Service Level Agreement for Distributed Mutual Exclusion in Cloud Computing

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

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

***Abstract*—In Cloud Computing, Service Level Agreement (SLA) is a contract that defines a level and a type of QoS between a cloud provider and a client. Since applications in a Cloud share resources, we propose two tree-based distributed mutual exclusion algorithms that support the SLA concept. The first one is a modified version of the priority-based Kanrar-Chaki algorithm [1] while the second one is a novel algorithm, based on Raymond algorithm [2], where a deadline is associated with every request. In both cases, our aim is to improve Critical Section execution rate and to reduce the number of SLA violations, which, for the first algorithm represents the number of priority inversions (i.e. a higher priority request is satisfied after a lower one) and for the second one, the number of requests whose deadline is not respected. Performance evaluation results show that our solutions significantly reduce SLA violations avoiding message overhead.**

a) Keywords: **Distributed mutual exclusion, SLA, priority-based algorithm, EDF, Cloud.**

I. Introduction

Cloud computing is a model aimed at providing ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. Cloud services are accessible over the Internet and hosted on a set of virtual machines running over a grid of physical machines in a datacenter. Virtual machines can be created, removed or migrated depending on the workload and availability of physical machines (cloud elasticity). In this context, Service Level Agreement (SLA) is a contract that defines a level and a type of QoS between a cloud provider (e.g., Amazon, Google) and a cloud client at any cloud level (IaaS, PaaS and SaaS) [3]. According to [4], cloud characteristics (elasticity and SLA) imply new obstacles for computer science and modify classical hypotheses for distributed algorithmic.

Since applications in a cloud share resources, concurrent accesses to them might be critical and should be controlled in order to avoid inconsistencies. In other words, accesses to such resources are considered critical sections and, therefore, a mutual exclusion service for controlling their access must be provided by the Cloud.

A mutual exclusion algorithm ensures that at most one process can execute the critical section (CS) at any given time. They can be divided into two families [5]: permission-based (e.g. Lamport [6], Ricart-Agrawala [7], Maekawa [8]) and token-based (Suzuki-Kazami [9], Raymond [2], Naimi-Trehel [10]). The algorithms of the first family are based on the principle that a node only enters a critical section after having received permission from all the other nodes (or a majority of them [7]). In the second group of algorithms, a system-wide unique token is shared among all nodes, and its possession gives a node the exclusive right to execute a critical section. Token-based algorithms present different solutions for the transmission and control of critical section requests of processes. 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. With regard to the number of nodes, token-based mutual exclusion algorithms present an average message traffic which is lower than that of permission-based ones. Thus, they are more suitable for controlling concurrent accesses to shared resources in Clouds whose number of nodes is often very large.

Nevertheless, current mutex algorithms are not suitable for Cloud applications because they do not take into account cloud characteristics such as SLA constraints. For instance, Google Clouds uses an optimized version of the Chubby lock algorithm [11] although it does not cope with the above mentioned dynamics of SLA.

Thus we propose in this article two SLA-oriented mutex algorithms to provide a mutex “service” for Clouds: (1) a priority’s request-based algorithm and (2) a request response time one. The first algorithm should be applied whenever QoS requirements can be associated to priority, i.e., different SLA-levels can be mapped to different priorities; in the second algorithm, Qos requirements are expressed in terms of request response time. Therefore, the goal for both algorithms is to minimize SLA violations. However, the meaning of “SLA violation” depends on the algorithm: avoidance of priority inversion for (1) and reduction of the number of requests which were not satisfied before a given response time (deadline) for (2). The two algorithms are based on the token-based approach since the latter provides scalability. Algorithm (1) is an extension of Kanrar-Chaki [1] algorithm (cf. Section II) while (2) is a novel algorithm that we have conceived.

The rest of the paper is organized as follows. Section II discusses some existing priority-based mutual exclusion distributed algorithms and gives a brief description of the Kanrar-Chaki algorithm. Our priority request-SLA and request response time-SLA distributed mutual exclusion solutions are presented in section III. Performance evaluation results of both SLA-

based mutual exclusion approaches are presented in Section IV. Finally, Section V concludes the paper.

II. Related Work

In distributed systems with time constraints such as request time deadline, critical section requests are usually ordered on the basis of their priority rather than the time when a CS request occurs. Several priority-based algorithms have been proposed to cope with real-time requirements. In this section we outline the main priority-based mutual exclusion algorithms and mention some works which explicitly address the problem of deadline constraints. Furthermore, as our SLA mutual exclusion algorithms are based on the Kanrar-Chaki [1] algorithm, the latter is described in more details.

Priority-based distributed mutex algorithms are usually an extension of some non-prioritized algorithms.

The Goscinksi algorithm [12] is based on the non-structured token-based Suzuki-Kasami algorithm and has a 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 and thus is not held by any node.

The Mueller algorithm [13] is inspired to the Naimi-Trehel 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 within each priority level. Its implementation is quite complex and the dynamic tree tends to become a queue because, unlike the Naimi-Trehel 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($\frac{N}{2}$).

The Housni-Trehel algorithm [14] 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 [2] and routers apply the Ricart-Agrawala algorithm [7]. Starvation is possible for lower priority processes if many higher priority requests are pending. Moreover a process can only send one request priority (that of his group).

Several algorithms propose to extend Raymond's algorithm in order to add a priority to requests.

Raymond's algorithm [2] 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, 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.

The Kanrar-Chaki algorithm [1] is based on Raymond's algorithm. It introduces a priority level for every process's token 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 first element of the processes's local queue. If such is the case, such a strategy is applied by all the nodes of the directed path till the root site. The algorithm then behaves like Raymond's, as described above. In order to avoid starvation, the priority level of pending requests of a process's local queue can be 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 [15] 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, 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 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 (in our case, number of SLA violations) when compared to the Kanrar-Chaki algorithm; performance evaluation discussion of both algorithms is presented in section IV. 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 a token message

In [16], Johnson and Newman-Wolfe present three algorithms for prioritized distributed locks. 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.

Some algorithms explicitly address real-time constraints. In [17], Han proposes a real-time fault-tolerant mutual exclusion

algorithm that takes into account deadlines of requests. It is a permission based algorithm where a majority of nodes agree on the same schedule of critical section accesses. A gossip protocol is used to broadcast requests. Each node maintains a queue that stores "feasible" requests ordered by their deadlines. In order to avoid priority inversions in real-time systems, synchronization algorithms use the Priority Ceiling Protocol (PCP), initially conceived by Sha and Rajkumar [18]. PCP prevents deadlocks and bounds blocking time. In [19] and [20], the authors propose some extensions of PCP to distributed systems for multiprocessors and CORBA respectively.

III. SLA-BASED MUTUAL EXCLUSION

When the SLA is based on request priority, we define a SLA violation as a priority inversion (i.e., a request that has been satisfied after a lower priority request). When the SLA is based on request response time, we define a SLA violation as a request which has been satisfied after its required deadline.

Since we consider that there is one process per node (virtual machine), the words node, process, and site are interchangeable.

A. Request Priority SLA-based mutex

Our solution is based on the Kanrar-Chaki algorithm because it is scalable with regard to the number of messages (complexity O(Log N)) and starvation does not exist thanks to the mechanism of priority increment. Our proposal is therefore to modify the Kanrar-Chaki algorithm to minimize the number of SLA 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 or the request response time.

To this end, we firstly applied Chang [15]'s message traffic optimization (see section II) to the Kanrar-Chaki algorithm and then two incremental heuristics: the *"level"* heuristic which postpones the priority increment of pending requests and the *"level-distance"* that uses in addition to *"Level"* heuristic the number of intermediate nodes from the current token holder to requesting nodes in order to decide which node will be the next token holder. The traffic message optimization and the two heuristics are described hereafter.

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 the latter a request to signify that the token must be returned later on. Hence, in order to lower communication traffic, this request can be piggybacked in the token message.

2) "Level" Heuristic: We have observed in the Kanrar-Chaki algorithm that requests, whose priority was originally low, were satisfied quite fast since their priority reached the maximum value due to the priority increment approach of the algorithm. Such a behavior characterizes in fact an inversion of priorities. Therefore, we have modified the algorithm in order to postpone the priority increment: the priority value of a pending request is not incremented at every insertion of a request with higher priority but only after X request insertions with such a priority. The X value 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 has a sufficiently high value so as to avoid that the original priority $p-1$ of a request becomes p before the original priority p of a second request becomes $p+1$.

3) "Level-Distance" Heuristic: In the Kanrar-Chaki algorithm, requests with the same priority are not ordered. We introduce a new parameter, denoted *request distance*, to take into account request locality when ordering such requests. The request distance from site R to site S is 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 smallest request distance with respect to the current token holder. It is worth pointing out that the tree topology has an impact in this heuristic. Since this heuristic is orthogonal with the previous "Level" heuristic, we have combined them in the "Level-Distance" heuristic.

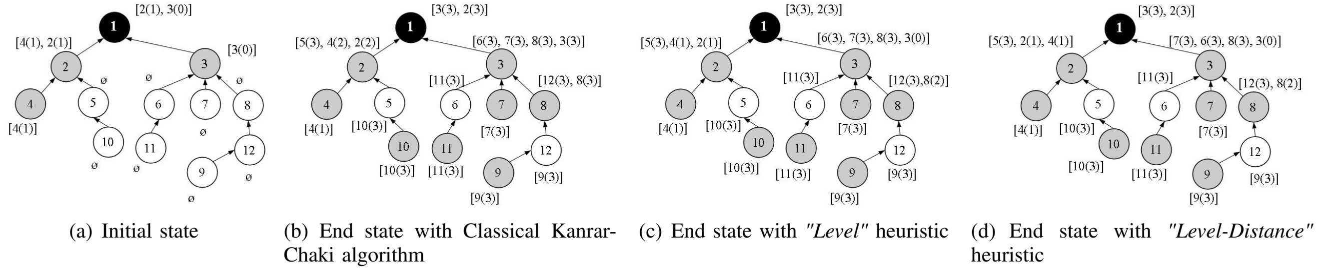
Figure 1 illustrates the impact of the two different heuristics with respect to the original Kanrar-Chaki algorithm. We consider a tree with 12 nodes. Pending requests, stored in local queues Q_i of each node, are sorted by decreasing order of priority. Each of them is separated by a coma and noted $x(y)$, 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. Such an initial state is shown in Figure 1(a).

Let's now consider that nodes n_{11}, n_{10}, n_7, n_8 and n_9 issue one request each with the following respective priorities:
(1) n_{11} sends a request with priority 3 denoted 11(3)
(2) n_{10} sends a request with priority 3 denoted 10(3)
(3) n_7 sends a request with priority 3 denoted 7(3)
(4) n_8 sends a request with priority 2 denoted 8(2)
(5) n_9 sends a request with priority 3 denoted 9(3).

Figures 1(b), 1(c), and 1(d) show the state of the tree after the five new requests have been taken into account by the Kanrar-Chaki algorithm, *"Level"* heuristic, and *"Level-Distance"* heuristic respectively. Notice that, in the three algorithms, all fathers of the requesting nodes have added the received requests in their respective local queues: n_6 has included 11(3) in Q_6, n_{12} has included 9(3) in Q_{12}, n_5 has included 10(3) in Q_5, and n_3 has included 7(3) and 8(2) in Q_3. Furthermore, in the case of the *"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 consequences on the state of the pending requests and local queues of the algorithms:

- original Kanrar-Chaki (Figure 1(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. Request 6(3) is included in Q_3;
  (2) The value 2 is assigned to the priority of n_2's and n_4's requests of Q_2. Priority of n_2's request in Q_1 becomes 3;
  (3) Request of n_3 in Q_3 is increased to 2;
  (4) No consequence over priorities.



(a) Initial state

(b) End state with Classical Kanrar-Chaki algorithm

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

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

Figure 1. Example of execution by heuristics

(5) Request of n_8 in Q_8 is increased to 3. The value 3 is assigned to the priority of n_8 and n_3 in Q_3

- *"Level"* heuristic (Figure 1(c)):
  (1) The priority level of the n_3's pending request in Q_1 becomes 3. Request 6(3) is included in Q_3;
  (2) The priority of n_2's request becomes 3;
  (3), (4), and (5) No consequence over priorities.
- *"Level-Distance"* heuristic (Figure 1(d)):
  (1) (respectively, (2) and (3)) The same consequences of *"Level"* heuristic but requests in Q_2 (respectively, Q_1 and Q_3) are rescheduled according to requester's distance. Request 6(3) is included in Q_3;
  (4) and (5) No consequence over priorities.

If we consider that no other request is issued until every pending request is satisfied, the order of node request satisfactions are the following:

- Kanrar-Chaki algorithm: n_{11} - n_7 - n_9 - n_8 - n_3 -n_{10} - n_4 - n_2
- *"Level"* heuristic: n_{11} - n_7 - n_9 - n_{10} - n_8 -n_4 - n_2 - n_3
- *"Level-Distance"* heuristic: n_7 - n_{11} - n_9 - n_{10} - n_8 -n_2 - n_4 - n_3

This execution example clearly shows that the different heuristics change the order in which requests are satisfied, particularly for node n_3: its request is the fifth one to be satisfied when the original algorithm is applied and the last one in the case of the *"Level"* heuristic and *"Level-Distance"* heuristic. We can also observe that both heuristics keep the original priority order. Furthermore, there are two SLA violations (n_8 and n_3) in the Kanrar-Chaki algorithm but none when either of the heuristics is applied.

B. Response time SLA-based mutex

We now consider that requests are satisfied according to the response time deadline associated to each request and propose a new algorithm. To this end, when a process issues a request, it informs two values: the maximum delay for the response and the duration of the critical section which it needs to execute. However, similarly to Cloud services whose QoS is defined by a SLA, before accepting the request, the mutual exclusion "service" must ensure that, taking into account the system's state, the request constraints can be satisfied. In the case of mutual exclusion, such constraints refer to the satisfaction of the request before its deadline. Therefore, a request must be submitted to an admission control before being accepted. If the admission control considers that the acceptance is not possible, the request is rejected and the process should issue, a less restrictive new request for instance. Otherwise, the request is accepted by the system and will be satisfied, with great probability, before its deadline. If the deadline of an accepted request is not respected (SLA violation), the request will fail and, therefore, the process will not have the right to execute the critical section. The aim of our proposed algorithm is thus to minimize SLA violations, i.e., deadline violations, and maximize critical section execution throughput.

Our algorithm is based on Raymond's algorithm: nodes are organized in a logical static tree whose links always form a directed path to the root. Requests are sorted at a process's local queue by their response time deadline, similarly to the real-time scheduling policy Earliest Deadline First (EDF). Notice that we consider that all nodes clocks are synchronized. Therefore, the deadline of the request in the head of the queue will be the first to expire.

1) Admission control: The feasibility of a request satisfaction should be checked before including the request in the system. Based on the requests already presented in the process's local queue, the admission control should firstly verify if a process request can be locally satisfied. If such is the case, a global admission policy is performed.

Local validation decision policy: As previously explained, requests of a process's local queue Q are sorted by their response time deadline. Upon reception of a new request R, the process computes the potential position P of R in its queue. It then evaluates if the satisfaction of R is feasible or not. R is feasible if: (1) requests before P in Q will respect R's constraints after its insertion; (2) R will respect the constraints of the next requests after position P in Q which have already been validated by the site.

In order to respect these two conditions, it is necessary to consider the scenario where all requests are satisfied at their deadline. Therefore, in order to ensure (1), the deadline of the request before P (denoted $P-1$) plus its CS execution duration plus the latency to send the token from $P-1$'s requesting node to R's should not violate R's deadline. Analogously, in order to ensure (2), the deadline of R plus its CS execution duration plus the latency to send the token between R's requesting node and $P+1$'s (i.e. the requesting node just after P) in Q should not violate $P+1$'s deadline.

Global validation decision policy: If a request is locally

satisfied according to the local decision policy then the process sends a request message to its father (same principle as Raymond's algorithm) which includes the maximum deadline for the response and the duration of the CS. Similarly, the father also submits the request to the admission control. Such a mechanism is recursively applied till the root node. Consequently, the root node will be aware of all the pending requests in the system.

If the request is rejected by a node N (a node in the path between the requesting node and the root, both of them included), it sends a reject message to its child that belongs to the path towards the requesting node. Such a message is forwarded until the requesting process which will finally discard it. On the other hand, in the case of a request acceptance, we propose two approaches for notification:

- **Acknowledgement approach**: The requesting process should wait for an acknowledgement message from its father which then confirms that its request has been accepted by the system. Hence, when the request is locally accepted, it is not immediately included in the process's local queue but in a temporary one. A request is added to the definitive local queue of the requesting node only after the reception of an acknowledgement message from its father. Since the root node knows all the requests in the system, it is the only one that can initiate the shipment of an acknowledgement message which is then forwarded to the requesting process. An accepted request will be removed from the process's local queue after the execution of the critical section or after a deadline violation detection.

- **Token approach**: In this case, the requesting processes and intermediate nodes directly add the request in their respective local queues. Consequently, they do not wait for any acknowledgement message. The request is removed from the process's local queue after the critical section execution, upon reception of a reject message, or a deadline violation detection.

2) Token scheduling: Basically, our algorithm follows the same principle of the Kanrar-Chaki algorithm for the token scheduling: upon reception of the token, a node N has the right to execute the critical section (CS) if its own request is in the head of its local queue (sorted by response time deadline); otherwise it sends the token to the node which corresponds to the first element of the queue. Moreover, N piggybacks its local queue in the token message to ensure that the new root node is aware of all the pending requests. When the new root node receives the token message, it merges the token's queue with its own local queue. However, in order to improve the critical section throughput, we introduce a preemption mechanism that takes into account requests' locality. We denote $NextHolder$ the next node that should get the token, according to the EDF policy. $NextHolder$ can be preempted by another node p, i.e., p executes a critical section before $NextHolder$, if the duration of p's critical section does not prevent the satisfaction of $NextHolder$'s deadline. In other words, it is possible to grant the token to other nodes if the duration of their critical section plus token transmission delay do not exceed $NextHolder$'s deadline. We denote such a condition *preemption condition*.

Therefore, node p, that receives the token and must forward it to the first element of its queue, might enter the critical section if the *preemption condition* is satisfied. Furthermore, if such a local preemption is not possible, p might grant the token to one of its 1-hop neighbors (not necessarily on the directed path to $NextHolder$), provided that the latter also ensures the *preemption condition*. If none of p's neighbors can satisfy the condition, p applies the same approach to its 2-hop neighbors and so forth (preemption distance size) till a threshold B, bounded by the diameter of the logical tree. If no preemption is possible at all, the token is forwarded to the first element of p's queue.

Let $NextHolder$ be the first requesting site of p's local queue Q and n a n-hop neighbor ($1 \leq n \leq Neighborhood_size$). Site n can preempt $NextHolder$ if the cost in time of the token's rerouting to n does not induce the violation of $NextHolder$'s deadline.

IV. Performance Evaluation

A. Experimental testbed and configuration

The experiments were conducted on a 20-nodes cluster (one process per node). Each node is equipped with two 2.8GHz Xeon processors and 2GB of RAM, running Linux 2.6. Nodes are linked by a 1 Gbit/s Ethernet switch. The algorithms were implemented using C++ and OpenMPI.

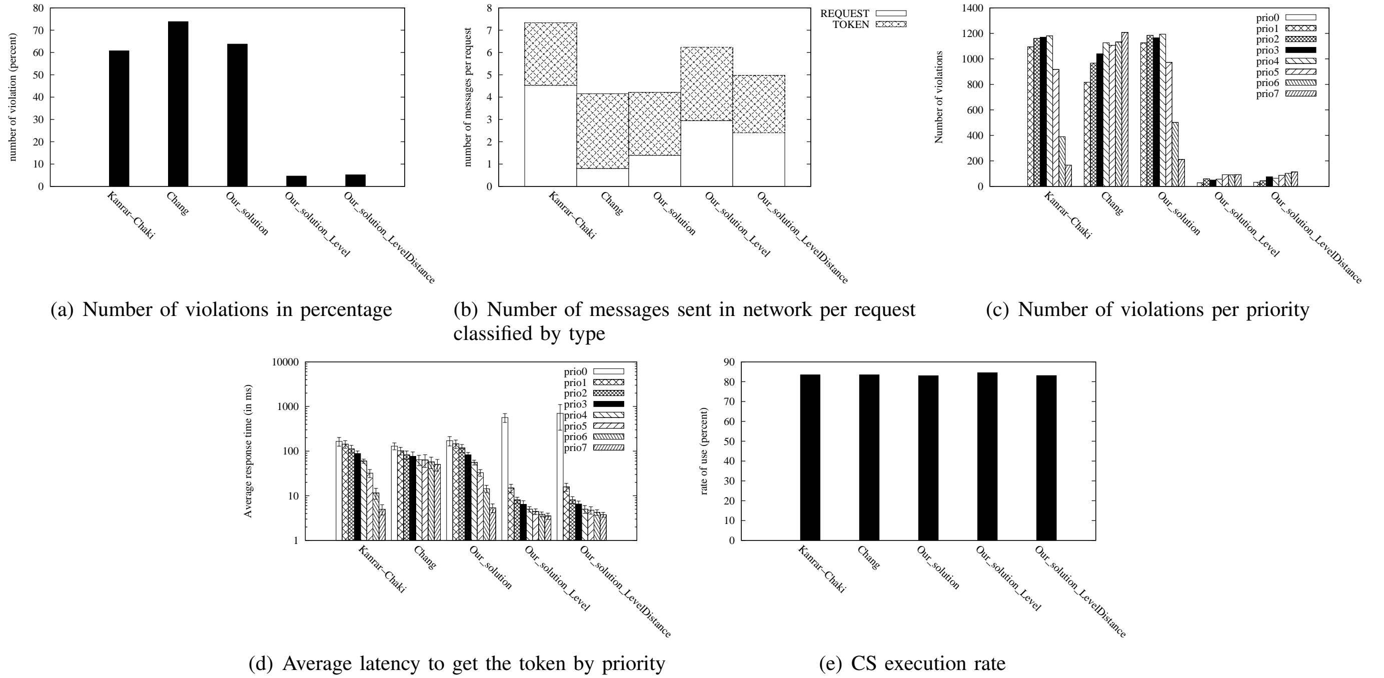
The following metrics were considered in our experiments:

- *Number of messages per request*: This metric depends both on the algorithm and the type of message. Priority-based algorithm: for a given type of message, it is the quotient between the total number of messages of this type and the total number of messages. It is similarly defined for the response time SLA-based algorithm, except for the token message which is defined as the number of token messages sent over the network divided by the number of accepted requests by the admission control (i.e., rejected requests are not considered for this type of message).
- *Number of SLA violations*: For the priority-based algorithm it expresses the number of requests satisfied after requests with lower priority; for the response time SLA-based algorithm it denotes the number of requests whose response time deadline was not respected.
- *Response time*: the time between the moment a node requests the CS and the moment it gets it.
- *CS execution rate*: ratio of critical section duration over token transmission time.

An application is characterized by:

- 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.
- ρ: the ratio β/α, which expresses the frequency with which the critical section is requested.

For all experiments, by calibrating ρ, the average number of pending requests is around 50 %, i.e., in average 10 nodes always wait for the token.



(a) Number of violations in percentage

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

(c) Number of violations per priority

(d) Average latency to get the token by priority

(e) CS execution rate

Figure 2. Performances of priority-based algorithm

B. Request Prioirity SLA-based mutex

We considered a logical binary tree topology and 8 different priority levels. Every process chooses a priority randomly and issues as many requests as possible during the duration of the experiment. Hence, we experience a stationary request rate: all processes issue requests during the whole experiment.

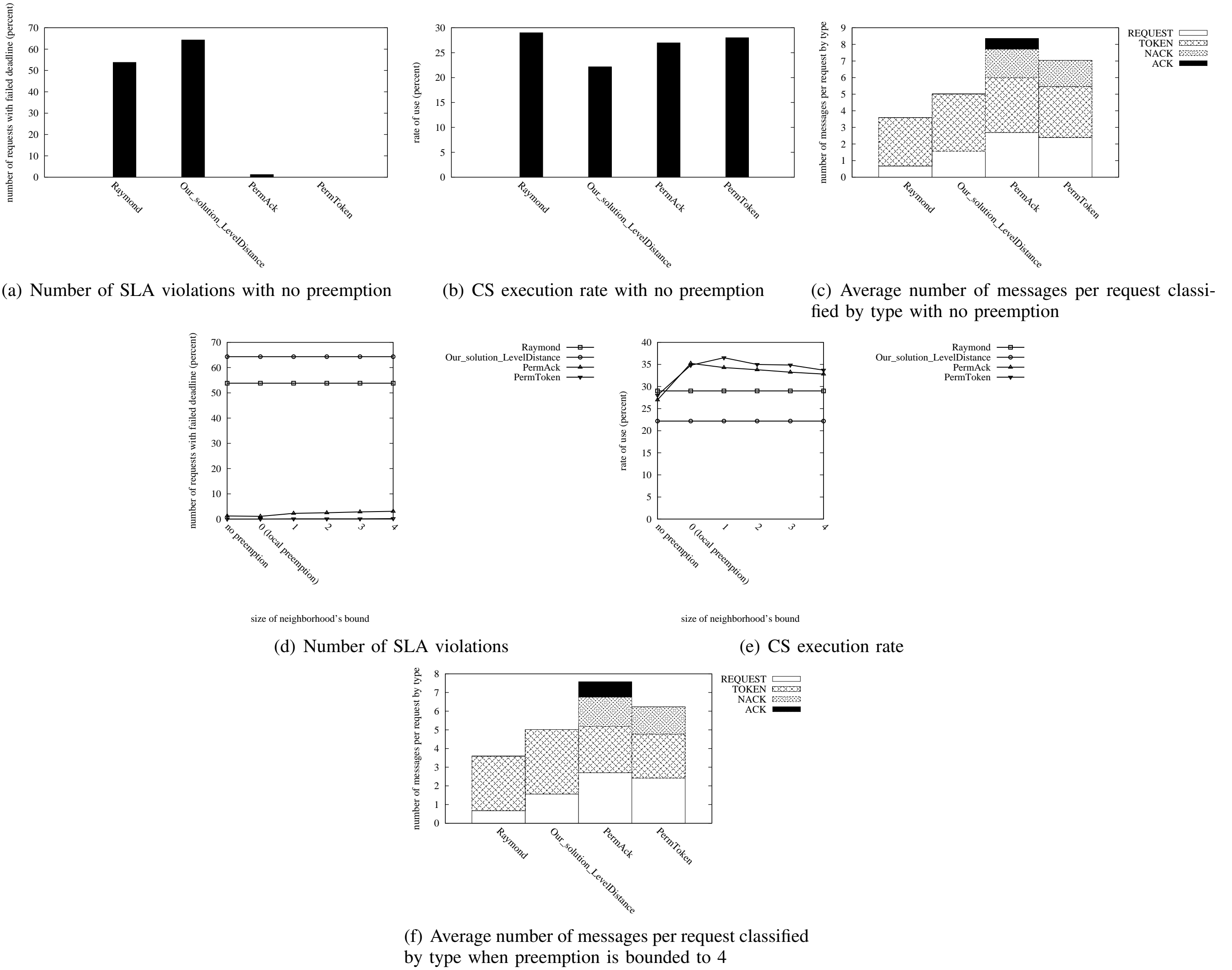
In the figures, $Our_solution$ corresponds to the modified Kanrar-Chaki algorithm with the piggybacking mechanism while $Our_solution_Level$ and $Our_solution_LevelDistance$ correspond to this message traffic optimized algorithm when the *"Level"*, and *"Level-Distance"*' heuristics are respectively applied to it. We have also included Chang's algorithm (see section II) in our performance evaluation experiments.

The number of priority violations and number of messages per request are shown in Figures 2(a) and 2(b) respectively. We can observe that the original Kanrar-Chaki algorithm and Chang's algorithm generate a lot of priority violations (around 60 % and around 75 % respectively). On the contrary, the "Level" heuristic strongly reduces such a number (around 10 %) but, according to Figure 2(b), at the expense of the number of messages which increases when compared to the original Kanrar-Chaki algorithm combined with the piggybacking mechanism. Consequently, the "Level" heuristic is very effective in reducing the numbers of violations but increases the number of messages. Notice that such an increase is mostly due to request messages because a site reaches the maximum priority more slowly and thus it is likely to forward more requests to its father. On the other hand, we observe in the same figures that Chang's algorithm reduces the number of messages in relation to the Kanrar-Chaki algorithm thanks to the piggybacking mechanism. The inclusion of the latter in the Kanrar-Chaki algorithm ($Our_solution$) does not induce much more message traffic overhead when compared to Chang's algorithm. However, in terms of the number of priority violations, its reduction is not very expressive, contrarily to the "Level" heuristic whose number of priority violations is much smaller than Chang's. Therefore, applying request locality to the *"Level"* heuristic, i.e., the *"Level-Distance"* heuristic, seems to be a good tradeoff for these metrics: the two figures confirm that the postponement of priority increment is essential for respecting the priority order while request locality is useful in reducing the number of messages generated by the algorithm.

Figure 2(c) shows the number of requests, grouped by priority, which have been violated by a lower priority request. The *"Level"* heuristic and the *"Level-Distance"* heuristic considerably reduce the number of violations of low and intermediate priorities. Hence, respect for priorities is improved with the *"Level"* heuristic.

Concerning request response time, we can observe in Figure 2(d) that in the original Kanrar-Chaki algorithm such a time has a regular behavior (shape of stairs), i.e., when the priority increases, response time decreases. However, postponement of priority increment strongly degrades response time of the lowest priority requests while the response time for higher priorities request is reduced when compared to the original algorithm. This happens because the former is in best-effort and thus rarely satisfied. The same figure shows that request locality (*"Level-Distance"* heuristic) has no impact on the response time.

As we can observe in Figure 2(e), the different heuristics have no impact (around 90 %) over the CS execution rate. Contrarily to the Response time SLA-based mutex, every request is accepted because there is no admission control.



(a) Number of SLA violations with no preemption

(b) CS execution rate with no preemption

(c) Average number of messages per request classified by type with no preemption

(d) Number of SLA violations

(e) CS execution rate

(f) Average number of messages per request classified by type when preemption is bounded to 4

Figure 3. Performances of response time-based algorithm

C. Response time SLA-based mutex

For the experiments, we considered four different SLA levels. Each level is related to a given response time. Before each request, processes randomly choose a SLA level and a critical section duration (within an interval of bounded values) and issues as many as requests as possible during the duration of an experiment (to obtain a stationary request rate along the same principle as the previous benchmark).

We have defined the following parameters:

- ***T_min***: the lowest response time corresponding to the highest SLA level (600 milliseconds);
- ***diff_SLA***: the difference in response time between two SLA levels (300 milliseconds);
- ***min_CS***: the smallest duration for a CS (25 milliseconds);
- ***max_CS***: the highest duration for a CS (50 milliseconds);
- ***T_exp***: duration of the experiment (60 seconds);
- ***lat_net***: transmission delay of a message between two neighbor nodes (30 milliseconds).

Since there is no distributed token-based mutual exclusion algorithm in the literature that takes into account the concept of SLA based on response time, we compared our two approaches of the algorithm (Acknowledgment and Token) to both Raymond's algorithm and our *"Level-Distance"* Kanrar-Chaki algorithm presented in section III-A3. Furthermore, as Kanrar-Chaki does not consider any time parameter when issuing a request, we mapped priority levels to SLA levels: the higher the time constraint, i.e. the smaller the maximum waiting response delay parameter, the higher the priority. We added a deadline missed detector for Raymond's and the Kanrar-Chaki algorithms. Hence, when a site detects that its request has missed its deadline, it removes the request from its local queue. This request is definitively lost. This site will issue a new request. Upon reception of the latter, all sites that still keep the old request will erase the latter replacing it by the new one respecting the queue order.

Figures 3(a) and 3(b) show the violation and CS execution

rate respectively when no preemption takes place. In Figures 3(d) and 3(e) we can observe the impact of the neighborhood's bound preemption distance over SLA violation and CS execution rate respectively. Finally, Figure 3(c) and 3(f) show the average number of messages per request grouped by type when, respectively, there is no preemption and preemption is bounded to 4-hop neighbors.

In terms of SLA violation, we observe in Figure 3(a) that the direct mapping of priorities to time constraints is not a suitable approach (around 65 % of requests miss their respective deadline). Contrarily, in the case of our *token approach* algorithm, all requests have been satisfied before their deadline. Furthermore, according to Figure 3(d) the size of neighborhood's bound preemption distance has no influence on the number of violations. Figure 3(e) confirms that some preemptions took place and the corresponding critical sections were executed.
In Figure 3(b), we observe that the CS execution rate for Raymond's algorithm is more effective when there is no preemption. We can thus deduce that Raymond's algorithm promotes request locality when network latencies are non negligible. When the token is preempted and the preemption distance size increases, the CS execution rate of the two response time algorithm approaches increases up to a value which is limited by the neighborhood (equal to 1-hop). However beyond this value, the CS execution rate decreases. Such a behavior can be explained: when the preemption distance increases, the token is likely to follow a longer deviation path of the tree. On the other hand, when network latencies are high, a long deviation path may be disadvantageous because it prevents the use of the token by a greater number of processes on the directed path to the $NextHolder$ node.

In Figure 3(c), we can observe that the *acknowledgement approach* generates much more messages than *token approach* due to the ACK messages in response to the accepted requests. Contrarily to the priority-based algorithm, Raymond's algorithm and *"Level-Distance"* present more token messages. Such a difference can be explained since, in these algorithms, the token is sent to the requesting node even if it has missed its deadline which implies useless token transmissions and, therefore, an increase in the average number of messages per request. Figure 3(f) shows that preemption reduces the number of token messages.

V. Conclusion

Our contribution is twofold: a heuristic-based version of the Kanrar-Chaki algorithm aimed at Cloud environments, and a novel algorithm that provides a mutual exclusion service which imposes predefined request response times and is thus also suitable for Clouds.

In the first algorithm, request priorities are dynamic in order to avoid starvation and thus ensure that requests with the lowest priorities are satisfied in a bounded time. However, dynamic priorities induce priority inversion (in our case, SLA violation). Therefore, it is necessary to find a tradeoff between starvation and priority inversion.

The second algorithm is based on Raymond's algorithm because it is scalable. Our approach is innovative and customized for clouds since it provides both an admission request control and a deadline-based request scheduling.

The evaluation results of section IV confirm that request locality improves the performance of the two algorithms: it reduces the number of messages sent in the network and in case of response time SLA-based, it increases the CS execution rate by diverting the token to the neighborhood of the token holder. They also show that our heuristics and novel algorithm reduce SLA violation.

As future work, we intend both to conduct our experiments in a real cloud environment and to extend our response time SLA-based algorithm to take into account node migration, which requires dynamic reconfiguration of the tree topology as in the Naimi-Trehel [10] algorithm.

References

[1] S. Kanrar and N. Chaki, "Fapp: A new fairness algorithm for priority process mutual exclusion in distributed systems," *JNW*, vol. 5, no. 1, pp. 11–18, 2010.

[2] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Trans. Comput. Syst.*, vol. 7, no. 1, pp. 61–77, 1989.

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *J. Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Tech. Rep., 2009.

[5] M. G. Velazquez, "A survey of distributed mutual exclusion algorithms," Tech. Rep., 1993.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[7] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Commun. ACM*, vol. 24, pp. 9–17, January 1981. [Online]. Available: http://doi.acm.org/10.1145/358527.358537

[8] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 145–159, May 1985. [Online]. Available: http://doi.acm.org/10.1145/214438.214445

[9] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 3, no. 4, pp. 344–349, 1985.

[10] M. Naimi and M. Trehel, "An improvement of the log(n) distributed algorithm for mutual exclusion," in *ICDCS*, 1987, pp. 371–377.

[11] K. Birman, G. Chockler, and R. van Renesse, "Toward a cloud computing research agenda," *SIGACT News*, vol. 40, pp. 68–80, 2009.

[12] A. M. Goscinski, "Two algorithms for mutual exclusion in real-time distributed computer systems," *J. Parallel Distrib. Comput.*, vol. 9, no. 1, pp. 77–82, 1990.

[13] F. Mueller, "Prioritized token-based mutual exclusion for distributed systems," in *IPPS/SPDP*, 1998, pp. 791–795.

[14] A. Housni and M. Trehel, "Distributed mutual exclusion token-permission based by prioritized groups," in *AICCSA*, 2001, pp. 253–259.

[15] Y.-I. Chang, "Design of mutual exclusion algorithms for real-time distributed systems," *J. Inf. Sci. Eng.*, vol. 11, no. 4, pp. 527–548, 1994.

[16] T. Johnson and R. E. Newman-Wolfe, "A comparison of fast and low overhead distributed priority locks," *J. Parallel Distrib. Comput.*, vol. 32, no. 1, pp. 74–89, 1996.

[17] K. Han, "Scheduling distributed real-time tasks in unreliable and untrustworthy systems," Ph.D. dissertation, Faculty of the Virginia Polytechnic Institute and State University - USA, 2010.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[19] R. Rajkuman, *Synchronization in Real-time Systems; a priority inheritance approach*. Boston: Kluwer Academic Publishers, 1991.

[20] C. Zhang and D. W. Cordes, "Simulation of resource synchronization in a dynamic real-time distributed computing environment," *Concurrency - Practice and Experience*, vol. 16, no. 14, pp. 1433–1451, 2004.