# A prioritized distributed mutual exclusion algorithm balancing priority inversions and response time

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

LIP6 - UPMC, CNRS and INRIA
4, place Jussieu
75252 Paris Cedex 05, France email: firstname.lastname@lip6.fr

***Abstract*—Distributed priority-based mutual exclusion algorithms may present starvation for low priority requests if the shared resource is continuously asked by high priority requests. To address this problem, several existing algorithms dynamically increment the priority of pending low-priority requests. The drawback of this approach is that it may lead to a great number of priority inversions, i.e., a pending request $p$ is satisfied before another one whose priority is higher than $p$'s. One solution to reduce this number, as we have proposed in [7], is to both postpone priority increments and prevent low priorities from increasing too fast. However, in this case, the response time of low priorities may considerably increase. Therefore, in this article, we propose a new algorithm, denoted *"Awareness"*, which aims at reducing the maximum response time whereas the number of priority violations remains low. To this end, a global view of pending requests of the system is necessary. Performance evaluation results confirm that our new algorithm provides a good tradeoff between response time and number of priority inversions.**

## I. Introduction

Processes in distributed and parallel applications require an exclusive access to one or more shared resources. Mutual exclusion is then one of the fundamental paradigms of distributed systems. It ensures that at most one process can access the shared resources at any time (safety property) and that all requests are eventually satisfied (liveness property). The critical section (CS) is the set of instructions of processes' code that access a shared resource. We find several distributed mutual exclusion algorithms in the literature (e.g. [6],[13], [8], [15], [12], [11]). We distinguish two families among these algorithms [16]: permission-based (e.g. Lamport [6], Ricart-Agrawala [13], Maekawa [8]) and token-based (Suzuki-Kazami [15], Raymond [12], Naimi-Tréhel [11]). In the first family, a node only enters a critical section after having received permission from all the other nodes (or a sub-set of them [13], [8]). In the second one, a system-wide unique token is shared among all nodes, and its possession gives the exclusive right to execute a critical section.

Distributed mutual exclusion algorithms usually, satisfy CS requests 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 strategy can not be suitable for all kind of applications such as those which some tasks have priority over the others, applications for real-time environments [2] [1], or applications where priority is associated to a quality of service requirement [14]. To overcome these constraints, some authors (e.g., [5], [2], [9], [10], [1], [7]) have proposed some mutual exclusion algorithms (usually a modified version of the above mentioned ones) where every request is associated to a priority level. The satisfaction of pending requests, whenever possible, respects the priority order. However, priority order induces starvation problems, i.e., a requesting process never gets access to the CS, which then violates liveness property. In priority-based mutual exclusion algorithm, starvation happens because higher priority requests prevent forever lower priority ones to execute the CS.

Hence, to address such a problem, Kanrar-Chaki [5] proposed a token-based algorithm, based upon Raymond's algorithm [12], where low priorities of pending requests are dynamically increased, eventually reaching the highest priority. The main advantage of this algorithm is that it avoids starvation. On the other hand, its dynamic priority mechanism can violate priority order of requests, leading to a great number of priority inversions, i.e., a pending request with an original low priority is satisfied before another one with a higher priority than the former.

In [1], Chang proposed a dynamic priority mutual exclusion algorithm, also based uped Raymond's algorithm, by introducing an *aging* strategy where a process $i$ increases the priority of old pending requests by the number of CS executions that take place after $i$ granted the token for the last time. Such a strategy reduces the gap in terms of average response-time between priorities (contrarily to the Kanrar-Chaki algorithm), but induces much greater number of priority inversions when compared to this algorithm.

In order to improve the respect of priority order but still ensuring the liveness property, we have proposed in [7] the *Level-Distance* algorithm which extends Kanrar-Chaki algorithm by introducing a mechanism that strongly reduces the number of priority violations. Basically, our approach slows down the upgrade of a priority request: for increasing from $p-1$ to $p$ the priority of a local pending request, the node that stores it should receive a number of requests with priority higher than $p-1$ which is equal to the value of an exponential function that depends on $p$. When compared to both Kanrar-Chaki and Chang algorithms, the *Level-Distance*

algorithm presents a smaller number of priority inversions and no performance degradation in terms of number of messages. However, we have observed that the maximum response time of low priorities requests considerably increased. In other words, by postponing the priority upgrade of requests, the response time of low priority requests was degraded.

Aiming at proposing an algorithm that offers a good tradeoff between number of priority inversions and maximum response time, we present in the current article a new algorithm, denoted *Awareness*, which is an extension of our previous *Level-Distance* algorithm. We should point out that achieving both goals are not trivial: on one hand, dynamic priority and, therefore, probable priority inversions, is necessary to avoid starvation; on the other hand, the reduction in the number of priority inversions leads to higher response time for low priority requests. For providing such a tradeoff, we argue that every node needs to have a global view of the current number of pending requests of the system. Such an information is then applied by every node to postpone priority upgrades of pending requests, contrarily to the above algorithms that increase priority based on requests received by each node. Furthermore, in the *Awareness* algorithm, the number of required global requests to upgrade a priority is defined by a user function that depends on the priority value itself. It is worth emphasizing that a consequence of this global-view upgrade mechanism is that old pending low priority requests are favored, and, therefore, their response time is reduced without increasing the number of priority inversions. Performance evaluation results confirm that the *Awareness* algorithm provides the proposed tradeoff.

The rest of the paper is organized as follows. Section II discusses some existing priority-based mutual exclusion distributed algorithms and gives a description of some algorithm with dynamic priorities. Section III presents some definitions as well some assumptions about the considered system. The *Awareness* algorithm is described in section IV. Performance evaluation results are presented in Section V. Finally, Section VI concludes the paper.

## II. Related Work

In this section we outline the main priority-based distributed mutual exclusion algorithms. We classified them in two families: static priorities and dynamic priorities. In the first family, requests keep their original priority till entering in critical section (CS): this mechanism induces a strict compliance with priorities but violates the liveness property if higher priority requests prevent forever lower priority ones to execute the CS. In the second family, priorities are incremented each time that a request of higher priority is issued in the system: this mechanism keeps the liveness property but degrades the priority order by generating priority inversions.

### A. Static prioritiy Algorithms

**Goscinksi algorithm** [2] is based on the token-based Suzuki-Kasami algorithm and has a complexity of O(N), N being the number of nodes. 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 and thus is not held by any node.

**Mueller algorithm** [9] is inspired in Naimi-Tréhel token-passing algorithm which exploits a dynamic tree as a logical structure for forwarding requests. Each node keeps a local queue and records the time of requests locally. These queues form a virtual global queue ordered by priority. The proposed implementation is quite complex and the dynamic tree tends to become a simple queue because, unlike the Naimi-Tréhel algorithm, the root node is not the last requester but the token holder. Therefore, in this case the algorithm presents a message complexity of $O(\frac{N}{2})$. 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.

**Housni-Tréhel algorithm** [3] adopts a hierarchical approach where processes are grouped by priority. Each group is represented by one router process. Within each group, processes are organized in a static logical tree like Raymond's algorithm [12] and routers apply the Ricart-Agrawala algorithm [13]. A process can only send requests with the same priority (that of its group).

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. Their third algorithm extends Raymond's algorithm. Similarly to the Kanrar-Chaki algorithm, each node 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.

### B. Dynamic priority Algorithms

Since all the algorithms described in this section are based on the Raymond algorithm, we describe this latter first. We then describe the Kanrar-Chaki algorithm [5], Chang algorithm [1] and our Level-Distance algorithm published in [7].

**Raymond's** algorithm [12] 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. Nodes thus form a directed path tree to the root. Excepting the root, every node has a father node. 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 node 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 node releases the token, it sends the token message to the first process of its own local queue and this node becomes its father. When a process receives the token, it removes the first request from its local queue. If the process's own request is the first element of its local queue, it executes the critical section otherwise it forwards the token to the first element of its local queue, and

the latter becomes its father. Moreover, when a node sends a token, if the local queue of the node is not empty, it sends to its new father a request on behalf of the first request of its queue.

**Kanrar-Chaki** [5] modify Raymond's algorithm introducing a priority level for every process CS request. The greater the level (an integer value), the higher the priority of the request. Then, pending requests of a process's local queue are 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 first element of the local queue or it the only one of the queue. In order to avoid starvation, the priority level of pending requests of a process's local queue is increased: whenever a 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 applying dynamic priorities to requests and at reducing communication traffic. For the priority, he added a mechanism denoted *aging* strategy: if process $p$ exits the CS or if it is a non requesting node that holds the token and receives a request, $p$ increases the priority of every request in its local queue. Furthermore, the token includes the current number of CS already achieved. Upon reception of the token, $p$ 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 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 evaluation discussion of both algorithms is presented in section V. To reduce communication overhead, Chang proposes the following mechanism. Since a request always follows the token from an intermediate node whose local queue contains more than one element, Chang's communication traffic optimization consists in piggybacking, whenever possible, a request on the token message.

Aiming at reducing the number of priority inversions, we have proposed in [7], the **Level-Distance** algorithm which is based on Kanrar-Chaki algorithm. To this end, we have introduced a *Level mechanism* where, contrarily to Kanrar-Chaki algorithm, a priority in a local queue is not incremented at every insertion in the local queue of a request with a higher priority but only after $X$ request insertions (a threshold) with such a priority. The value of the threshold depends on an exponential level, i.e., to upgrade its priority to $p$, a request of priority $p-1$ must wait $2^{p+c}$ insertions of requests with higher priority. The constant $c$ prevents that small priorities increase too fast. To reduce the cost in messages, we used the Chang's communication traffic optimization regarding the piggybacking of a request in the token message. In addition, in order to improve the throughput of the algorithm, we proposed the *Distance mechanism* that orders requests of same priority based on their locality. We denote the request distance from site $R$ to site $S$ the number of intermediate nodes between $R$ and $S$ that the token must travel. 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.

## III. Definitions and assumptions

We consider a distributed system consisting of a finite set of 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. Processes do not share memory and communicate by sending and receiving messages. Sites and links are not prone to failures. Nodes are organized in a static logical tree and the root and only the direction of links between two neighbor nodes can change.

Applications behave correctly: a process requests a CS by calling the $Request_CS$ procedure and releases it by calling the $Release_CS$ procedure. Furthermore, a process can not request a new CS before its previous one has been satisfied.

A priority is associated to each request. Let $P$ be the finite ordered set of possible request priorities. Like in Kanrar-Chaki algorithm, priority $p$ is higher than priority $p'$, if $p > p'$.

A level function denoted $\mathcal{F}(p)$ is a function which defines the increment policy, i.e., the number of necessary requests of priority higher than $p-1$ for upgrading pending requests of $p-1$ priority to $p$ priority. This function is monotone, increasing, and positive.

## IV. The awareness priority-based mutex algorithm

In this section, we present our new algorithm, denoted *Awareness*, which is an extension of the *Level-Distance* algorithm (see section II) and whose aim is to provide a good tradeoff between the number of priority inversions and response time. The *distance* mechanism has not been changed but the *level* mechanism has been replaced. In the previous *level* mechanism, a node takes into account only the requests received by itself while in our new proposal, a node considers the total number of pending requests in the system to increase the priority of the pending requests that it keeps in its own local queue. Furthermore, the number of requests required to upgrade a priority $p-1$ depends on the function level $\mathcal{F}(p)$.

The algorithm uses the token to propagate the knowledge about requests issued by all nodes of the system. In other words, a vector of $|P|$ entries (one entry per priority) is piggybacked in the token. Each entry of this vector is eventually equal to the total number of issued requests of the corresponding priority.

Figures 1 and 2 show the pseudo code of the *Awareness* algorithm.

```
1  Local variables :
2  begin
3      in_CS : boolean;
4      father : Site or ∅;
5      last_token : Vector of |P| integers ;
6      pending : Vector of |P| integers ;
7      local_queue : list of <site,priority,level,distance> ;
8  end
9  Initialization
10 begin
11     in_CS ← false;
12     last_token[i] ← 0 ∀i ∈ [1, |P|];
13     pending[i] ← 0 ∀i ∈ [1, |P|];
14     local_queue ← ∅;
15     father ← according to the topology;
16     if self = 0 then
17         father ← ∅;
18 end
19 UpdateLocalQueue(V : Vector of |P| integers)
20 begin
21     for p_i from 2 to |P| do
22         for n from 1 to V[p_i] do
23             foreach req in local_queue do
24                 if p_i > req.priority or (p_i = req.priority =
                   head(local_queue).priority) then
25                     req.level ← req.level + 1;
26                     if req.level = F(req.priority + 1) then
27                         req.priority ← req.priority + 1;
28                         req.level ← 0;
29     reorder (local_queue) ;
30 end

31 Request_CS(Priority p)
32 begin
33     if father ≠ ∅ then
34         addqueue(< self, p, 0, 0 >,local_queue) ;
35         if < self, p, 0, 0 >=head(local_queue) then
36             Send Request(p,1) to father;
37         else
38             pending[p] ← pending[p] + 1;
39         Wait(in_CS = true) ;
40     else
41         pending[p] ← pending[p] + 1;
42         in_CS ← true;
43     /* CRITICAL SECTION                                        */
44 end
45 Release_CS
46 begin
47     in_CS ← false;
48     UpdateLocalqueue(pending);
49     last_token ← last_token + pending ;
50     pending[i] ← 0 ∀i ∈ [1, |P|];
51     if local_queue ≠ ∅ then
52         request next ← dequeue(local_queue);
53         request head ← head(local_queue);
54         Send Token(min (|P|,head.priority), head.distance + 1,
           last_token) to next.site;
55         father ← next.site;
56 end
```

Figure 1. The awareness algorithm (initialization, update function, request and release handlers)

```
57 On_receive request from Sj(Pj, Dj)
58 begin
59     if father = ∅ and in_CS = false then
60         last_token[Pj] ← last_token[Pj] + 1;
61         Send Token(∅,∅,last_token) to Sj ;
62         father ← Sj ;
63     else if Sj ≠ father then
64         request before ← head(local_queue);
65         if ∃e ∈ local_queue such that e.site = Sj then
66             if e.priority ≤ Pj then
67                 e.priority ← Pj ;
68                 e.distance ← Dj ;
69                 reorder(local_queue) ;
70         else
71             addqueue(< Sj, Pj, 0, Dj >,local_queue) ;
72         request after ← head(local_queue);
73         if after ≠ before then
74             Send Request(min (|P|,after.priority), after.distance + 1) to
               father;
75         else
76             pending[Pj] ← pending[Pj] + 1;
77     else
78         pending[Pj] ← pending[Pj] + 1;
79 end

80 On_receive Token from Sj(Pj, Dj, Vtok)
81 begin
82     father ← ∅ ;
83     request next ← dequeue(local_queue);
84     Vtok ← Vtok + pending;
85     UpdateLocalqueue(Vtok − last_token);
86     last_token ← Vtok;
87     pending[i] ← 0 ∀i ∈ [1, |P|];
88     if Pj ≠ −1 then
89         addqueue(< Sj, Pj, 0, Dj >,local_queue) ;
90     if next.site = self then
91         /* process can enter in CS                             */
92         in_CS ← true;
93     else
94         request head ← head(local_queue);
95         Send Token(min (|P|,head.priority), head.distance + 1,
           last_token) to next.site;
96         father ← next.site;
97 end
```

Figure 2. The awareness algorithm(message handlers)

### A. Local variables and messages

For each site $S_i$, the algorithm defines the following local variables:

- $in_CS$: true if $S_i$ is in the CS; false, otherwise;
- $father$: the identifier of $S_i$'s neighbor on the path leading to the process that holds the token (root);
- $local_queue$: local queue of pending requests received by $S_i$, sorted by decreasing order of priority, increasing order of distance in case of equal priorities, and then FIFO order in case of equal distances. Each entry of the queue has the format $< site, prioirity, level, distance >$ where $site$=the neighbor of $S_i$ which issued or forwarded the request; $priority$=the current priority of the request; $level$=the current number of pending requests that has already been counted up in order to increase $req$'s priority to $priority + 1$ ; $distance$=the distance from the node that issued the request and the current node (distance mechanism).
- $pending$ (vector of $|P|$ integers): used to count up the global number of pending requests known by $S_i$ since the last time it either releases the CS or received the token. $pending[j]$ corresponds to the number of requests with priority $j$ that have not been taken into account yet for incrementing priorities. A request is registered exactly once in a $pending$ vector of just one site.
- $last_token$ (vector of $|P|$ integers): stores the last vector of requests kept by the token when $S_i$ received it.

A *request* and *token* messages respectively keep the following information:

- request ($< p, d >$): $p$= the current priority of the request; $d$=the distance from the node that issued the request and the current node.
- token ($< p, d, v >$): $p$=the current priority of the request which is piggybacked in the token message, if it is the case; otherwise this field is equal to -1; $d$=the distance value of the piggybacked request; $v$=the vector with the number of global issued requests per priority.

The following functions handle the $local_queue$ variable:

- addqueue (): includes a request ($< s, p, l, d >$) in the local queue, according to the ordering policy described above.
- reorder($local_queue$): sorts the local queue, according to the same ordering policy.
- dequeue($local_queue$): considering that the local queue is not empty, this function returns the first request of the local queue and removes it.
- head($local_queue$): returns the first request of the local queue. The message is kept in the local queue. If the latter is empty, each field ($< s, p, l, d >$) of the returned element is equal to $-1$.

### B. Description of the algorithm

When calling the *Request_CS* function (line 31), if $S_i$ does not have the token, it includes the request into its local queue (line 34). Furthermore, if the request is the head of the queue, $S_i$ sends it to its father (line 36). On the other hand, if $S_i$ does not forward the request (lines 38 and 41), it registers the request by incrementing the entry corresponding to the request's priority of its $pending$ vector. Having the token, $S_i$ enters the CS.

Node $S_i$ releases the CS by calling the function *Release_CS* (line 45). The priority of the requests of $S_i$'s queue is updated upon calling the $UpdateLocalQueue$ function (described in the following). Then, $S_i$ updates the token vector by adding the number of pending requests for each priority( line 49) and resets the pending request vector (line 50). If the local queue is not empty, $S_i$ sends the token to $S_j$, the node at the head of its local queue (line 54), removing the corresponding request from the queue (line 52). If $S_i$'s local queue is not empty, it also piggybacks in the token message the request of the head of its local queue, but keeps the latter in its queue. It then changes its $father$ variable to $S_j$ (line 55).

Upon reception of a request from $S_j$ (Figure 2, line 57), if $S_i$ keeps the token without using, i.e., it is the root node but it is not in CS, $S_i$ updates the token by registering the request in $last_token$. Then, it sends back the token (line 61) to $S_j$ and sets its father to $S_j$. On the other hand, if $S_i$ is not the root, it adds the new request in its local queue (line 71) or updates the corresponding request if $S_i$ has already received a request from $S_j$ (lines 65 – 69), reordering the queue, if necessary. Then, $S_i$ forwards the request to its own father (line 74) if the head of $p$'s local queue has changed; otherwise it registers the request (line 76).

Similarly to the $Release_CS$ function, whenever $S_i$ receives the token (line 80), it adds the $pending$ vector to the received token vector (line 84). It updates its local queue by calling the function $UpdateLocalQueue$ with the new requests it received since the last time it obtained the token which is the input parameter of the function (line 85). Then, variable $last_token$ is updated and $pending$ is reset. If the token piggybacks a request, $S_i$ includes the received request in its local queue (line 89). If its own request is the head of the queue (i.e., it has the highest priority), the node enters the CS (line 92). Otherwise, the token is forwarded to the node at the head of the local queue (line 95). The token message also piggybacks the head request of $S_i$'s local queue, if this one is not empty. Finally, the $father$ variable is updated (line 96).

*Local queue updating (UpdateLocalQueue):* As already mentioned, a site updates the priorities of requests stored in its local queue by calling the $UpdateLocalQueue$ function (line 19) whenever either it receives the token or releases the CS.

For each priority $p_i$ of the vector $V$, the function increments by one the level field of each request $req$ of the local queue whose priority is smaller than $p_i$ (line 24). Whenever the level field of $req$ is equal to the number of pending requests necessary to increase by one the priority of $req$, the latter is incremented and $req$'s level field is reset. Notice that the first condition of the test of line 24 prevents the priority of $req$ to be greater than $p_i$. The second condition of the test allows to solve possible starvation problems induced by the *distance*

mechanism, which aims at reducing the number of messages sent by the algorithm by taking into account request locality: if two requests of a local queue have the same priority, the token will be sent to the closest node in terms of number of hops. However, it might happen that the token infinitely travels over a portion of the tree where there are some nodes, which continuously request the CS with the same priority. Such a behavior can eventually lead to starvation problem whenever a node, which issues requests with this same priority, is located far from this portion of the tree. To overcome this problem, a priority of a request can increase beyond the maximum priority value and, therefore, eventually preempts those sites with the same priority and a smaller distance value.

### C. Discussion

The minimization of both the number of inversions and the response time requests with low priority are two conflicting objectives: on one hand, if there is no priority inversion, the response time can be infinite, which is the case of static priority algorithms; on the other hand, a low response time for low priority requests implies a great number of inversions. In fact, for a given value of load (i.e., number of pending requests), the tradeoff between response time and number of inversions depends on three factors, discussed bellow in ascending dependency order:

(1) *The global knowledge of pending requests*: This knowledge allows the nodes to learn about the total number of pending requests of the system and their respective priority. Therefore, the greater the number of sites which are aware of such an information, the higher their flexibility to balance both objectives.

(2) *Dynamic priority strategies.* The level function defines the increment policy for increasing priority $p-1$ to $p$: the higher the increasing behavior of the function, the smaller the number of priority inversions but the faster the maximum response time increases.

(3) *Mapping of nodes over the logical tree.* If we consider that a process issues requests always with the same priority, the position of the node in the tree has an impact: whenever a great number of nodes in the low levels of the tree issue high priority requests, the number of priority inversions naturally decreases, because these requests are favored by their position in the tree.

## V. Performance Evaluation

In this section, we present some evaluation results comparing *Awareness* with Kanrar-Chaki, Chang, and *Level-Distance* algorithms.

### A. Experimental testbed and configuration

The experiments were conducted on a 64-nodes cluster with one process per node. There is no network contention since there is 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 (64 in our case).
- $\alpha$: time to execute the critical section (CS). (equal to 2.5 ms)
- $\beta$: mean time interval between the release of the CS by a node and its request by this same node.
- $\gamma$: network transmission delay between two neighbor nodes. (equal to 2.5 ms)
- $\rho$: 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 our experiment, we have considered:
  - **High load** ($\rho = 0,1N$): a scenario where the majority of application processes request the critical section;
  - **Intermediate load** ($\rho = 0,5N$): a scenario where some sites compete to get the CS;

We start collecting data for evaluation only after a warm-up phase of the experiment where the rate of requests is stationary.

All the algorithms are based on a static logical tree topology and, therefore, nodes' position has an impact in performance. Consequently, we define a priority mapping policy for the initial tree topology: the deeper the initial position of the node in the tree, the lower its priority. Hence, nodes at tree level 0 (initial root node) and 1 have the highest priorities. The set of nodes in the same initial tree level have the same priority. Therefore, a node has a strictly lower priority than its initial father (except for nodes at tree level 1) but has a strictly higher priority than its initial children (except for the initial root node). This configuration is considered ideal since it presents the following two advantages:

- the token will be more frequently in the lowest levels of the tree where processes that issue requests with the highest priorities are located. These processes will therefore benefit from both the route of the token and the fact that new requests are included in the token, reducing the number of sent messages.
- the number of priority inversions can be intrinsically reduced: when the token moves away from the lowest levels of the tree in order to satisfy low priority requests, it is likely to satisfy, during its travel, intermediate priority requests in descending order. It will thus respect the order of priorities.

Based on such a mapping policy, the number of priorities is equal to the total height of the tree. Since in the experiments, we have considered 64 nodes organized in a binary tree, the number of priorities is equals to 6 ($Log_2(64)$). Thereafter, we denote priorities 0 and 1 *"low priorities"*, priorities 2 and 3 *"intermediate priorities"* and finally priorities 4 and 5 *"high priorities"*.

The Figures that follow show evaluation results comparing our new algorithm *Awareness* with *Level-Distance*, Kanrar-Chaki, and Chang algorithms (see Section II). Kanrar-Chaki and Chang algorithms are denoted *no-level* algorithms while the *Level-Distance* and the *Awareness* algorithms are denoted *level* algorithms.

### B. Experiments with a given level function

For the current experiments, we have considered the same level function used in [7] which was $\mathcal{F}(p) = 2^{p+c}$ with $c = 6$.

We are going to evaluate the following metrics:

- *Response time*: the delay between the moment a node requests the CS and the moment it gets access to it.
- *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.
- *Average percentage of priority inversions*: At each CS access, the percentage of pending requests that were penalized (i.e., they have higher priority than the one that obtained the CS) is measured. This metric expresses than the average of such percentages.
- *CS execution rate*: ratio of the sum of all requested critical section execution durations over the time of the experiment. It expresses the percentage of time corresponding to critical section executions.

In figure 3, we show for the two considered values of load, results concerning the above metrics.

*Response time:* In Figures 3(a) and 3(b), we observe that *no-level* algorithms have a regular behavior (shape of stairs), i.e., the higher the priority, the lower the average response time, except for Kanrar-Chaki algorithm where we can observe that high priorities have the same average response time. On the other hand, in *level* algorithms, response time differences between priorities are not so regular: response time of priority 0 of *Awareness* algorithm is higher when compared to *no-level* algorithms (a "best-effort" approach), while high priorities have presented a strong improvement related to CS access time. Generally, we observe that the average response time of *level* algorithms respect priority levels. This is all the more confirmed by the fact standard deviations do not overlap.

However, in the *Level-Distance* algorithm, some low priority levels have no response time (priorities 0, 1, and 2 for $0.1N$ and priority 0 for $0.5N$). Such results correspond to a huge response time for these priority levels since no request has been satisfied during the experiment. The *Level-Distance* algorithm penalizes too much low priorities. 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.

For all algorithms, we note that the response time of low and intermediate priorities is overall higher in high load than in intermediate load. In fact, in intermediate load, low and intermediate priorities have a better chance to be satisfied faster since the frequency of high priority requests is lower than in high load.

Comparing the *Level-Distance* 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.

*Number of messages per request:* Figures 3(c) and 3(d) show that the number of messages (request messages and token messages) per request is greater in intermediate load than in the high load for all algorithms. Let's firstly analyze requests messages. In our priority mapping policy described above, a request message reception on site $S$ concerns a request which probably has a lower priority than $S$'s priority (message from $S'$'s child). Consequently, in case of high load, requests are less likely to be forwarded to the upper area of the tree since the algorithms only forward a request that is in the head of the queue, i.e., the one with the highest priority. Thus, the number of request messages is reduced in high load. Let's now analyse token messages. The transfer of the token from a given area $A$ of the tree to an area of lower priority is due to two conditions:

(1) there is no more pending request in area $A$;
(2) it exists at least one lower priority request which has been incremented and thus equals to the priority of area $A$.

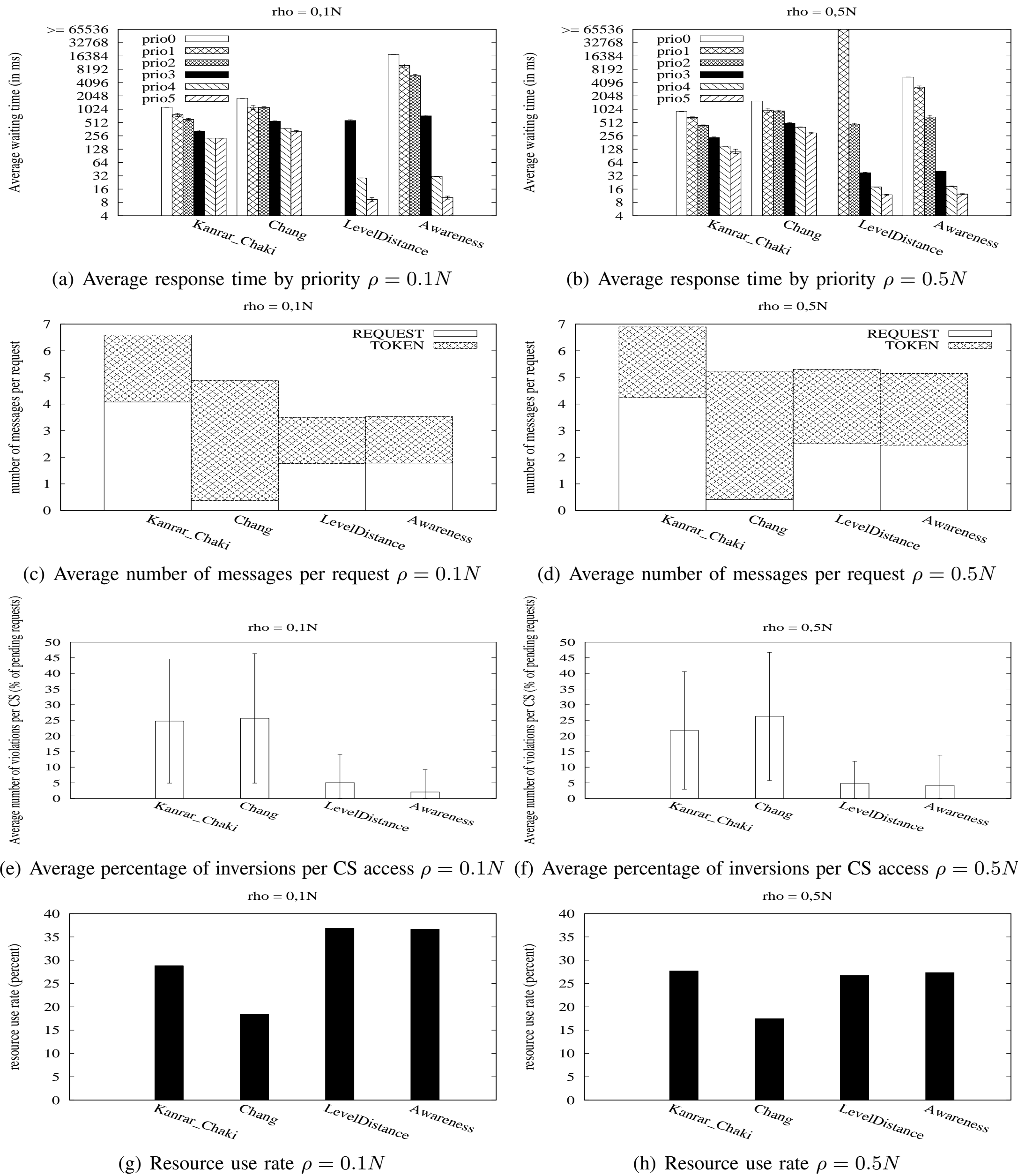
In case of high load, the token has less chance to leave a high priority area moving to an area of lower priority due to the second condition. The token travels mainly in the low levels of the tree and leaves this area only in case of a priority increment. In case of intermediate load, the first condition happens more frequently which implies that the token can leave the low levels area and go away from it more easily. Such a behavior of the token induces more token messages in the system.

Hence, if we consider both kinds of messages, in case of high load, *level* algorithms present a smaller number of messages than *no-level* algorithms. Compared to the Kanrar-Chaki algorithm, *level* algorithms reduce the number of request messages thanks to the piggybacking mechanism (see Section II-B). On the other hand, *level* algorithms present a small number of messages when compared to Chang algorithm because of:

- exploitation of request locality with the distance mechanism (see section II-B);
- a great respect of priorities. For instance, in Chang algorithm, sites get the maximum priority fast which induces more token transfers.

Comparing both Figures, in case of intermediate load, *level* algorithms present a message increase of around 50% while *no-level* algorithms have a message increase of around only 5%. We have previously pointed out that a decrease in load induces an increase of the number of messages. We can then emphasize that this decrease has a higher impact in *level* algorithms than in *no-level* ones.

*Percentage of priority inversions:* Figures 3(e) and 3(f) show that for any load value, there is a great difference between *level* and *no-level* algorithms in regard to the number of priority inversions due to the increasing priority mechanisms of the former (around 25 % for the *no-level* algorithms against less than 5% for the *level* algorithms). We also observe that the *Awareness* algorithm presents a smaller percentage of priority inversions when compared to *Level-Distance* which confirms that upgrading priority based in a global knowledge of issued requests is more effective than just in a local view of pending requests.




(a) Average response time by priority $\rho = 0.1N$

(b) Average response time by priority $\rho = 0.5N$

(c) Average number of messages per request $\rho = 0.1N$

(d) Average number of messages per request $\rho = 0.5N$

(e) Average percentage of inversions per CS access $\rho = 0.1N$

(f) Average percentage of inversions per CS access $\rho = 0.5N$

(g) Resource use rate $\rho = 0.1N$

(h) Resource use rate $\rho = 0.5N$

Figure 3. Study of metrics with two different loads

When the load decreases, we observe that Chang algorithm does not change its behavior whereas in Kanrar-Chaki algorithm the percentage of priority inversions decreases. Such a reduction can be explained because the overall number of requests decreases as well. Concerning the *Level-Distance* algorithm, the average percentage of inversion does not change excepting the standard deviation which is greater for high load. Finally, the percentage increases for the *Awareness* algorithm when the load decreases since sites receive less information about the system pending requests due to both the smaller number of messages that travel in the system and the non negligible network latency when compared to the critical section duration.

*Resource use rate:* Since network latency equals to the CS execution duration ($\alpha = \gamma$) in the experiments, the threshold of the resource use rate is 50 %. Thus, whatever the algorithm, in the best case, (the granting of the token necessarily implies that the receiver will enter in CS), it spends the same amount of time in communication as in critical section.

In general, *no-level* algorithms are less effective in obtaining the CS in high load, particularly Chang's algorithm. Such a behavior is a direct consequence of its high number of inversions: whenever processes reach fast the maximum priority, the token is more likely to travel longer distance between two critical sections, which degrades the resource use rate when

the network latency is not negligible comparing to the CS execution time.

In conclusion, the *level* and *awareness* mechanisms are essential for reducing the number of priority inversions. These mechanisms do not induce message overhead compared to Chang. Moreover, the latter is less effective in terms of resource use rate. We can also state that the *Awareness* algorithm significantly reduces the response time of lower priorities while keeping a low number of inversions. Moreover, it presents no performance degradation in terms of number of messages nor resource use rate when compared to the *Level-Distance* algorithm.

### C. Impact of the level function on priority violations and maximum response time

In this section we evaluate the impact of the level functions on the performance of the *level* algorithms (*Level-Distance* and *Awareness*). To this end, we have defined five function families:

- *constant*: $\mathcal{F}_c(p) = c$
- *linear*: $\mathcal{F}_c(p) = p * c$
- *polynomial*: $\mathcal{F}_c(p) = p^c$
- *exponential*: $\mathcal{F}_c(p) = c^p$
- *power of two*: $\mathcal{F}_c(p) = 2^{p+c}$ (previous experiments)

For a given load and constant $c$, we have measured the number of priority inversions and maximum response time. By varying the constant $c$, we can study the behavior of the *level* algorithms for different functions. Figure 4 summarizes such a study for high and intermediate load scenarios. Each sub-figure shows the impact of a level function for a given load on the algorithms. A coordinate of a point corresponds to the two following metrics: x-axis is the ratio of priority inversions which is equal to the total number of inversions divided by total number of requests and the y-axis is the maximum response time. Thus, for a given ratio of priority inversions, we can compare the maximum response time between the different algorithms. Since the minimization of x and y values is conflicting, the objective of the function is to find a point that is the closest to point 0.

We observe for both *level* algorithms different global inversion ratios. However, for a given value of constant $c$, the range of points for both curves are not the same, which entails some comparison difficulty. We have also included in the sub-figures the *no-level* algorithms, Kanrar-Chaki and Chang, for comparison sake. Both are represented by one point since they do not use level functions. Consequently, these points are the same for every sub-figure of a given load.

Overall, we can observe that for any load and any level function, the *Awareness* algorithm has a shorter maximum response time than the *Level-Distance* algorithm (at least 2 times shorter). Interestingly that for a given load, all curves in the sub-figures of the *Awareness* algorithm are quite similar, regardless of the level function. In fact, its knowledge mechanism reduces the impact of the choice of the level function family.

Finally, for a given level function, the gradient is sloping in case of intermediate load. We can thus point out the impact of the load on the level function: the higher the load, the greater the number of high priority requests and, thus, the faster the level increment of request. Consequently, there is a reduction in the maximum response time.

## VI. Conclusion

We have presented in this paper a new distributed priority-based mutual exclusion algorithm which considerably reduces the maximum waiting time of low priority requests without increasing the number of priority inversions. Such a behavior is due to the "awareness" approach of the algorithm which gives more flexibility for priority increments. Furthermore, level functions allow to better configure the threshold between the maximum waiting time and the number of priority inversions. We have also observed that, thanks to the distance mechanism, the algorithm message complexity does not degrade.

## VII. 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] Y.-I. Chang. Design of mutual exclusion algorithms for real-time distributed systems. *J. Inf. Sci. Eng.*, 11(4):527–548, 1994.

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

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

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

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

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

[7] J. Lejeune, L. Arantes, J. Sopena, and P. 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] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3:145–159, May 1985.

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

[10] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. *Real-Time Systems Symposium, IEEE International*, 0:340, 1999.

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

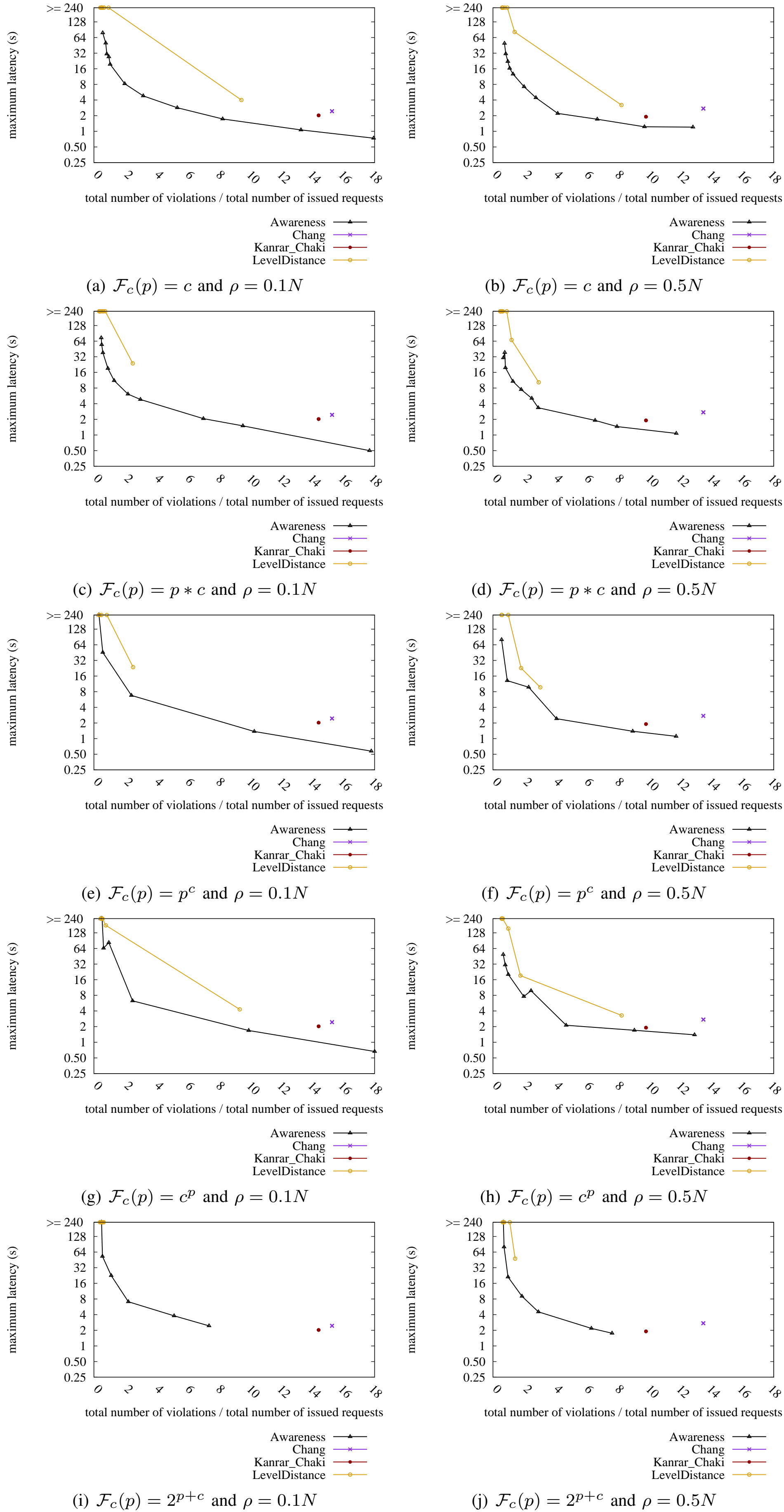
[12] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.

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

[14] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens. Towards QoS-Oriented SLA Guarantees for Online Cloud Services. In *13th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID'13)*. IEEE Computer Society Press, May 2013.

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

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




(a) $\mathcal{F}_c(p) = c$ and $\rho = 0.1N$

(b) $\mathcal{F}_c(p) = c$ and $\rho = 0.5N$

(c) $\mathcal{F}_c(p) = p * c$ and $\rho = 0.1N$

(d) $\mathcal{F}_c(p) = p * c$ and $\rho = 0.5N$

(e) $\mathcal{F}_c(p) = p^c$ and $\rho = 0.1N$

(f) $\mathcal{F}_c(p) = p^c$ and $\rho = 0.5N$

(g) $\mathcal{F}_c(p) = c^p$ and $\rho = 0.1N$

(h) $\mathcal{F}_c(p) = c^p$ and $\rho = 0.5N$

(i) $\mathcal{F}_c(p) = 2^{p+c}$ and $\rho = 0.1N$

(j) $\mathcal{F}_c(p) = 2^{p+c}$ and $\rho = 0.5N$

Figure 4. Study of five level fonctions with two different loads