we will see in the Sect. 5.3, process i may receive at most two REQUEST messages from a process of Q_i before i exits a phase. This problem is due to the asynchronous evolution of processes and arbitrary network communication delays. Therefore, Q_w must be a two

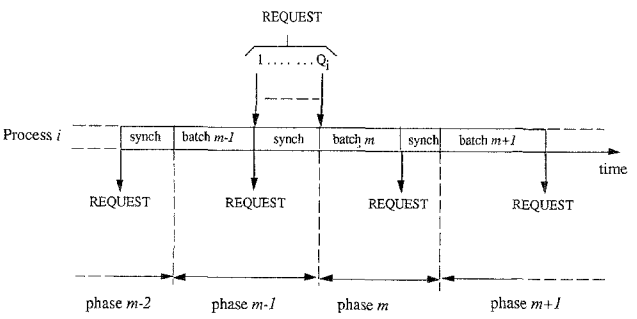


Fig. 5.1. Process activities

columns array. Hence, upon receiving a REQUEST message, process i will set to P the first column component of Q_w related to the sender only if its current value is -1. If that value is greater than -1, the priority is stored in the second column of Q_w whose component will be copied in the first when i exits the synchronization.

5.2 Synchronization

A process, say i, enters the synchronization iff the following precondition β is verified (i.e., all the requests of the current batch have been served from the i's point of view):

$\beta \equiv (Q_{c_i}$ is empty) and (sync_phase$_i$ = FALSE)

Process i enters next phase, executing the following phase transition protocol [15]:

(1) i sends a REQUEST message to all processes in R_i, and

(2) i waits until the phase transition predicate α is true (i.e., the next batch is ready to be served)

$\alpha \equiv (\forall j \in Q_i :: Q_{w_i}[j, 1] > -1)$

Then, authentic requests, if they exist, will be moved from Q_{w_i} to Q_{c_i} in P-order and i will start executing the next phase granting requests of the next batch.

From Fig. 5.2 we see that the boolean variable *requesting* controls that REQUEST messages are sent only once for each synchronization. Note that initially β is true in each process. Therefore, the first request generates the first synchronization.

5.3 Correctness

Definition 1. Two processes, i and j, are directly connected iff $((j \in R_i) \vee (i \in R_j))$

Definition 2. Two processes, i and j, are indirectly connected iff $(\exists k \neq i, j: k \in R_i \cap R_j)$

Lemma 1. *Any pair of processes (i, j) is connected (directly or indirectly) if Maekawa's conditions* (i) *and* (ii) *are satisfied.*

(i) $(\forall i \in U :: i \in R_i)$
(ii) $(\forall i, j \in U :: R_j \cap R_i \neq \emptyset)$

Proof. From (i) and (ii)

$R_i \cap R_j \neq \emptyset$

if $(i \vee j) \in R_i \cap R_j$ then i and j are *directly* connected, otherwise i and j are *indirectly* connected. □

Lemma 2. *After a finite time, β becomes true at each process.*

Proof. The claim follows from assumptions (A1), (A2), (A3) and (A4) and considering that there are a finite number of processes and that β is initially true in each process. □

Theorem 1. *Conditions of Lemma 1 guarantee that in a finite number of message exchanges, all the processes enter and exit the same synchronization.*

Proof. Entering phase: Let us suppose a process, say i, enters a synchronization. It sends a REQUEST message to the processes in R_i. From Lemma 1 and Lemma 2, REQUEST messages will be propagated, at most in two steps, to all processes. Hence, processes not in synchronization yet are forced to enter (step 7 of Fig. 5.2).

Exiting phase: Process i exits the synchronization when the transition phase predicate α is true. It is guaranteed that process i will receive the needed REQUEST messages because:

- all processes enter synchronization phase;
- each process sent a REQUEST message to its R_i's members at the entering phase;
- assumptions (A1), (A2), and (A3). □

Lemma 3. *At any given time, all the processes are at most in two successive phases.*

Proof. Assume the contrary. There are processes in three successive phases $m-2$, $m-1$ and m. Let us suppose i to be a process in the phase $m-2$ and j to be a process in the phase m. In such a case, j must have just received the REQUEST message of phase $m-1$ from all members of Q_j. This is impossible since process i cannot send to R_i's members the REQUEST message related to $m-1$ phase conclusion before entering that phase. In doing so, given the assumption (A3) and the constraint (ii) of Lemma 1, i blocks the entrance in the phase m of all other processes. □

Notation. $a \overset{r}{\Rightarrow} b$ indicates that request "a" precedes request "b" in Q_c of site r (denoted in the following as Q_{c_r}).

Theorem 2. *Any two requests "a" and "b" made when the processes were in the m-th phase will be served in the $m+1$-st phase, and if they are in two distinct Q_{c_s} (Q_{c_h} and Q_{c_k} respectively) we have that*

if $a \overset{h}{\Rightarrow} b$ then $a \overset{k}{\Rightarrow} b$

Proof. Each request made when the process was in the m-th phase will be send out during the transition protocol between the m-th and the $m+1$-st phase (step 2 of Fig. 5.2).

- The requests are temporarily stored in Q_{w_s} $(s = h, k)$ (step 6);
- Given Theorem1, h and k exit from the synchronization (step 8), and "a" and "b" will move from Q_{w_s} to Q_{c_s} $(s = h, k)$ (step 8.a);
- In the $m+1$-st phase requests are queued in Q_{c_s} $(s = h, k)$ in P-order (step 8.b);

therefore,

if $a \overset{h}{\Rightarrow} b$ then $a \overset{k}{\Rightarrow} b$ □

Theorem 3. *P/GB algorithms guarantee mutual exclusion.*

Proof. Assume the contrary. Two processes, i and j, are simultaneously in their CS. In such a case, both of them have received a GRANT message from each member of their request set R. i.e., $\exists i \in U, \exists j \in U: R_i \cap R_j = \emptyset$. Given constraint (ii) of Lemma 1, this is impossible. Therefore, the assumption is contradicted and the claim follows. □

Requesting the resource (P)

```
   begin
1.    requesting := TRUE;
2.    wait until (β=TRUE)                                   /* precondition */
         then begin
            sync_phase := TRUE;
            for each j∈Ri do send REQUEST(i, P) to j;
         end;
3.    wait until (∀j∈Ri : GRANT[j]=TRUE) then for each j∈Ri do GRANT[j]:=FALSE;
      <enter CS>
4.    for each j∈Ri do send RELEASE to j;
5.    requesting := FALSE;
   end.
```

Upon receiving a REQUEST(j, P) message:

```
   begin
6.    if QW[j,1]= -1    then QW[j,1]:=P;
                        else QW[j,2]:=P;
7.    if ((β=TRUE) and (not requesting))                    /* precondition */
         then begin
               sync_phase := TRUE;
               for each j∈Ri do send REQUEST(i, Ø) to j;    /* dummy request */
         end;
8.    if (α=TRUE)                                  /* phase transition predicate*/
         then begin
8.a.           for each j∈Qi do if (QW[j, 1]>0) then insert(j) in QC in P-order;
               for each j∈Qi do QW[j, 1]:=QW[j, 2];
               for each j∈Qi do QW[j, 2]:= -1;
                sync_phase:= FALSE;
8.b.           if (QC is not empty)
                  then begin
                     send GRANT(i) to head(QC);
                     remove(head(QC));
                  end;
         end;
   end.
```

Upon receiving a GRANT(j) message:

```
   begin
      GRANT[j] := TRUE;
   end.
```

Upon receiving a RELEASE message:

```
   begin
      (acts like 8.b);
      (acts like 7);
   end.
```

Fig. 5.2. P/GB algorithm code

Theorem 4. *Given assumptions (A1), (A2), (A3), and (A4) P/GB algorithms are freedom from starvation.*

Proof. Because

- There are a finite number of processes;
- Requests are serialized according to the P/GB discipline;
- Theorem 2;

then there is a total ordering among all the requests. This guarantees that the waiting time for each request is finite, so starvation is impossible. □

Theorem 5. *P/GB algorithms are freedom from deadlocks.*

Proof. Let us assume the contrary, i.e., the algorithm is in a deadlock. Then, no process is able to enter its CS since processes in deadlock block all the other processes (Lemma 1). In such a case, no process has received all GRANT messages from each member of its request set and there is no GRANT message in transit. Two cases are possible depending upon the number of processes in deadlock:

(a) two processes are in deadlock, i.e.,

$\exists k \in U, \exists h \in U$ such that

$$R_i \cap R_j \supseteq \{h, k\} \text{ and } req(k) \overset{i}{\Rightarrow} req(h) \text{ and } req(h) \overset{j}{\Rightarrow} req(k)$$

where the function $req(x)$ returns the request made by process x;

Such a case is impossible from Theorem 2.

(b) more than two processes are in deadlock.

Let us assume x, y, z to be the processes in deadlock and A_1, A_2, A_3 to be the arbiters of the pairs (x, y), (x, z), and (y, z) respectively. A deadlock situation occurs when there is a cycle among the requests stored in Q_{cs} of the arbiters, i.e.,

$$req(x) \overset{A_1}{\Rightarrow} req(y),\ req(z) \overset{A_2}{\Rightarrow} req(x),\ req(y) \overset{A_3}{\Rightarrow} req(z)$$

All the arbiters use the same criterion to order requests, therefore if $req(x) \overset{A_1}{\Rightarrow} req(y)$ and $req(z) \overset{A_2}{\Rightarrow} req(x)$, then $req(z) \overset{A_3}{\Rightarrow} req(y)$ therefore case (b) is impossible.

Hence, the assumption is contradicted and the claim follows. □

5.4 A note on the timestamp management and the implementation

The use of the phase-based approach has another important consequence: messages do not need to piggyback timestamps. Indeed, according to Lamport's rules [8] and Lemma 1, upon exiting each synchronization, logical clocks will get the same value in each process (see *rule F* of [6]), with the consequence that they could be reset to zero. Moreover, requests are collected in batches during synchronizations, and in each phase, actions to serve a batch are totally ordered. i.e., a GRANT message, a CS execution, a RELEASE message, a GRANT message and so on. Such motivations show that logical clocks, and then timestamps are unnecessary.

Regarding the implementation, P/GB algorithms need more data structures (Q_w) and more code due to synchronization than Maekawa-type algorithms. On the other hand, P/GB algorithms need neither additional messages and procedures to recover from deadlock nor logical clocks to dispense timestamps. This results in an algorithm that is only slightly more complex than that of Maekawa.

Finally we would like to underline that in Maekawa-type algorithms processes that have not participated in any action will have a top priority (i.e., low timestamp) when they issue a request. This could be an undesirable property. P/GB algorithms solve such a problem.

6 Performance

In this section, the performance of P/GB algorithms are compared with the ones of Singhal and Maekawa. Such a comparison is done taking into account the number of messages exchanged per CS execution (NM) and the synchronization delay between two successive CS executions (D). NM is analyzed first for the general case and then for the optimal configurations of the request sets of each class of algorithms.

6.1 Number of messages exchanged per CS execution

In P/GB algorithms, NM is equal to the number of GRANT and RELEASE messages generated per CS execution plus a fraction of REQUEST messages necessary to manage the synchronization. Such a fraction is evaluated taking into account the number of requests in a batch (S_n), i.e.,

$$NM_i = 2|R_i| + \frac{\sum_{k=1}^{N} |R_k|}{S_n} \tag{6.1}$$

where

$$1 \leqq S_n < N \tag{6.2}$$

The number of messages per CS execution of Maekawa-type algorithms is given by $3|R_i|$ in the best case and $5|R_i|$ in the worse. Comparing (6.1) with the worst case of Maekawa-type algorithms, it is possible to see that P/GB algorithms exhibit a behaviour comparable to that of Maekawa when

$$\frac{\sum_{k=1}^{N} |R_k|}{S_n} \leqq 3|R_i| \tag{6.3}$$

6.1.1 NM under optimal conditions

In this section we analyze P/GB, Maekawa-type, and Singhal-type algorithms in the case of configuration of requests sets that minimize NM. Hence, we compare P/GB and Maekawa-type algorithms in case of symmetric request sets' configuration and the optimal Singhal's solution. In the case of optimal configuration of requests sets (6.1) becomes:

$$NM = 2\lceil \sqrt{N-1} \rceil + \frac{N\lceil \sqrt{N-1} \rceil}{S_n} \tag{6.4}$$

The number of messages per CS execution of Maekawa symmetric algorithm is given by (3.1) and (3.2). Comparing (6.4) with (3.1), it is possible to see that the symmetric P/GB algorithm exhibits a behaviour comparable with that of Maekawa when:

$$S_n \geqq N/3 \tag{6.5}$$

Comparing (6.4) with (3.2), it is possible to see that the symmetric P/GB algorithm can never require a lesser average number of messages per CS execution than the best case of Maekawa's symmetric algorithm, since this implies $S_n \geqq N$ contradicting (6.2).

The number of messages per CS execution of Singhal's algorithm is given by the (3.3). Comparing (6.4) and (3.3), the symmetric P/GB algorithm is better than optimal Singhal's solution if

$$S_n > \frac{2N}{3\lceil \sqrt{N-1} \rceil - 4} \tag{6.6}$$

The better or worse behaviour of P/GB algorithm depends on the average number of requests issued within a batch as shown in Figs. 6.1 and 6.2. In such Figures, NM versus the average S_n value is plotted in a distributed system with 9 and 100 processes, respectively. In both

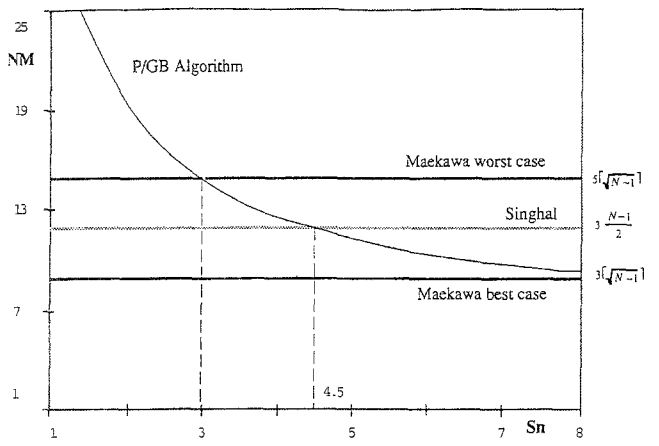


Fig. 6.1. (Nm) versus (Sn) in a distributed system with 9 processes

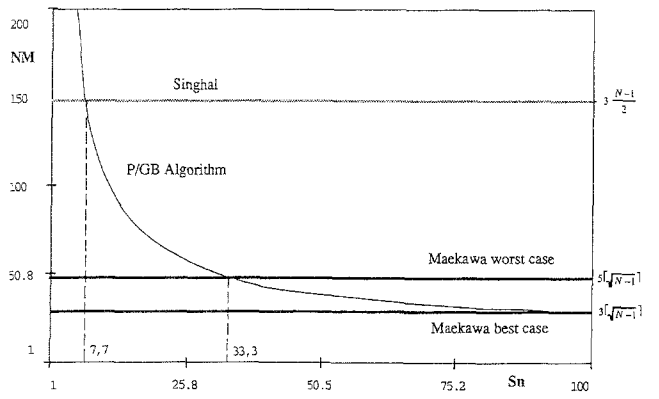


Fig. 6.2. (Nm) versus (Sn) in a distributed system with 100 processes

Figures, the symmetric P/GB algorithm becomes comparable with that of Maekawa if the average S_n value is greater than $N/3$ (i.e., more than 33.3% of the processes make a request in a phase). Moreover, in the symmetric P/GB algorithm, increasing the average request rate, NM goes to $3\lceil\sqrt{N-1}\rceil$ (being unaffected by deadlocks), whereas, in the symmetric Maekawa algorithm, deadlock situations will be more frequent leading to an increase in the average NM value [10].

From Fig. 6.1, we see that P/GB algorithm is better than Singhal's if the average S_n value is greater than 4.5 (i.e., more than 50% of the processes make a request in a phase). In Fig. 6.2, the P/GB algorithm becomes better than the one of Singhal if the average S_n value is greater than 7.7 (i.e., more than 7.7% of the processes make a request in a phase).

6.2 Synchronization delay between two successive CS executions

To define the delay D between two successive CS executions in P/GB algorithms, we have to analyze two different situations:

(1) the process that exits from its CS is not the last one of the current batch;

(2) the process that exits from its CS is the last one of the current batch.

In the case (1), D is equal to that of Maekawa-type algorithms (i.e., D is at most $2T$). In the case (2), D is equal to $3T$. Indeed, the delay between the exit of the last process of the previous batch, say i, and the execution of the first process of the current batch, say j, is given by three consecutive messages. These massages are:

the RELEASE message from i to the arbiter of (i,j);
the REQUEST message from the arbiter of (i,j) to i and j;
the GRANT message from the arbiter of (i,j) to j.

Hence, the average value of D in P/GB algorithms is

$$D_{\text{Maekawa}} + \frac{T}{S_n} \tag{6.7}$$

Therefore, in P/GB algorithms, D is always greater than Singhal-type one (i.e., $D = T$). Under heavy demand, it is slightly greater than the one of Maekawa.

7 Conclusions

In this paper we evaluated the impact of the insertion of a priority discipline in Maekawa-type algorithms, avoiding starvation. Priority serialization disciplines were inserted by means of a *gated batch* mechanism. The insertion of such a mechanism in "token-based" algorithms with concurrent entries to a critical section has been thoroughly investigated in [4]. The advantages induced by the use of the Gated Batch Priority serialization discipline are the following:

- it permits the serialization of the accesses to a CS according to any kind of discipline including priority-based. In particular, P/GB algorithms may take an SJF into account; this leads to a smaller response time than the FCFS discipline adopted by Maekawa-type algorithms;
- messages do not need to piggyback timestamps and logical clocks are unnecessary;
- P/GB algorithms are inherently deadlock-free. Then, extra messages to detect and resolve deadlocks are no longer necessary;
- under heavy demand (i.e., high average request rate), there is even a benefit in the number of messages exchanged compared to Maekawa-type algorithms irrespective of the serialization discipline.

The cost we have to pay to avoid starvation is a greater message traffic and a greater synchronization delay compared to Maekawa-type algorithms in the case of low demend.

Acknowledgements. We are grateful to Giacomo Cioffi for his support, encouragement and helpful comments. In particular, Giacomo contributed to the basic idea of the gated batch processing. Andrea Schaerf and Arif Ishaq provided valuable comments that improved the clarity of exposition. Authors are also grateful to three anonymous referees for their numerous and constructive suggestions and criticisms that greatly improved the presentation and contents of the paper.

References

1. Agrawal D, Abbadi AE: An efficient and fault-tolerant solution for distributed mutual exclusion. ACM Trans Comput Syst 9(1): 1–20 (1991)
2. Baldoni R: An $O(N^{M/(M+1)})$ distributed algorithm for the k-out of-M resources allocation problem. In: Proc of 14-th Int Conf on Distributed Computing Systems. IEEE Press 1994, pp 81–87
3. Baldoni R: Mutual exclusion in distributed systems. Ph.D dissertation, Dipartimento di Informatica e Sistemistica, Unversity of Rome "La Sapienza", Italy, 1994
4. Baldoni R, Ciciani B: Distributed algorithms for multiple entries to a critical section with priority. Inf Process Lett 50(3): 165–172 (1994)
5. De Vries RC: Reducing NULL messages in Misra's distributed discrete event simulation method. IEEE Trans Soft Eng 16(1): 97–107 (1990)
6. Fidge CJ: Logical time in distributed computing systems. IEEE Comput 24(8) 28–33 (1991)
7. Kleinrock L: Queuing systems volume II: computer applications. Wiley 1976
8. Lamport L: Time, clocks and the ordering of events in a distributed system. Commun ACM 21(7): 558–565 (1978)
9. Lavenberg SS: Computer performance modeling handbook. Academic Press, New York 1983
10. Maekawa M: A $\sqrt{N}$ algorithm for mutual exclusion in decentralized system. ACM Trans Comput Syst 3(2): 145–159 (1985)
11. Mostefaoui A, Raynal M: Causal multicast in overlapping groups: towards a low cost approach. In: Proc of Int Conf on Future Trends of distributed Systems. IEEE Press 1993, pp 136–142
12. Raynal M: A simple taxonomy for distributed mutual exclusion algorithms. ACM Operat Syst Review 25(1): 47–51 (1991)
13. Ricart G, Agrawala AK: An optimal algorithm for mutual exclusion in computer networks. Commun ACM 24(1): 9–17 (1981)
14. Sanders BA: The information structure of distributed mutual exclusion algorithms. ACM Trans Comput Syst 5(3): 284–299 (1987)
15. Schneider FB: Synchronization in distributed programs. ACM Trans Prog Lang Syst 4(2): 179–195 (1982)
16. Singhal M: A class of deadlock-free Maekawa-type algorithms for mutual exclusion in distributed systems. Distrib Comput Syst 4: 131–138 (1991)
17. Singhal M: A taxonomy of distributed mutual exclusion. J Paralel Distrib. Comput 18: 94–101 (1993)
18. Takagi H: Queueing analysis volume 1: vacation and priority systems. North-Holland 1991