EAI Endorsed Transactions
on Cloud Systems
Research Article EAI.EU

A Permission-Based Distributed Mutual Exclusion Algorithm allowing Quality of Service (PBDMEAQoS)

E. D. NGOUNOU NTOUKAM[1, *], V. C. KAMLA [2] and J. C. KAMGANG[3]

[1] The university of NGAOUNDERE, P. O. Box 455 CAMEROON (ngounoudimitry@gmail.com)
[2] The university of NGAOUNDERE, P. O. Box 455 CAMEROON (vckamla@gmail.com)
[3] The university of NGAOUNDERE, P. O. Box 455 CAMEROON (jckamgang@gmail.com)

Abstract

The main purpose of mutual exclusion in a distributed environment is to control access to a shared resource. Large-scale distributed systems such as clouds or grids provide shared informatics resources to its clients. In this type of environment, Service Level Agreement (SLA) allows for the definition of a type of quality of service (QoS) between a resource provider and a client. This means that some constraints like priority, response time or reliability must be taken into consideration to maintain a good QoS. Permission-based algorithms are costly in messages, not easily extensible and naturally more robust, pertaining to failures when compared to token algorithms. In this paper, we propose two mutual exclusion algorithms, integrating priority and time constraints for each request, via deadline and execution time in the critical section, with the aim of ensuring a proper service quality. The proposed algorithms are based on a logical structure of nodes in complete binary trees. The algorithms named PBDMEAQoSα and PBDMEAQoSβ are SLA (Service Level Agreement) based. They integrate priority dynamics, which cumulates with the age of a request. PBDMEAQoSα requires $3\log_2 N$ messages per access to critical section and a synchronization delay of $2\log_2 N$ for a set of N nodes competing for the critical resource. PBDMEAQoSβ requires $2\log_2 N$ messages per access to critical section and a synchronization delay of $\log_2 N$.

Keywords: Distributed algorithm, mutual exclusion, time constraints, SLA, QoS

Received on 3 October 2017, accepted on 30 November 2017, published on 20 December 2017

doi: 10.4108/_____

1. Introduction

Distributed systems have been the centre of interest in computer science for the last three decades. Distributed systems are a collection of autonomous computers connected by communication network. One of the main goals of distributed system is to provide an efficient and convenient environment for sharing resources. Certain resources can't be accessed simultaneously by different processes, mutual exclusion therefore permits us to solve this problem. This kind of resource is called a critical resource and the execution inside such resource is called critical section. Many distributed mutual exclusion algorithms have been proposed in literature. The taxonomy proposed by [4], [5] and [6] distinguishes two families of algorithm. The first family is permission based. In this algorithm family, a node enters the critical section after the permission of a set of nodes. The second family is token-based, where the system has a single token and the possession of this token by one site gives him the right to enter in critical section. The main goal of the algorithms cited above was to reduce the number of messages

* Corresponding author: ngounoudimitry@gmail.com

exchanged per critical section. In most of those algorithms the order of critical section request was FIFO (First In First Out). The response time was finite but not bound. This type of algorithm is not always suitable for new distributed systems such as cloud computing which has some requirements in term of Quality of Service (priority, response time, . . .) [7]. The QoS requirements are important for any application since these are recorded as an agreement between the customer and the system designer. Any violation from QoS may lead to customer dissatisfaction, hence, must be taken seriously [8]. Many mutex algorithms have been proposed to this new type of distributed system. We can list some priority-based algorithms (eg.: Goscinksi, Housni-Trhel, Mueller,KanrarChaki) where a node of system has a privilege which determines the weight of his request in the system. There are some algorithms that are dedicated to cloud [19] [21] [7].

We observe that all these new algorithms for modern distributed systems are essentially token-based and are in most case either essentially priority-based or time constraint algorithms. Token-based mutual exclusion algorithms are easy to put in place, present weak average message traffic and are easy to extend, but the loss of token or failure of a node is difficult to manage. Permission-based mutual exclusion algorithms present heavy average message traffic and are not easy to extend but are naturally fault-tolerant. There is no type of algorithm better than another, but their uses are contextualized for a specific objective performance [9]. Thus, we propose in this article two permission-based mutual exclusion algorithm which consider QoS requirement defined in Agreement. QoS requirements defined in our algorithm are time constraint and priority. The time constraint includes the deadline of the request and its execution time in critical resource. Therefore, the purpose of our algorithm is to minimize the violation of the contract. This means that there is a reduction of the number of requests which were not satisfied before a given response time (deadline of request). Our algorithms are permission based algorithm which is more robust to of failures than token-based. They use the complete Binary Tree topology which provides a better bandwidth (in order of $log(N)$) as compared to other algorithms. The rest of the paper is organized as follows; Section 2 discusses about some existing priority-based mutual exclusion distributed algorithms and some mutual exclusion algorithms for cloud environment. Section 3 presents some definitions as well as some assumptions about the considered system. Our mutual exclusion algorithms named PBDMEAQoSα and PBDMEAQoSβ are presented in Section 4 and 5. The comparison between PBDMEAQoSα and PBDMEAQoSβ is given in section 6. Finally, Section 7 concludes the paper.

2. Related Work

Priority is the level of importance of something. To be sure that each process will have access to the critical resource in time, a priority must be affected to each process. Several priorities-based algorithms have been proposed for realtime systems. According to Lejeune et *al.* in [11] we distinguish two families of priority-based algorithms: static priorities and dynamic priorities

- Static priority Algorithms

In the static priority algorithms, the priority of a process remains the same until it enters the critical section. The entry of a process in the CS respects the order of priorities without inversion of priorities. Nevertheless, the starvation of nodes with weak priority remains always possible if nodes with greater priority perpetually request the CS. **Goscinksi** [12] ameliorated Suzuki-Kasami's [13] algorithm by inserting a priority. Requests in waiting CS are recorded in a global queue and are ordered by priority. The queue is included in the token. The algorithm presents a message complexity of $O(N)$. **Housni-Trhel algorithm** [14] adopted a hierarchical approach where nodes are grouped by priority. In each group one router node represents the group close to other groups. Routers nodes use Ricart-Agrawala's [15] algorithm between them. In each group the algorithm applied is Raymond's algorithm [16]. A process can only send requests with the same priority (that of its group). **Mueller's algorithm** [17] was inspired from the Naimi-Trehel algorithm where the circulation of the token uses a dynamic tree as a logical structure for forwarding requests. Each node stores the date of reception of each request and keeps it in local queue. These queues form a virtual global queue ordered by priority.

- Dynamic priority Algorithms

In the dynamic priority algorithms, the priority of a request is incremented with time to assume liveliness property. Generally, the priority of hanging requests is increased at each record of a new request with high priority. The **Kanrar-Chaki** [1] algorithm is based on Raymond's algorithm. They introduced a priority level for every process CS request. The greater is the priority level, the more urgent is the CS request. Every pending request of a local process queue is ordered by decreasing priority levels. A process that wishes the token sends a request message to its father as Raymond's algorithm. To avoid starvation, the priority level of pending requests of a local process 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. Aiming at reducing the number of priority inversions, **Lejeune** et ***al.*** [7] propose a new algorithm with a reduction of incrementation of priorities. Though, this reduction of incrementation always assumes the liveliness property, a process with low priority could have enormous waiting time in certain configuration. To resolve the problem of waiting time, Lejeune et *al.* proposed another algorithm in [18]. This algorithm is based on the circulation of token inside a static tree topology and considerably reduces the waiting time of process to a given rate inversion. Cloud computing essentially outsources the computing infrastructure of a user or an organization to data centers. These centers allow thousands of their clients to share their computing resources through the Internet for efficient computing at an affordable cost [3]. To achieve satisfaction of client Service Level Agreement (SLA) is defined between

the client and cloud provider. Any SLA violations can lead to client dissatisfaction. In comparison with classical distributed systems, the cloud computing environment has to be dealt with differently because of following characteristics: Scalability, Fault tolerance, Quality of Services, Different Priorities [19]. To consider all these characteristics, some mutual exclusion algorithms have been dedicated to cloud environment.
Emondson-Schmidt-Gokhle's [19] algorithm called Prioritizable Adaptive Distributed Mutual Exclusion (PADME) requires that customers be differentiated by the price they pay for different services. In the other hand, the customers who pay more than other customers must have greater priority. PADME is based on a spanning tree where higher requests are pushed towards the root. PADME uses three types of messages: **Request**, **Reply** and **Release. Emondson** Schmidt-Gokhle also presented a fault tolerant version of PADME. Token-based distributed mutual exclusion algorithm has been proposed by Lejeune et *al.* [7] with the aim to support SLA. To respect SLA Lejeune and al. put in place an admission control to accept or reject requests. Accepted requests will be satisfied, with great probability, before their deadline. The Lejeune et *al.* [7] algorithm is based on Raymond's algorithm. Requests are sorted at a process local queue by their response time deadline, similarly to the real-time scheduling policy Earliest Deadline First (EDF). They proposed two mechanisms to minimize SLA violations. **Admission control**: the feasibility of a request satisfaction should be checked before including the request in the system and **Preemption mechanism**: which permits to decide which path the token must follow inside the topology in order to make profitable its utilization.

Recently, gossip-based algorithms have received much attention due to their inherent scalable and fault-tolerant properties, which offer additional benefits in distributed systems. In gossip-based algorithms, each node maintains several neighbours, called a partial view. With this partial view, at each cycle (round), every node in the system selects f(fanout) number of nodes randomly and then communicates using one of the following ways: 1) Push, 2) Pull, and 3) Push pull mode [21]. Applications of gossip-based algorithms include message dissemination, failure detection services, data aggregation etc. Lim et *al.* [21] proposed a gossip-based mutual exclusion algorithm for cloud computing systems with dynamic membership behaviour. The amortized message complexity of our algorithm is $O(n)$, where n is the number of nodes. Their simulation results show that their proposed algorithm is attractive regarding dynamic membership behaviour.

3. The system model

In this section, we present some definition and assumptions, variables and messages exchanged between nodes, and finally the criterion of sort of pending queues.

3.1 Definitions and assumptions

We suppose that N nodes of our distributed system communicate under a network which is reliable, and nodes are not prone to failures. The words node, process, and site are interchangeable. The delay of a message delivery is supposed constant γ. The treatment time of request e at each time is supposed constant. The physical clock of each node does not change during the execution of our algorithm. Our algorithm runs on a distributed system of N physically dispersed autonomous computers sites that logically form a complete binary tree and communicate with one another only by sending messages. It is assumed that the sites numbered from 1 to N form a tree topology. The implementation of this kind of tree is made in a way that each node i has two children node ($2i$ and node $2i + 1$). Each node i also keeps inside his local queue all request of CS that he received. We assume that each node has two layers: the **application layer** and the **network layer**. Every request emanates from the application layer. It is the layer which changes state (calm, requesting, in CS). The network layer permits to receive and transfer the messages from the network layer of other node or from its own application layer.

3.2 Local variables and messages

For each site Si, the algorithm defines the following local variables which are updated:
– **Flag** allows us to know if the node is inside the CS. It is 1 if the node is inside and 0 otherwise;
– **Granted** permits us to know if the node gave an authorisation to another node to enter inside CS, 1 if it is the case and 0 otherwise;
– **State** is used to know if the node is the requesting state or not, 1 if it is the case and 0 otherwise;
– **Q***c* stores the local queue of all request of CS received ;
– $N_{i,r}$ gives information about the number of links between the root of tree and node i. The construction of the tree is done in a way that each node added in system takes the $N_{i,r}$ of its parent and adds one into it to determine its own $N_{i,r}$ value.

Three types of messages are used: **REQUEST** (to request access to the critical section), **REPLY** (to grant access to the critical section) and **EXIT** (to release the critical section). A message keeps the following information:
–id_i its sender-id, id_p its receiver-id;
– *Bi* which is the rest of request waiting time, *Ai* which is the request constant, *msgi* which is the message type. Note that for **REPLY** and **EXIT** message parameters *Bi* and *Ai* are not necessary. The following functions used in our algorithm: *init()* which initiates our environment; *Request_section()* which describes the instruction for requesting the critical resource; *Exit_section()* which describes how a process exits the CS; Treatment message() which describes the behaviour of node at the arrival of a

message; *InsertMessage(Message$_m$)* which describe how a new request is inserted in a queue. Other variables are necessary to be put in place in our algorithm.

- p_i: priority of initiator site of request ;
- h_i : the initial Lamport's timestamp [20];
- d_{sci}: latest date of CS exit (as illustrated in figure 1) ;
- d_{scLi}: latest local date of CS exit;
- t: current date at given site ;
- t_{2i}: latest date of entrance the CS (like illustrated in figure 1) ;
- t_{1i}: latest authorisation date of access to critical resource (as illustrated in figure1) ;
- t_0: emission date of the request;
- $N_{c,r}$: number of intermediate links separate current site where the request found and the root of the tree;
- e: treatment time of request at a site (supposed constant for every request);
- s_i, s_c, s_p, s_r and root are respectively request site, current site, parent site of current site, message receiver site, root site;
- msg: message Object;
- d_a: waiting time of request;
- Message: each message exchanged or stored;
- $V_{i,c(t)}$: Value *V* of request *i* at a site *c* at a moment *t*.

3.3 Criterion of sort

In known algorithms, local queues are sorted by FIFO policy. This policy doesn't consider waiting time, priority and execution time of the process. These parameters are important for QoS. They must be considered during the insertion of a request in the local queue. We draw up a formula accounts for these parameters. The formula obtained must permit us sort or insert a new process in a pending queue.

$$V_{i,c}(t) = \frac{t - h_i + 1}{d_{sci} - (d_{e_i} + (\gamma + e) \times (N_{i,r} + N_{c,r}) + e) - t} \times p_i \times \frac{1}{d_{e_i}} \quad (1)$$

With $1 \le i \le N$ and $1 \le j \le N$

We suppose that:

$$\forall t, d_{sc_i} - (d_{e_i} + (\gamma + e) \times (N_{i,r} + N_{c,r}) + e) - t \neq 0$$

$$\forall P_i, d_{e_i} \text{ and } h_i \neq 0$$

$V_{i,c}(t)$ obtained is the priority of the process *Pi* inside a queue of processes at a given time *t*. This priority is **dynamic** and **temporal** because it varies during the execution and depends essentially on the waiting time of request. The value $V_{i,c}(t)$ is the criteria admission of the request in the CS at a given time. *Vi,c*(*t*) can be divided into three parts. We have:

$$A = h_i - 1 \qquad C = p_i \times \frac{1}{d_{e_i}}$$

$$B = d_{sc_i} - (d_{e_i} + (\gamma + e) \times (N_{i,r} + N_{c,r}) + e)$$

Therefore, equation 1 can be summarise as:

$$V_{i,c}(t) = \frac{t - A}{B - t} \times C \quad (2)$$

The parameter *A* represents the emission date of the request. Thus, the age of a request is the difference between the current time and *A*. We added 1 to h_i and obtained *A*, this is to avoid having request having zero age. In *C*, we observe that the value of *C* doesn't change with time, due to its parameters (p_i, *dei*) which are statics. On the other hand, wherever the request is found the value of *C* is the same. *pi* is the priority of initiator node of request. To prioritize request with a lower execution time we use $1/d_{ei}$ in our formula. *B* represents the rest of the waiting time in the expression of $V_{i,c(t)}$, it's situated between t_{0i} and t_{2i} as shown in figure 1. The difference between the deadline of request and the CS time, transition time between the initiator node and the root of the tree must not exceed the latest authorisation date of access to critical resource t_{1i}. This is done to satisfy a request before its deadline. **Notice that we consider that all nodes have a virtual clock that are synchronized**.

4. PBDMEAQoSα Algorithm

4.1 Description of the PBDMEAQoSα algorithm

Figure 2 shows the pseudo code of PBDMEAQoS*α* algorithm. When a site s_i wants to enter the critical section at time *t*, it defines its expiration date $d_{sci} > t$, and estimates the duration of execution d_{ei} in the critical section (line **11** to **15**). It inserts into its own list of queries a REQUESTtype message which it transmits itself. It then sends a request to its parent's site s_p in case it's not the root of the tree (line **19** to **21**) via its network layer. Upon reception of a REQUEST message by the current site s_c from its application layer or the network layer of one of its son nodes, it is verified that the waiting time has not yet been reached (ie $t > B_i + (\gamma + e)$) for demands from its daughter sites). If this is the case, this message is deleted (line **56**). Otherwise, the current site s_c updates the B_i parameter ($B_i = B_i + (\gamma + e)$) of the message (line **58**). It then inserts the message in its own line of requests (line **59**). The current site s_c transmits the message to its parent site s_p by substituting itself to the sender (line **60**). In case the current site is the root of the tree (ROOT), it is checked that the waiting time is not completed ($B_i + (\gamma + e) > t$) (line **43** to **44**). If this is the case, this message is inserted into the local queue. Otherwise it is removed from the system. If an entry into the CS has not been authorized (Granted == 0) the most urgent request in the queue will be searched for and an authorization message sent to it. Upon reception of a **REPLY** message at a site s_c (line **61**), if this **REPLY** message belongs to it, it checks whether the deadline of its request has not yet been reached. If that is the case, it sets its Flag to 1 and enters the critical section. Otherwise, it goes in its demand queue *Qc*, searches for the highest value demand $V_{i(t)}$ still permissible in CS while

withdrawing requests with due dates attained. As soon as it finds it, it sends a **REPLY** message and sets its Granted variable to 1(Figure 2, line **83** to **99**). At the output of the critical section, the site deletes its request from its own local queue and sets its Flag and Granted to 0. If it is the root, it sends an **EXIT** message to itself; Otherwise, it sends an **EXIT** message to its parent site (line **24** to **37**). Upon reception of an **EXIT** message by a s_c (line **71**), the latter sets its Granted variable to 0 and looks for the initiator process of the **EXIT** message in its pending queue, removes it while withdrawing the demands with deadlines reached (line **73** to **76**). If the site s_c is the root site (line **77**), it sends a **REPLY** message to itself if its queue is not empty. If site *sc* is not the root site, it sends an **EXIT** message to its parent site (line **78** to **82**). The waiting time for each request is fixed at the initiation of the request. If this duration is zero, the value *Vi*(*t*) will also be. When the due date is reached (Vi(*t*) <= 0), the message is simply deleted from the list. If this message has been emitted by the current site, the application layer informs the network layer that the waiting time has expired for this request.

4.2 Theoretical analysis of PBDMEAQoSα algorithm

For a mutual exclusion algorithm to be considered correct, it must satisfy the properties of safety and liveliness.

Proof of liveliness: Liveliness avoids blocking and starvation situations in processes requiring access to a resource. In our algorithm, we define the waiting time for the process. This allows us to cancel the process once this waiting time is exceeded and notify a rejection message to the process initiator. The main aim is to minimize the number of rejections. We also define the execution time and priority of a process. To avoid the greatest number of rejections, the most urgent process is chosen according to *Vi*(*t*) (combination of wait time, priority and run time parameters). This is to allow a large number of pending applications to have a chance to use the critical resource efficiently.

ASSERTION 1. *Deadlock is impossible* The system of nodes is said to be deadlocked when no node is in its critical section and no requesting node can ever proceed to its own critical section [15].

PROOF. Attending to the most urgent request of a pending queue allows to assign the resource to one process at least. Once it has been executed, it must be reported (**EXIT** message) to the entire system. Removing queries from the local queue of a site once their due date reached allows for an increase in the allocation of the critical resource. Thus, at time *t*, a process *i* has a value of V_i of a low emergency: At a time, $t + \Delta t$, a process *i* will have a value of V_i with a high degree of emergency; At time $t + (\Delta + 1)\ t$, a process *i* will have a value of *Vi* which could turn to zero or less. We can have in a queue two processes *i* and *j* that have the same value of *V* ($V_i = V_j$) at a certain time. This can only happen at a given moment, for there exists a single *t* for which *Vi* = *Vj*. In this type of situation (*Vi* = *Vj*), if our queue has only two processes, and an authorization message input to CS arrives, one of the two processes *i* and *j* we be randomly chosen. This ensures a constant evolution of the system, hence the vivacity of our system.

ASSERTION 2. *Starvation is impossible* Starvation occurs when node must wait indefinitely to enter its critical section even though other nodes are entering and exiting their own critical sections [15].

PROOF: In our case, a process can only wait for the time it has previously defined. Consider two processes *i* and *j*. At a time, *t*, $V_i > V_j$ if *i* runs at CS at this time. Once process *i* is taken out of the SC, it can only run a second time after *j* because for process *i* to emit a request again, it would have to signal its output (**REPLY** message) and cover all the intermediate links that separate it from the current node ($\gamma + e$) *Ni,r*. This is being done for a certain duration Δt. Therefore, at a time $t + \Delta t$, *Vj* will necessarily be more urgent because *Vi* changes with time. Therefore, process *I* would like to enter a second time in SC, process *Vi* would be greater or equal to *Vi*, therefore $Vj \geq Vi$. This avoids starvation situations. However, repetitive transmission of several requests with small parameters to a single node near the root could lead to the famine of a process in the pending queue of this node.

Proof of safety

ASSERTION 3. *Mutual exclusion is achieved.* Mutual exclusion is achieved when no pair of nodes ever simultaneously in its critical section. For any pair of nodes, one must leave its critical section before the other may enter [15].

PROOF. Since our topology is a complete binary tree, only the root node can have knowledge of the whole state of the system. However, it can give its authorization (**REPLY** message) only to one of its children node. Entry into the critical section from a site is only possible after the authorization from all its parent sites from the root site. Thus, the root site can authorize input into critical section only if it has received an **EXIT** message, meaning the release of the critical resource. We deduce that our algorithm can admit at most a single process in critical section at a given time. It does verify the safety criterion.

Bandwidth analysis: The aim here is to evaluate the number of messages exchanged to enter critical section in a set of N sites in total. As with the previous algorithm, for a request to reach the root site, it takes $log_2(N)$ messages in the worst case. This represents the average size of a complete binary tree of size N. To enter the critical section, a process must send in the worst case $log_2(N)$ **REQUEST** messages, $log_2(N)$ **REPLY** messages and $log_2(N)$ **EXIT** messages. This means that in the worst case a total of $3log_2(N)$ posts per access to the critical resource. However, in our algorithm, a request can be cancelled ($V_{i,c(t)} = 0$) on the way to the root if its expected response time is reached. The worst case corresponds to a query leaving a leaf node to the root, and reaches its due date at the level of the root. This corresponds to $log_2(N)$ **REQUEST** messages. This will make it possible to release the bandwidth of messages with expired waiting time ($V_{i,c(t)} = 0$), and give way to priority messages.

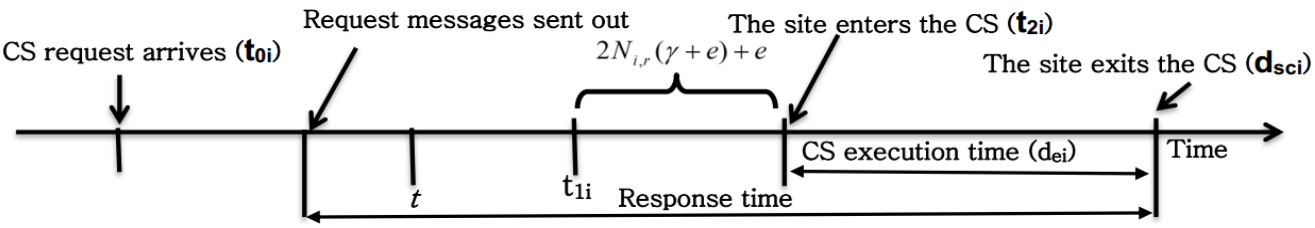

Figure 1: Description of urgency of request

Synchronization delay: The aim here is determining the number of messages between the output of the critical section by a process and the input of the other. At the end of its execution, a process must send an **EXIT** message to the root. In the worst case it will cost $log_2(N)$ messages. Then, it will take $log_2(N)$ **REPLY** messages from the root to allow the next process to enter the critical section. Thus, we have a synchronization delay of $2log_2(N)$ in the worst case.

Cost of queue processing: The aim here is to evaluate the complexity of the processes carried out on the pending queue. Upon reception of a **REPLY** message by a site, the entire pending queue is checked in search of the highest priority request. This has a complexity of order of $\Theta(n)$ for an n elements queue. Upon reception of an **EXIT** message, the current node must go to its local queue and remove the process that sent the message if it still exists in the queue. The worst of cases would be that during the search, the process issuing the message **EXIT** is not in the queue or at the end of the queue. In this case, the complexity of this processing is of the order of $\Theta(n)$.

5. PBDMEAQoSβ Algorithm

5.1 Description of the PBDMEAQoSβ algorithm

Figure 3 shows the pseudo code of PBDMEAQoSβ algorithm. In this algorithm, the granting of input authorization in CS is done automatically. This is done through a timer at the root. This timer allows one to know when to send an authorization. When a site s_i wants to enter the critical section, it defines its expiration date (d_{sci}, this date must always be greater than the current date, t) and estimates the duration of execution (d_{ei}) in the critical section (equation l). It fits into its own list of queries a REQUEST type message which it transmits itself. Then it sends its request to its parent site s_p if it is not the root of the tree via its network layer (Figure 2, line **109** to **122**). Upon reception of a REQUEST message by the current site s_c from its application layer or the network layer of one of its child node (line **144**), it is verified that the waiting time has not yet been reached (ie $t > B_i + (\gamma + e)$) (line **145**). If this is the case, this message is deleted (line **146**). Otherwise, the current site s_c updates the setting B_i ($B_i = B_i + (\gamma + e)$) of the message (line **148**). Then, it inserts the message in its own demand queue (line **149**). The current site s_c sends the message to its parent site s_p by substituting itself to the sender (line **150**). In this algorithm, the root of the tree functions differently from the other nodes. Upon reception of a REQUEST message by the root (line **125**), it is checked that the waiting time is not completed ($Bi + (\gamma + e) > t$) (line **126**). If this is the case, this message is inserted in the local file (line **128**). Otherwise it is removed from the system (line **130**). The CS is initially assigned to the first requests and the end date of execution of the process in CS is defined (*dateFinTimer*). This date is the sum of the current date and the journey time of the number of intermediate links between the root and the node (node to be executed) plus the execution time of the process. The next authorization to enter CS is automatically granted once this date has been reached or exceeded (line **133** to **134**). Once *dateFinTimer* is reached, the root goes into its local file, searches for the most urgent query and defines the variable *dateFinTimer*. Then, it goes through the line, withdraws any requests that cannot survive till the next authorization in CS (line **137** to **140**). Upon reception of a **REPLY** message by a site s_c (line **151**), If this **REPLY** message belongs to it, it sets its Flag to 1 and enters the critical section (line **152** to line **155**). Otherwise, it goes in its query queue Qc, removes from its queue the process

initiating the message **REPLY** received, this while withdrawing requests by deadlines reached (line **156** to line **161**). As soon as it removes the process, it sends a **REPLY** message (line **162** to line **163**).

5.2 Proof of PBDMEAQoSβ algorithm

```
1  Initialization ;
2  Procedure init();
3  begin
4      Flag ← 0;
5      Granted ← 0;
6      msg ← NIL;
7      Q_c ← ∅;
8      Treatment_message() ;
9  Procedure Request_section() ;
10 begin
11     define (d_sc_i);
12     define (d_e_i);
13     A_i ← h_i + 1;
14     B_i ← d_sc_i − (d_e_i + (γ + e) × 2 × N_i,r + e) ;
15     C_i ← p_i × 1/d_e_i;
16     calculate (V_i);
17     if V_i > 0 then
18         msg_i ← REQUEST;
19         if S_i != ROOT then
20             Send Message(id_i,A_i,B_i,C_i,h_i, msg_i) to S_p ;
21         insertMessage (Message) in Q_c ;
22     else
23         notify rejection message
24 Procedure Exit_section() ;
25 begin
26     Flag ← 0;
27     Granted ← 0;
28     msg_i ← EXIT;
29     repeat
30         if (t > B_i) Or IdS_i == IdMessage_i then
31             Remove Message_i from Q_c;
32     until IdS_i == IdMessage_i;
33     if S_c == ROOT then
34         Send Message(id_i,h_i, msg_i) to ROOT ;
35     else
36         Send Message(id_i,h_i, msg_i) to S_p of S_c ;
37     Treatment_message() ;
```

```
38 Procedure Treatment_message() ;
39 begin
40     while Flag == 0 do
41         Receive( Message) from S_i ;
42         if (msg == REQUEST) then
43             if ( S_c == ROOT) then
44                 if (B_i + (γ + e) > t) then
45                     B_i ← B_i + (γ + e);
46                     insertMessage(Message) ;
47                 else
48                     Delete Message_i ;
49                 if (Granted == 0) then
50                     Granted ← 1;
51                     Message_k ← ResearchMax(Q_c) ;
52                     msg_k ← REPLY ;
53                     Send Message (id_k,h_k, msg_k) to S_k ;
54             else
55                 if (t > B_i + (γ + e)) then
56                     Delete Message_i ;
57                 else
58                     B_i ← B_i + (γ + e);
59                     insertMessage(Message) ;
60                     Send Message(id_c,A_i,B_i,h_i, msg_i) to S_p ;
61         if (msg == REPLY) then
62             if (S_c == S_i) then
63                 Flag ← 1;
64                 Granted ← 1;
65                 Enter in Critical section;
66             else
67                 if Q_c != ∅ then
68                     Message_k ← ResearchMax(Q_c) ;
69                     Granted ← 1;
70                     Send msg REPLY to sender of S_k ;
71         if (msg == EXIT) then
72             Granted ← 0;
               /* with Message_k from Q_c
73             repeat
74                 if (t > B_k) Or IdS_i == IdMessage_k then
75                     Remove Message_k from Q_c;
76             until IdS_i == IdMessage_k;
77             if S_c == ROOT then
78                 if Q_c != ∅ then
79                     Granted ← 1;
80                     Send msg REPLY to ROOT ;
81             else
82                 Send msg EXIT to S_p ;
```

Figure 2: The PBDMEAQoSα algorithm

Proof of liveliness

ASSERTION 4. *Deadlock is impossible*
PROOF: At a time, *t* a process *i* is either served if the CS has just been released, or withdrawn from the file because its due date has been reached or it can only wait until the end of the process to which the CS has just been assigned. This ensures a constant evolution of the system, hence the vivacity of our system. The fact that the root is the only node to assign CS periodically allows to ensure the liveliness of the system.

ASSERTION 5. *Starvation process is impossible*
PROOF: In our case, a process can only wait for the time it has previously defined. Consider two processes *i* and *j*. At a time, *t* $V_i > V_j$ if *i* runs at CS at this time. Once process *I* will be taken out of the SC, it will be able to run a second time only after *j*, because for process *i* to issue a request again, it should have to signal its exit (**REPLY** message) and cross all the intermediate links between it and the current node ($\gamma + e$) $N_{i,r}$, this being done for a certain duration Δt. Thus, at any instant $t+\Delta t$, *Vj* will necessarily be more urgent because *Vi* changes with time. Process *i* would like to enter a second time in SC, process *Vj* would be greater or equal to *Vi*, hence $V_j \geq V_i$. This avoids starvation situations. However, repetitive transmission of several requests with small parameters with a single node near the root could lead to the famine of a process in the pending queue of this node.
Proof of safety: It consists in ensuring that the critical resource at a time is used only by one and only one process. Indeed, initially no process has access to the critical resource. Entry into the critical section by a site is only possible after the authorization of the root site. Thus, the root site can only authorize a critical section entry if the release of the critical resource is effective (*dateFinTimer* <= *t*). From this we deduce that our algorithm can admit at most one process in critical section at a given time. Our algorithm thus verifies the safety criterion.
ASSERTION 6. *The algorithm ensures mutual exclusion*
PROOF: Since our topology is a complete binary tree, only the root node can have knowledge of the whole state of the system. It can however only give its authorization to only one of its children node.
Bandwidth analysis: The aim here is to evaluate the number of messages exchanged to enter critical section in a set of *N* sites in total. As with the previous algorithm, it takes $log_2(N)$ messages for a request to reach the root site. This represents the average size of the binary tree of size *N*. To enter a critical section, a process must send in the worst case $log_2(N)$ **REQUEST** messages and $log_2(N)$ **REPLY** messages. This sums to a total of $2log_2(N)$ messages per access to the critical resource in the worst case. However, in our algorithm, a request can be cancelled ($V_{i,c(t)} = 0$) on its way to the root if its expected response time is reached. The worst case corresponds to a query leaving from a leaf node to the root, and reaches its due date at the level of the root. This corresponds to $log_2(N)$ **REQUEST** messages. However, this will make it possible to release the bandwidth of messages with expired waiting time ($Vi,c(t) = 0$), and give way to priority messages.
Synchronization delay At the end of the execution of a process in SC, another process is allowed automatically through the root of the tree. Thus, in the worst case, $log_2(N)$ messages for the effective entry of the process into SC. Thus, we have a synchronization deadline of $log_2(N)$ in the worst case.

Cost of queue processing: The aim here is to evaluate the complexity of the processing carried out on the pending queue. When a **REPLY** message is received, the entire queue must be scanned to withdraw the initiating process from the received **REPLY** message, while removing the requests with due dates reached. In this worst case, the complexity of this treatment is of the order of $\Theta(n)$. At the end of the execution of a process in SC, the root must look for the most urgent process in its queue. This has a complexity of the order of $\Theta(n)$ for an n-element queue.

6. Comparison between the PBDMEAQoSα and PBDMEAQoSβ algorithms

In addition to the comparisons made in Table 1, the processing cost makes it possible to distinguish between the two algorithms. The PBDMEAQoS*α* and the PBDMEAQoS*β* algorithms differ in the fact that in the PBDMEAQoS*β* algorithm, only the root of the tree performs the search for the largest element of its local queue. This makes it possible to say that the algorithm PBDMEAQoS*β* has a low processing cost compared to the PBDMEAQoS*α* algorithm. We can superficially say that the algorithm PBDMEAQoS*β* is better than the PBDMEAQoS*α* algorithm. It is important to note that the PBDMEAQoS*β* algorithm is more appropriate when one wants an algorithm with an efficient use of the critical resource. The PBDMEAQoS*α* algorithm is better adapted when one wishes to privilege processes with high dynamic priorities (according to $V_{i,c(t)}$).

Table 1. Comparison between the PBDMEAQoSα and PBDMEAQoSβ algorithms

	PBDMEAQoSα	PBDMEAQoSβ
Bandwidth	**$3log_2(N)$**	**$2log_2(N)$**
Synchronization delay	**$2log_2(N)$**	**$log_2(N)$**

```
83  Function ResearchMax(Qc) : Message ;
84  begin
85      if Qc == ∅ then
86          return Null;
87      else
88          Message msgMax ← Qc[0] ;
89          VmsgMax calculate (V(msgMax);
90          repeat
91              if (t > Bk) then
92                  Remove Messagek from Qc;
93              Vk ← calculate (V(Messagek));
94              if Vk > VmsgMax then
95                  msgMax ← Messagek;
96                  VmsgMax ← Vk;
97              k++;
98          until (Messagek == end of Qc);
99          Return msgMax ;
100 Procedure init() ;
101 begin
102     Flag ← 0;
103     Granted ← 0;
104     dateFinTimer ← 0 ( this line is only for root node) ;
105     msg ← NIL;
106     Qc ← ∅;
107     Treatment_message() ;
108 Procedure Request_section() ;
109 begin
110     define (dsci);
111     define (dei);
112     Ai ← hi + 1;
113     Bi ← dsci − (dei + (γ + e) × 2 × Ni,r + e) ;
114     Ci ← pi × 1/dei;
115     calculate (V);
116     if Vi > 0 then
117         msgi ← REQUEST;
118         if Si != ROOT then
119             Send Message(idi,Ai,Bi,Ci,dei,Ni,r, msgi) to Sp ;
120         insertMessage (Message) in Qc ;
121     else
122         notify rejection message
123 Procedure TreatmentAtTheRootNode() ;
124 begin
125     Receive( Message) from Si ;
126     if (Bi + (γ + e) > t) then
127         Bi ← Bi + (γ + e);
128         insertMessage(Message) ;
129     else
130         Delete Messagei ;
131     /* this code is automatically execute */;
132     if (dateFinTimer < 0 Or dateFinTimer <= t) then
133         Messagek ← ResearchMax(Qc) ;
134         dateFinTimer ← Nk,r × (γ + e) + dek ;
135         Granted ← 1;
136         Send msg REPLY to sender of Sk ;
137         repeat
138             if (dateFinTimer > Bk) then
139                 Remove Messagek from Qc;
140         until Qc == end of Qc;
141 Procdure Treatment_message() ;
142 begin
143     while Flag == 0 do
144         if (msg == REQUEST) then
145             if (t > Bi + (γ + e)) then
146                 Delete Messagei ;
147             else
148                 Bi ← Bi + (γ + e);
149                 insertMessage(Message) ;
150                 Send Message(idc,Ai,Bi,hi, msgi) to Sp ;
151         if (msg == REPLY) then
152             if (Sc == Si) and ( Bi > t) then
153                 Flag ← 1;
154                 Remove Si from Qc;
155                 Enter in Critical section;
156             else
157                 if Qc != ∅ then
158                     repeat
159                         if (t > Bi) Or IdSi == IdMessagei then
160                             Remove Messagei from Qc;
161                     until IdSi == IdMessagei;
162                     Granted ← 1;
163                     Send msg REPLY to sender of Si ;
```

Figure 3: The PBDMEAQoSβ algorithm

7. Conclusion and perspectives

We have proposed in this paper two permission-based distributed mutual exclusion algorithms called PBDMEAQoSα and PBDMEAQoSβ. In PBDMEAQoSα and PBDMEAQoSβ the quality of service is considered by priority, deadline and execution time of process. The dynamic variation of our classification parameter (Vi,c(t)) is used to reduce to number the SLA violation. This policy here is that processes that cannot wait to obtain authorization and the indication of CS exit by other by the current process which have authorization, should be removed from the system. Thus, process with reached deadlines will be deleted to free the bandwidth. PBDMEAQoSβ algorithm is more appropriate for an efficient use of the critical resource. The PBDMEAQoSα algorithm is better adapted when one wishes to privilege processes with high dynamic priorities (according to $V_{i,c(t)}$).

As future work, we might extend his work using a dynamic root system. The root can be a bottleneck in our current system. we might also consider fault tolerance and adapting our solutions to k-mutual exclusion and group mutual exclusion.

Acknowledgements.

We thank the laboratory LAMEX (Laboratory of Experimental Mathematics) of the Doctoral Training Unit UFD-MIAP.

References

[1] KANRAR, S., & CHAKI, N. (2010). FAPP: A New Fairness Algorithm for Priority Process Mutual Exclusion in Distributed Systems. *JNW*, 5(1), 11-18.

[2] KSHEMKALYANI, A. D., & SINGHAL, M. (2011). *Distributed computing: principles, algorithms, and systems*. Cambridge University Press.

[3] GHOSH, S. (2014). *Distributed systems: an algorithmic approach.* CRC press.

[4] Raynal, M. (1991). A simple taxonomy for distributed mutual exclusion algorithms. **ACM SIGOPS Operating Systems Review**, 25(2), 47-50.

[5] Velazquez, M. G. (1993) A survey of distributed mutual exclusion algorithms. *Tech. rep.*, **Colorado state university**.

[6] Saxena, P. C., & Rai, J. (2003). A survey of permission-based distributed mutual exclusion algorithms. *Computer standards & interfaces*, 25(2), 159-181.

[7] Lejeune, J., Arantes, L., Sopena, J., & Sens, P. (2012, May). Service level agreement for distributed mutual exclusion in cloud computing. *In Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (pp. 180-187). IEEE Computer Society.

[8] Singh, J., Dutta, M., & Swaroop, A. (2015). A QoS Aware Self Adaptive General Scheme to Solve GME Problem. *International Journal of Computer Applications*, *109*(7), 25-29.

[9] Sopena, J. (2008). Algorithmes d'exclusion mutuelle : tolérance aux fautes et adaptation aux grilles (*Doctoral dissertation, Paris 6*).

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

[11] Lejeune, J., Arantes, L., Sopena, J., & Sens, P. (2013, January). Un algorithme équitable d'exclusion mutuelle distribuée avec priorité. In *9ème Conférence Française sur les Systèmes d'Exploitation (CFSE'13), Chapitre français de l'ACM-SIGOPS, GDR ARP*.

[12] Goscinski, A.M. (1990). Two Algorithms for Mutual Exclusion in Real-Time Distributed Computer Systems. *J. Parallel Distrib. Comput.*, *9*, 77-82.

[13] Kasami, T., & Suzuki, I. (1985). A Distributed Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.*, *3*, 344-349.

[14] Housni, A., & Trehel, M. (2001). Distributed mutual exclusion token-permission based by prioritized groups. In *Computer Systems and Applications, ACS/IEEE International Conference on. 2001* (pp. 253-259). IEEE.

[15] Agrawala, A.K., & Ricart, G. (1981). An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Commun. ACM*, *24*, 9-17.

[16] Raymond, K. (1989). A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Trans. Comput. Syst.*, *7*, 61-77.

[17] Mueller, F. (1998). Prioritized Token-Based Mutual Exclusion for Distributed Systems. *IPPS/SPDP*.

[18] Lejeune, J., Arantes, L., Sopena, J., & Sens, P. (2013, October). A prioritized distributed mutual exclusion algorithm balancing priority inversions and response time. In *Parallel Processing (ICPP), 2013 42nd International Conference on* (pp. 290-299). IEEE.

[19] Edmondson, J., Schmidt, D., & Gokhale, A. (2011). QoS-enabled distributed mutual exclusion in public clouds. *On the Move to Meaningful Internet Systems: OTM 2011*, 542-559.

[20] Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, *21*, 558-565.

[21] Lim, J., Chung, K. S., Chin, S. H., & Yu, H. C. (2012). A gossip-based mutual exclusion algorithm for cloud environments. *Advances in Grid and Pervasive Computing*, 31-45.