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

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

A Fair Starvation-free Prioritized Mutual Exclusion Algorithm for Distributed Systems

Jonathan Lejeune[a], Luciana Arantes[a], Julien Sopena[a], Pierre Sens[a]

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

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.

Keywords: Distributed system, mutual exclusion, priority, algorithm

1. Introduction

Many distributed and parallel applications require that their processes obtain exclusive access 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-Kazami [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

Email addresses: jonathan.lejeune@lip6.fr (Jonathan Lejeune), luciana.arantes@lip6.fr (Luciana Arantes), julien.sopena@lip6.fr (Julien Sopena), pierre.sens@lip6.fr (Pierre Sens)

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 kind 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], [11], [10], [1], [7]) have proposed some mutual exclusion algorithms where every request 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 have an average lower message cost and thus present 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 compare 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. Goscinksi 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 [11] 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 Figure 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 (Figure 1.(a)). When B releases the CS, it sends the token to A, updates its father link and sends a new request to A (Figure 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; Figure 1.(c) shows the final state when both requests were satisfied.

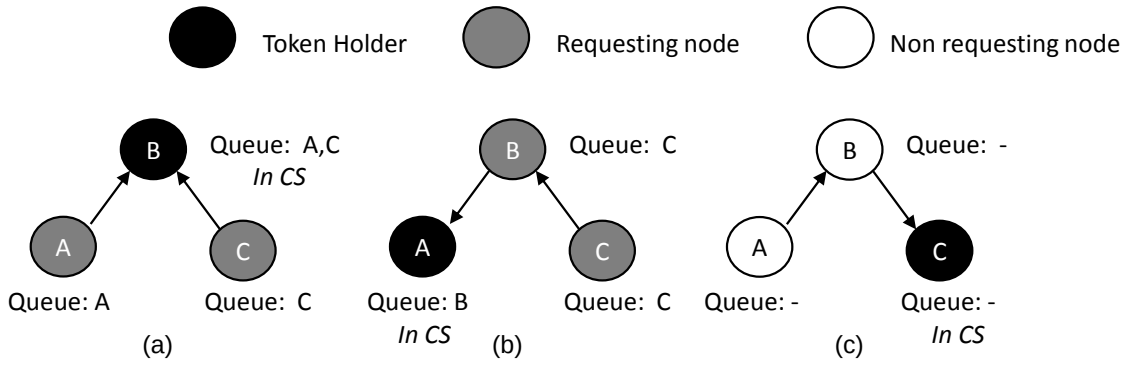


Figure 1: Raymond's algorithm

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 priority of all its old requests (i.e., those requests that were already pending when p releases the token for the last time) by the number of CS that were executed since the last time p had the token. On the one hand, such a priority approach reduces the gap in terms of average response time between priorities (contrarily to the Kanrar-Chaki algorithm). On the other hand, it induces a greater number of priority inversions when compared to the Kanrar-Chaki algorithm. Performance comparison evaluation results of both algorithms are presented in section 4. Since a request always follows the token from an intermediate process whose local queue contains more than one element, (2) communication traffic optimization consists in piggybacking, whenever possible, a request on a token message

In [4], Johnson and Newman-Wolfe present three algorithms for prioritized distributed mutual exclusion. Two of the algorithms use a path compression technique for fast access and low message overhead. One of the algorithm extends Raymond algorithm. Similarly to the Kanrar-Chaki algorithm, each process maintains a local priority queue of requests that it has received. Only new requests with a higher priority than the ones in the queue are forwarded to the father.

In order to prevent priority inversion, Mueller proposes in [10] a token-based prioritized mutual exclusion algorithm which is enhanced with priority ceiling protocol or priority inheritance protocol [15].

3. Priority-based mutual exclusion

We consider a distributed system consisting of a finite set $\Pi = \{s_1, s_2, ..., s_N\}$ of N nodes. There is one process per node. Hence, the words node, process, and site are interchangeable. Nodes are assumed to be connected by means of reliable and FIFO communication links and are organized in a static logical tree. They communicate by sending and receiving messages. Nodes and links are not prone to failures.

Applications behave correctly: a process requests a CS by calling the $Request\_CS$ procedure if and only if its previous request has been satisfied, or it is its first call, and it has released the CS by calling the $Release\_CS$ procedure. A priority is associated to each request. Let $\mathcal{P} = \{p_{min}, p_{min}+1, ..., p_{max}-1, p_{max}\}$ be the set of possible request priorities. Like in Kanrar-Chaki we note $p > p'$ iff priority p is higher than priority p'.

3.1. Priority violation definition

We define that a priority violation happens whenever the priority order of request satisfaction is not respected.

When a priority violation occurs, we distinguish two classes of requests:

- A **favored request** is a request that is satisfied before a pending request with higher priority.

- A **penalized request** is a pending request waiting for the token but a request with priority lower than latter gets it.

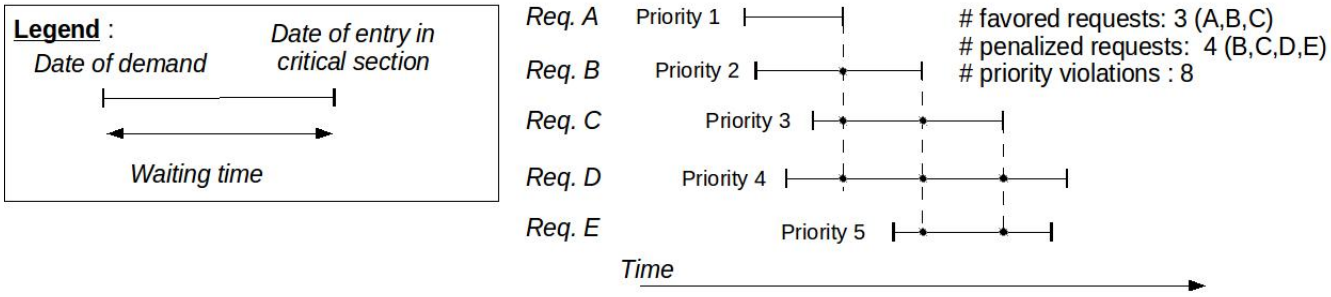


Figure 2: Priority violation

Figure 2 shows 5 requests with their respective original priority where the horizontal lines represent the pending interval of each request (critical section execution starts at the end of each pending interval but is not shown in the figure). For instance, request A is a favored one since it is satisfied while request B is pending. Therefore, request B is a penalized request. Notice that a request can be both a favored and penalized one (for instance, requests B and C).

We discretize the global time by events of the algorithm execution such as CS request or token acquisition. We denote T such a discrete-time. Let the triplet $(p, t_r, t_a) \in R \subset \mathcal{P} \times t_r \times t_r$ be a request where p is the priority of the request, t_r is the time when the request was issued, and t_a is the time when the token was acquired by the requesting process.

The priority violation can then be formalized by:

- the number of **favored** requests:
  $\#\{(p, t_r, t_a) \in R \mid \exists (p', t'_r, t'_a) \in R, p < p' \wedge t_a \in]t'_r, t'_a[\}$
- the number of **penalized** requests:
  $\#\{(p, t_r, t_a) \in R \mid \exists (p', t'_r, t'_a) \in R, p' < p \wedge t'_a \in]t_r, t_a[\}$
- the total number of **priority violations**:
  $\#\{((p, t_r, t_a), (p', t'_r, t'_a)) \in R^2 \mid p' < p \wedge t'_a \in]t_r, t_a[\}$

In Figure 2, the above violation concepts are illustrated as follows:

- The number of vertical dotted lines corresponds to the number of favored requests (3 in the example);
- The number of horizontal lines (requests) which have at least one dot (cross with a vertical line) corresponds to the number of penalized requests (4 in the example);

- The number of dots corresponds to the total number of priority violations; this number corresponds to the total number of priority inversions. (8 in the example)

This example clearly shows that the number of priority violations is not equal to the sum of the number of penalized requests plus favored requests. Such a difference motivates the need to formally define this concept.

Note that in our previous work [7], the number of violations was equal to the number of penalized requests.

3.2. Our priority-based request algorithm

Our solution is based on the Kanrar-Chaki algorithm. It is scalable with regard to the number of messages (complexity $\mathcal{O}(logN)$) and starvation never takes place thanks to the mechanism of priority increment. Our proposal is therefore to modify the Kanrar-Chaki algorithm to minimize the number of priority violations but without introducing much overhead nor degrading the performance of the algorithm. In other words, without increasing either the number of messages sent over the network nor the request response time.

To this end, we firstly applied Chang [1]'s message traffic optimization to the Kanrar-Chaki algorithm which includes requests in token messages (see section 2). Then, we applied two incremental heuristics: the *"Level"* heuristic which postpones priority increment of pending requests and the *"Level-distance"* which, in addition to *"Level"* heuristic, uses the number of intermediate nodes from the current token holder to requesting nodes for deciding which node will be the next token holder.

The pseudo-code of the algorithm is shown in Figure 3. We start by describing the body of the algorithm. Then, the traffic message optimization and the two heuristics are explained.

For each site s_i, the algorithm defines the following local variables (line 1):

- *state*: *idle* if s_i does not require the CS; *requesting* if s_i waits for the CS; $inCS$ if s_i executes the CS;
- $father$: the identifier of s_i's neighbor on the path leading to the process that holds the token (root);
- Q: local queue of pending requests received by s_i. Each element of this queue is a quadruplet $(s, p, l, d) \in \Pi \times \mathcal{P} \times \mathbb{N} \times \mathbb{N}$ where *site*=the neighbor of s_i which issued or forwarded the request; p=the current priority of the request in the local queue; l=the current number of pending requests that has already been counted up in order to increase req's priority to $p + 1$; d=the distance from the node that issued the request and the current node (distance mechanism). This queue is sorted by decreasing order of priority, increasing order of distance in case of equal priorities, decreasing order of delay level in case of equal distance and then FIFO order in case of equal delay levels.

The following functions handle the Q variable:

- **add**$((s, p, l, d))$: includes a request in the local queue, according to the ordering policy described above.
- **dequeue**(Q): considering that the local queue is not empty, this function returns the first request of the local queue and removes it.
- **head**(Q): returns the first request of the local queue. The request is kept in the local queue. If the latter is empty, each field (s, p, l, d) of the returned element is equal to nil.
- **reorder**(Q): reorder the queue according to its ordering policy.

In *Request_ CS* function (line 38), a node s_j includes its request into its local queue (line 42) and, if this request is in the head of the queue, s_j sends it to its father s_i (line 44). Upon reception of the request (line 13), if s_i is the root node but not in CS, it grants the token (line 16) to s_j. If it is not the root, it adds the new request in its local queue and updates the priority of requests of its local queue in order to avoid starvation, according to the heuristics described below (lines 19 to 33). Then, if s_i has added the request in the head of its queue, it forwards the request to its own father (line 37). Note that the test on the line 18 is useful in the case where s_i has just sent the token to s_j and that the request message and the token message cross each other on the link from s_i to s_j. In this case s_i does not store the request from s_j in its local queue in order to avoid cycles since s_j is also the new s_i's father.

Whenever a node receives the token, if the token piggybacks a request (see section 3.2.1), it updates the priority of requests of its local queue and also includes the received request in its local queue (lines 60 to 68). Then, if its own request is at the head of the queue (i.e., it has the highest priority), the node enters the CS (line 72). Otherwise, the token is forwarded to the node at the head of the local queue (line 75).

Finally, when a node releases the CS by calling the function *Release_ CS* (line 49), if its local queue is not empty, it sends the token to the node at the head of its local queue, removing the corresponding request from the queue. Furthermore, if there still exist pending requests in its local queue, the node also includes the first one in the token message, but keeps it in its queue.

3.2.1. Communication traffic optimization

In the Kanrar-Chaki algorithm, whenever a site whose local queue is not empty grants the token to another process, it also sends to the latter a request to inform that the token must be returned later on. Hence, in order to reduce communication traffic, this request can be piggybacked in the token message (lines 54 and 75). To this end, the pending request which has the maximum priority in the local queue (i.e., the request at the head of the local queue) is added to the token message. When the token is received, the request is added in the queue of the receiver (line 68).

Moreover, in the Kanrar-Chaki algorithm, a non token holder process which wants to enter in CS with a priority p, systematically sends a request message to its father even if there is a request in its local queue

which has a priority higher than p. Thus, in order to reduce the number of messages per request, this process will only send the request if the latter has been included in the head of the local queue since it has the highest priority (line 43).

3.2.2. "Level" Heuristic

We have observed in the Kanrar-Chaki algorithm that requests, whose priority was originally low, are satisfied quite fast since their priority reaches the maximum value due to the increment of their respective priority in order to avoid starvation. Therefore, we have modified the algorithm aiming at postponing the priority increment. A level function, denoted $\mathcal{F}(p)$, defines the increment policy, i.e., the number of necessary requests of priority $p-1$ for upgrading pending requests of $p-1$ priority to p priority. (lines 28 to 31 and lines 64 to 67). This function is monotone, increasing, positive, and can be seen as a parameter of the algorithm. Since our main objective is to reduce as much as possible the number of violations, we have considered an exponential level function where $\mathcal{F}(p) = 2^{p+c}$ for the experiments (see Section 4). The constant c prevents that small priorities increase too fast and can be seen as a parameter of the level function.

3.2.3. "Level-Distance" Heuristic

We have introduced a new parameter, denoted *request distance*, in order to take into account pending requests' locality. We then use the distance to order requests of the same priority. The request distance from site s_r to site s_i is the number of intermediate nodes between s_r and s_i. Hence, if two pending requests have the same highest priority, the token will be sent to the one with the shortest request distance with respect to the current token holder. It is worth pointing out that the tree topology has an impact in this heuristic. However, such a mechanism can introduce starvation since it might happen that the token infinitely travel over a part of the tree where some processes, which continuously request the CS with the same priority, are located. Such a behavior can eventually induce a starvation problem whenever a process, with the same priority, is located far from this part of the tree. To overcome this problem, when a node s_n receives a request with priority p', it increments the parameter l of all requests with priority p such that $p' > p$ or p is the highest local priority and $p = p'$ (lines 27 and 63). Notice that it thus is possible that a request has a local priority equal to $p_{max} + 1$, which ensures that all requests will eventually be in the head of a local queue.

Since this heuristic is orthogonal with the previous *"Level"* heuristic, we have combined them in the *"Level-Distance"* heuristic.

3.2.4. Impact of the heuristics in the number of priority violations

Figure 4 shows the impact of the two different heuristics with respect to the original Kanrar-Chaki algorithm. We consider a tree with 8 nodes. Pending requests, stored in local queues Q_i of each node, are sorted by decreasing order of priority and by FIFO order in case of equal priority. Each of them is separated

```
1  Local variables :
2  begin
3      state ∈ {idle, requesting, inCS};
4      father : site ∈ Π or nil;
5      Q : queue of (s, p, l, d) ∈ Π × P × ℕ × ℕ ;

6  Initialization
7  begin
8      state ← idle;
9      Q ← ∅;
10     father ← according to the initial topology;
11     if self = s_0 then
12         father ← nil;

13 On_receive Request(p_j ∈ P, d_j ∈ ℕ) from s_j
14 begin
15     if father = nil and state = idle then
16         Send Token(∅,∅) to s_j ;
17         father ← s_j ;
18     else if s_j ≠ father then
19         (s_old, p_old, l_old, d_old) ← head(Q);
20         foreach (s, p, l, d) ∈ Q do
21             (s_head, p_head, l_head, d_head) ← head(Q);
22             if s = s_j then
23                 if p_j ≥ p then
24                     p ← p_j ;
25                     d ← d_j ;
26                     l ← 0 ;
27             else if p_j > p or (p_j = p and
               p = p_head) then
28                 l ← l + 1;
29                 if l = F(p + 1) then
30                     p ← p + 1;
31                     l ← 0;
32         if ∄(s, p, l, d) ∈ Q, s = s_j then
33             add (s_j, p_j, 0, d_j) in Q;
34         reorder (Q) ;
35         if father ≠ nil then
36             if (s_old, p_old, l_old, d_old) ≠ head(Q)
               then
37                 Send Request(p_j, d_j + 1) to father;

38 Request_CS(p ∈ P)
39 begin
40     state ← requesting;
41     if father ≠ nil then
42         add (self, p, 0, 0) in Q ;
43         if (self, p, 0, 0) = head(Q) then
44             Send Request(p,1) to father;
45         wait(father = nil);
46     state ← inCS;
47     /* CRITICAL SECTION                    */

48 Release_CS
49 begin
50     state ← idle;
51     if Q ≠ ∅ then
52         (s_next, p_next, l_next, d_next) ← dequeue(Q);
53         (s_head, p_head, l_head, d_head) ← head(Q);
54         Send Token(min(p_head, p_max),d_head + 1) to s_next;
55         father ← s_next;

56 On_receive Token(p_j ∈ P, d_j ∈ ℕ) from s_j
57 begin
58     father ← nil ;
59     (s_next, p_next, l_next, d_next) ← dequeue(Q);
60     if p_j ≠ ∅ then
61         foreach (s, p, l, d) ∈ Q do
62             (s_head, p_head, l_head, d_head) ← head(Q);
63             if p_j > p or (p_j = p and p = p_head) then
64                 l ← l + 1;
65                 if l = F(p + 1) then
66                     p ← p + 1;
67                     l ← 0;
68         add (s_j, p_j, 0, d_j) in Q;
69     reorder (Q) ;
70     if s_next = self then
71         /* process can enter in CS             */
72         notify(father = nil);
73     else
74         (s_head, p_head, l_head, d_head) ← head(Q);
75         Send Token(min(p_head, p_max),d_head + 1) to s_next;
76         father ← s_next;
```

Figure 3: Our solution with the Level-Distance heuristic

by a coma and noted $x(y)$, where x represents the requester and y the local priority of the request. Node n_1 is the root, i.e., it owns the token and is in critical section. Nodes n_2, n_3, and n_4 have requested the token with priority 0, 0, and 1 respectively. Such an initial state is shown in Figure 4(a).

Let's now consider that nodes n_5, n_8, and n_7 issue one request, in this order, with priority 2, 3, and 3 respectively, noted 5(2), 8(3), and 7(3) respectively.

Figures 4(b), 4(c), and 4(d) show the state of the tree after each of the three new requests has been taken into account by the original Kanrar-Chaki algorithm, the *"Level"* heuristic algorithm, and the *"Level-Distance"* heuristic algorithm respectively. Notice that, in the three algorithms, all fathers of the requesting nodes have added the received requests in their respective local queues. Furthermore, in the case of *"Level"* and *"Level-Distance"* heuristics, we consider that $c = 2$ which implies that 8 (respectively, 16 and 32) insertions of higher requests are required to a 0-level (respectively, 1-level and 2-level) priority request to be upgraded to level 1 (respectively, level 2 and 3).

Each one of the new requests has the following results on the state of the pending requests and local queues of the algorithms:

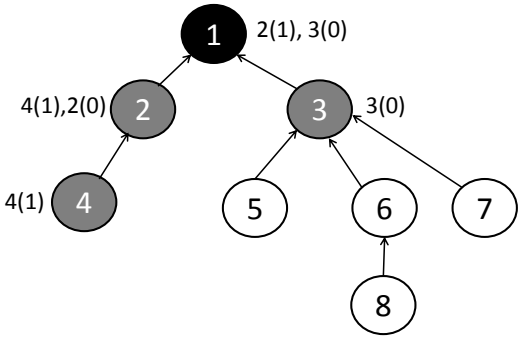
- original Kanrar-Chaki (Figure 4(b)):
  (1) The priority of n_3's pending request in both Q_3 and Q_1 as well as the priority of n_2's pending request in Q_1 are increased till 3.
  (2) The final satisfaction order of requests is: 4(1),5(2),8(3),3(0),7(3),2(0).

- *"Level"* heuristic (Figure 4(c)):
  (1) The priority level of the n_3's pending request in Q_1 is increased till 3 but does not change in Q_3.
  (2) The final satisfaction order of requests is: 8(3),7(3),5(2),4(1),2(0),3(0).

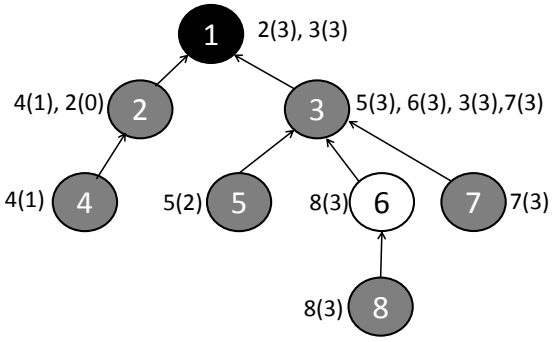
- *"Level-Distance"* heuristic (Figure 4(d)):
  (1) Requests in Q_3 are rescheduled according to requester's distance.
  (2) The final satisfaction order of requests is: 7(3),8(3),5(2),4(1),2(0),3(0).

This execution examples clearly show that the different heuristics change the order in which requests are satisfied. We can also observe that both heuristics keep the original priority order. According to the concepts of priority violation defined in section 3.1, we have:

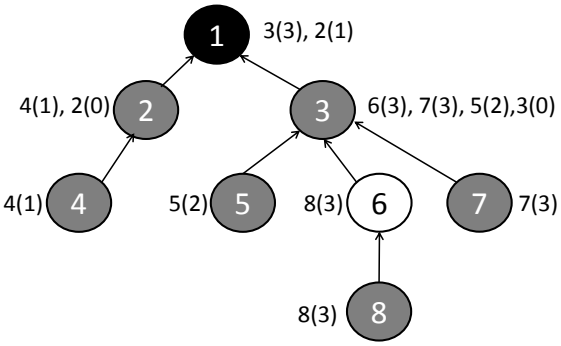
- three favored requests (n_4, n_5, n_3), three penalized requests (n_5, n_8, n_7) and six priority violations for the Kanrar-Chaki algorithm
- no priority violation when the heuristics is applied.



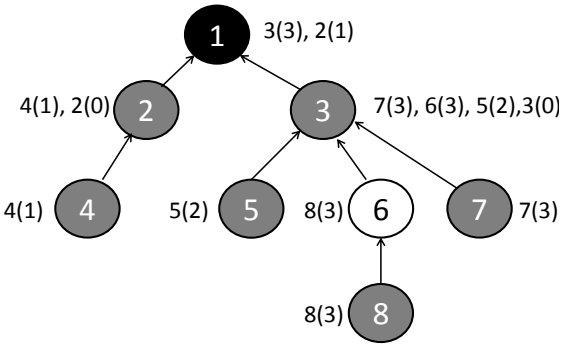
(a) Initial state



(b) End state with Classical Kanrar-Chaki algorithm



(c) End state with *"Level"* heuristic



(d) End state with *"Level-Distance"* heuristic

(e) Legend

Figure 4: Example of execution by heuristics

4. Performance Evaluation

Compared to our previous work [7], we have significantly extended the performance evaluation study of the algorithms. The results are presented in this section.

4.1. Experimental testbed and configuration

The experiments were conducted on a 32-nodes cluster with one process per node. It is worth emphasizing that there is no network contention since there is one process per network card. Therefore, the side effect due to the network is limited since there is just one process per network card. Each node has two 2.5GHz Xeon processors and 16GB of RAM, running Linux 2.6. Nodes are linked by a 20 Gbit/s Ethernet switch. The algorithms were implemented using C++ and OpenMPI.

An application is characterized by:

- N: number of processes.
- α: time to execute the critical section (CS).
- β: mean time interval between the release of the CS by a node and its request by this same node.
- γ: network transmission delay between two neighbor nodes.
- ρ: the ratio $\beta/(\alpha+\gamma)$, which expresses the frequency with which the critical section is requested. The value of this parameter is inversely proportional to load: a low value implies a high request load and vice-versa. In other words:
  - **High load** ($0.1N \leq \rho < 0.375N$): a scenario where the majority of application processes request the critical section;
  - **Intermediate load** ($0.375N \leq \rho < 3N$): a scenario where some sites compete to get the CS;
  - **Low load** ($3N \leq \rho \leq 10N$): a scenario where concurrent requests to the CS are rare.

  The parameter γ must be take into account whenever its value is not negligible. In this case, the transfer of the token message can be seen as an extension of the critical section time α, decreasing, therefore ρ.
- θ: the duration of the experiment.

The following metrics were considered in our experiments:

- *Number of messages per request*: for a given type of message, it is the ratio between the total number of messages of this type and the total number of requests.

- *Number of priority violations*: described in section 3.1. However, we express it as the percentage of issued requests, i.e., it is normalized with regard to the number of requests.
- *Response time*: the delay between the moment a node requests the CS and the moment it gets access to it.
- *CS execution rate*: ratio of the sum of all requested critical section execution durations over θ.

For all experiments, we have considered a logical binary tree topology and 8 different priority levels. In the figures that follow, *CommOpti* corresponds to a modified version of Kanrar-Chaki algorithm with the communication optimization of Chang algorithm described in section 2, while *CommOpti_Level* and *CommOpti_LevelDistance* correspond to this message traffic optimized algorithm when the *"Level"*, and *"Level-Distance"* heuristics are respectively applied to it. We have also included Chang's algorithm (see section 2) in our performance evaluation experiments.

We classify the above algorithms in two classes: (1) *"no-level"* which comprises Kanrar-Chaki, Chang, and *CommOpti* algorithms and (1) *"level"* composed by *"Level"*, and *"Level-Distance"*algorithms.

4.2. Constant load during an experiment

In this section, we present and discuss some performance evaluation results when request load within the same experiment does not vary. Processes issue a request periodically. The interval between two requests of the same process is chosen randomly according to a Poisson distribution where the average is computed using the parameter ρ (load). For each new request, every process randomly chooses a priority according to a uniform distribution.

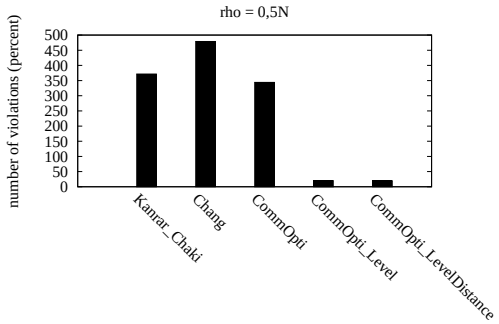
In order to have a stationary request rate scenario, the first five CS accesses of each site are not taken into account. Consequently, the average request load keeps constant during the whole experiment.

Our study was divided in two phases. We firstly considered one fixed average load and then, we evaluated the impact of different loads, but each one does not vary during each experiment, on the behavior of the algorithms.

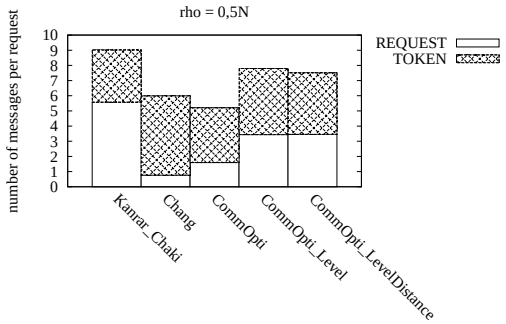
4.2.1. Single constant load

Figure 5 summarizes the behavior of the algorithms with regard to the metrics described in section 4.1 when the load is fixed to $\rho = 0.5N$ (around of 51 % processus are waiting to access the CS).

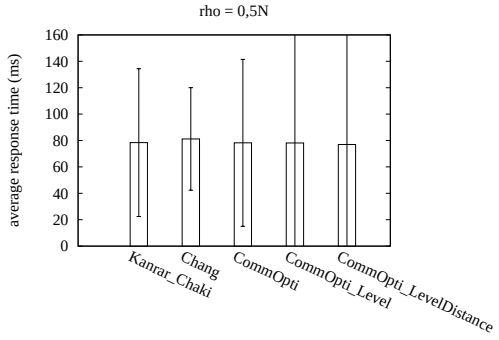
We can observe in the Figure 5(a) that the number of priority violations of Kanrar-Chaki algorithm is 25% smaller than Chang's algorithm. However, this gain is obtained at the expense of message complexity (Figure 5(b)), i.e., 45 % of additional messages. On the other hand, the performance of *CommOpti* confirms that the simple addition of the message traffic optimization mechanism in Kanrar-Chaki algorithm is enough



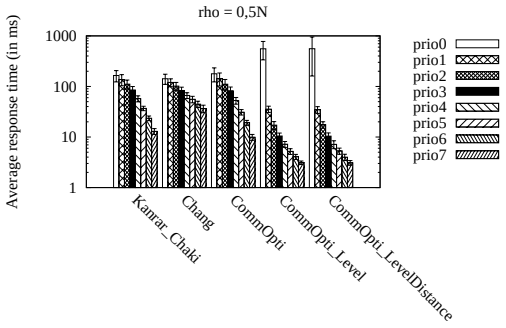
(a) Total number of violations in percentage of request



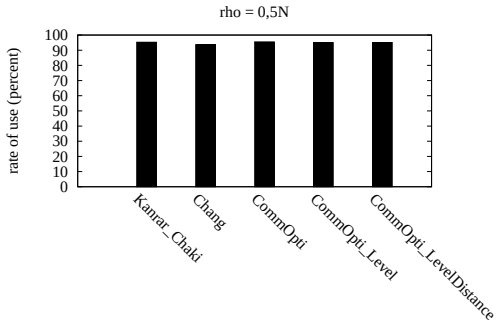
(b) Number of messages sent in network per request ordered by type



(c) Response time to obtain the token



(d) Response time to obtain the token by priority



(e) CS execution rate

Figure 5: Performance of priority-based algorithms

to obtain a message complexity which is of the same order of Chang algorithm with the same number of violations of the original algorithm.

Nevertheless, the number of priority violations in *CommOpti* is still very high. The *"level"* heuristic, i.e., *CommOpti_Level*, considerably reduces the number of violations: a reduction ratio of 25. We should point out that this amazing reduction takes place despite the increase in the number of messages (40 % of message overhead). Such a behavior can be explained since processes reach the highest priority more slowly when compared to *CommOpti* and, therefore, are more likely to forward a greater number of requests that originally had a higher priority. Aiming at reducing the message overhead generated by the *"level"* heuristic, the *"distance"* heuristic was introduced in the algorithm: *CommOpti_LevelDistance* presents a reduction of 15 % of messages when compared to *CommOpti_Level* and still has a very small number of priority violations, similarly to the latter.

Concerning the average response time, we can see in Figure 5(c) that the global average response time is the same for the different algorithms but the standard deviation is quite high, especially for *"level"* algorithms. Indeed, Figure 5(d) shows that the waiting depends on the priority. The original algorithm of Kanrar-Chaki has a regular behavior (shape of stairs), i.e., the higher the priority, the shorter the average response time. However, the other algorithms do not present such a regular behavior for different priorities: response time of priority 0 is hugely increased (a "best-effort" approach) while the highest priorities present a strong improvement in CS access time. When we compare the response time of *CommOpti_Level* and *CommOpti_LevelDistance*, we can observe that there is no much difference in terms of average. On the other hand, the reduction in the number of messages of *CommOpti_LevelDistance* induces an increase in the standard deviation of the lowest priorities.

Finally, Figure 5(e) shows that the heuristics do not degrade the overall performance: the CS execution rate is the same for all of them (around 95%).

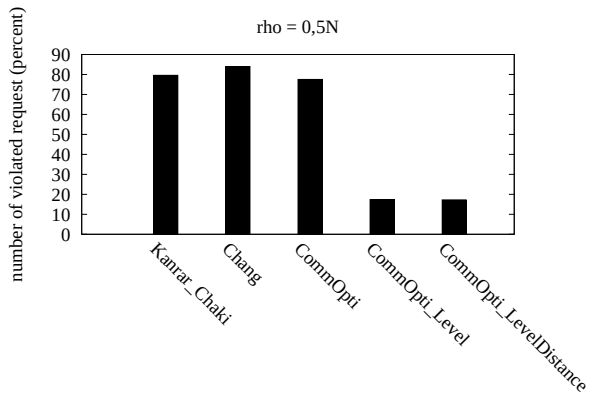
In conclusion, the above results confirm that the postponement of priority increment is essential for respecting priority order while request locality is effective in reducing the number of messages generated by an algorithm.

Study of priority violation:

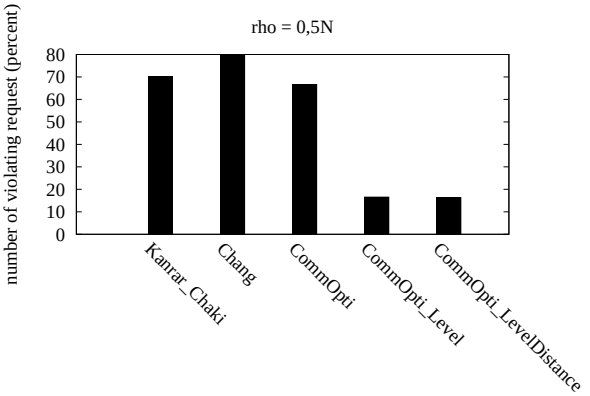
Figure 6 presents some evaluation results aiming at thoroughly studying priority violations.

For $\rho = 0.5N$, the sub-figures show (see section 3.1):

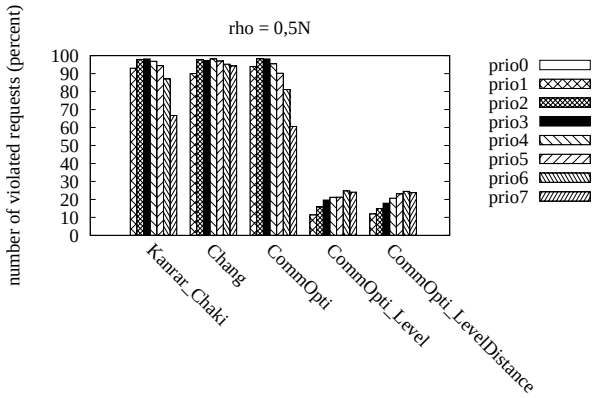
- Figure 6(a) and 6(b): the percentage of penalized requests and favored requests respectively;
- Figure 6(c) and 6(d): for each priority level, the percentage of penalized requests and favored requests respectively;



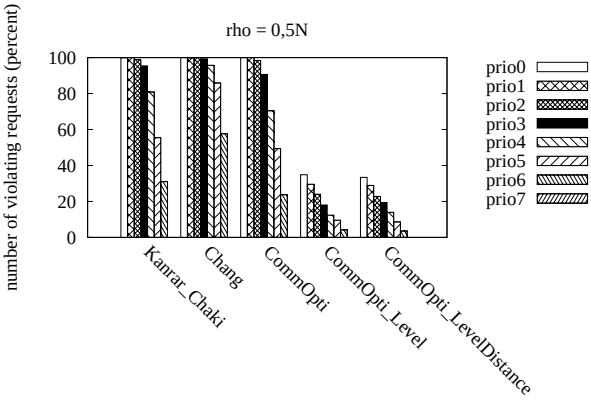
(a) Number of penalized requests in percentage



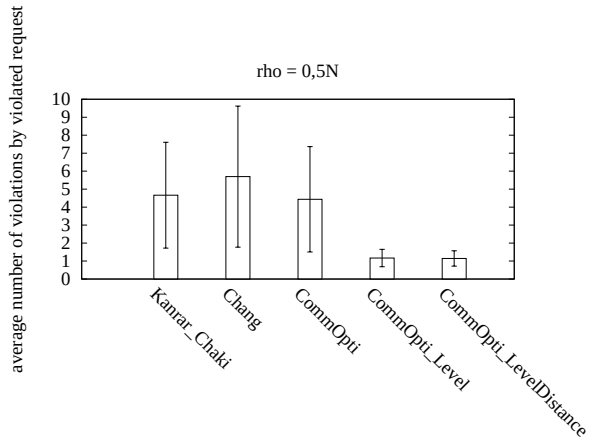
(b) Number of favored requests in percentage



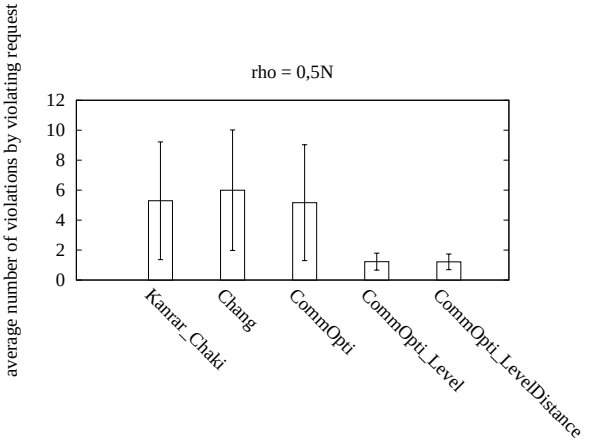
(c) Number of penalized requests per priority



(d) Number of favored requests per priority



(e) average number of favored requests per penalized request



(f) average number of penalized requests per violating request

Figure 6: Priority violation analysis

- Figure 6(e) and 6(f): the average number of penalized requests per favored request and the average number of favored requests per penalized request respectively as well as their respective standard deviation.

As we can note in Figures 6(a) and 6(b), *"no-level"* algorithms induce more penalized requests than favored ones, while in *"level"* algorithms the number of both type of requests is the same and respectively smaller than the former.

In Figure 6(c), we observe the above difference in absolute value for each priority. However, such a difference for penalized requests is not the same for all classes of algorithms: in *"level"* algorithms, the number of penalized requests increases linearly with the priority while in *"no-level"* algorithms, the most penalized requests are those whose priority has an intermediate value.

To better understand these differences, we should remember that the chance of penalization of a request depends on two factors:

- the number of requests that surpass a higher priority request depends only on the initial priority of this request, and not on the algorithm itself: the higher the priority, the greater the number of requests that is likely to surpass it.
- the probability of a request to surpass another one depends on both the priority increment mechanism and the initial priorities.

In *"level"* algorithms, the increment postponement approach renders the second factor negligible. The first factor thus explains the linearity between the number of violations and priority. Conversely, in *"no-level"* algorithms, priority increment is fast and, therefore, non negligible. We then observe that:

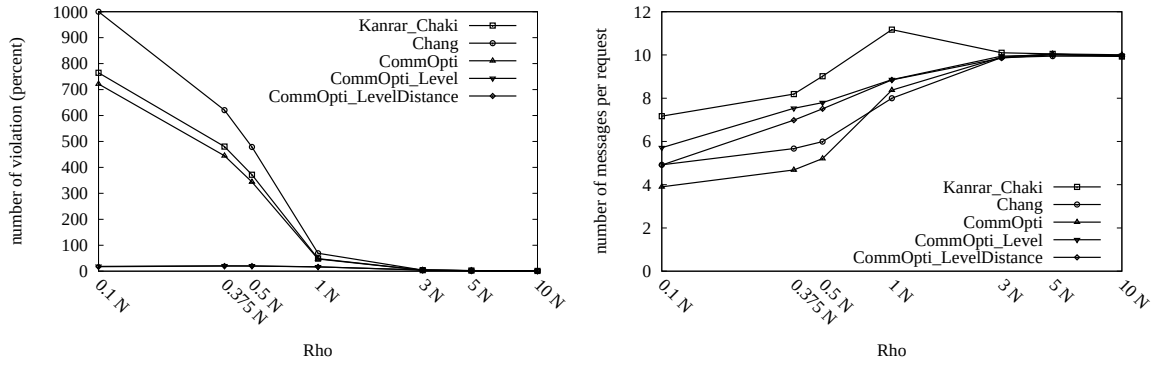
- The rate of increments is faster for lower priority (priority 0 and 1) since they have higher chance of being surpassed by a higher priority requests. Such a higher increment rate strongly penalizes priority 3 (an intermediate priority).
- Requests with the highest priority values (6 and 7) may be surpassed by many requests. However, these requests either have priority values 5 and 4 which increase slowly or have priority values which are much smaller than 6 and 7 (priorities 0 and 1).

Therefore, in *"no-level"* algorithms there is a trade-off between the two factors which tends to penalize more those requests with intermediate priorities than the others.

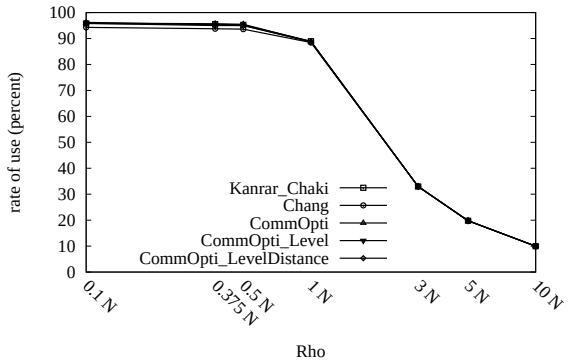
Figure 6(e) shows that in *"level"* algorithms, penalized requests are surpassed only once (small standard deviation), while in *"no-level"* algorithms requests are surpassed in average 4 to 6 times. Furthermore, by Figures 6(a) and 6(b), we can conclude that in these algorithms the number of penalized requests is small and the latter are surpassed by a small number of requests. Such a behavior explains the very good results in terms of priority violations of Figure 5(a). This same observation also applies to Figure 6(f).

4.2.2. Impact of different loads

We present now some performance evaluation results related to the impact of different loads on each algorithm. We conducted several experiments varying ρ, proportionally to the number of processes N: $0.1N$, $0.375N$, $0.5N$, $1N$, $3N$, $5N$, and $10N$ which respectively correspond to 84.3%, 61.2%, 50.9%, 11.6%, 0.5%, 0.2% and 0.07% of processes waiting for accessing the CS (x-axis of figures 7 and 8).



(a) Total number of violations in percentage of request (b) Number of messages sent in network per request



(c) CS execution rate

Figure 7: Impact of the load over the number of messages, CS execution rate, and the number of violations

Figure 7(a) shows the total number of priority violations. We can observe that *"level"* algorithms are insensitive to high loads: the number of violations remains low regardless of the value of ρ. In contrast, the number of violations increases significantly for *"no-level"* algorithms when the load increases. Such a behavior can be explained since higher loads present more concurrent requests which lead lower priority requests to quickly upgrade their priority to the highest value. These algorithms can then no longer distinguish priorities, generating, therefore, a large number of priority violations.

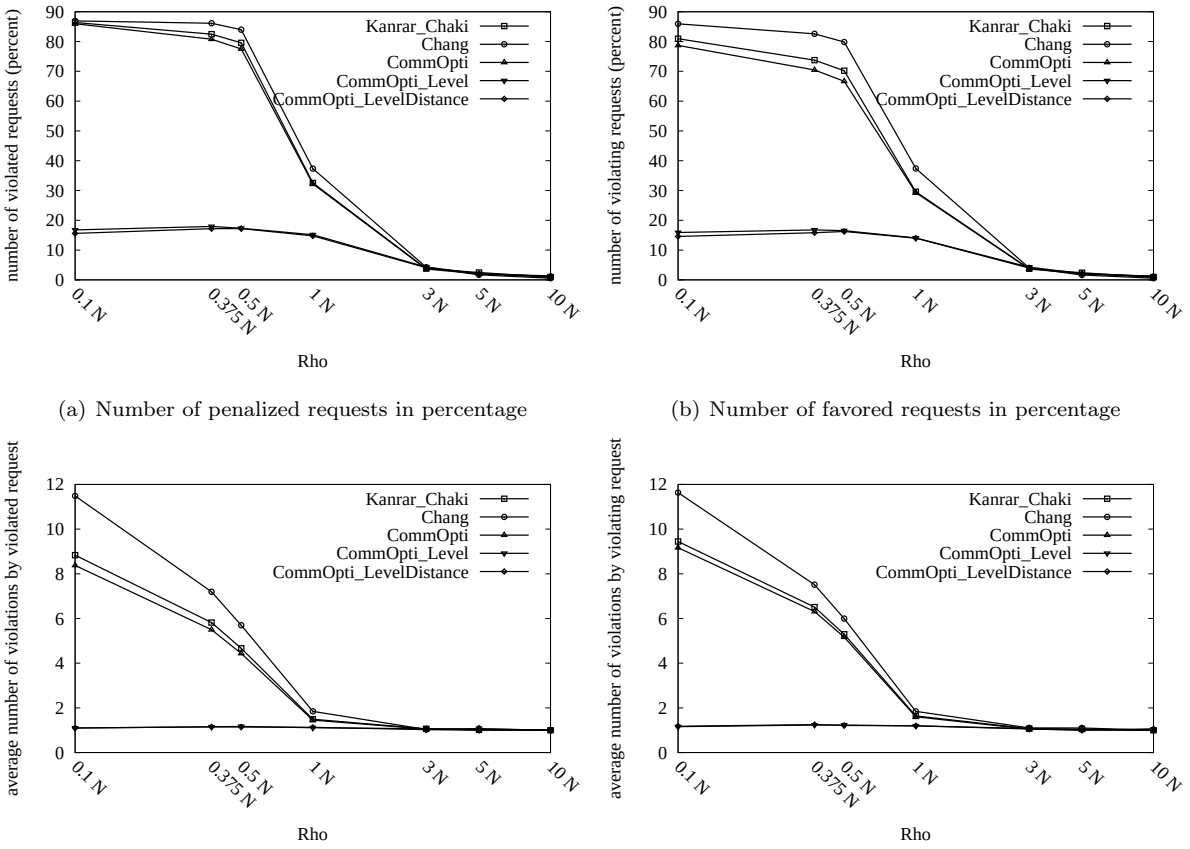
On the other hand, we observe in Figure 7(b) that the number of messages decreases when the load increases whichever the algorithm. This behavior is a direct consequence of Raymond's algorithm: a process does not retransmit the request if it is already requesting the critical section. This figure also shows, as

discussed in section 4.2.1, the message overhead generated by the *"level"* heuristic, and the gain in terms of number of messages reduction provided by the *"distance"* heuristic. It is worth remarking that the effectiveness of the *"distance"* heuristic is all the more important as the load increases. The algorithm becomes more effective for a load of $0.1N$ (84.3% of waiting processes). This is particularly useful for applications with peak loads.

Concerning the access to critical section, we note in Figure 7(c) that all algorithms have the same behavior, i.e., for a given value of ρ, any algorithm satisfies the same number of requests.

It is important to emphasize that the three sub-figures of Figure 7 confirm that in the case of low load, algorithms *"no-level"* and *"level"* behave similarly.

Study of priority violation



(a) Number of penalized requests in percentage

(b) Number of favored requests in percentage

(c) average number of favored requests by penalized request

(d) average number of penalized requests by favored request

Figure 8: Impact of Load in Priority violation

Figures 8(a), 8(b), 8(c), and 8(d) respectively show the number of penalized requests, the number of favored requests, the average number of times when a request is penalized, and the average number of times

when a request is favored.

In Figure 8(a), we observe that in the case of low load ($10N$), there is no penalized request whichever the algorithm. When the load starts increasing (from $10N$ to $3N$), some few requests are penalized. On the other hand, with an intermediate load (from $3N$ to $0.5N$), only in *"no-level"* algorithms, the number of penalized requests increases significantly (up to 80%), while in *"level"* algorithms such a number increases slightly. Finally, when the load is high (from $0.375N$ to $0.1N$), the percentage of penalized requests increases to 85% in *"no-level"* algorithms which corresponds to the proportion of requests having an initial priority strictly greater than zero. Notice that requests whose initial priority is 0 can not be penalized. In other words, in *"no-level"* algorithms, 100% of requests prone to be penalized, denoted penalizable, are eventually penalized. We find a similar behavior for favored requests (Figure 8(b)).

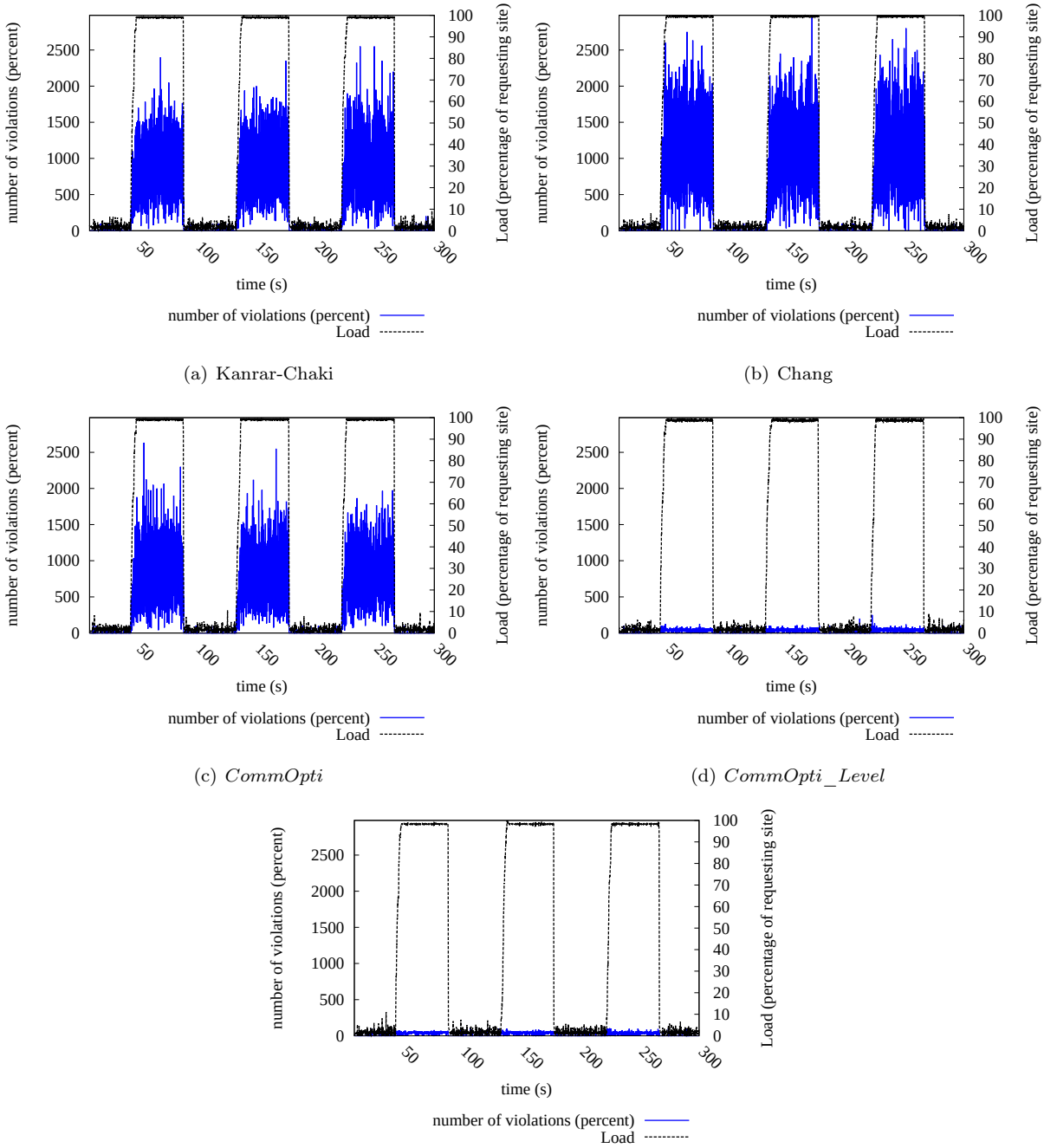
Figure 8(c) focuses on the average number of requests exceeding a penalized request. Comparing this figure with Figure 8(a), we note that when the number of penalized requests becomes relatively high (40%) for an intermediate load ($1N$), they are surpassed in average twice. Beyond the threshold of $0.5N$, almost all penalizable requests are penalized, but the number of surpassing requests continues to grow strongly which explains why the total number of violations continues to increase in heavy load (between $0.5N$ and $0.1N$) in Figure 7(a). We can, therefore, explain the bad performance in high load scenarios: all penalizable requests are surpassed around 10 times.

4.3. Dynamic load during an experiment

The goal of the experiments described in this section is to evaluate the number of priority violations when load varies during the same experiment. We also discuss the algorithms' adaptiveness to load variation in regard to priority violations. Aiming at ensuring a regular behavior of the algorithms, peak loads are injected at regular interval during an experiment. We consider that load is characterized by the percentage of processes which are waiting for the token.

Figures 9(a), 9(b),9(c), 9(d), and 9(e) show the number of violations for Kanrar-Chaki, Chang, $CommOpti$, $CommOpti\_level$, and $CommOpti\_levelDistance$ respectively.

Figure 9 shows, for all algorithms, the number of priority violations at a given interval of each experiment. A given abscissa point in each sub-figure is a sample equals to an interval of 50 milliseconds. For a given time sample, a point of a curve corresponds to the ratio of the total number of priority violations over the total number of satisfied requests within the corresponding 50 millisecond interval. We have considered the percentage of violations and not the absolute number of them because the throughput of critical section is not exactly the same among different algorithms. Figures 9(a), 9(b), and 9(c) confirm that the percentage of the number of priority violations increases significantly during the whole peak load for *"no-level"* algorithms. In fact, this metric varies between a minimum value (around 100 %) and a maximum value (around 2500 %). Such a result is in accordance with Figure 7(a) where we could observe that no violation takes place



(a) Kanrar-Chaki

(b) Chang

(c) *CommOpti*

(d) *CommOpti_Level*

(e) *CommOpti_LevelDistance*

Figure 9: Number of violations during the experiment for a dynamic load

in case of low load since the number of requests increases faster than the number of violations. In contrast, *"level"* algorithms are insensitive to peak load. The number of violations is bounded by a maximum value which is smaler than the minimum value of *"no-level"* algorithms. Such a result is consistent with both Figures 8(c) and 8(d) where a request is penalized, on average, just once, regardless the load.

In conclusion, the present study of dynamic load confirms that *"level"* algorithms are load adaptive with regard to priority violations.

4.4. Constant load and priority per process

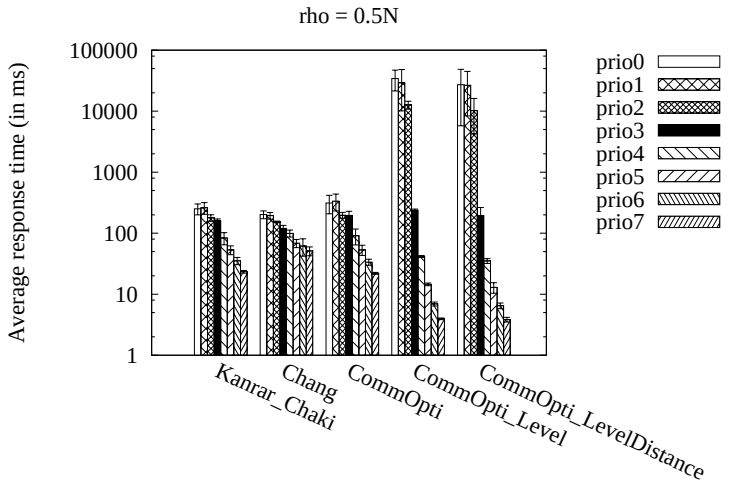
In the previous experiments, priorities were randomly chosen at each new request. However, in many applications, priorities are assigned to processes. Thus, contrarily to the previous experiments, in such applications, a process issues all its requests with the same priority during the whole application execution. On the other hand, in our approach, algorithms are based on a static logical tree topology and, therefore, nodes' position has an influence performance. The aim of the current section is to study the impact of priority distribution over the tree in the performance of the algorithms.

We denote **center** of the graph the set of vertices whose eccentricity is equal to the graph's radius. Therefore, the maximum distances between vertices of the center (central points) and other vertices of the graph are minimized.

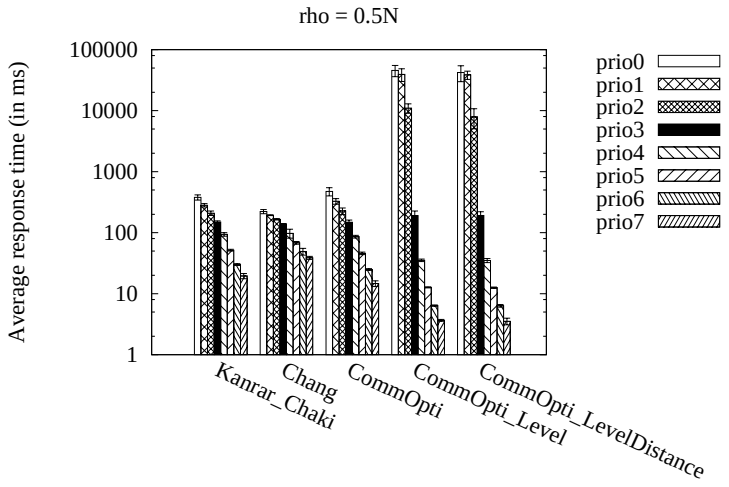
The experiments have been conducted with constant load ($\rho = 0.5N$). Every process issues its requests with the same given priority. We have considered three different priority distributions:

- **Random**: processes are randomly distributed over the tree independently of their respective priority (Figure 10(a)).
- **High center**: processes that issue requests with the highest priorities are assigned to sites of the center. Thus, the further the process is from the center, the lower its priority. (Figure 10(b)).
- **Low center**: processes that issue requests with the lowest priorities are assigned to sites of the center. Hence, the further the process is from the center, the higher its priority. (Figure 10(c)).

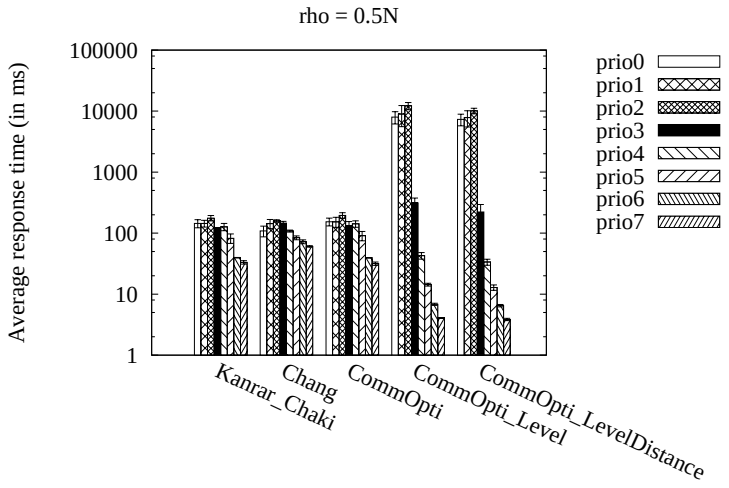
On the one hand, regardless of the configuration, we observe the same behavior of the algorithms of the previous experiments where processes issued requests with different priorities. *CommOpti_Level* and *CommOpti_LevelDistance* algorithms present the best performance. On the other hand, the current experiments show that the topology has an impact in the performance of the algorithms. By assigning processes whose request have the highest priority in the center of the graph (High center, Figure 10(b)), the critical section access time is reduced when compared to the random distribution for all algorithms (Figure 10(a)). This result is quite obvious (see Tables 11(a) and 11(b)) since sites of the center have more chance to intercept the token, i.e., requests of the processes in the center, which have high priorities, will be satisfied



(a) Randomly priority positioning



(b) High priority graph center positioning



(c) Low priority graph center positioning

Figure 10: Average response time according to priority position in the tree

Centre non prioritaire	prio0		prio1		prio2		prio3		prio4		prio5		prio6		prio7	
_	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type
Kanrar_Chaki	144,21	21,85	143,93	18,13	176,29	18,16	118,65	05,10	126,79	17,46	82,35	14,55	39,63	00,25	33,18	02,12
Chang	108,58	21,64	143,00	24,33	158,06	07,05	145,92	10,33	108,09	03,88	84,04	06,10	72,31	05,24	60,53	01,61
CommOpti	153,41	22,41	152,98	28,77	195,29	21,10	131,80	22,92	141,68	17,07	90,65	15,76	39,42	00,18	31,66	01,99
CommOpti_level	7963,77	1815,43	8946,36	3387,89	12166,91	1638,32	312,45	62,72	42,74	05,21	14,33	00,75	06,81	00,27	04,07	00,06
CommOpti_level_distance	7326,17	1582,07	7820,12	2334,06	10075,06	1079,55	221,74	73,01	33,55	03,79	12,86	01,29	06,48	00,29	03,84	00,17

(c) Low priority graph center positioning (table of values)

Centre prioritaire	prio0		prio1		prio2		prio3		prio4		prio5		prio6		prio7	
_	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type
Kanrar_Chaki	379,09	37,13	281,63	17,56	208,94	17,85	149,36	06,87	93,83	06,41	51,62	01,73	29,97	01,05	19,49	01,93
Chang	221,79	17,36	193,30	02,48	166,10	04,25	138,61	01,70	97,51	16,24	68,19	03,49	49,06	06,18	38,66	02,23
CommOpti	473,59	73,12	326,99	34,75	226,17	28,11	148,14	12,63	86,01	04,44	45,89	02,19	24,89	00,83	14,69	01,58
CommOpti_level	45528,14	9531,29	39304,08	9208,16	11007,86	1920,55	190,47	36,32	35,24	01,70	12,70	00,26	06,37	00,16	03,65	00,13
CommOpti_level_distance	42140,91	12315,02	38438,71	5899,04	7891,51	2848,17	191,07	29,20	34,99	02,69	12,54	00,33	06,32	00,30	03,52	00,44

(b) High priority graph center positioning (table of values)

Aléatoire	prio0		prio1		prio2		prio3		prio4		prio5		prio6		prio7	
_	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type	moy	Ec-type
Kanrar_Chaki	250,64	50,61	261,27	56,61	178,62	22,08	162,29	09,17	83,25	18,66	53,14	08,80	35,26	04,84	23,38	01,19
Chang	202,14	29,32	194,14	24,42	153,29	04,57	120,03	14,09	99,40	13,02	68,58	10,63	61,49	19,43	51,18	08,47
CommOpti	312,89	105,34	334,91	102,74	197,40	22,05	193,45	34,42	90,61	26,49	53,29	10,13	33,64	03,91	21,98	00,77
CommOpti_level	34233,85	12746,94	29158,51	19071,79	12675,00	1871,17	238,27	10,42	41,50	01,56	14,53	00,73	06,93	00,52	03,95	00,12
CommOpti_level_distance	27194,55	21432,00	26597,70	18229,90	10205,80	5973,76	194,48	69,97	35,05	03,13	12,93	02,54	06,47	00,70	03,83	00,34

(a) Randomly priority positioning (table of values)

Figure 11: Average response time according to priority position in the tree

before the others reducing the distances traveled by the token. The experiments also provide two other interesting results:

- overall performance gain is the same despite of the algorithm: average response time of high priority requests are reduced by 40%.
- performance gain of $Chang$ and $Kanrar - Chaki$ algorithms with a high center configuration is worse than those of $CommOpti\_Level$ and $CommOpti\_LevelDistance$ algorithms with a random configuration. In other words, high (respectively, low) priority requests of the latter present shorter (respectively, higher) response time than the former. On the other hand, all *"no level"* algorithms present better performance in high center configuration than in random one which confirms that, for these algorithms, priority distribution has an impact in response time (see Tables 11(a) and 11(b)).

Contrarily to the other two configurations (Random and High Center), grouping processes that issue requests with low priorities in the center of the network (Low center) degrades performances of the algorithms (Figure 10(c)). We can observe an inversion in the average response time: priority 2 requests have higher response time than requests with priorities 0 and 1. This behavior shows the impact of priority distribution in the algorithm. In such a configuration, there is in fact a tradeoff between priority and location in the graph: low (respectively, high) priority requests such as 0 and 1 (respectively, 6 and 7) are favored (respectively, penalized) by their center location but penalized (respectively, favored) by their respective priority value. On the other hand, requests with intermediate priority (2 and 3) do not take advantage of the center location neither of the priority value. Consequently, they are not favored at all which explains their worst performance gain.

To conclude, when a process requests is associated to a given priority, the location in the graph has an impact in the average response time of the requests.

4.5. Synthesis

We have compared our *"Level"*, and *"Level-Distance"* algorithms with Chang and Kanrar-Chaki algorithms in two configurations. In the first one where processes can issue requests with different priorities, we could observe two results in medium and high load scenarios:

- the increment postponement of the *"level"* algorithms respects more the priority order than *"no-level"* algorithms;
- exploitation of request locality reduces message overhead induced by the increment postponement.

In the second configuration where processes issues requests always with the same priority, we observed the impact of the topology on the waiting time. If low priority processes are located in the center of the

graph, then the most penalized requests those with medium priority. Conversely, if processes that issue low priority requests are located in the graph edges, between two successive priorities, the waiting time of the highest priority requests does not increase very much, i.e., there is a better respect of priorities.

5. Trade-off between the waiting time and the number of violations

A system configuration where priorities are assigned to processes with a "**high center**" topology policy described in the section 4.4 may present a high waiting time for low priorities.

Let's remember that in such a topology, the deeper the initial position of the node in the tree, the lower the priority. Consequently, if we consider both a high number of processes in the system and request load, the token will stay most of time in the center of the graph. A process q increases priorities in its local queue only if it receives a higher priority request from its current sub-tree which, most of the time, is composed of processes with lower priority than p. Thus, priorities in a process's local queue increase only when the process receives the token with a higher priority pending request, which is quite rare for processes with low priority. Therefore, the latter presents a quite high waiting response time. To overcome this problem, we have proposed an extension of the *Level-Distance* algorithm in [8], called the *Awareness* algorithm. It provides a mechanism which allows every process to eventually know the total number of issued requests for each priority. Thus, priorities are increased by considering requests from the whole system. Consequently, requests of upper-areas of the graph will be taken into account and, therefore, requests with low priorities will be less penalized.

Figure 12 illustrates the differences between the *Kanrar-Chaki* algorithm, the *Level-Distance* algorithm, and the *Awareness* algorithm, in such a topology. This example shows the number of issued requests with an initial priority p necessary to reach the configuration 2 from configuration 1. In the configuration 1, the token is in the priority p area (the center of the graph), and processes S1 and S2 are waiting for the token with priorit $p-2$ and $p-1$ respectively. In configuration 2, S2 holds the token and the priority of $S1$ in the $S2$'s local queue has been incremented. This example clearly shows that there is a difference of factor $\mathcal{F}(p)$ between the *Level-Distance* algorithm and the *Awareness* algorithm. This difference is even greater when the level function is increasing. Table 13 summarizes for each algorithm the order of magnitude of the number of requests issued with priority p necessary for a request with initial priority p' to receive the token.

In [8], we evaluated the *Awareness* algorithm with 64 processes and high load ($p = 0.1N$). Contrarily to the performances presented in section 4.4 for the high center priority distribution, the number of priorities is directly linked to the height of the initial topology: nodes at tree level 0 (initial root node) and level 1 have the highest priorities and every other node has a strictly lower priority than its initial father (except nodes at level 1). Since there are 64 processes organized in a binary tree, the number of priorities is equal to 6 ($Log_2(64)$) instead of 8 (see section 4). In Figure 14, we compare the *Kanrar-Chaki* algorithm, the *Chang*

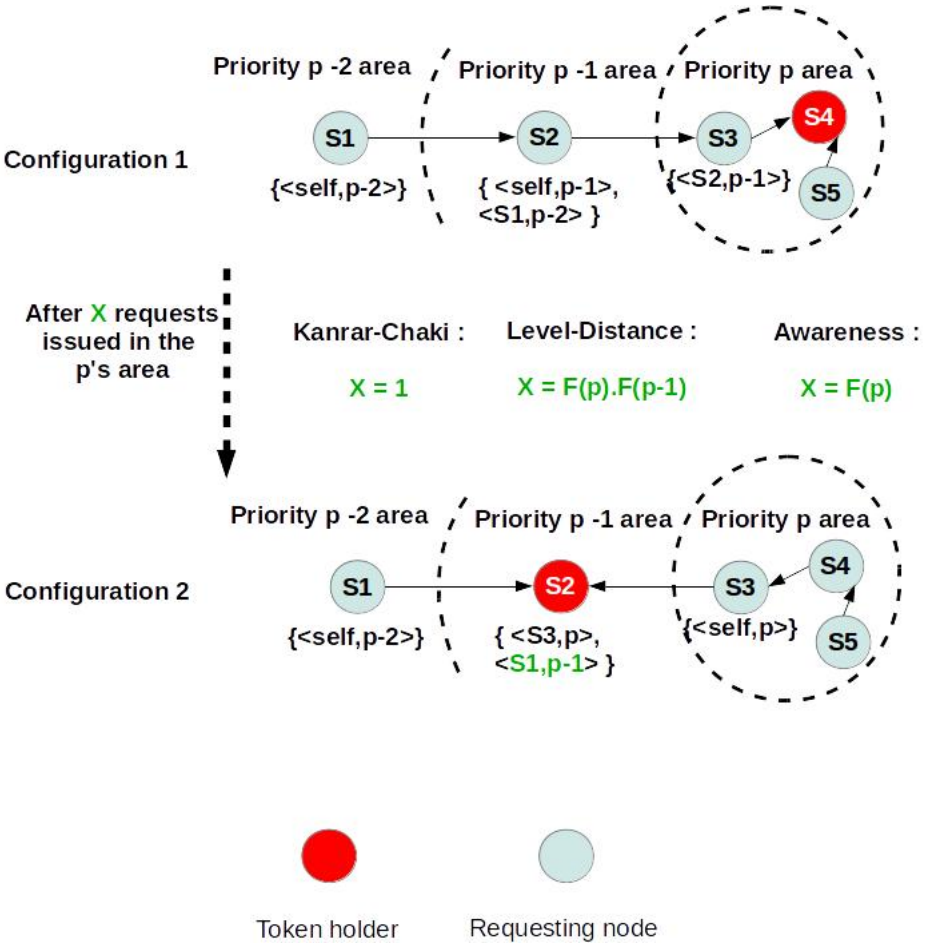
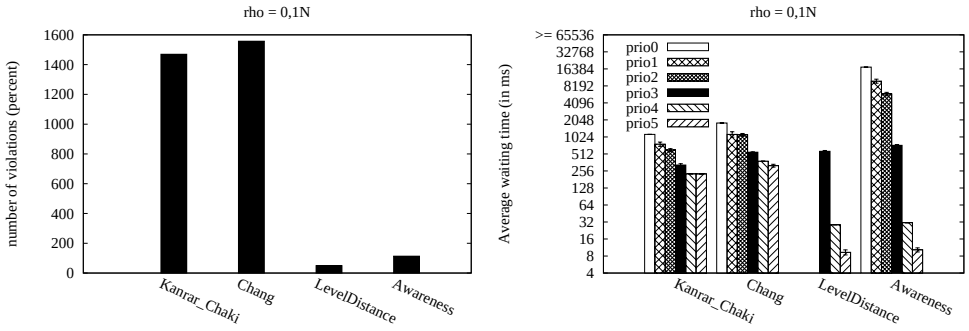


Figure 12: Differences between the three algorithms

Algorithm	order of
Kanrar-Chaki	$p - p'$
Level-Distance	$\sum_{i=p'+1}^{p} \left(\prod_{j=i}^{p} \mathcal{F}(j) \right)$
Awareness	$(p - p')\mathcal{F}(p)$

Figure 13: Order of magnitude of the number of requests issued with priority p necessary for a request with initial priority p' to receive the token

algorithm, the *Level-distance* algorithm, and the *Awareness* algorithm in terms of the number of priority violations (Figure 14(a)) and average waiting time (Figure 14(b)). In the *Level-Distance* algorithm, some requests with low priority have no response time (priorities 0, 1, and 2 for $0.1N$). Such results correspond to a huge response time for these priority levels since no request has been satisfied during the experiment, i.e., the *Level-Distance* algorithm strongly penalizes low priorities requests. We denote such a delay a *"pseudo-starvation"* since the starvation cannot occur in theory but low priority requests are satisfied within a too long interval. Comparing the later with the *Awareness* algorithms, we observe that high priorities (4 and 5) present almost the same response time in both algorithms. On the other hand, intermediate priorities (2 and 3) are more penalized in the *Awareness* algorithm than in the *Level-Distance* algorithm while low priorities (0 and 1) are much less penalized. However, this reduction of response time for the lowest priorities comes at the cost of a small overhead in terms of priority violations. Since minimizing both the number of violations and the waiting time metrics are two contradictory objectives, it is necessary to find a trade-off which depends on the application needs which is possible by defining a suitable level function. In [8], we study the impact of different level function families on these two metrics in the above algorithms. We have observed that, considering a given number of violations, for any level function, the *Awareness* algorithm considerably reduces the waiting time of low priority requests. Consequently, contrarily to the *Level-Distance* algorithm, performance of the *Awareness* algorithm does not depend on the priority position on the graph and, therefore, the above mentioned trade-off only depends on the level function.



(a) Number of violations (percentage of requests) (b) average waiting time per priority

Figure 14: Amount of violations and average waiting time

6. Conclusion

Based on Kanrar-Chaki algorithm, we have presented in this article an effective starvation-free priority-based mutual exclusion algorithm. Priorities associated to requests can dynamically increase in order to ensure that requests with low priority are satisfied within a bounded time. Starvation are thus avoided.

However, dynamic priorities induce priority inversion. Aiming at minimizing such a constraint, we have proposed two heuristics. Evaluation results confirm that by postponing priority increment ("Level" heuristic), the number of priority violations can be strongly reduced but at the expense of message overhead. On the other hand, by taking into account request locality ("Level-distance" heuristic), the number of messages sent over the network decreases.

We have also shown that our algorithm is load adaptive since there is no much variation in the number of violations for different loads. Hence, our algorithms are quite suitable for applications with peak loads. On the other hand, as observed in performance evaluation results, the postponement of priority increments induces a higher response time for the lowest priorities and the location of processes on the logical tree topology has an impact over performance. Therefore, in future work, we plan to propose priority-based algorithms based on dynamic tree topologies.

7. Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] Ye-In Chang. Design of mutual exclusion algorithms for real-time distributed systems. *J. Inf. Sci. Eng.*, 11(4):527–548, 1994.

[2] Andrzej M. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *J. Parallel Distrib. Comput.*, 9(1):77–82, 1990.

[3] Ahmed Housni and Michel Trehel. Distributed mutual exclusion token-permission based by prioritized groups. In *AICCSA*, pages 253–259, 2001.

[4] Theodore Johnson and Richard E. Newman-Wolfe. A comparison of fast and low overhead distributed priority locks. *J. Parallel Distrib. Comput.*, 32(1):74–89, 1996.

[5] Sukhendu Kanrar and Nabendu Chaki. Fapp: A new fairness algorithm for priority process mutual exclusion in distributed systems. *JNW*, 5(1):11–18, 2010.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[7] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. Service level agreement for distributed mutual exclusion in cloud computing. In *12th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'12)*. IEEE Computer Society Press, May 2012.

[8] Jonathan Lejeune, Luciana Arantes, Julien Sopena, and Pierre Sens. A prioritized distributed mutual exclusion algorithm balancing priority inversions and response time. In *42th International Conference on Parallel Processing (ICPP'13)*. IEEE Computer Society, October 2013.

[9] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3:145–159, May 1985.

[10] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 340 –349, 1999.

[11] Frank Mueller. Prioritized token-based mutual exclusion for distributed systems. In *IPPS/SPDP*, pages 791–795, 1998.

[12] Mohamed Naimi and Michel Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. In *ICDCS*, pages 371–377, 1987.

[13] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

[14] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24:9–17, January 1981.

[15] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[16] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4):344–349, 1985.

[17] Martin G. Velazquez. A survey of distributed mutual exclusion algorithms. Technical report, Colorado state university, 1993.

Annexe: Proof of correctness

Model

We consider that T is discretized by the primitives algorithm execution. A **request** $req \in R_{t'}$ is a triplet (s, t, p) of the set $\Pi \times T \times \mathcal{P}$ where (s, t, p) represents a critical section request by the site s at $t \leq t'$ with the priority p.

A **request message** $Mreq$ in transit is a quadruplet denoted

$$\langle (s_{ini}, t_{ini}, p_{ini}), p, d, s_i, s_j \rangle_{req}$$

of the set $R_t \times \mathcal{P} \times \mathbb{N} \times \Pi \times \Pi$, $t_{ini} \leq t$ which represents the message transition for the initial request $(s_{ini}, t_{ini}, p_{ini})$ from site s_i to site s_j with a priority p and a distance d between s_j and s_{ini}.

On the same principle, a **token message** $Mtok$ is a quadruplet denoted

$$\langle (s_{ini}, t_{ini}, p_{ini}), p, d, s_i, s_j \rangle_{tok}$$

of the set $R_t \times \mathcal{P} \times \mathbb{N} \times \Pi \times \Pi$ except that $(s_{ini}, t_{ini}, p_{ini})$, p and d can have $\emptyset$ as a value if no request is piggybacked in the token.

We denote M^t the set of transiting messages in the network at $t \in T$.

A process s_i can be at $t \in T$ in one of the three following states : $idle^t_{s_i}$, $requesting^t_{s_i}$ and $inCS^t_{s_i}$. Its local variables are :

- $father^t_{s_i} \in \Pi$ indicating the father of site $s_i \in \Pi$ at $t \in T$. If s_i is the root site at t then $father^t_{s_i} = nil$.
- $Q^t_{s_i}$ is the set which represents the local queue of site $s_i \in \Pi$ at $t \in T$. An element of this queue is a 6-uplet $((s_r, t_r, p_r), d, s_n, p, l, t') \in R_t \times \mathbb{N} \times \Pi \times \mathcal{P} \times \mathbb{N} \times T$ where (s_r, t_r, p_r) represents the request, d is the distance in number of links between site s_i and site s_r, s_n is the neighbor of s_i in direction to s_r, p is the local priority of $req = (s_r, t_r, p_r)$ in $Q^t_{s_i}$, l is the current level value and $t' \leq t$ is the moment where req has been added in $Q^t_{s_i}$. This set is totally ordered by the relation $\prec$ such that $\forall$ $(req_k, d_k, s_k, p_k, l_k, t_k) \in Q^t_{s_i}$ and $\forall$ $(req_{k'}, d_{k'}, s_{k'}, p_{k'}, l_{k'}, t_{k'}) \in Q^t_{s_i}$

$$(req_k, d_k, s_k, p_k, l_k, t_k) \prec (req_{k'}, d_{k'}, s_{k'}, p_{k'}, l_{k'}, t_{k'}) \Leftrightarrow$$

$$(p_k > p_{k'}) \vee (p_k = p_{k'} \wedge d_k < d_{k'}) \vee (p_k = p_{k'} \wedge d_k = d_{k'} \wedge l_k < l_{k'})$$
$$\vee (p_k = p_{k'} \wedge d_k = d_{k'} \wedge l_k = l_{k'} \wedge t_k < t_{k'})$$

The increment procedure $incr(Q^t_{s_i}, p)$ is modeled as :
$\forall (req_k, d_k, s_k, p_k, l_k, t_k) \in Q^t_{s_i}$:

1. $p > p_k \vee \big(p = p_k \wedge p_k = PH(Q^t_{s_i})\big) \wedge l_k + 1 \geq \mathcal{F}(p_k + 1) \Rightarrow (req_k, d_k, s_k, p_k + 1, 0, t_k)$
2. $p > p_k \vee \big(p = p_k \wedge p_k = PH(Q^t_{s_i})\big) \wedge l_k + 1 < \mathcal{F}(p_k + 1) \Rightarrow (req_k, d_k, s_k, p_k, l_k + \ 1, t_k)$
3. $\neg\Big(p > p_k \vee \big(p = p_k \wedge p_k = PH(Q^t_{s_i})\big)\Big) \Rightarrow (req_k, d_k, s_k, p_k, l_k, t_k)$

where $PH(Q^t_{s_i})$ is the highest local priority stored in $Q^t_{s_i}$.

Safety property

Lemma 1. *If there exists a root node, there is no pending token token message. Formally,*

$$\forall t \in T,\ \exists s_i \in \Pi,\ father^t_{s_i} = nil \ \Leftrightarrow \nexists Mtok \in M^t$$

Proof.

* We prove by recurrence $\nexists Mtok \in M^t \Rightarrow \exists s_i \in \Pi father^t_{s_i} = nil$: The property is true at t_0 ($M^{t_0} = \emptyset$ and an only s_i where $father^t_{s_i} = nil$) By assuming this property true until the moment t_k, we will prove it at the moment t_{k+1}. If the next moment is a *Request_CS* procedure execution on a site s_i, the root site does not send a $Mtok$ message and $father^{t_k+1}_{s_i} = nil$. If the next moment is a *Release_CS* procedure execution on a site s_i : $father^{t_k+1}_{s_i} \neq nil \Rightarrow \exists Mtok \in M^{t_k+1}$ if $Q^{t_k}_{s_i} \neq \emptyset$ or $father^{t_k+1}_{s_i} = father^{t_k}_{s_i}$ otherwise. This can be applied if the next moment is a *Receive_Requeqt* procedure execution on a site s_i. Since we suppose $\nexists Mtok \in M^{t_k}$, the execution of the *Receive_Token* procedure is impossible.

* We prove by recurrence $\exists s_i \in \Pi father^t_{s_i} = nil \Rightarrow \nexists Mtok \in M^t$: The property is true at t_0 ($father^t_{s_0} = nil$) By assuming this property true until the moment t_k, we will prove it at the moment t_{k+1}. If the next moment is a *Request_CS* procedure execution on a site s_i where $father^{t_k}_{s_i} = nil$ $M^{t_k+1} = M^{t_k}$ and according to the recurrence assumption, $\nexists Mtok \in M^{t_k+1}$. If the next moment is a *Release_CS* procedure execution on a site s_i, if $Q^{t_k}_{s_i} \neq \emptyset$ then $father^{t_k+1}_{s_i} \neq nil$ and a token message is sent, otherwise $father^{t_k}_{s_i} = father^{t_k+1}_{s_i} = nil$ and no token message is sent. If the next moment is a *Receive_Request* procedure execution on a site s_i, if $father^{t_k}_{s_i} = nil$ and $idle^{t_k}_{s_i}$ then $father^{t_k+1}_{s_i} \neq nil$ and $\exists Mtok \in M^{t_k+1}$. Finally, if the next moment is a *Receive_Token*, either $father^{t_k+1}_{s_i} = nil$ and $\nexists Mtok \in M^{t_k+1}$ either the token is just forwarded and $father^{t_k+1}_{s_i} \neq nil$ and $\exists Mtok \in M^{t_k+1}$.

□

Lemma 2. *$\forall t \in T$, there is at most one token message in M^t*

Proof. Since we consider reliable channels (no loss, no duplication), there is an only token message at each token sending in the network. Moreover following functions induces a token message sending :

- In *Receive_Req* when $father^t_{s_i} = nil$. In this case, according to lemma 1 $\nexists Mtok \in M^t$.

- In *Receive_Token* upon a receipt of the token. The token message is then removed in M^t if the current site enters in critical section, otherwise it forwards the token.

Consequently, it exists at most one message token in M^t $\forall t \in T$. □

Lemma 3. *These is at most a site where* $father = nil$. *Formally,*

$$\forall t \in T,\ \exists s_i \in \Pi,\ father^t_{s_i} = nil \Rightarrow \nexists s_j \in \Pi,\ father^t_{s_j} = nil$$

Proof. Except at the initialization, $father^t_{s_i} = nil$ uniquely when a site s_i receives the token at the moment t. At each token sending, the $father$ variable becomes systematically $\neq nil$. Since there exists at most one token message in the system according to lemma 2 there is at most one root site $\forall t \in T$.

□

Theorem 1 (Safety)**.** *The "Level-distance" algorithm ensures the safety property*

$$\forall t \in T,\quad \exists s_i,\ inCS^t_{s_i} \Rightarrow \nexists s_j \in \Pi,\ inCS^t_{s_j}$$

Proof. The application of lemmas 1, 2 and 3 implies that there exists at most one token in the system : either the token is in the network, either it is owned by a unique root site. Since a site can enter in critical section if it is requesting and if its variable $father = nil$, it exists at most one process in critical section.

□

Liveness property

Lemma 4. *If at* $t \in T$ *a request* req *is at the head of local queue of a site* s_i, *then* $\exists t' > t$ *where* req *will be stored in the local queue of* $father^t_{s_i}$. *Formally,* $\forall t \in T$, $\forall s_i \in \Pi$, $\exists t' > t$,

$$(req, \_, \_, \_, \_, \_) = \mathcal{H}(Q^t_{s_i}) \wedge father^t_{s_i} \neq nil \Rightarrow (req, \_, \_, \_, \_, \_) \in Q^{t'}_{father^t_{s_i}}$$

Proof. When a new element is added in the head of $Q^t_{s_i}$, the request is sent to the father. Upon receipt of this message at t' by s_i's father, the request may be forwarded till reach the root site s_k at t_k. If s_k exits the critical section and has to forward the token to a next holder, the head element is piggybacked in the token message and added at t'' in $Q^{t''}_{father^{t_k}_{s_k}}$ when $father^{t_k}_{s_k}$ will receive the token.

□

Lemma 5. *If a request* $(req_a, d_a, s_a, p_a, l_a, t_a)$ *belong* Q_{s_i} *at* $t > t_a$ *then there exists* $t' > t$ *from which, every new insertion of request* $(req_b, d_b, s_b, p_b, l_b, t_b)$, $t_b > t'$ *in* Q_{s_i} *always verifies*

$$(req_a, d_a, s_a, p_a, l_a, t_a) \prec (req_b, d_b, s_b, p_b, l_b, t_b)$$

.

Proof. Suppose that such instant t' does not exist. The element a of Q_{s_i} can forever be overtaken at each new insertion of an element b. Each insertion implies the application of function *incr*:

- either $p_b > p_a$: l_a is increased and potentially p_a. This implies eventually, $\exists t_{ph} \in T$ where $p_a = PH(Q_{s_i}^{t_{ph}})$
- either $p_b = p_a \wedge d_b < d_a$: Two cases are possible for the function *incr* :
  - $p_a < PH(Q_{s_i}^t)$: as we saw previously, this state is provisional till the time t_{ph}.
  - $p_a = PH(Q_{s_i}^t)$: l_a is increased and potentially p_a. This implies eventually, $\exists t' \in T$ where $p_a > PH(Q_{s_i}^{t'})$. Since the priority value of received requests is bounded by p_{max} then every new received request with priority p_b is eventually always lower than p_a (which in this case would be equal to $p_{max} + 1$) implying that $a \prec b$ will be true at t'. Consequently, there is a contradiction with our assumption.
- either $p_b = p_a \wedge d_b = d_a \wedge l_b > l_a$: this case is impossible because $t_a < t_b$ by assumption
- either $p_b = p_a \wedge d_b = d_a \wedge l_b = l_a \wedge t_a > t_b$: this case is impossible because $t_a < t_b$ by definition.

□

Lemma 6. *If the local queue of s_i is non empty at time $t \in T$, then $\exists t' > t$ where s_i will receive the token.*

Proof. $Q_{s_i}^t \neq \emptyset$ is equivalent to say that there exists a head element associated with a request *req*. By applying lemma 4, we can deduce that *req* is stored in the local queue of s_i's father. This can be applied recursively by message request forwarding till a site s_k which can be at $t_k \geq t$:

- either s_k does not insert the element associated with *req* in the head of $Q_{s_k}^{t_k}$: according to lemma 5, it will exist in a finite time a number of elements between the head of queue and the element associated with *req* which will decrease at each token receipt. Consequently, the element associated with *req* will be at the head of queue of Q_{s_k} in a finite time. We can thus apply again the same reasoning by taking s_k as starting.
- either s_k holds the token. We consider then three cases:
  - s_k is in critical section and the element associated with *req* in $Q_{s_k}^{t_k}$ is not the head: we apply the same reasoning as the previous point.

- s_k is in critical section and the element associated with req in $Q_{s_k}^{t_k}$ is the head: since the critical section time is assumed bounded, s_k will execute eventually the $Release\_CS$ procedure. The token will be sent in the direction of s_i. Since the transmission delay is assumed finite, s_i eventually receives the token.
- s_k is not in critical section: the token is sent in the direction of s_i which eventually receives the token.

□

Theorem 2 (Liveness)**.** *The "Level-distance" algorithm ensures the liveness property*

$$\forall s_i \in \Pi \; \forall t \in T, \; \exists t' \in T, \, t' > t, \\ requesting_{s_i}^{t} \Rightarrow inCS_{s_i}^{t'}$$

Proof. To be at the $requesting$ state at t, a site s_i has to execute the $Request\_CS$ procedure. If s_i already owns the token ($father_{s_i}^{t} = nil$) it enters directly in critical section. Otherwise, if $father_{s_i}^{t} \neq nil$ then the s_i's request is stored in $Q_{s_i}^{t}$. According to the lemma 6, site s_i eventually receives the token. When s_i receives the token at $t' > t$, the element associated with the request (s_i, t, p) is the head of the local queue, then $inCS_{s_i}^{t'}$ is true. Otherwise according to lemmas 5 and 6, s_i's request will be eventually at the head of $Q_{s_i}^{t''}$ at $t'' > t$ and then $inCS_{s_i}^{t'''}$ will be true $t''' > t'' > t$ upon token receipt.

□