Reducing synchronization cost in distributed multi-resource allocation problem

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

Reducing synchronization cost in distributed multi-resources allocation problem

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

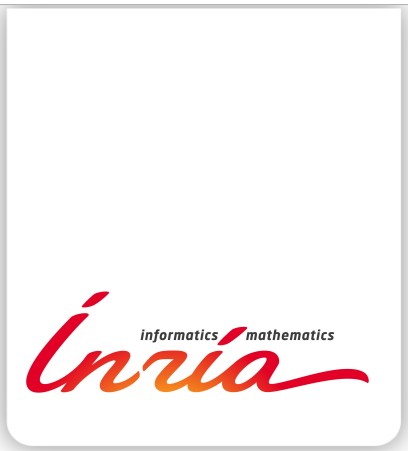
RESEARCH REPORT

N° 8689

February 2015

Project-Teams Regal and ASCOLA

ISSN 0249-6399 ISRN INRIA/RR--8689--FR+ENG



Reducing synchronization cost in distributed multi-resources allocation problem

Jonathan Lejeune, Luciana Arantes, Julien Sopena, Pierre Sens

Project-Teams Regal and ASCOLA

Research Report n° 8689 — February 2015 — 31 pages

Abstract: Generalized distributed mutual exclusion algorithms allow processes to concurrently access a set of shared resources. However, they must ensure an exclusive access to each resource. In order to avoid deadlocks, many of them are based on the strong assumption of a prior knowledge about conflicts between processes' requests. Some other approaches, which do not require such a knowledge, exploit broadcast mechanisms or a global lock, degrading message complexity and synchronization cost. We propose in this paper a new solution for shared resources allocation which reduces the communication between non-conflicting processes without a prior knowledge of processes conflicts. Performance evaluation results show that our solution improves resource use rate by a factor up to 20 compared to a global lock based algorithm.

Key-words: distributed algorithm, generalized mutual exclusion, multi-resources allocation, drinking philosophers, performance evaluation

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Réduire les coûts de synchronisation dans le problème de l'allocation distribuée de ressources

Résumé : Les algorithmes d'exclusion mutuelle généralisée permettent de gérer les accès concurrents des processus sur un ensemble de ressources partagées. Cependant, ils doivent assurer un accès exclusif à chaque ressource. Afin d'éviter les interblocages beaucoup de solutions reposent sur l'hypothèse forte d'une connaissance préalable des conflits entre les requêtes des processus. D'autres approches, qui ne requièrent pas une telle connaissance, utilisent un mécanisme de diffusion ou un verrou global, dégradant ainsi la complexité en messages et augmentant le coût de synchronisation. Nous proposons dans cet article un nouvel algorithme pour l'allocation de ressources partagées qui réduit les communications entre processus non conflictuels sans connaître à l'avance le graphe des conflits. Les résultats de nos évaluations de performances montrent que notre solution améliore le taux d'utilisation d'un facteur 1 à 20 comparé à un algorithme se basant sur un verrou global.

Mots-clés : algorithme distribué, exclusion mutuelle généralisée, allocation multi-ressources, cocktail des philosophes, évaluation de performances

1 Introduction

Processes in distributed and parallel applications require an exclusive access to one or more shared resources. In the case of a single shared resource, one of the standard distributed mutual exclusion algorithms (e.g. [12],[24],[28], [20], [17], [18]) is usually applied in order to ensure that at most one process uses the resource at any time (safety property) and that all requests are eventually satisfied (liveness property). The set of instructions of processes' code that access the shared resource is called the critical section (CS). However, most of distributed systems such as Clouds or Grids are composed of multiple resources and processes may ask access to several of them simultaneously. In such a problem, a generalization of the mutual exclusion one, a process can start executing the respective critical section, which concerns exclusive access to all requested resources, only after having acquired the right to use all of them. On the other hand, requests for accessing resources can cause conflicts if the sets of the asked resources are not disjoint. To solve such a problem, we can define a conflict graph whose vertices represent the processes and two of them are linked if the corresponding processes concurrently ask for the same resource. Consequently, a third property, called concurrency property, which ensures that two non-conflicting processes execute their critical section concurrently, has been introduced.

Nevertheless, exclusive access to each resource is not enough to ensure the liveness property since deadlock scenarios can take place. In the context of multiple resources allocation, such a problem can happen when two processes are waiting for the release of a resource owned by the other one. This multi-resource problem, also called AND-synchronization, has been introduced by Dijkstra [9] with the dining philosopher problem where processes require the same subset of resources all the time. Later, it was extended by Chandy-Misra [7] to the drinking philosopher problem where processes can require different subset of resources.

In the literature, we distinguish two families of algorithms which solve the multi-resource problem: incremental ([13, 27]) and simultaneous (e.g. [2, 5, 10, 14, 3]). In the first family, a total order is defined for the set of resources and processes must acquire them respecting such an order. In the second one, algorithms propose some mechanisms which allow them to acquire the set of resources atomically without entailing conflicts. On the one hand, many of the proposed solutions of the incremental family consider that the conflict graph is known a priori and does not change during the algorithm execution, which implies very strong assumptions about the application. On the other hand, some solutions of the simultaneous family do not require any knowledge about the conflict graph. However, in order to serialize the requests, these solutions have a huge synchronization cost which entails performance degradation of both resource use rate and average waiting time. Other solutions exploit one or several coordinators to order the requests and avoid, thus, deadlocks, but, since they are not fully distributed, they can generate some network contentions when the system load is high. Finally, some algorithms use broadcast mechanisms which render them not scalable in terms of message complexity.

In this paper, we propose a new decentralized approach for locking multiple resources in distributed systems. Our solution does not require the strong hypothesis of a priori knowledge about the conflict graph and does not need any global synchronization mechanism. Moreover, it dynamically re-orders resource requests in order to exploit as much as possible the potential parallelism of non-conflicting requests. Performance evaluation results confirm that our solution improves performance in terms of resources use rate and average request waiting time.

The rest of the paper is organized as follows. Section 2 discusses some existing distributed algorithms which solve the multi-resources allocation problem. A general outline of our proposal and its implementation are described in sections 3 and 4 respectively. Section 5 presents performance evaluation results of this implementation by comparing them with two existing solutions of the literature. Finally, Section 6 concludes the paper.

2 Related Work

The original mutual exclusion problem can be generalized with different aims:

- a shared resource which can be accessed concurrently in the same session (group mutual exclusion problem [11, 4, 1])
- several copies (or units) of the same critical resource (k-mutex problem) [19, 21, 26, 16, 8, 25, 6, 22]
- several types of resources (the multi-resource problem)

In this paper we will focus on the last approach and, in this section, we outline the main distributed multi-resources algorithms. They are divided into two families: incremental and simultaneous.

2.1 Incremental family

In this family, each process locks incrementally its required resources according to a total order defined over the global set of resources. The mutual exclusion to each resource can be ensured with a single-resource mutual exclusion algorithm. However, such a strategy may be ineffective if it presents a domino effect when waiting for available resources[1]. The latter affects the concurrency property and, therefore, may hugely degrade resources use rate.

In order to avoid the domino effect, Lynch [13] proposes to color a dual graph of the conflict graph. Then, it is possible to define a partial order over the resources set by defining a total order over the colors set. This partial order reduces the domino effect and improves the parallelism of non-conflicting requests.

Aiming at reducing the waiting time, Styer and Peterson [27] consider an arbitrary coloring (preferably optimized) which also supports request cancelation: a process can release a resource even if it has not use it yet. Such an approach dynamically breaks possible waiting chains.

2.2 Simultaneous family

In this family, resources are not ordered. Algorithms implement some internal mechanisms in order to avoid deadlocks and atomically lock the set of resources required by the process.

Chandy and Misra [7] have defined the drinking philosophers problem where processes (= philosophers) share a set of resources (= bottles). This problem is an extension of the dining philosophers problem where processes share forks. Contrarily to the latter, where a process always asks for the same subset of resources, i.e the same two forks, the drinking philosopher problem let a process to require a different subset of resources at each new request. The communication graph among processes corresponds to the conflict graph and has to be known in advance. Each process shares a bottle with each of its neighbors. By orienting the conflict graph we obtain a precedence graph. Note that if cycles are avoided in the precedence graph, deadlocks are impossible. It has been shown that the dinning philosophers problem respects this acyclicity but it is not the case for the drinking philosophers one. To overcome this problem, Chandy and Misra have applied dinning procedures in their algorithms: before acquiring a subset of bottles among its incident edges, a process firstly needs to acquire all the forks shared with its neighbors. Forks can be seen as auxiliary resources that serialize bottle requests in the system and are released when the process has acquired all the requesting bottles. Serialization of requests avoids cycles

[1] A process waits for some resources which are not in use but locked by other processes that wait for acquiring other resources before releasing the former.

in the precedence graph and, therefore deadlocks are avoided. On the other hand, the forks acquisition phase induces synchronization cost.

Ginat et al. [10] have replaced the dinning phase of the Chandy-Misra algorithm by logical clocks considering a complete conflict graph. When a process asks for its required resources, it timestamps the request with its local logical clock value and sends a message to each concerning neighbor. Upon receipt of a request, the associate shared bottle is sent immediately if the request timestamp value is smaller than the current clock value of the receiver. The association of a logical clock value and a total order over identifiers of processes defines a total order over requests which prevents deadlocks. However, message complexity becomes high whenever the conflict graph is unknown (equivalent to a complete graph) since the algorithm uses, in this case, a broadcast mechanism.

In [23], Rhee presents a request scheduler where each processes is a manager of a resource. Each manager locally keeps a queue that can be rescheduled according to new pending requests avoiding, therefore, deadlocks. This approach requires several dedicated managers which can become potential bottlenecks. Moreover, the coordination protocol responsible for avoiding deadlocks between managers and application processes is quite costly.

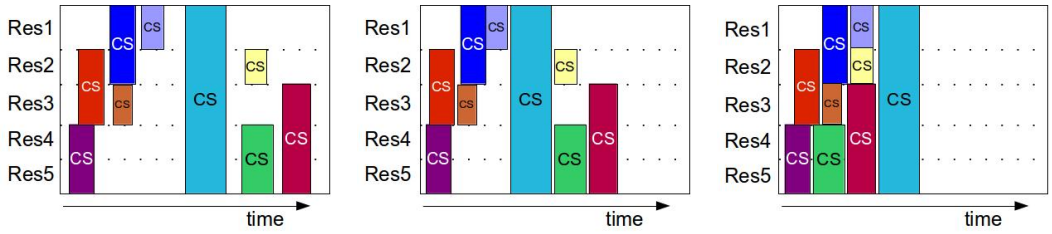
Maddi [14] proposed an algorithm which is based on a broadcast mechanism and each resource is represented by a single token. Each process request is timestamped with the local clock value of the process and broadcast to all other processes. Upon reception, the request is stored in a local queue of the receiver, ordered by the request timestamps. This algorithm can be seen as multiple instances of Susuki-Kasami mutual exclusion algorithm [28], presenting, thus, high messages complexity.

The Bouabdallah-Laforest token-based algorithm [5] is described in more details in this section because it is the closest one to our solution and, therefore, the performance of both algorithms will be evaluated and compared in section 5. A single *resource token* and a distributed queue are assigned to each resource. For having the right to access the resource, a process must acquire the associated *resource token*. Furthermore, before asking for a set of resources, the requester must firstly acquire a *control token*, which is unique in the system. A Naimi-Tréhel [17] mutual exclusion algorithm is responsible for handling this control token. This algorithm maintains a dynamic distributed logical tree such that the root of the tree is always the last process that will get the token among the current requesting ones. It also keeps a distributed queue of pending requests. The *control token* contains a vector with M entries (the total number of resources of the system) where each entry corresponds to either the *resource token* or the identifier of the latest requester of the resource in question. Thus, when a requesting process receives the *control token*, it acquires all the required resources already included in the *control token* and sends an INQUIRE message to the respective latest requester for each *resource token* which is not in the *control token*. We point out that the *control token* serializes requests, ensuring that a request will be registered atomically in the different distributed waiting queues. Hence, no cycle takes place among all distributed waiting queues. This algorithm presents a good message complexity, but the control token serialization mechanism can induce bottlenecks when the system has few conflicts, i.e., in a scenario where concurrency is potentially high.

3 General outline of our solution

3.1 Model and assumptions

We consider a distributed system consisting of a finite set Π of reliable N nodes, $\Pi = \{s_1, s_2, ...s_N\}$ and a set of M resources, $\mathcal{R} = \{r_1, r_2, ..., r_M\}$. The set Π is totally ordered by the order relation $\prec$ and $s_i \prec s_j$ iff $i < j$. There is one process per node. Hence, the words node, process, and site



(a) With global lock, static scheduling (b) Without global lock, static scheduling (c) Without global lock, dynamic scheduling

Figure 1: Illustration of the impact of our objectives on the resource use rate

are interchangeable. Nodes are assumed to be connected by means of reliable (neither message duplication nor message loss) and FIFO communication links. Processes do not share memory and communicate by sending and receiving messages. The communication graph is complete, *i.e.*, any node can communicate with any other one. A process can not request a new CS before its previous one has been satisfied. Therefore, there are at most N pending requests. We also assume no knowledge about the conflict graph.

3.2 Discussion

Similarly to our solution, simultaneous solutions found in the literature do not assume a prior knowledge of the conflict graph. Their control mechanisms totally order requests avoiding, thus, deadlocks. However, they may be inefficient since they induce communication between non conflicting processes which have no need to interact with each other.

Since it presents a logarithmic message complexity, on the one hand we consider that Bouabdallah-Laforest [5] is a very effective multi-resource algorithm. On the other hand, we point out two of its limitations which degrade the resource use rate:

- two non conflicting sites communicate with each other in order to exchange the *control token*, inducing additional cost in terms of synchronization;
- request scheduling is static: it depends only on the acquisition order of the *control token* by the requesting processes. Consequently, a new request is not able to preempt another one which obtained the control token before it, preventing, therefore, a dynamic scheduling that would increase resource usage rate.

Hence, our objective is twofold:

- not to use a global lock to serialize requests in order to avoid useless communication between non conflicting processes,
- to schedule requests dynamically.

Figure 1 shows as Gantt diagrams, the impact of our two objectives (lack of global lock and dynamic schedule) on the resource use rate when compared to Bouabdallah-Laforest's algorithm [5] in a system with five shared resources:

- the lack of global lock reduces the time between two successive conflicting critical sections (Figure 1(b)).

- the dynamic scheduling makes possible the granting of resources to processes in idle time periods (white spaces) where resources are not used (Figure 1(c)).

3.3 Suppression of global lock

We describe now the principles of our mechanism which ensures the serialization of requests without the use of a global lock.

3.3.1 Counter mechanism

The goal of the control token in Bouabdallah-Laforest's algorithm is to provide a unique scheduling order over the whole requesting waiting queues associated to resources. In order to remove this global lock, we have assigned one counter per resource. Each counter provides then a global request order for the resource to which it is related. Hence, there are M counters in the system that should be accessed exclusively, i.e., there is a token associated to each counter whose current value is kept in the token. Therefore, a requesting process should firstly obtain for each requested resource, the current value of the respective counter. Then, the token holders atomically increments the counters in order to ensure different values at each new request. Once a process has acquired all the required counter values, its request can be associated with a single vector of M integers in the set $\mathbb{N}^M$. Entries of the vector corresponding to non required resources are equal to zero. Consequently, every request is uniquely identified regardless of the time when it has been issued as well as the set of required resources. Then, a process can request its resources independently. Note that this counter mechanism and the exclusive access to a resource are independent: it is always possible to ask for the value of a resource counter while the resource in question is currently in use.

3.3.2 Total order requests

A request req_i issued by the site $s_i \in \Pi$ for a given resource is associated with two pieces of information: the identifier of the requesting process s_i and the respective associated vector $v_i \in \mathbb{N}^M$. Deadlocks are avoided if a total order over requests is defined. To this end, we firstly apply a partial order over the vector values by defining a function $\mathcal{A} : \mathbb{N}^M \rightarrow \mathbb{R}$ which transforms the values of a counter vector in a real value. Since such an approach guarantees just a partial order, we use the identifier of sites identifiers to totally order the requests. Therefore, we define this total order, denoted, $\lhd$ by $req_i \lhd req_j$ iff $\mathcal{A}(v_i) < \mathcal{A}(v_j) \vee (\mathcal{A}(v_i) = \mathcal{A}(v_j) \wedge s_i \prec s_j)$. Thus, if $\mathcal{A}$ returns the same real value for two requests' vector values, the identifiers of the corresponding requesting sites break the tie. Although this mechanism avoids deadlocks by ensuring that all requests can be differentiated, the satisfaction of the algorithm's liveness property depends of the choice of a suitable function $\mathcal{A}$. In other words, $\mathcal{A}$ should avoid starvation by ensuring that every request will have, in a finite time, the smallest real value among all pending requests according to the order $\lhd$. The function $\mathcal{A}$ is a parameter of the algorithm and, basically, defines the scheduling resource policy.

3.4 Dynamic scheduling

The introduction of a *loan mechanism* into the algorithm could improve the resource use rate. Requested resources are acquired progressively but are actually used once the process got the right to access all of them. Thus, some or even many resources are locked by processes which are not able to use them. Such a behavior reduces the overall resources use rate. The idea of the dynamic scheduling is then to restrict as much as possible the right to access a resource only

to critical section execution, i.e., offer the possibility to lend the right to access a resource to another process. However, for sake of the liveness property, the loan mechanism has to ensure that eventually a site get back the right, previously acquired, to access the resource. In other words, it must avoid starvation and deadlocks.

3.4.1 Starvation avoidance

Since the lending of the right to access a resource will not necessarily ensure that a borrower process will own all the set of resources it has required, starvation problems may occur. To overcome this problem, we propose a simple mechanism by restricting the loan to only one process at a time. Thus, we guarantee that the lender process will obtain again all the lent resource access rights in a finite time since the critical section time of the borrower is bounded by assumption.

3.4.2 Deadlock avoidance

Resources borrowed from multiple processes can lead to cycles in the different resources waiting queues and, therefore, to deadlocks. To avoid it, we propose to restrict the loan to a single site provided that the lender process owns all the resource access rights which are missing to the borrower process. Consequently, upon reception of the rights, the latter can immediately execute its critical section.

4 Description of the implementation

In this section we describe the implementation of our algorithm. This description references lines of the pseudo-code given in annex A.

Each resource is associated with a unique token which contains the resource counter. The process that holds the token is the only one which has the right to access and increment the counter value ensuring, therefore, an exclusive access.

Each token is controlled by an instance of a simplified version of the Mueller algorithm [15]. The latter is a prioritized distributed token-based mutual exclusion algorithm that logically organizes the processes in a dynamic tree topology where the root of the tree is the token holder of the corresponding resource. Every token also keeps the queue of pending requests related to the resource it controls.

For instance, in Figure 3, we consider 3 processes (s_1, s_2, and s_3) and 2 resources (r_{red} and r_{blue}). Figure 3(a) shows the initial tree topologies related to each of the resources where s_1 and s_3 hold the token associated with r_{red} and r_{blue} respectively. Notice that s_2 has 2 fathers, s_1 (red tree) and s_3 (blue tree), while s_1 (respectively, s_3) has a single father: s_3 (blue tree) and s_2 (red tree), respectively. In Figure 3(c), the topologies of the trees have changed since s_2 got the two tokens and it is, therefore, the root of both trees.

We should point out that the choice of a prioritized algorithm as Mueller's one makes possible the rescheduling of pending requests of a given resource queue whenever a request, with a higher priority according to the $\triangleleft$ order, regarding this resource, is received.

4.1 Process states

A process can be in one of the following four states:

- *Idle*: the process is not requesting any resource;

- $waitS$: the process is waiting for the requested counter values;
- $waitCS$: the process is waiting for the right to access all the requested resources.
- $inCS$: the process is using the requested resources (in critical section).

Figure 2 shows the global machine states of a process.

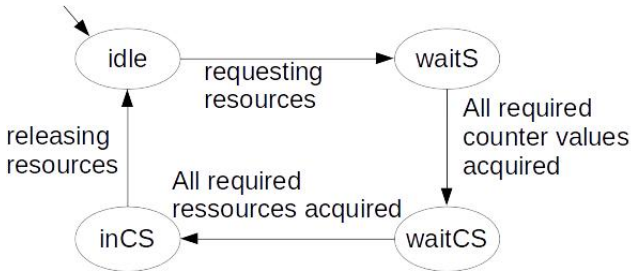


Figure 2: Machine state of a process

4.2 Messages

We define five types of message where their pseudo-code is given Figure 8:

- $ReqCnt(r, s_{init}, id)$: sent by s_{init} for the critical section request id when requesting the current value of the counter associated with r. (line 1)
- $Counter(r, val)$: sent by the token holder associated with the resource r as a reply to a $ReqCnt$ request. It contains the value val of the r counter that the token holder has assigned to the request in question. (line 19)
- $ReqRes(r, s_{init}, id, mark)$: sent whenever s_{init} requests the right to access resource r for the critical section request id. The request is tagged with $mark$, the value returned by function $\mathcal{A}$. (line 6)
- $ReqLoan(r, s_{init}, id, mark, missingRes)$: sent by s_{init} whenever it requests a loan of resource r tagged with $mark$ value (return of function $\mathcal{A}$) for the critical section request id. The message also contains the set of missing resources $missingRes$ which s_{init} is waiting for. (line 12)
- $Token(r, counter, lastReqC, lastCS, wQueue, wLoan, s_{lender})$: The token message related to resource r which contains the latest value of the associated counter and the waiting queue $wQueue$ of pending requests in increasing order of the respective $marks$. The queue $wLoan$ contains the set of pending requested loans concerning r and, if the latter is currently lent, s_{lender} corresponds to the identifier of the lender site. Array $lastReqC$ maintains for each site the id of the last $ReqCnt$ received and processed by the token owner. Array $lastCS$ maintains for each site the id of the id of the last critical section request which has been satisfied. (line 23)

We can classify these five message types in two families :

- **Request messages** which are forwarded from the sender s_{init} till the token holder along the corresponding tree structure: *ReqCnt*, *ReqRes*, and *ReqLoan* types.
- **Response messages** which are sent directly to the requester: *Counter* and *Token* types.

4.2.1 Problems due to request messages

Note that the graph representing the tree topology dynamically changes during the forwarding of a request message. This leads two problems:

- a cycle message may appear and consequently a message may be forwarded indefinitely. For avoiding this problem, we have included in every request message the identifiers of the nodes already visited by the request. Thus, the forwarding is stopped if it has reached the token holder or if the father belongs to the visited nodes set (lines 167 and 185).
- a request message can never be took into account if the token is transiting to a node already visited by this message. This must introduces starvation. For avoiding this problem, each site keeps a local history of received request messages (local variable *pendingReq*, line 42). In order to discard obsolete messages (already took into account by the token owner), we introduced a timestamp mechanism by comparing the id. included in the request and the two arrays $lastReqC$ and $lastCS$. Thus, a request message for a resource r from the site s_{init} is obsolete (lines 165 and 147) if $id \leq lastCS[s_{init}]$ in the token r. In the same principle, to avoid to send more than once a counter value for the same counter request id for the same resource r, a counter request ($ReqCnt$) is obsolete if $id \leq lastReqC[s_{init}]$ in the token r.

4.2.2 Aggregation mechanism

It is worth pointing out that in order to reduce the number of messages in our implementation, whenever possible, messages with same type related to the same resource and addressed to the same site can be combined into a single message of this type. Consequently, the message receipt concerns a set of resources. We define the function **buffer(Site** s_{dest}**, Type** t**,** *data***)** which stores temporarily the message of type t to site s_{dest}. We define two functions which send to their corresponding recipient, stored messages by the **buffer** function calls. Thus we define:

- **SendBufReq(***visited* **: set of sites)** for request messages. The set *visited* contains all sites visited by the message (see section 4.2.1).
- **SendBuf()** for response messages.

4.3 Local variables

Each process maintains the following local variables (figure 9):

- *state*: the current state of the current process
- *tokDir*: array of M sites, where each entry indicates the father in the tree of the corresponding resource (nil value if the process is the root site of the tree).
- $MyVector$: the vector of counters of the current request
- $lastTok$: array of M token structures which stores locally for each $r \in \mathcal{R}$ the last snapshot of the corresponding token.

- *TRequired*: set of required resources of the current request.
- *TOwned*: set of owned tokens.
- *CntNeeded*: set of required resources where the process has not yet received the corresponding counter value.
- *curId*: id of the current request. This value changes at each new CS request (line 70).
- *pendingReq* : array of M sets of request messages (*ReqCnt*, *ReqRes* and *ReqRes*) which have potentially not yet been processed by the corresponding token owner.
- *TLent*: set of lent resources.
- *loanAsked*: boolean set to true if the current site has sent a *ReqLoan* message for the current request, false otherwise.

4.4 The counter mechanism

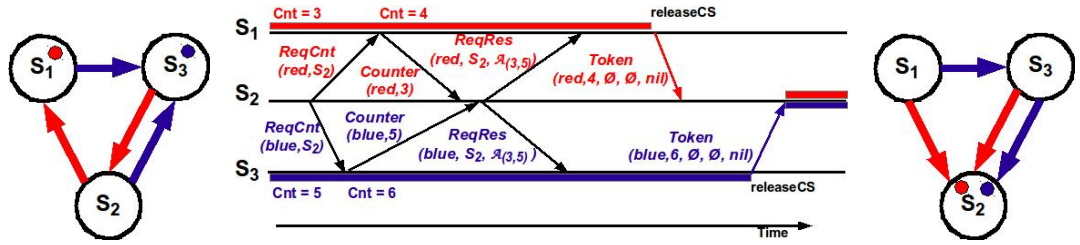
When process s_i wishes to access a set of resources (*Request_CS* procedure call, line 68), it changes its state from *Idle* to *waitS* (line 72). Then, it has to get the current value of the counters associated with all these resources (lines 73 to 80). If s_i already owns the tokens associated with some of the required resources, it reserves to its request the current value of the respective counters and increases them (lines 74 to 76). We should remind that only the token holder (s_i in this case for the tokens it holds) has the right to increase the counters associated with the resources in question. Otherwise, for each missing counter value, it sends a *ReqCnt* message to one of its fathers, i.e., the one which is its father in the corresponding resource tree (lines 77 to 79). It also registers in its local *CntNeeded* set variable the id. of the missing resources (line 78). Process s_i then waits to receive the missing counter values.

When s_j receives the request *ReqCnt* message for resource r from s_i, if it does not hold the token related to r, it forwards the message to its father which belongs to the r tree (line 185 to 187). However, if s_j is the token holder, but does not require r, it directly sends the token to s_i (lines 170 and 171). Otherwise, s_j keeps the token and sends a *Counter* message, which contains the current value of the counter to s_i and then, increments the counter (lines 172 to 175).

Upon reception of a *Counter* message for the resource r (line 255), s_i removes r from its *CntNeeded* set (line 259). When *CntNeeded* becomes empty (call to procedure **processCntNeedeEmpty()**, line 108), s_i has obtained counter values for all its required resources. Note that these values are uniquely assigned to the requests of s_i. It then changes its state to *waitCS* (line 111) and for each of these resources which it does not hold yet, it sends a *ReqRes* message to the respective father (lines 112 to 115).

Similarly to the *ReqCnt* message, when receiving a *ReqRes* for a resource r, process s_j forwards the message to its father if it does not hold the token associated with r (line 185 to 187). If s_j holds the token and does not require r or is in the *waitS* state, it sends the token directly to s_i (lines 170 and 171). Otherwise, s_i and s_j are in conflict and it is necessary to take a decision about which of them has the right to hold the resource r. If s_j is in critical section (*inCS* state) or if the priority of its request is higher than the s_i's request ($req_j \triangleleft req_i$), it keeps the right (lines 183 and 184). In this case, s_i's request is registered in the r token queue (*wQueue*). Otherwise, s_j has to grant the right to access r to s_i. To this end, it registers its own request in the r token queue (*wQueue*) and sends the token directly to s_i, i.e., a *Token* message (lines 179 to 181).

When s_i receives a *Token* message related to r, first it makes two set of updates (**processUpdate()** procedure call, line 133):



(a) Initial state (b) Execution (c) Final state

Figure 3: Execution example

(1) it includes r in its set of owned tokens (line 137). If r belongs to $CntNeeded$, i.e., s_i has not yet received all the counter values required of the current CS request, it registers the current value of the token counter in the vector, increments the counter and removes r from $CntNeeded$ (lines 139 to 142)

(2) s_i takes into account pending messages of the local history for the concerned resource ($pendingReq[r]$): it replies by a $Counter$ message to each site that has issued a $ReqCnt$ message (lines 149 to 152) and adds in $wQueue$ (respectively $wLoan$) of the token information related to $ReqRes$ (resp. $ReqLoan$) messages (lines 153 to 158).

Then, site s_i can enter in critical section ($inCS$ state) if it owns the right to access all the requested resources (lines 213 to 215). If it is not the case, it can change its state to $waitCS$ provided its $CntNeeded$ set is empty (i.e., s_i got all the asked counter values). In this case, s_i sends $ReqRes$ messages for each missing resources (line 225). Because of the second set of updates, site s_i has to ensure that its request has the highest priority according to the $\triangleleft$ order (lines 226 to 238). If it is not the case, the token is granted to the site having the highest priority. Site s_i is now able to processes loan request stored in $wLoan$ concerning the other token that it keeps (lines 241 to 247). Finally, s_i can initiate a loan request (lines 249 to 252), if necessary (see section 4.5).

When the process exits the critical section ($Release\_CS$ procedure call, line 85), it dequeues the first element of waiting queue of all owned resource tokens and sends to their next holder (or potentially the lender site) the associated token. Finally s_i's state becomes $Idle$.

Let's take up the example of Figure 3 with the 3 processes (s_1, s_2 and s_3) and the 2 resources (r_{red} and r_{blue}), where the initial configuration is given in Figure 3(a), that we have previously described. Processes s_1 and s_3 are in critical section accessing r_{red} and r_{blue} respectively. Figure 3(b) shows the messages that processes exchange when s_2 requires both resources. First, s_2 sends to each of its fathers, s_1 (red tree) and s_3 (blue tree), a $ReqCnt$ request in order to obtain the associated current counter values. When s_2 has received the two requested counter values, it sends $ReqRes$ messages along the trees asking for respective resources. Upon exiting the critical sections s_1 and s_3 respectively send r_{red} token and r_{blue} token to s_2, which can thus enter the critical section once it received both tokens. The final configuration of the logical trees is shown in Figure 3(c).

4.5 The loan mechanism

The execution of a loan depends on some conditions related to both the lender and the borrower sites:

- Upon reception of a token, a process s_i can request a loan provided it is in the *waitCS* state (i.e., it got all the needed counter values), the number of missing resources is smaller or equal to a given threshold and if *loanAsked* is false (line 249). If it is the case, s_i sends a *ReqLoan* message to the respective father of the missing resources trees (lines 251 and 252). Similarly to a *ReqRes* message, a *ReqLoan* message for a resource is forwarded till the token holder associated with this resource.

- When receiving a *ReqLoan* message for resource r (lines 169 and 247), the token holder s_j calls the procedure **processReqLoan** (line 190). First it checks if the loan is possible. All required tokens in the message (*missingRes* set) can be lent if the following conditions are met (lines 119 to 125):

  - s_j owns all the requested resources;
  - none of the resources owned by s_j is a loan;
  - s_j has not lent resources to another site;
  - s_j is not in critical section;
  - s_i's request has a higher priority than s_j's request if the both have sent a loan request.

If the loan is feasible, the tokens associated with the resources are sent to s_i with s_{lender} equals to s_j (lines 198 to 202). Otherwise, if s_j does not require the resource of the request or is in *waitS* state, it sends the token directly to the borrower site s_i (line 205). Otherwise the loan request is included in the *wLoan* of the corresponding token to be potentially satisfied later upon receipt of new tokens (line 207).

When s_i receives borrowed tokens and if it does not enter in critical section (e.g. if it has yield other tokens for higher priority requests in the meantime), then the loan request has failed. Consequently, the loan request is canceled and s_i immediately returns borrowed tokens to s_{lender} (lines 217 to 223). This avoids an inconsistent state where a site owns borrowed and unused tokens.

Finally, when exiting the critical section, s_i sends back these tokens directly to s_j (line 98).

4.6 Optimizations

4.6.1 Synchronisation cost reduction of single resource requests

It is possible to reduce the synchronization cost of requests requiring a single resource by directly changing the state of the requester from *Idle* to *waitCS*. Since such requests require only one counter, stored in the token, the root site of the corresponding tree is able to apply $\mathcal{A}$ and then consider the *ReqCnt* message as a *ReqRes* message. Hence, such an optimization reduces messages exchanges.

4.6.2 Reduction of *ReqRes* messages

Once a process s_i gets all the requested resource counter values, it sends, for each of these resources, a *ReqRes* message that will travel along the corresponding tree till the token holder (root site). The number of these forward messages can be reduced by:

- shortcuting the path between the requesting site s_i and the root site s_j: upon reception of a *Counter* message from s_j, s_i sets its father pointer to s_j since it is the last token owner from the viewpoint of s_i (line 260).

- stopping message forwarding before the message reaches the root site. When receiving a $ReqRes$ message for a resource r, a process s_j does not forward the message if (1) it is in the $waitCS$ state, also requires r and its request has a higher precedence than s_i's request or (2) s_j has lent the token. If one of these two conditions are met, s_j knows that it will get the token corresponding to r before s_i. The request of s_i will eventually be stored in the waiting queue $wQueue$ of the token.

5 Performance Evaluation

In this section, we present some evaluation results comparing our solution with two other algorithms:

- An algorithm, which we have denoted **incremental algorithm** which uses M instances of the Naimi-Tréhel algorithm [18], one of the most efficient mutual exclusion algorithm thanks to its messages complexity comprised between $\mathcal{O}(LogN)$ and $O(1)$
- The **Bouabdallah-Laforest** algorithm [5] (see Section 2).

In order to show the impact of the loan mechanism, we consider two versions of our algorithm named **Without loan** and **With loan** which respectively disable and enable the loan mechanism. In the latter, a site asks for a loan when it has just one missing requesting resource.

We are interested in evaluating the following two metrics: (1) the resource use rate and (2) the waiting time to have the right to use all the requested resources, i.e., enter the critical section.

As previously explained, our algorithm requires a function $\mathcal{A}$ as input. For performance evaluation, our chosen function $\mathcal{A}$ computes the average of non null values of the counter vector. This function avoids starvation because counter values increase at each new issued request which implies that the minimum value returned by $\mathcal{A}$ increases at each new request. Thus, the liveness property is ensured. We should emphasize that the advantage of this approach lies in the fact that starvation is avoided only by calling the function and not inducing any additional communication cost.

5.1 Experimental testbed and configuration

The experiments were conducted on a 32-nodes cluster with one process per node. Therefore, the side effect due to the network is limited since there is just one process per network card. Each node has two 2.4GHz Xeon processors and 32GB of RAM, running Linux 2.6. Nodes are linked by a 10 Gbit/s Ethernet switch. The algorithms were implemented using C++ and OpenMPI. An experiment is characterized by:

- N: number of processes (32 in our experiments).
- M: number of total resources in the system (80 in our experiments).
- α: time to execute the critical section (CS) (it varies from 5 ms to 35 ms).
- β: mean time interval between the release of the CS by a node and the next new request issued by this same node.
- γ: network latency to send a message between two nodes (around 0,6 ms for our experiments).

- ρ: the ratio $\beta/(\alpha+\gamma)$, which expresses the frequency with which the critical section is requested. The value of this parameter is inversely proportional to the load: a low value implies a high request load and vice-versa.

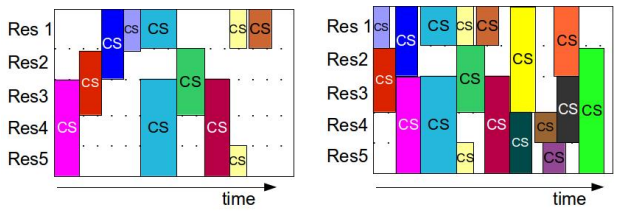
- ϕ: the maximum number of resources that a site can ask in a single request which ranges for 1 and M. The greater the value of this parameter, the lower the potential parallelism of the application and thus, the higher the probability to have conflicting requests.

At each new request, a process chooses x resources. The critical section time of the request depends on the value of x: the greater its value, the higher the probability of a long critical section time since a request requiring a lot of resources is more likely to have a longer critical section execution time.

For each metric, we show performance results corresponding to both medium and high load scenarios.

5.2 Resource use rate

This metric expresses the percentage of time that resources are in use (e.g., 100 % means that all resources are in use during the whole experiment). It can be seen as the percentage of colored area in the diagrams of Figure 4. We can observe that resources are used more effectively in the example of execution of Figure 4(b) than in the example of Figure 4(a), i.e., the former presents fewer white areas.



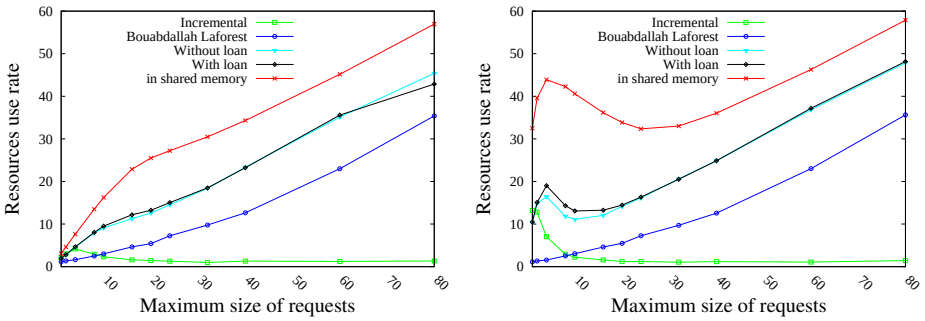
(a) Inefficient execution (b) Efficient execution

Figure 4: Illustration of the metric of resource use rate

By varying ϕ, we show in Figure 5 the impact of the number of asked resources within a request, denoted *request size*, on the resource use rate in the case of medium (figure 5(a)) and high (figure 5(b)) loads. The request size x may be different for each new request and it is chosen according to a uniform random law from 1 to ϕ.

In addition to the considered algorithms, we have included in both figures a fifth curve which represents a distributed scheduling algorithm executed on a single shared-memory machine with a global waiting queue and no network communication. The aim of such a curve is the evaluation of the synchronization cost of the different algorithms since the former is a resource scheduling algorithm without any synchronization.

Overall, in both figures, whenever ϕ increases, the resource use rate increases too. When the request size is minimal, the maximal number resources in use is equal to the number of process N which is smaller than the number of the total resources M. On the other hand, when the average request size increases, each critical section execution concerns a larger number of resources which, therefore, increases the use rate.



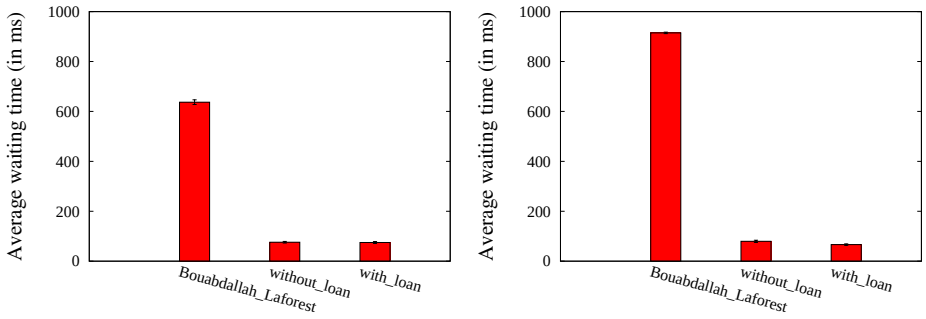
(a) Medium load (b) High load

Figure 5: Impact of request size over resource use rate

Note that in high load scenario (Figure 5(b)) the shape of the curve of the scheduling algorithm without synchronization firstly increases, then decreases till a threshold value at $\phi = 20$, and finally increases again. The curve has such a shape due to a threshold effect. In the first rise of the curve, the probability of having conflicts is small compared to both the request size and the difference between N and M. After $\phi = 4$, the number of conflicts starts to increase and the drop that follows is a consequence of the serialization of conflicting requests. Finally, when ϕ is greater than 20, the probability of having requests conflicts is maximum but each critical section access requires a lot of resources which increases the global use rate. Therefore, the subsequent rise of the curve is not caused by the increase of non-conflicting requests concurrency, but by the increase of requests' size.

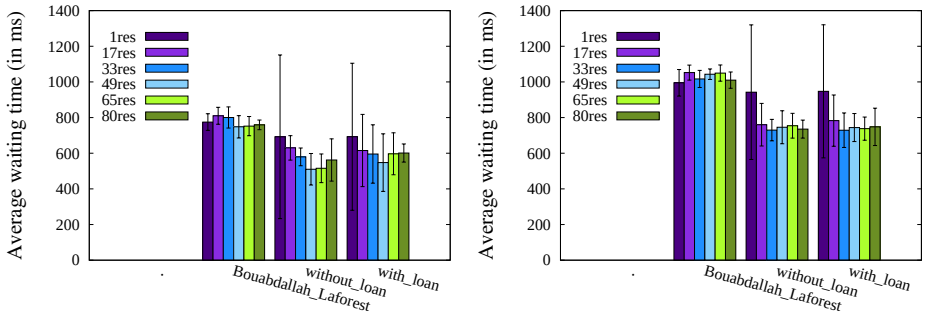
We should also point out that, when the average request size increases, the shapes of the resource use rate curves of the different algorithms are not the same. For the **incremental algorithm**, the resource use rate decreases and stabilizes since this algorithm does not benefit from the increase in the request size due to the domino effect (see 2.2).The resource use rate of the **Bouabdallah-Laforest** algorithm increases regularly. Although this algorithm is very disadvantaged by the global lock bottleneck whenever there are few conflicts (especially in high load), its use rate increases with the average request size. We observe that in this algorithm the resource use rate increases faster because it can take advantage of concurrent requests. However, it is not as much effective as our algorithms: independently of the request size, the latter present a higher resource use rate than the former, whose performance is affected by the bottleneck of its control token as well as its static scheduling approach. Notice that, depending on the request sizes, our algorithms have resource use rate values from 0.4 to 20 times higher than Bouabdallah-Laforest algorithm.

The curves related to the resource use rate of **our two algorithms** have the same shape than the one of the scheduling without synchronization. When the loan mechanism is enabled, the respective algorithm presents a higher resource use rate in high load scenario when the request size lies between 4 and 16 (improvement of up to 15%). Such a behavior shows that the loan mechanism is effective in reducing the negative effect of conflicts induced by medium requests and does not degrade performance when request size is big.



(a) Medium load (b) High load

Figure 6: Average waiting time ($\phi = 4$)



(a) Medium load (b) High load

Figure 7: Average waiting time to get a given number of resources ($\phi = 80$)

5.3 Average waiting time

In this section we study the average waiting time of a request which corresponds to the interval from the time the request was issued till the time when the requesting process got the right to access the resources whose identifiers are in the request.

For both high and medium loads, Figures 6 and 7 respectively show the average waiting time for processes to enter in critical section, considering a small ($\phi = 4$) and the highest ($\phi = 80$) maximum request size. In Figure 7 we detail the waiting time of different request sizes. We have not included in the figures the performance of the incremental algorithm because it is strongly disadvantaged by the domino effect: the average waiting time was too high compared to the experiment duration.

We can note in Figures 6(a) and 6(b) that our algorithms have a lower average waiting time than the Bouabdallah-Laforest algorithm when request size is small (around 11 times lower in high load and 8 times lower in medium load). Such a behavior confirms that our algorithms benefit from its lower synchronization cost. We also observe an improvement of 20% when the loan mechanism is activated in the high load scenario which is consistent with the previous figures related to resource use rate.

On the other hand, contrarily to our algorithms, both the waiting time and the standard

deviation of Bouabdallah-Laforest algorithm do not vary much when request size varies, as shown in Figures 7(a) and 7(b). Although our algorithm is the most efficient, its scheduling penalizes requests of small size. We can observe in the same figures that the average waiting time of small requests is the highest one as well as the respective standard deviation. Indeed, due to our chosen scheduling policy, i.e., our function $\mathcal{A}$, the access order of a single resource request depends on the value of the corresponding counter. In other words, the vector value average returned by the function concerns, in this case, just one counter value which is increased according to the frequency with each the associated resource is required: a highly requested resource will have a higher counter value when compared to other ones which are less requested.

6 Conclusion and future work

We have presented in this paper a new distributed algorithm to exclusively allocate a set of different resources. It does not require a prior knowledge about the conflicts graph and reduces communication between non conflicting processes since it replaces a global lock by a counter mechanism. The totally order of requests can be ensured with the definition of a function $\mathcal{A}$, given as input parameter of the algorithm. Performance evaluation results confirm that the counter mechanism improves the resource use rate and reduces the average waiting time. However, it can not completely avoid the domino-effect which increases the waiting time of pending requests. To overcome this drawback, we have include in the algorithm a loan mechanism that dynamically reschedules pending requests, reducing the probability that the domino-effect takes place.

Since our solution limits communication between non conflicting processes, it would be interesting to evaluate our algorithm on a hierarchical physical topology such as Clouds. Indeed, the lack of global lock of our algorithm would avoid useless communication between two distant geographic sites reducing, therefore, requests waiting time when compared to other control token based multi-resources algorithms. Performance results show that initiating a loan request when a process misses just one resource (threshold =1) improves use rate in scenarios with medium size requests. Thus, it would be interesting to evaluate the impact of this threshold on other metrics.

7 Acknowledgment

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

References

[1] Aoxueluo, Weigang Wu, Jiannong Cao, and Michel Raynal. A generalized mutual exclusion problem and its algorithm. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 300–309, 2013.

[2] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *FoCS, 1990. Proceedings., 31st Annual Symposium on*, pages 65 –74 vol.1, oct 1990.

[3] Valmir C. Barbosa, Mario R. F. Benevides, and Ayru L. Oliveira Filho. A priority dynamics for generalized drinking philosophers. *Inf. Process. Lett.*, 79(4):189–195, 2001.

[4] Vibhor Bhatt and Chien-Chung Huang. Group mutual exclusion in $O(\log\ n)$ RMR. In *PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 45–54, 2010.

[5] Abdelmadjid Bouabdallah and Christian Laforest. A distributed token/based algorithm for the dynamic resource allocation problem. *Operating Systems Review*, 34(3):60–68, 2000.

[6] Shailaja Bulgannawar and Nitin H. Vaidya. A distributed k-mutual exclusion algorithm. In *ICDCS*, pages 153–160, 1995.

[7] K. Mani Chandy and Jayadev Misra. The drinking philosopher's problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[8] Naomi S. DeMent and Pradip K. Srimani. A new algorithm for k mutual exclusions in distributed systems. *Journal of Systems and Software*, 26(2):179–191, 1994.

[9] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[10] David Ginat, A. Udaya Shankar, and Ashok K. Agrawala. An efficient solution to the drinking philosophers problem and its extension. In *WDAG (Disc)*, pages 83–93, 1989.

[11] Yuh-Jzer Joung. Asynchronous group mutual exclusion (extended abstract). In *PODC*, pages 51–60, 1998.

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

[13] Nancy A. Lynch. Upper bounds for static resource allocation in a distributed system. *J. Comput. Syst. Sci.*, 23(2):254–278, 1981.

[14] Aomar Maddi. Token based solutions to m resources allocation problem. In *SAC*, pages 340–344, 1997.

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

[16] Mohamed Naimi. Distributed algorithm for k-entries to critical section based on the directed graphs. *SIGOPS Oper. Syst. Rev.*, 27(4):67–75, October 1993.

[17] Mohamed Naimi and Michel Trehel. How to detect a failure and regenerate the token in the log(n) distributed algorithm for mutual exclusion. In *WDAG*, pages 155–166, 1987.

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

[19] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30(4):189–193, 1989.

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

[21] Michel Raynal. A distributed solution to the k-out of-m resources allocation problem. In *ICCI*, pages 599–609, 1991.

[22] Vijay Anand Reddy, Prateek Mittal, and Indranil Gupta. Fair k mutual exclusion algorithm for peer to peer systems. In *ICDCS*, pages 655–662, 2008.

[23] Injong Rhee. A modular algorithm for resource allocation. *Distributed Computing*, 11(3):157–168, 1998.

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

[25] R. Satyanarayanan and C. R. Muthukrishnan. Multiple instance resource allocation in distributed computing systems. *J. Parallel Distrib. Comput.*, 23(1):94–100, 1994.

[26] Pradip K. Srimani and Rachamallu L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 41(1):51–57, 1992.

[27] Eugene Styer and Gary L. Peterson. Improved algorithms for distributed resource allocation. In *PODC*, pages 105–116, 1988.

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

Annex A: Pseudo-code of the algorithm

```
1  Type ReqCnt :
2  begin
3      r : resource;
4      s_init : site;
5      id : integer ;

6  Type ReqRes :
7  begin
8      r : resource;
9      s_init : site;
10     id : integer ;
11     mark : float;

12 Type ReqLoan :
13 begin
14     r : resource;
15     s_init : site;
16     id : integer ;
17     mark : float;
18     missingRes : set of resources;

19 Type Counter :
20 begin
21     r : resource;
22     val : integer ;

23 Type Token :
24 begin
25     r : resource;
26     counter : integer;
27     lastReqC : array of N integers;
28     lastCS : array of N integers;
29     wQueue : sorted list of ReqRes ;
30     wLoan : sorted list of ReqLoan ;
31     s_lender : site or nil;
```

Figure 8: Data structures carried by messages

```
32 Local variables :
33 begin
34     state ∈ {Idle, waitS, waitCS, inCS};
35     tokDir : array of M sites;
36     MyVector : array of M integers;
37     lastTok : array of M Token;
38     TRequired : set of resources;
39     TOwned : set of resources;
40     CntNeeded : set of resources;
41     curId : integer ;
42     pendingReq : array of M sets of request messages;
43     TLent : set of resources;
44     loanAsked : boolean ;

45 Initialization
46 begin
47     if self = elected_node then
48         tokDir[r] ← nil ∀r ∈ R;
49         TOwned ← R ;
50     else
51         tokDir[r] ← elected_node ∀r ∈ R;
52         TOwned ← ∅ ;
53     TRequired ← ∅;
54     CntNeeded ← ∅;
55     TLent ← ∅;
56     state ← Idle;
57     curId ← 0;
58     loanAsked ← false;
59     foreach resource r ∈ R do
60         MyVector[r] ← 0;
61         lastTok[r].r ← r;
62         lastTok[r].counter ← 1;
63         lastTok[r].lastReqC[s] ← 0 ∀s ∈ Π;
64         lastTok[r].lastCS[s] ← 0 ∀s ∈ Π;
65         lastTok[r].wQueue ← ∅;
66         lastTok[r].wloan ← ∅;
67         pendingReq[r] ← ∅;
```

Figure 9: Local variables and initialization

```
68  Request_CS(D : set of resources)
69  begin
70      curId ← curId + 1;
71      TRequired ← D;
72      state ← waitS;
73      foreach resource r ∈ TRequired do
74          if tokDir[r] = nil then
75              MyVector[r] ← lastTok[r].counter;
76              lastTok[r].counter ←
                lastTok[r].counter + 1;
77          else
78              CntNeeded ← CntNeeded ∪ {r};
79              buffer(tokDir[r],ReqCnt,
                < self, r, curId, nil > );
80      SendBufReq({self});
81      if TRequired ⊈ TOwned then
82          wait(TRequired ⊆ TOwned);
83      state ← inCS;
84      /* CRITICAL SECTION                    */
```

```
85  Release_CS
86  begin
87      state ← Idle;
88      loanAsked ← false;
89      foreach ressource r ∈ TRequired do
90          lastTok[r].lastCS[self] ← curId;
91          site s_lender ← lastTok[r].lender;
92          if lastTok[r].wQueue ≠ ∅ and s_lender = nil then
93              < s, r', id, res, m >←
                dequeue(lastTok[r].wQueue);
94              SendToken(s,r);
95          else if s_lender ≠ nil then
96              lastTok[r].wQueue ← lastTok[r].wQueue − {<
                s_lender, r, _, res, _, >};
97              lastTok[r].lender ← nil ;
98              SendToken(s_lender,r);
99      TRequired ← ∅;
100     MyVector[r] ← 0 ∀r ∈ R;
101     SendBuf();
```

Figure 10: Request and release CS procedures

```
102 SendToken(s_dest: site, r : resource)
103 begin
104     /* precond :  r∈TOwned                 */
105     buffer(s_dest,Token, lastTok[r] );
106     tokDir[r] ← s_dest;
107     TOwned ← TOwned − {r};

108 processCntNeededEmpty()
109 begin
110     /* precond :  state = waitS ∧ CntNeeded = ∅   */
111     state ← waitCS;
112     foreach resource r ∈ TRequired do
113         ReqRes
            myReq ←< self, r, curId, A(MyVector) >;
114         if r ∉ TOwned then
115             buffer(tokDir[r],Req, myReq );
116     SendBufReq({self});

117 canLend(req : ReqLoan) with boolean result
118 begin
119     if req.missingRes ⊆ Towned
120     and ∄r ∈ Towned, lastTok[r].lender ≠ nil
121     and TLent = ∅
122     and state ≠ inCS then
123         if state = waitCS then
124             ReqRes
                myReq ←< self, r, curId, A(MyVector) >;
125             if loanAsked = false or req ◁ myReq then
126                 return true;
127             else
128                 return false;
129         else
130             return true;
131     else
132         return false;
```

```
133 processUpdate(t : Tok)
134 begin
135     resource r ← t.r;
136     lastTok[r] ← t;
137     TOwned ← TOwned ∪ {r};
138     tokDir[t.r] ← nil;
139     if r ∈ CntNeeded then
140         CntNeeded ← CntNeeded − {r};
141         MyVector[r] ← lastTok[r].counter;
142         lastTok[r].counter ← lastTok[r].counter + 1;
143     if r ∈ TLent then
144         TLent ← TLent − {r};
145     foreach Req req ∈ pendingReq[r] do
146         site s_i ← req.s_init;
147         if req is obsolete then
148             continue;
149         if req is a ReqCnt then
150             lastTok[r].lastReqC[s_i] ← req.id;
151             buffer(s_i,Counter,< r, lastTok[r].counter > );
152             lastTok[r].counter ← lastTok[r].counter + 1;
153         else if req is a ReqRes then
154             if req ∉ lastTok[r].wQueue then
155                 add(lastTok[r].wQueue,req);
156         else if req is a ReqLoan then
157             if req ∉ lastTok[r].wLoan then
158                 add(lastTok[r].wLoan,req);
```

Figure 11: Auxiliary methods

```
159 Receive Request (visitedNodes : set of sites, ReqsRecv
    : set of request messages) from s_from
160 begin
161     foreach Req req ∈ ReqsRecv do
162         resource r ← req.r;
163         site s_init ← req.s_init;
164         integer id ← req.id;
165         if req is obsolete then
166             continue;
167         if r ∈ TOwned then
168             if req is a ReqLoan then
169                 processReqLoan(req);
170             else if r ∉ TRequired or (state = waitS and
                req is not a ReqCnt) then
171                 SendToken(s_init, r);
172             else if req is a ReqCnt then
173                 lastTok[r].lastReqC[s_init] ← id;
174                 buffer(s_init,Counter,<
                    r, lastTok[r].counter > );
175                 lastTok[r].counter ←
                    lastTok[r].counter + 1;
176             else if req is a ReqRes then
177                 ReqRes myReq ←<
                    self, r, curId, A(MyVector) >;
178                 if req ∉ lastTok[r].wQueue then
179                     if state = waitCS ∧ req ◁ myReq
                        then
180                         add(lastTok[r].wQueue,myReq);
181                         SendToken(s_init, r);
182                     else
183                         /* (waitCS∧myReq◁req)∨inCS
                            */
184                         add(lastTok[r].wQueue,req);
185         else if tokDir[r] ∉ visitedNodes then
186             add(pendingReq[r],req);
187             buffer(tokDir[r],Req, req );
188     SendBufReq(visitedNodes ∪ {self});
189     SendBuf();

190 processReqLoan(req : ReqLoan)
191 begin
192     /* req.r ∈ Towned                                  */
193     resource r ← req.r;
194     site s_init ← req.s_init;
195     integer id ← req.id;
196     if req is not obsolete then
197         if canLend(req) then
198             TLent ← req.missingRes;
199             foreach resource r' ∈ TLent do
200                 lastTok[r'].lender ← self;
201                 lastTok[r'].wQueue ←
                    lastTok[r'].wQueue − {req};
202                 SendToken(s_init, r');
203         else
204             if r ∉ TRequired or state = waitS then
205                 SendToken(s_init, r);
206             else if req ∉ lastTok[r].wLoan then
207                 add(lastTok[r].wLoan,req);

208 Receive Token (ToksRcv : sets of Tok) from s_from
209 begin
210     /* t.r must be in TRequired                        */
211     foreach Tok t ∈ ToksRcv do
212         processUpdate(t);
213     if TRequired ⊆ TOwned then
214         state ← inCS;
215         notify(TRequired ⊆ TOwned);
216     else
217         foreach resource r ∈ TOwned do
218             site s_lender ← lastTok[r].lender;
219             if s_lender ≠ nil then
220                 /* the loan has failed: s_i is not in CS  */
221                 /* return borrowed tokens to its legitimate
                       owner                                */
222                 SendToken(s_lender,r);
223                 loanAsked ← false;
224         if state = waitS ∧ CntNeeded = ∅ then
225             processCntNeededEmpty();
226         foreach resource r ∈ TOwned do
227             if lastTok[r].wQueue ≠ ∅ then
228                 Req req ← Head(lastTok[r].wQueue);
229                 site s_i ← req.s_init;
230                 if state = waitS then
231                     dequeue(lastTok[r].wQueue);
232                     SendToken(s_i, r);
233                 else if state = waitCS then
234                     Req myReq ←<
                        self, r, curId, res, A(MyVector) >;
235                     if req ◁ myReq then
236                         dequeue(lastTok[r].wQueue);
237                         add(lastTok[r].wQueue,myReq);
238                         SendToken(s_i, r);
239                     else
240                         /* IMPOSSIBLE                   */
241         foreach resource r ∈ TOwned do
242             if lastTok[r].wLoan ≠ ∅ then
243                 copy : Set of ReqLoan;
244                 copy ← lastTok[r].wLoan;
245                 lastTok[r].wLoan ← ∅;
246                 foreach ReqLoan req ∈ copy do
247                     processReqLoan(req);
248     MissingRes ← TRequired − (TOwned ∩ TRequired);
249     if |MissingRes| = givenThreshold and state = waitCS
        and loanAsked = false then
250         loanAsked ← true;
251         foreach resource r ∈ MissingRes do
252             buffer(tokDir[r],ReqLoan,
                < self, r, curId, A(MyVector), MissingRes > );
253     SendBuf();
254     SendBufReq({self});

255 Receive Counter(CntsRcv : set of Count) from s_from
256 begin
257     foreach Count c ∈ CntsRcv do
258         MyVector[c.r] ← c.val;
259         CntNeeded ← CntNeeded − {c.r};
260         tokDir[r] ← s_from;
261     if CntNeeded = ∅ then
262         processCntNeededEmpty();
```

Figure 12: Messages processing

Annex B: Proof of correctness

Hypothesis

Model

We point out that we consider a distributed system consisting of a finite set Π of reliable N nodes, $\Pi = \{s_1, s_2, ...s_N\}$ and a set of M resources, $\mathcal{R} = \{r_1, r_2, ..., r_M\}$. The set Π is totally ordered by the order relation $\prec$ and $s_i \prec s_j$ iff $i < j$.

We define the set T, a set of instants t which discretizes the time of the system execution where t_0 represents the moment of the initialization. The time is discretized by the primitives algorithm execution:

- $ReqCS(s_i, rcs)^t$ when the site s_i wishes to enter in critical section at t with the critical section request rcs
- $RelCS(s_i, rcs)^t$ when the site s_i wishes to release the critical section at t for the critical section request rcs
- $RecvReq(s_i, Mreq)^t$ when the site s_i receives the request message $Mreq$ at t
- $RecvCnt(s_i, Mcnt)^t$ when the site s_i receives the counter message $Mcnt$ at t
- $RecvTok(s_i, Mtok)^t$ when the site s_i receives the token message $Mtok$ at t

We denote $R^t_{CS} \subset \Pi \times T \times \mathcal{P}(\mathcal{R})$, the **set of issued critical section requests** (satisfied and pending) at $t \in T$. A **critical section request** $rcs \in R^{t'}_{CS}$ is a triplet (s, t, D) representing a critical section request by the site s at $t \leq t'$ with the set of required resources D.

We denote $R^t_{cnt} \subset \mathcal{R} \times R^t_{CS}$ the **set of issued counter requests** at $t \in T$. A **counter request** $rcnt \in R^t_{cnt}$ is a couple (r, rcs) representing the request of the counter value associated with r for the critical section request rcs.

We denote $R^t_{res} \subset \mathcal{R} \times R^t_{CS} \times \mathbb{R}$ the **set of issued resource requests** at $t \in T$. A **resource request** $rres \in R^t_{res}$ is a triplet (r, rcs, m) representing for the critical section request rcs, the request of the resource token associated with r and m, the resulting value of $\mathcal{A}$.

We denote $R^t_{loan} \subset \mathcal{R} \times R^t_{CS} \times \mathbb{R} \times \mathcal{P}(\mathcal{R})$ the **set of issued loan requests** at $t \in T$. A **loan request** $rloan \in R^t_{loan}$ is a quadruplet $(r, rcs, m, \mathcal{M})$ representing for the critical section request rcs and the resource r, the loan request associated with the set of missed resources $\mathcal{M}$ and m, the resulting value of $\mathcal{A}$.

We denote $TOK^t \subset \mathbb{N} \times \mathcal{P}(R^t_{res}) \times \mathcal{P}(R^t_{loan}) \times \Pi$ **the set of tokens** at $t \in T$ A token associated with a resource r at $t \in T$ is denoted tok^t_r and is a quadruplet $(vcount, Q^t_r, QL^t_r, s_{lender})$. $vcount$ is equal to the current value of the counter associated with r, Q^t_r (respectively QL^t_r) is the queue of resource requests (resp. loan requests) for the resource r at $t \in T$. The total order $\lhd$ is applicable on each Q^t_r and each QL^t_r.

A **request message** $Mreq$ in transit at $t \in T$ is a quadruplet denoted

$$\langle req, V, s_{from}, s_{to} \rangle_{req}$$

of the set $R^t_{cnt} \cup R^t_{res} \cup R^t_{loan} \times \Pi \times \Pi$ representing the request message transition for the request req from the site s_{from} to the site s_{to}. The set V contains all sites already visited by the message.

A **counter message** $Mcnt$ in transit at $t \in T$ is a quadruplet denoted

$$\langle r, val, s_{from}, s_{to} \rangle_{cnt}$$

of the set $\mathcal{R} \times \mathbb{N} \times \Pi \times \Pi$ representing the counter message transition concerning the resource r from the site s_{from} to the site s_{to} with the value v.

A **token message** $Mtok$ in transit at $t \in T$ is a triplet denoted

$$\langle tok_r^t, s_{from}, s_{to} \rangle_{tok}$$

representing the token message transition concerning the resource r from the site s_{from} to the site s_{to}.

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

A process s_i can be at $t \in T$ in one of the four following states: $Idle_{s_i}^t$, $waitS_{s_i}^t$, $waitCS_{s_i}^t$ and $inCS_{s_i}^t$. Its local variables are:

- $tokdir_{(s_i,r)}^t \in \Pi$ indicates the father of s_i in the tree of the corresponding resource r at $t \in T$. If s_i is the root site of the tree associated with r at t then $tokdir_{(s_i,r)}^t = nil$.
- $Towned_{s_i}^t \subset \mathcal{P}(TOK^t)$ is the set of owned tokens by site s_i at $t \in T$.
- $currcs_{s_i}^t \in R_{CS}^t$ represents the issued critical section request of site s_i at t. If $Idle_{s_i}^t$, $currcs_{s_i}^t = (s_i, nil, \emptyset)$
- $vector_{s_i}^t \in \mathbb{N}^M$ represents the vector of the current CS request of s_i at t. $vector[r]_{s_i}^t$ is value of received counter for the resource r.
- $CntNeeded_{s_i}^t \in \mathcal{P}(\mathcal{R})$ represents the set of required resources where the process s_i has not yet received the corresponding counter value at $t \in T$.
- $pendingReq_{s_i}^t \in \mathcal{P}(R_{cnt}^t \cup R_{res}^t \cup R_{loan}^t)$ represents the set of received requests which have potentially not yet been processed by the corresponding token owner at $t \in T$.
- $Tlent_{s_i}^t \subset \mathcal{P}(\mathcal{R})$ is the set of lent resources by site s_i at $t \in T$.
- $loanAsked_{(s_i,rcs)}^t \in \{true, false\}$ true if s_i has sent a loan request for the critical section request rcs at $t \in T$, false otherwise.

According to the pseudo-code, we assume that

$$\forall t \in T, \forall r \in \mathcal{R}, \forall s_i \in \Pi, tokdir_{(s_i,r)}^t = nil \Leftrightarrow tok_r^t \in Towned_{s_i}^t$$

.

Definition 1. We define a total order, denoted, $\lhd$ on R_{res}^t where $rres_i = (r_i, (s_i, t_i, D_i), m_i) \in R_{res}^t$ and $rres_j = (r_j, (s_j, t_j, D_j), m_j) \in R_{res}^t$:

$$rres_i \lhd rres_j \Leftrightarrow m_i < m_j \vee (m_i = m_j \wedge s_i \prec s_j)$$

Definition 2. We define the function $\mathcal{H}(E)$ applicable on all set E where the total order relation $\lhd$ can be defined. This function return the smallest element according to the $\lhd$ order, or in others words the head of E if E is a queue. Formally,

$$\mathcal{H}(E) = e \Leftrightarrow \nexists e' \in E, e' \lhd e$$

Hypothesis 1. Nodes are assumed to be reliable and be connected by means of reliable (neither message duplication nor message loss) and FIFO communication links.

Hypothesis 2. Processes do not share memory and communicate by sending and receiving messages

Hypothesis 3. The communication graph is complete, any node can communicate with any other one.

Hypothesis 4. A process can not request a new CS before its previous one has been satisfied.

Hypothesis 5. All critical section execution time is finite.

Hypothesis 6. The definition of $\mathcal{A}$ ensures that every request will have, in a finite time, the smallest real value m among all pending requests according to the order $\lhd$.

Safety property

Definition 3. We define the predicate $Conf(s_i, s_j, t)$ which is verified when s_i and s_j are conflicting processes at t. Formally,

$$Conf(s_i, s_j, t) \Leftrightarrow \Big(currcs^t_{s_i} = (s_i, t_i, D_i) \wedge currcs^t_{s_j} = (s_j, t_j, D_j) \Rightarrow D_i \cap D_j \neq \emptyset\Big)$$

Lemma 1. For all resource r, if it exists a root node in the corresponding tree, the corresponding token message is not in M^t. Formally,

$$\forall t \in T, \forall r \in \mathcal{R},\ \exists s_i \in \Pi,\ tokdir^t_{(s_i,r)} = nil \ \Leftrightarrow \nexists \langle tok^t_r, s_{from}, s_{to}\rangle_{tok} \in M^t$$

Proof.

* We prove by recurrence $\forall t \in T, \forall r \in \mathcal{R}, \nexists \langle tok^t_r, s_{from}, s_{to}\rangle_{tok} \in M^t \Rightarrow \exists s_i \in \Pi, tokdir^t_{(s_i,r)} = nil$:

The property is true at t_0 ($M^{t_0} = \emptyset$ and $\forall r \in \mathcal{R}$ it exists an only site s_i where $tokdir^t_{(s_i,r)} = nil$). By assuming this property true until the moment t_k, we will prove it at the moment $t_k + 1$.

. If the next moment is a $reqCS$ procedure execution on a site s_i, the root site of any resource r does not send a $Mtok$ message and $tokdir^{t_k+1}_{(s_i,r)} = nil$.

. If the next moment is a $relCS$ procedure execution on a site s_i, the $SendToken$ procedure is called $\forall (vcount, Q^{t_k}_r, QL^{t_k}_r, s_{lender}) \in Towned^{t_k}_{s_i}$, if $Q^{t_k}_r \neq \emptyset$ or if $s_{lender} \neq nil$. This procedure implies that $tokdir^{t_k+1}_{(s_i,r)} \neq nil \Rightarrow \exists \langle tok^{t_k+1}_r, s_i, s_{to}\rangle_{tok} \in M^{t_k+1}$. Otherwise, $tokdir^{t_k+1}_{(s_i,r)} = tokdir^{t_k}_{(s_i,r)}$.

. If the next moment is a $RecvReq(s_i, \langle req, V, s_{from}, s_i\rangle_{req})$ procedure execution, we can apply the same reasoning of $Release\_CS$ procedure execution: $SendToken$ procedure is called only if s_i owns the corresponding resource r whatever the type of req. Otherwise, if the $SendToken$ procedure is not called, $tokdir^{t_k+1}_{(s_i,r)} = tokdir^{t_k}_{(s_i,r)}$.

. If the next moment is a $RecvCnt(s_i, \langle r, val, s_{from}, s_i\rangle_{cnt})$ procedure execution, no token message for resource r can be sent from s_i. Consequently the property is still true at $t_k + 1$.

. Since we suppose $\nexists Mtok \in M^{t_k}$, the execution of the $RecvTok$ procedure is impossible.

* We prove by recurrence $\forall t \in T, \forall r \in \mathcal{R}, \exists s_i \in \Pi, tokdir^t_{(s_i,r)} = nil \Rightarrow \nexists \langle tok^t_r, s_{from}, s_{to}\rangle_{tok} \in M^t$:

The property is true at t_0 ($\forall r \in \mathcal{R}$, $tokdir^{t_0}_{(s_{elected},r)} = nil$ and there is no token message in M^{t_0}). By assuming this property true until the moment t_k, we will prove it at the moment $t_k + 1$.

. If the next moment is a $reqCS$ procedure execution on a s_i where $\forall r \in \{r_i | tokdir^{t_k}_{(s_i,r_i)} = nil\}$ and according to the recurrence assumption $\nexists \langle tok^{t_k+1}_r, s_i, s_{to}\rangle_{tok} \in M^{t_k+1}$.

. If the next moment is a $relCS$ procedure execution on a site s_i, $\forall (vcount, Q^{t_k}_r, QL^{t_k}_r, s_{lender}) \in Towned^{t_k}_{s_i}$, if $Q^{t_k}_r \neq \emptyset$ or if $s_{lender} \neq nil$ then $tokdir^{t_k+1}_{(s_i,r)} \neq nil$ and the token message corresponding to r is sent, otherwise $tokdir^{t_k}_{(s_i,r)} = tokdir^{t_k+1}_{(s_i,r)} = nil$ and no token message corresponding to r is sent.

. If the next moment is a $RecvReq(s_i, \langle req, V, s_{from}, s_i\rangle_{req})$ procedure execution for the resource r, if $tokdir^{t_k}_{(s_i,r)} = nil$ and if the $SendToken$ procedure is called then $tokdir^{t_k+1}_{(s_i,r)} \neq nil$ and $\exists \langle tok^{t_k+1}_r, s_i, s_{to}\rangle_{tok} \in M^{t_k+1}$.

. If the next moment is a $RecvCnt(s_i, \langle r, val, s_{from}, s_i\rangle_{cnt})$ procedure execution, $tokdir^{t_k}_{(s_i,r)} \neq nil$ and $tokdir^{t_k}_{(s_i,r)} \neq nil$ and according to the recurrence assumption, the property is true at $t_k + 1$.

. If the next moment is a $RecvTok(s_i, \langle tok^t_r, s_{from}, s_i\rangle_{tok})$ procedure execution, either $tokdir^{t_k+1}_{(s_i,r)} = nil$ (s_i keeps the token) and $\nexists \langle tok^{t_k+1}_r, s_{si}, s_{to}\rangle_{tok} \in M^{t_k+1}$, either the token corresponding to r is yielded or lent to another site. In this case, the $SendToken$ procedure is called then $tokdir^{t_k+1}_{(s_i,r)} \neq nil$ and $\exists \langle tok^{t_k+1}_r, s_i, s_{to}\rangle_{tok} \in M^{t_k+1}$

□

Lemma 2. $\forall t \in T, \forall r \in \mathcal{R}$, there is at most one token message $\langle tok^t_r, s_{from}, s_{to}\rangle_{tok}$ in M^t

Proof. Since we consider reliable channels (no loss, no duplication) according to the hypothesis 1, $\forall r \in \mathcal{R}$ there is an only corresponding token message at each token sending in the network . Moreover following functions induces a token message sending:

. In $RelCS(s_i, rcs)^t$, when $\forall r_k, tok^t_{r_k} \in Towned^t_{s_i}$ then $tokdir^t_{(s_i,r_k)} = nil$ and according to the lemma 1 $\nexists \langle tok^t_{r_k}, s_{from}, s_{to}\rangle_{tok} \in M^t$.

. In $RecvReq(s_i, \langle req, V, s_{from}, s_i\rangle_{req})^t$ concerning the resource r_k when $tokdir^t_{(s_i,r_k)} = nil$. In this case, according to the lemma 1 $\nexists \langle tok^t_{r_k}, s_{from}, s_{to}\rangle_{tok} \in M^t$.

. In $RecvTok(s_i, \langle tok^t_{r_k}, s_{from}, s_i\rangle_{tok})^t$ upon a receipt of the token concerning the resource r_k. This token message is then removed in M^t if the current site enters in critical section or keep the token while waiting for other required resources, otherwise the token is yielded or lent.

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

Lemma 3. $\forall r \in \mathcal{R}$, there is at most one site s_i where $tokdir^t_{(s_i,r)} = nil$. Formally,

$$\forall t \in T, \forall r \in \mathcal{R}, \exists s_i \in \Pi, tokdir^t_{(s_i,r)} = nil \Rightarrow \nexists s_j \in \Pi, tokdir^t_{(s_j,r)} = nil$$

Proof. Except at the initialization, for a resource r $tokdir^t_{(s_i,r)} = nil$ uniquely when a site s_i receives the token corresponding to r at the moment t. At each token sending (*sendToken* procedure), the corresponding *tokdir* variable becomes systematically $\neq nil$. Since it exists at most one token message per resource in the system according to lemma 2 there is at most one root site per resource $\forall t \in T$. □

Theorem 1 (Safety)**.** The algorithm ensures the safety property

$$\forall t \in T, \quad \forall (s_i, s_j) \in \Pi^2,\ Conf(s_i, s_j, t) \wedge inCS^t_{s_i} \Rightarrow \neg inCS^t_{s_j}$$

Proof. The application of lemmas 1, 2 and 3 implies that it exists at most one token in the system per resource: either a token is in the network, either it is owned by a unique root site. Since a site can enter in critical section if it is not in the $Idle$ state and if it owns all required resources, conflicting processes are not able to execute their critical section simultaneously. □

Liveness property

Definition 4. For each resource $r \in \mathcal{R}$ and for each instant $t \in T$, we define an oriented graph, called **precedence graph** and denoted $\mathcal{G}^t_r = (\mathcal{N}^t_r, \mathcal{E}^t_r)$ where

- $\mathcal{N}^t_r \subseteq \Pi$ is the vertices set and a site s_i belongs to $\mathcal{N}^t_r$ iff it exists a resource request $rres$ in Q^t_r where s_i is the initiator of $rres$. Formally,

$$s_i \in \mathcal{N}^t_r \Leftrightarrow \exists (r, (s_{ini}, t_{ini}, D_{ini}), m) \in Q^t_r \wedge s_{ini} = s_i$$

- $\mathcal{E}^t_r \subset \mathcal{N}^t_r \times \mathcal{N}^t_r$ is the arrows set of couple (s_i, s_j) where s_i precedes s_j iff it exists two different resource requests $rres_i$ and $rres_j$ in Q^t_r where s_i is the initiator of $rres_i$ and s_j is the initiator of $rres_j$ and $rres_j$ is the direct successor of $rres_i$ in Q^t_r according to total order relation $\lhd$. Formally,

$$(s_i, s_j) \in \mathcal{E}^t_r \Leftrightarrow \quad \exists rres_i, rres_j \in Q^t_r, rres_i \neq rres_j \wedge rres_i \lhd rres_j \\ \wedge\ \nexists rres_k \in Q^t_r,\ rres_i \lhd rres_k \lhd rres_j$$

Definition 5. For each instant $t \in T$, we define an oriented graph, called **global precedence graph** and denoted $\mathcal{G}^t = (\mathcal{N}^t, \mathcal{E}^t)$ where $\mathcal{N}^t = \bigcup_{r \in \mathcal{R}} \mathcal{N}^t_r$ and $\mathcal{E}^t = \bigcup_{r \in \mathcal{R}} \mathcal{E}^t_r$

Lemma 4. Every precedence graph $\mathcal{G}^t_r$ is empty or a degenerated tree.

Proof. We can prove it by recurrence. At t_0, $\forall r \in \mathcal{R}$, $\mathcal{G}^{t_0}_r$ is empty. Let's suppose that the property is true until the moment t_k. If the moment at $t_k + 1$ is:

. $reqCS(s_i, rcs)$: the property is still true at $t_k + 1$ since this procedure execution does not modify any $Q^{t_k}_r$.

. $relCS(s_i, rcs)$: the first element of each queue of owned resources r is removed (root site for $\mathcal{G}^{t_k}_r$) if r has not been lent, otherwise the lender is potentially removed from $Q^{t_k}_r$. Thus $\mathcal{G}^{t_k+1}_r$ is still either empty either a degenerated tree.

. $RecvReq(s_i, \langle req, V, s_{from}, s_i \rangle_{req})$: if req is a loan request or a counter request for resource r, $Q^{t_k}_r$ does not change. Otherwise, $s_i \in \mathcal{N}^{t_k+1}_r$ or $s_{init} \in \mathcal{N}^{t_k+1}_r$. $s_i \notin \mathcal{N}^{t_k}_r$ because he owns the token, and $s_{init} \notin \mathcal{N}^{t_k}_r$ according to the hypothesis 4. Consequently, the property is true at $t_k + 1$.

. $RecvCnt(s_i, \langle r, val, s_{from}, s_i\rangle_{cnt})$: the property is still true at $t_k + 1$ since this procedure execution does not modify any $Q_r^{t_k}$.

. $RecvTok(s_i, \langle tok_r^t, s_{from}, s_i\rangle_{tok})$: in the procedure *processUpdate*, all stored request message in $pendingReq_{s_i}^{t_k}$ related to a non obsolete resource request req is added in the corresponding queue $Q_r^{t_k}$. If req is not already in $Q_r^{t_k}$, then $s_{init} \notin \mathcal{N}_r^{t_k}$ according to the hypothesis 4. If $waitS_{s_i}^{t_k}$ and $Q_r^{t_k} \neq \emptyset$ then the first element of $Q_r^{t_k}$ is just removed. If $waitCS_{s_i}^{t_k}$ and $Q_r^{t_k} \neq \emptyset$, then we can apply the same reasoning of the execution of *RecvReq* procedure. Consequently, the property is true at $t_k + 1$.

□

Corollary 1. For all precedence graph $\mathcal{G}_r^t$, it does not exist an oriented cycle in $\mathcal{G}_r^t$.

Lemma 5. $\forall t \in T$, it does not exist an oriented cycle in $\mathcal{G}^t$.

Proof. Let's suppose that an oriented cycle in $\mathcal{G}^t$ may occurs. Since an oriented cycle in a single given $\mathcal{G}_r^t$ is impossible according to the corollary 1, the cycle concerns an union of $\mathcal{G}_r^t$. Consequently it exists a path P of k nodes in $\mathcal{G}^t$ denoded $(s_1, s_2, ..., s_{i-1}, s_i, s_{i+1}, ..., s_{k-1}, s_k, s_1)$. By applying the definition of $\mathcal{G}_r^t$ and if we note $rres_x = (r_x, (s_x, t_x, D_x), m_x)$,

$$rres_1 \lhd rres_2 \lhd ... \lhd rres_k$$

by transitivity

$$rres_1 \lhd rres_k$$

However it exists an arrow (s_k, s_1) and then

$$rres_k \lhd rres_1$$

It is impossible since $\lhd$ is a total order. Then there is a contradiction and consequently it does not exist an oriented cycle in $\mathcal{G}^t$, $\forall t \in T$. □

Theorem 2. The algorithm is deadlock free.

Proof. This property is a direct consequence of lemma 5. Since oriented cycles in $\mathcal{G}^t$ can not occur, $\forall t \in T$ any pending CS request is distinguishable from each other and then deadlocks are impossible.

□

Lemma 6. For all request message $Mreq = \langle req, V, s_{from}, s_{to}\rangle_{req}$ concerning a resource $r \in \mathcal{R}$ issued at $t \in T$, it exists $t' \in T$, $t < t'$ where the token holder of r is aware of req's existence.

Proof. Let's suppose it exists a request message $Mreq = \langle req, V, s_{from}, s_{to}\rangle_{req}$ that req will never be known by the corresponding token owner. This can happen in the following cases:

. a $Mreq$ message has been lost: impossible according to hypothesis 1.

. the message forwarding has been stopped by a site s_k at $t_k \in T$:

  * s_k is the token holder: there is a contradiction with our hypothesis.

  * s_k has detected that req is obsolete: the token holder has already processed req. There is a contradiction.

  * $tokdir^{t_k}_{(s_i,r)} \in V$: in this case we are sure that the token will be eventually received by a site s_i which has already forwarded req. Since req is stored in $pendingReq^t_{s_i}$, req will be processed by the token holder. There is a contradiction.

. req is forwarded indefinitely: the number of elements in the set of visited nodes V increases indefinitely at each message forwarding. This is impossible because $V \subset \Pi$ and the set of sites Π is assumed finite and static. Consequently the token holder will be eventually a site having already forwarded req.

□

Lemma 7. $\forall t \in T, \forall s \in \Pi$, if $\exists t \in T$ where $waitS^t_s$ then $\exists t' \in T, t < t'$ where $currcs^t_s = currcs^{t'}_s$ and $CntNeeded^{t'}_s = \emptyset$.

Proof. According to the lemma 6, for all required resource r by a site s_{ini} at $t \in T$, it exists $t_k > t$ where each token holder s_{holder} of a resource r:

. either sends a counter message to s_{ini} if s_{holder} requires r

. either sends the token directly to s_{ini} if s_{holder} does not require r

In both cases, when s_{ini} receives the message at t'_k, an element of $CntNeeded^{t_k}_{s_{ini}}$ is removed and eventually it exists $t' > t$ where $CntNeeded^{t'}_{s_{ini}} = \emptyset$

□

Corollary 2. $\forall t \in T, \forall s \in \Pi$, if $waitS^t_s$ then $\exists t' \in T, t < t'$ where $waitCS^{t'}_s$

Lemma 8. For all pending CS request (s, t, D) and for all $r \in D$, it exists $t' \in T, t < t'$ where:

. either it exists a resource request $rres = (r, rcs, m) \in Q^{t'}_{res}$ and $rcs = (s, t, D)$.

. either s owns $tok^{t'}_r$

Proof. According to the lemma 6, for all required resource r by a site s_{ini} at $t \in T$, it exists $t_k > t$ where each token holder s_{holder} of a resource r:

. either stores the resource request in $Q^{t_k}_r$ if s_{holder} requires r and is in $inCS$ state or s_{holder}'s resource request has a higher priority than s_{ini}'s resource request according to the $\triangleleft$ order.

. either sends the token directly to s_{ini} and s_{holder} potentially puts its resource request in $Q^{t_k}_r$ if s_{holder} are conflicting and s_{ini}'s resource request has a higher priority than s_{holder}'s resource request according to the $\triangleleft$ order.

□

Lemma 9. $\forall t \in T, \forall r \in \mathcal{R}$, if $rres_a \in Q^t_r$ then it exists $t' \in T, t < t'$ from which, every new insertion of resource request $rres_b$ in $Q^{t'}_r$ always verifies $rres_a \triangleleft rres_b$.

Proof. Let's suppose that t' does not exist for a resource request $rres_a = (r, rcs_a, m_a)$ in Q^t_r. Then the element $rres_a$ of Q^t_r can forever be overtaken at each new insertion. Consequently m_a is never the smallest element in the order $\triangleleft$. This is impossible because of the hypothesis 6. □

Lemma 10. All lent resources will be returned to the lender in a finite time. Formally,
$\forall t \in T, \forall s \in \Pi, \exists\ t' > t$ where $\forall r \in Tlent^t_s$, $tok^{t'}_r \in Towned^{t'}_{s_i}$.

Proof. When a site $s_{borrower}$ receives a set of tokens as a loan from a site s_{lender}:

. either $s_{borrower}$ may enter in critical section: according to hypothesis 5, it will call in a finite time the $releaseCS$ procedure where the lent tokens will be returned to s_{lender}.

. either $s_{borrower}$ may stay in $waitCS$ state: in this case the borrowed tokens are immediately returned to s_{lender}.

Consequently s_{lender} will recover the lent tokens in finite time. □

Lemma 11. $\forall t \in T$, $\forall r \in \mathcal{R}$, if $rres = (r, (s_{ini}, t_{ini}, D_{ini}), m) = \mathcal{H}(Q_r^t)$ then it exists $t' \in T$, $t < t'$ where $tok_r^{t'} \in Towned_{s_{ini}}^{t'}$

Proof. We denote s_{holder}, the site holding the token tok_r^t at t and we consider that $currcs_{s_{holder}}^t = (s_{holder}, t_{holder}, D_{holder}) \neq nil$. Let's suppose that t' does not exist for the resource r. This can happen in the following cases:

. s_{holder} executes indefinitely its critical section: impossible according to hypothesis 5.

. s_{holder} is in $waitCS$ state and waits indefinitely missing resources r' preventing it to enter in critical section because:

  * either it exists a deadlock: impossible according to theorem 2.
  * either s_{holder} has lent some resources to s_{lender} and the latter keeps indefinitely the lent resources: impossible according to lemma 10.
  * either all non owned tokens $tok_{r'}^t$ are kept indefinitely by conflicting sites with an higher priority according to the $\triangleleft$ order: this case is impossible by applying recursively the same reasoning for all r' and by pointing out that D_{holder} and $\mathcal{R}$ are finite sets.

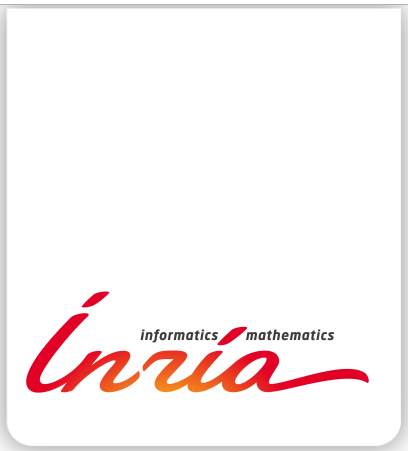
□

Theorem 3 (Liveness)**.** The algorithm ensures the liveness property
$\forall s_i \in \Pi\ \forall t \in T,\ \exists t' \in T,\ t' > t,$

$$waitS_{s_i}^t \Rightarrow inCS_{s_i}^{t'}$$

Proof. The corollary 2 implies that $\forall s_i \in \Pi$ it exists a moment $t_a > t$ where $waitCS_{s_i}^{t_a}$. Consequently, all s_i in the $waitCS$ state sent a resource request for all missing resources r_{miss}. Then, it exists a moment at $t_b > t_a$ according to the lemma 8 where all resource request sent are stored in the corresponding waiting queue $Q_{r_{miss}}^{t_b}$. The lemmas 9 and 11 imply that for all resource request $rres$ stored in a queue $Q_r^{t_b}$, it exists $t_c > t_b$ where $rres = \mathcal{H}(Q_r^{t_c})$. Finally the lemma 11 implies that all required resource will be owned by s_i at the moment $t' > t_c$. Thus, s_i is in the $inCS$ state at $t' > t$.

□



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399