J. Parallel Distrib. Comput. 83 (2015) 13–29

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A fair starvation-free prioritized mutual exclusion algorithm for distributed systems

Jonathan Lejeune *, Luciana Arantes, Julien Sopena, Pierre Sens

Sorbonne Universités, UPMC Univ Paris 06, CNRS, Inria, LIP6, 4, place Jussieu 75252 Paris Cedex 05, France

HIGHLIGHTS

- Our algorithm is based on two mechanisms aiming at postponing priority increment and reducing messages overhead.
- Priority increment postponing reduces the amount of priority violations without introducing starvation.
- Taking into account request locality allows to reduce the message overhead due to the priority postponement.
- Priority increment increases significantly the waiting time of the lowest priorities.
- The location of processes on the logical tree topology has an impact over performance.

ARTICLE INFO

Article history:
Received 20 December 2013
Received in revised form
15 November 2014
Accepted 2 April 2015
Available online 12 April 2015

Keywords:
Distributed system
Mutual exclusion
Priority
Algorithm

ABSTRACT

Several distributed mutual exclusion algorithms define the order in which requests are satisfied based on the priorities assigned to requests. These algorithms are very useful for real-time applications ones or those where priority is associated to a quality of service requirement. However, priority based strategies may result in starvation problems where high priority requests forever prevent low priority ones to be satisfied. To overcome this problem, many priority-based algorithms propose to gradually increase the priority of pending requests. The drawback of such an approach is that it can violate priority-based order of requests leading to priority inversion. Therefore, aiming at minimizing the number of priority violations without introducing starvation, we have added some heuristics in Kanrar–Chaki priority-based token-oriented algorithm in order to slow down the frequency with which priority of pending requests is increased. Performance evaluation results confirm the effectiveness of our approach when compared to both the original Kanrar–Chaki and Chang's priority-based algorithms.

1. Introduction

Many distributed and parallel applications require that their processes obtain exclusive access to one or more shared resources. Mutual exclusion is then one of the fundamental building bricks of distributed systems. It ensures that at most one process can access the shared resources at any time (safety property) and that all critical section requests are eventually satisfied (liveness property). The set of instructions of processes' code that access a shared resource is denoted a critical section (CS).

Several distributed mutual exclusion algorithms exist in the literature (e.g. [6,14,9,16,13,12]). They can be divided into two families [17]: permission-based (e.g. Lamport [6], Ricart–Agrawala [14], Maekawa [9]) and token-based (Suzuki–Kasami [16], Raymond [13], Naimi–Tréhel [12]). The algorithms of the first family are based on the principle that a process only enters a critical section after having received permission from all the other processes (or a sub-set of them [14,9]). In the second group of algorithms, a system-wide unique token is shared among all processes, and its possession gives a process the exclusive right to execute a critical section.

In the majority of distributed mutual exclusion algorithms, CS requests are satisfied in First-Come-First-Served (FCFS) time-based event order such as the logical time of the requests or the physical time when the token holder receives a request. However, this approach is not suitable for all kinds of applications such as, for instance, applications where some tasks have priority over the others, real-time environments [2,1], or applications where priority is associated to a quality of service requirement [7]. To overcome these constraints, some authors (e.g., [5,2,10,11,1,7]) have proposed some mutual exclusion algorithms where every request

* Corresponding author.
E-mail addresses: jonathan.lejeune@lip6.fr (J. Lejeune), luciana.arantes@lip6.fr (L. Arantes), julien.sopena@lip6.fr (J. Sopena), pierre.sens@lip6.fr (P. Sens).

http://dx.doi.org/10.1016/j.jpdc.2015.04.002
0743-7315/

is associated to a priority. The satisfaction of pending requests respects, whenever possible, the priority order. However, priority order induces starvation problems, i.e., the infinite delay for granting access to the CS to a process, which then violate liveness property. Starvation happens when higher priority requests forever prevent lower priority ones from executing the CS. Hence, in order to avoid such a problem, low priorities of pending requests are dynamically increased in these algorithms, eventually reaching the highest priority. The drawback of this strategy is that it can violate priority-based order of requests, i.e., it can lead to priority inversion where a request with an original low priority will be satisfied before another one with higher priority.

We propose in this paper some priority-based distributed mutual exclusion algorithms that reduce request priority violations without introducing starvation. We particularly focus our work on token-based mutual exclusion algorithms since the latter usually has an average lower message cost and thus presents better scalability.

Token-based algorithms exploit different solutions for the forwarding of critical section requests of processes and token transmission. Each solution is usually expressed by a logical topology that defines the paths followed by critical section request messages which might be completely different from the physical network topology. Our algorithm is an extension of Kanrar–Chaki [5] algorithm where distributed processes are organized in a static logical tree. By applying some heuristics, our algorithm postpones the increasing of priority of pending requests and, therefore, the number of priority violations is reduced when compared to both the original Kanrar–Chaki algorithm and Chang's priority-based algorithm [1], as confirmed by the results of our thorough performance evaluation experiments. Furthermore, we also show that the heuristics have a low message overhead compared to the original algorithm while keeping the same waiting time. Moreover, they tolerate quite well peaks of request load. A first version of our algorithm has been presented in [7] but oriented to Service Level Agreement constraints in the context of cloud computing.

The rest of the paper is organized as follows. Section 2 discusses some existing priority-based mutual exclusion distributed algorithms and gives a description of the Kanrar–Chaki algorithm. Our priority request distributed mutual exclusion solutions are described in Section 3. Performance evaluation results are presented in Section 4. A discussion about a trade-off between the response time and the priority violation is presented in Section 5. Finally, Section 6 concludes the paper.

2. Related work

In this section we outline the main priority-based mutual exclusion algorithms. Furthermore, since our priority-based mutual exclusion (mutex) algorithms are based on the Kanrar–Chaki [5] algorithm, the latter is described in more details.

Prioritized distributed mutex algorithms are usually an extension of some non-prioritized algorithms. Goscinski algorithm [2] is based on the token-based Suzuki–Kasami algorithm and has a message complexity of $O(N)$. Pending requests are stored in a global queue and are piggybacked on token messages. Starvation is possible since the algorithm can lose requests while the token is in transition since in this case, it is not held by any process.

Mueller algorithm [10] is inspired in Naimi–Tréhel token-passing algorithm which exploits a dynamic tree as a logical structure for forwarding requests. Each process keeps a local queue and records the time of requests locally. These queues form a virtual global queue ordered by priority within each priority level. Its implementation is quite complex and the dynamic tree tends to become a simple queue because, unlike the Naimi–Tréhel algorithm, the root process is not the last requester but the token holder. Therefore, in this case the algorithm presents a message complexity of $O(\frac{N}{2})$.

Housni–Tréhel algorithm [3] adopts a hierarchical approach where processes are grouped by priority. Each group is identified by one router process. Within each group, processes are organized in a static logical tree like Raymond's algorithm [13] and routers apply the Ricart–Agrawala algorithm [14]. Starvation is possible for processes that issued low priority processes if many high priority requests are pending. Moreover, a process can only send requests with the same priority (that of its group).

Several algorithms, such as Kanrar–Chaki [5] and Chang [1] algorithms, propose to extend Raymond's [13] token-based algorithm in order to assign priorities to requests. Since our heuristics are applied to Kanrar–Chaki algorithm, we describe both Kanrar–Chaki and Raymond algorithms.

Raymond's algorithm [13] is a token-based mutex algorithm where processes are organized in a static logical tree: only the direction of links between two processes can change during the algorithm's execution. Processes thus form a directed path tree to the root. Excepting the root, every process has a father process. The root process is the owner of the token and it is the unique process which has the right to enter the critical section. When a process needs the token, it sends a request message to its father. This request will be forwarded till it reaches the root or a process which also has a pending request. Every process saves its own request and those received from its children in a local FIFO queue. When the root process releases the token, it grants the token message to the first process of its own local queue and this process becomes its father. Then, if its queue is not empty, it sends a request to its new father, to eventually get the token back. When a process receives the token, it removes the first request from its local queue. If this request was issued by the process itself, it executes the critical section; otherwise it forwards the token to the process that issued it, and the latter becomes its father. Moreover, if the local queue of the process is not empty, it sends to its new father a request on behalf of the first request of its queue.

An example of Raymond algorithm execution with 3 processes is shown in Fig. 1 where arrows represent father links. Initially, process B, the root process, is in critical section, and both processes A and C have issued a request (Fig. 1(a)). When B releases the CS, it sends the token to A, updates its father link and sends a new request to A (Fig. 1(b)) on behalf of C request. In its turn, when A releases the token, it sends it to B that forwards it to C. Finally, the token is received by C; Fig. 1(c) shows the final state when both requests were satisfied.

Kanrar–Chaki algorithm [5] extended Raymond algorithm in order to introduce a priority level for every process CS request. The greater the level (an integer value), the higher the priority of the request. Hence, pending requests of a process's local queue is ordered by decreasing priority levels. Similarly to Raymond's algorithm, a process that wishes the token sends a request message to its father. However, upon reception, the father process includes the request in its local queue according to the request priority level and only forwards it if the request priority level is greater than the one of the previous first element of the processes's local queue. In order to avoid starvation, the priority level of pending requests of a process's local queue is increased: when the process receives a request with priority p, every pending request of its local queue whose priority level is smaller than p is increased by 1.

Similarly to the Kanrar–Chaki algorithm, Chang has modified Raymond's algorithm in [1] aiming both at (1) applying dynamic priorities to requests and (2) reducing communication traffic. For the priority, he added a mechanism denoted *aging* strategy: if process p releases the CS or if it is a non requesting process that holds the token and receives a request, p increases the priority of every request in its local queue; furthermore, upon reception of the token, which includes the number of CS executions, p increases the