A Composition Approach to Mutual Exclusion Algorithms for Grid Applications

Julien Sopena, Fabrice Legond-Aubry, Luciana Arantes, and Pierre Sens

LIP6 - Université de Paris 6 - INRIA Rocquencourt

4, Place Jussieu 75252 Paris Cedex 05, France.

Phone: (33).1.44.27.34.23 - Fax : (33).1.44.27.74.95

email: [julien.sopena,fabrice.legond-aubry,luciana.arantes,pierre.sens]@lip6.fr

Abstract

We propose a new composition approach to mutual exclusion algorithms for applications spread over a grid which is composed of a federation of clusters. Taking into account the heterogeneity of communication latency, our hierarchical architecture combines intra and inter cluster algorithms. We focus on token-based algorithms and study different compositions of algorithms. Performance evaluation tests have been conducted on a national grid testbed whose results show that our approach is scalable and that the choice of the most suitable inter cluster algorithm depends on the behavior of the application.

Keywords: distributed mutual exclusion algorithm, Grid, performance evaluation.

1 Introduction

By gathering geographically distributed resources, a Grid offers a single large-scale environment suitable for the execution of computational intensive applications. A Grid usually comprises of a large number of nodes grouped into clusters. Nodes within a cluster are often linked by local networks (LAN) while clusters are linked by a wide area network (WAN). Therefore, Grids present a hierarchy of communication delays: the cost of sending a message between nodes of different clusters is much higher than that of sending the same message between nodes within the same cluster.

Distributed or parallel applications that run on top of a Grid usually require that their processes get exclusive access to some shared resource (critical section). Thus, the performance of mutual exclusion algorithms is critical to Grid applications and it is the focus of this paper. A mutual exclusion algorithm ensures that exactly one process can execute the critical section at any given time (*safety* property) and that all critical section (CS) requests will eventually be satisfied (*liveness* property).

Mutual exclusion algorithms can be divided into two families: *permission-based* (e.g. Lamport [7], Ricart-Agrawala [15], Maekawa [9]) and *token-based* (Suzuki-Kazami [17], Raymond [14], Naimi-Tréhel [12], Martin [10]. The algorithms of the first family are based on the principle that a node enters a critical section only after having received permission from all the other nodes (or a majority of them [9]). In the second group of algorithms, a system-wide unique token is shared among all nodes, and its possession gives a node the exclusive right to enter a critical section. Token-based algorithms present different solutions for the transmission and control of critical section requests from processes. Each solution is usually expressed by a logical topology that defines the paths followed by critical request messages. For instance, Martin's algorithm [10] considers that nodes are organized in a logical ring and CS requests should be forwarded along this ring; Naimi-Tréhel's algorithm [12] maintains a dynamic logical tree for transmitting CS requests, such that the root of the tree is always the last site that will get the token among the current requesting ones; in Suzuki-Kazami's algorithm [17], a process broadcasts its CS request to the others.

Token-based mutual exclusion algorithms present an average lower message traffic with regard to the number of nodes when compared to permission-based ones. Thus, they are more suitable

for controlling concurrent access to shared resources of Grids whose number of nodes is often very large. However, existing token-based algorithms do not take into account the above-mentioned hierarchy of communication latency. Therefore, we propose in this article a composition approach which allows the combination of any two token-based mutual exclusion algorithms: one at *intra-cluster* level and a second one at *inter-cluster* level. By using our composition mechanism several algorithms can be easily "plugged in" on both levels thus providing an interesting framework for comparing the performance of different combination of mutual exclusion algorithms. Furthermore, our approach has confirmed that the good choice for an *inter-cluster* mutual exclusion algorithm depends on the application behavior i.e., the frequency with which the application processes request the shared resource.

The contribution of this paper is then twofold: a hierarchical mutual exclusion composition approach which easily allows the combination of different *inter-cluster* and *intra-cluster* algorithms and a detailed performance evaluation of two-level mutual exclusion algorithms which shows that the behavior of parallel or distributed applications has an impact on the choice of *inter-cluster* algorithms. The token-based algorithms that we have chosen for our performance tests are Martin's, Naimi-Trehel's and Suzuki-Kazami's since they present distinct solutions for managing processes CS requests on top of different logical topologies.

The remainder of this paper is organized as follows. Section 2 presents the three above mentioned token-based algorithms which are based on different logical topology. In section 3, we describe our composition approach for mutual exclusion algorithms. Performance evaluation results combining the three algorithms is presented in section 4. Some related work is given in section 5. The last section concludes our work.

2 The chosen token-based algorithms

The three token-based algorithms that we have chosen are **Martin**'s, **Naimi-Trehel**'s, and **Suzuki-Kasami**'s which respectively consider a *ring*, a *tree*, and a *complete logical connection graph* for transmitting critical section requests. As they present distinct solutions for transmitting requests and controlling the algorithm's liveness, they present different message complexity with regard to the number of nodes.

The performance of a mutual exclusion algorithm is usually measured by the number of messages exchanged per critical section and the delay for getting access to the shared resource i.e., the time between the moment a node requests the CS and the moment it gets it. The latter, which we called the *obtaining time* in this paper, comprises the delay for transmitting a token request T_{req} plus the delay for granting the token T_{token}. However, if the time for waiting for the current pending requests T_{pendCS} is higher than T_{req}, the *obtaining time* is equal to T_{pendCS} plus $Tkoken$.

The three algorithms have different advantages and drawbacks when compared to each other. For instance, Suzuki-Kasami's and Martin's algorithms do not scale, while Naimi-Threhel's does. On the other hand, Martin's solution is much more simple than the others. However, by diffusing the request to all sites, Suzuki-Kasami's is more resilient to failures than the other two and a request takes just one message time.

2.1 Martin's algorithm

Martin's algorithm considers that nodes are organized in a logical ring. Requests for the token move along one direction while the token on the opposite direction.

When node i, which does not hold the token, wants to enter the critical section it asks for the token by sending a request message to its successor j in the ring. If j does not keeps the token it forwards the request to its successor. The request will travel along the ring till it reaches the site k which keeps the token. On receiving the request, if k is not in critical section itself, it forwards the token to its predecessor. Each node between k and i will do the same. Therefore, the token will eventually reach i, which can then enter the critical section. Notice that before the token reaches i, nodes between i and k might have requested the token too. Thus, when k forwards the token on behalf of i all pending requests of nodes between k and i will be satisfied when they receive the token.

If the number of nodes between i and k is x, $0 \leq x < N-1$, the total number of messages exchanged per critical section invocation is $2*(x+1)$: $(x+1)$ messages for the request + $(x+1)$ messages for the token. In average, such a number is equal to $2*(N/2) = N$ messages.

Considering T as the average message delay, a request message delay T_{req} and the token granting delay T_{token} are both equal to $(x+1)*T$ ($N*T$ in the average case).

Notice that for optimization reasons, upon receiving a request from its predecessor, a node that is also requesting the token does not need to forward the request of the predecessor. It just keeps the information that after satisfying its own request, it must send the token to its predecessor.

2.2 Naimi-Tréhel's algorithm

Naimi-Tréhel's algorithm establishes that nodes are organized in a logical tree and that a node always sends a token request to its father on the tree. It thus keeps two data-structures:

- A logical dynamic tree structure such that the root of the tree is always the last node which will get the token among the current requesting ones. Hence, requesting nodes form a logical tree of probable token owners that point to the root. Initially, the root is the token holder, elected among all nodes. We call this tree, the *last tree*, since each node keeps a local variable called *last* that points to the last probable owner of the token.

- A distributed queue which keeps critical section (CS) requests that have not been satisfied yet. We call this queue, the *next queue*, since each node i keeps a local variable called *next* that points to the next node to whom the token will be granted after i leaves the critical section.

When a node i wants to enter the critical section, it sends a request to its *last*. Node i then sets its *last* variable to itself and waits for the token. Node i becomes the new root of the tree.

Receiving i's token request message, node j can take one of the following actions: (1) j is not the root of the tree. It forwards the request to its *last*; (2) j is the root of the tree. If j holds the token but it is not in critical section, it directly sends the token back to i. On the other hand, if j either holds the token but is in the critical section or is waiting itself for the token, j sets its variable *next* to i. In both case, node j updates its *last* variable to i. Notice that the *last tree* is modified dynamically; At the end of the critical section, j sends the token to its *next*.

Tree-based algorithms result in an average number of message per CS equal to $\mathcal{O}(log(N))$ with regard to the number of nodes. A request message delay T_{req} takes in average $\mathcal{O}(log(N)) * T$ while T_{token} takes T.

2.3 Suzuki-Kasami's algorithm

In the Suzuki-Kazami's algorithm, when a node i, which does not hold the token, attempts to enter the critical section, it diffuses a request message to the other $N-1$ nodes. Such a message contains the identifier i of the node and a sequence number x which indicates the xth critical section invocation of i. As in the previous token-based algorithms, when node i receives the token, it enters the critical section.

Each node i keeps an array RN_i of size N where it stores the largest token invocation (sequence number) of each node of which it is aware. Whenever i receives a request from j, it updates $RN_i[j]$ with the sequence number of the request.

The *token* message includes a queue Q of nodes whose requests are pending and an array LN of size N which keeps the sequence number of the most recent satisfied request from each node. When node i exits the critical section, it updates $LN_i[i]$ with its current $RN_i[i]$ in order to indicate that its request has been satisfied. Then, it appends to Q all nodes not yet in Q for which it knows that their requests have not been satisfied yet. If Q is not empty, the first node j is removed from Q and the token is sent to j.

The algorithm requires N message exchanges for each mutual exclusion invocation. Both the request message delay T_{req} and token granting delay T_{token} are equal to T.

3 Composition approach to mutual exclusion algorithms

Our approach consists in having a hierarchy of token-based mutual exclusion algorithms: a per cluster mutual exclusion algorithm that controls critical section requests from processes within the same cluster and a second algorithm that controls *inter-cluster* requests for the token. The former is called the *intra* algorithm while the latter is called the *inter* algorithm. An *intra* algorithm of a cluster runs independently from the other *intra* algorithms.

The application is composed of a set of processes which run on the nodes of the Grid. We consider one process per node and we call it an *application* process. When an *application* process wants to access the shared resource, it calls the function *IntraCSRequest()*. It then executes its critical section. After executing it, the process calls the function *IntraCSRelease()* to release it.

Both functions are provided by the *intra* token algorithm.

3.1 Composition algorithm

Within each cluster there is a special node, the *coordinator*. The *inter* algorithm runs on top of the *coordinators* allowing them to request the right of accessing the shared resource on behalf of *application* nodes of their respective cluster. *Coordinators* are in fact hybrid processes which participate in both the *inter* algorithm with the other *coordinators* and the *intra* algorithm with their cluster's *application* processes. However, even if the *intra* algorithm sees a *coordinator* as an *application* process, the *coordinator* does not take part in the application's execution i.e, it never requests access to the shared resource for itself.

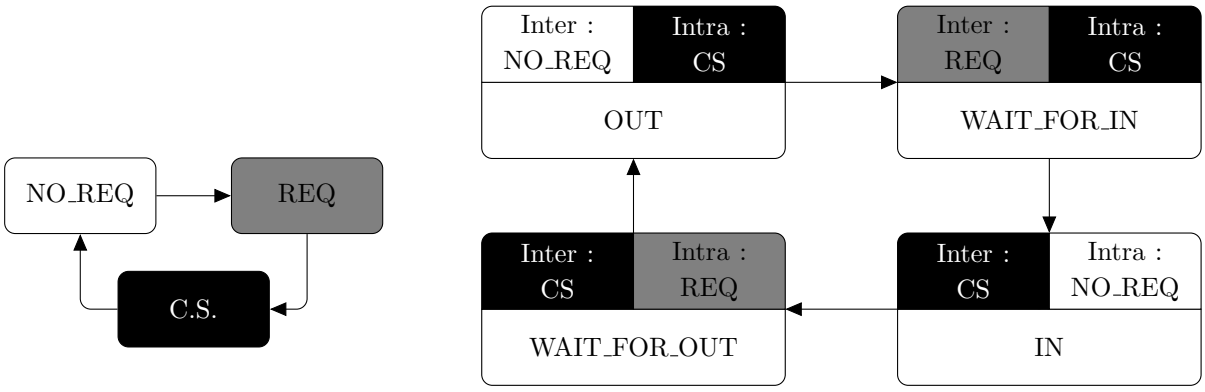
The key feature of our approach is that the two hierarchical algorithms are clearly separated since an *application* process gets access to the shared resource just by executing the *intra* algorithm of its cluster. Another important advantage is that the chosen algorithms for both layers do not need to be not modified. Hence, it is very simple to have different compositions of algorithms.

An *intra* algorithm controls an *intra* token while the *inter* algorithm controls an *inter* token. Thus, there is one *intra* token per cluster but a single *inter* token of which only the *coordinators* are aware. Holding the *intra* token is sufficient and necessary for an *application* process to enter the CS since the local *intra* algorithm ensures that no other local *application* node of the cluster has the *intra* token. On the other hand, considering the hierarchical composition of algorithms, our solution must also guarantee that no other *application* process of the other clusters is also in critical section when holding an *intra* token (per cluster *safety* property). In other words, the *safety* property of the *inter* algorithm must ensure that at any time only one cluster has the right of allowing its *application* processes to execute the CS. This property can be asserted by the possession of the *inter* token by a *coordinator*.

Similarly to a classical mutual exclusion algorithm, the *coordinator* calls the *InterCSRequest()* and the *InterCSRelease()* functions for respectively asking or releasing the *inter* token. However, when a *coordinator* is in critical section, it means that *application* processes of its cluster have the right of accessing the resource. The *inter* token is hold by the *coordinator* of this cluster which is then considered to be in critical section by the other *coordinators*.

The guiding principle of our approach is described in the pseudo code of figure 2. Initially, every *coordinator* holds the *intra* token of its cluster. When an *application* process wants to enter the critical section, it sends a request to its local *intra* algorithm by calling the *IntraCSRequest()* function. The *coordinator* of the cluster, which is the current holder of the *intra* token, will also receive such a request. However, before granting the *intra* token to the requesting *application* process, the *coordinator* must first acquire the *inter* token by calling the *InterCSRequest()* function (line 9) of the *inter* algorithm. Therefore, upon receiving the *inter* token, the *coordinator* gives the *intra* token to the requesting *application* process by calling the *IntraCSRelease()* function (line 11).

A *coordinator* which holds the *inter* token must also treat the *inter* token requests received from the *inter* algorithm. However, it can only grant the *inter* token to another *coordinator* if it holds its local *intra* token too. Having the latter ensures it that no *application* process within its cluster is in the critical section. Thus, if the *coordinator* does not hold the *intra* token, it sends a request to its *intra* algorithm asking for it by calling the *IntraCSRequest()* function (line 16). Upon obtaining the *intra* token, the *coordinator* can give the *inter* token to the requesting *coordinator* by calling the *InterCSRelease()* function (line 18).



(a) Classical mutex automata (b) Composition coordinator automata

Figure 1: Mutual exclusion automatas

3.2 Coordinator Automata

In a classical mutual exclusion algorithm, a process can be in one of the three following states : requesting the critical section (REQ), not requesting it (NO_REQ), or in the critical section(CS),

as shown in figure 1.(a). If a process does not want to access the resource, its state is *NO_REQ*.

The behavior of a *coordinator* process can be summarized by a state automata. However, a *coordinator* process is in one of the above three states with regard to both algorithms. Therefore, in the automata of figure 1.(b), *Intra* and *Inter* refer to the *coordinator* state related to the *intra* algorithm and *inter* algorithm respectively. Furthermore, a *coordinator* has additional states with respect to the global state of the composition, which can be one of the following: OUT, IN, $WAIT\_FOR\_OUT$, $WAIT\_FOR\_IN$.

```
1  Coordinator Algorithm ()
2      IntraCSRequest()
3      /* Hold Intra CS */
4      while TRUE do
5          if ¬ IntraPendingRequest then
6              state ← OUT
7              Wait for IntraPendingRequest
8          state ← WAIT_FOR_IN
9          InterCSRequest()
10         /* Hold Inter CS */
11         IntraCSRelease()
12         if ¬ InterPendingRequest then
13             state ← IN
14             Wait for InterPendingRequest
15         state ← WAIT_FOR_OUT
16         IntraCSRequest()
17         /* Hold Intra CS */
18         InterCSRelease()
```

Figure 2: Coordinator algorithm

If the coordinator is in the OUT state, no local *application* process of its cluster has requested the CS. Thus, it holds the *intra* token (*Intra* = *CS*) and does not hold the *inter* token (*Inter* = *NO_REQ*).

When the *coordinator* is in the $WAIT\_FOR\_IN$ state, it means that there are one or more pending *intra* algorithm requests. It still holds the local *intra* token (*Intra* = *CS*) but is waiting for the *inter* token (*Inter* = *REQ*).

In the IN state, the coordinator holds the *inter* token (*Inter* = *CS*) but has granted the *intra* algorithm token (*Intra* = *NO_REQ*) to one of the *application* process of its cluster.

Finally, when the coordinator is in the $WAIT\_FOR\_OUT$ state, it still holds the *inter* token (*Inter*= *CS*) but it is requesting the *intra* token to the *intra* algorithm (*Intra* = *REQ*) in order to be able to satisfy an *inter* algorithm pending request.

It is worth remarking that only one coordinator can be either in IN or in $WAIT\_FOR\_OUT$ state at any given time. All the other nodes are either in OUT or in $WAIT\_FOR\_IN$ state.

4 Performance Results

Considering various application behaviors, this section presents some performance evaluation results aimed at comparing the efficiency of several mutual exclusion algorithm compositions.

4.1 Environment and Parameters

The evaluation performance experiments were conducted on Grid5000, a French large-scale grid experimental testbed[1]. Grid5000 comprises 17 clusters located in 9 different cities all over France. Whichever the cluster, every node has a Bi-Opteron CPU and 2GB of RAM. Clusters are connected by dedicated 10Gb/s bandwidth links.

Our experiments used 9 of the 17 clusters, each one with 20 nodes, located in a different city. Figure 3 presents the average latency between the clusters.

from \ to	orsay	grenoble	lyon	rennes	lille	nancy	toulouse	sophia	bordeaux
orsay	0.034	15.039	9.128	8.881	4.489	95.282	15.556	20.239	7.900
grenoble	14.976	0.066	3.293	15.269	12.954	13.246	10.582	9.904	16.288
lyon	9.136	3.309	0.026	12.672	10.377	10.634	7.956	7.289	10.078
rennes	8.913	15.258	12.617	0.059	11.269	11.654	19.911	19.224	8.114
lille	10.000	10.001	10.001	10.001	0.001	10.001	20.000	20.001	10.001
nancy	5.657	13.279	10.623	11.679	9.228	0.032	98.398	17.215	12.827
toulouse	15.547	10.586	7.934	19.888	19.102	17.886	0.043	14.540	3.131
sophia	20.332	9.889	7.254	19.215	16.811	17.238	14.529	0.051	10.629
bordeaux	7.925	16.338	10.043	8.129	10.845	12.795	3.150	10.640	0.045

Figure 3: Grid5000 RTT Latencies (average ms)

The mutual exclusion algorithms as well as the coordinator are written in C using UDP sockets. Each application process that runs on a single node executes 100 critical sections. Each of them

[1]Grid'5000 is an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https:// www.grid5000.fr)

lasts 10ms, which is the same order of magnitude as a data packet hop time between two clusters. Every experiment was executed 10 times and the presented results represent the average value.

An application behavior is characterized by:

- α: time taken by a node to execute the critical section;
- β: mean time interval between the release of the CS by a process and its next request.
- ρ: the ratio β/α, which expresses the frequency with which the critical section is requested.

We have developed several applications having **low**, **intermediate**, and **high** levels of parallelism.

Considering N as the total number of *application* processes (180 in our experiment), the three levels of parallelism can be expressed respectively by :

- **Low Parallelism**: $\rho \leq N$: An application where the majority of *application* processes request the critical section. Thus, almost all *coordinators* wait for the *inter* token in the *inter* algorithm. In other words, almost all clusters have one or more *application* processes in the *requesting* state.
- **Intermediate parallelism**: $N < \rho \leq 3N$: A parallel application where some sites compete to get the CS. Only some *coordinators* are in the *requesting* state with respect to the *inter* algorithm on the whole Grid i.e., just some clusters have one or more *application* processes in requesting the token.
- **High Parallelism**: $3N \leq \rho$: A highly parallel application where concurrent requests to the CS are rare. The whole number of requesting *application* processes is small and usually distributed over the grid. Hence, only one or a few clusters have one or more *application* processes in the *requesting* state regarding the *inter* algorithm.

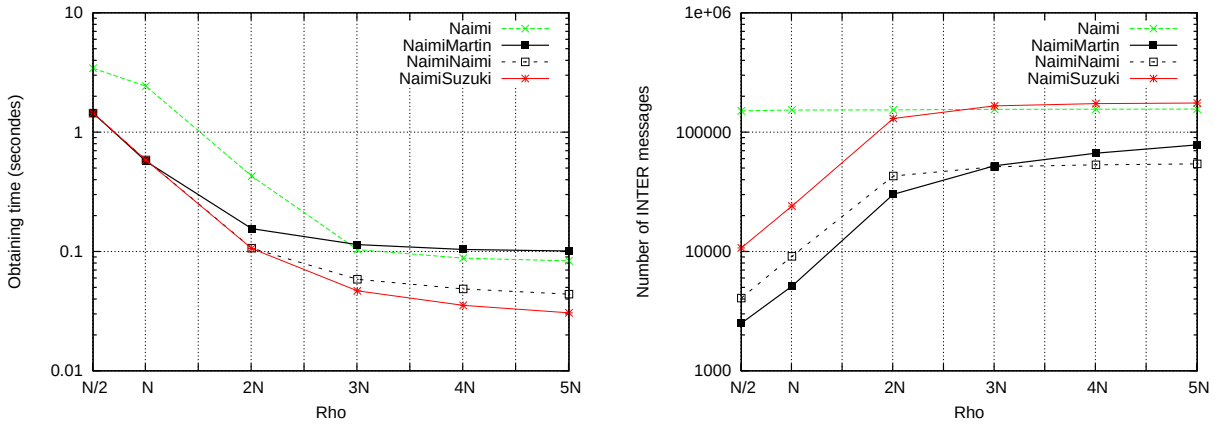
In order to evaluate our compositional approach, three metrics have been considered: the **obtaining time** i.e., the time between the moment a node requests the CS and the moment it gets it, the **number of sent messages**, and the **standard deviation** of the obtaining time.

For the sake of simplicity, we call the Naimi-Tréhel and Suzuki-Kasami algorithms respectively Naimi's and Suszuki's and for all figures of this section we have adopted the notation "*Intra algorithm-Inter Algorithm*" to denote a two level algorithm composition. For instance, "Naimi-Martin" denotes a composition where Naimi's is used as the *intra* algorithm of every cluster and

Martin's as the *inter* algorithm. Furthermore, the abscissae of the curves always represent the ρ parameter (degree of parallelism). Hence, when analyzing the curves the reader must keep in mind that when ρ increases, the number of processes that concurrently request the critical section decreases.

As we observed that the *inter* algorithm has a much stronger influence in the overall performance than the *intra* algorithm, the experiments of sections 4.3 and 4.4 have been performed by fixing the latter to Naimi's algorithm. Therefore, the variation of application processes *obtaining time* and number of *inter-cluster sent messages* is only due to the *inter* algorithm. The latter comprises the number of messages for delivering *inter* token requests plus the number of messages for granting the *inter* token.

The impact of the *intra* algorithm choice on the overall performance of our composition approach as well as the advantages of choosing Naimi's for the *intra* algorithm are explained in section 4.6.



(a) Obtaining time average

(b) Total sent inter-messages

Figure 4: Composition evaluation

4.2 Performance evaluation without composition

In figure 4.(b), we can see that, independently of ρ, the original Naimi-Tréhel always presents the same number of inter cluster sent messages ($\mathcal{O}(log(N))$). This constant behavior can be explained since the routing of both a CS request and a token granting message from a node does not depend on its location. A message is arbitrarily routed through nodes which are within the same cluster or belong to different clusters. On the other hand, when a compositional approach is used, inter cluster messages are managed by coordinators which gather token request messages from *application*

processes into just one *inter* token request. Hence, the number of inter cluster sent messages decreases when compared to the original algorithm, as we can observe in the same figure for all three algorithm compositions. Nevertheless, when applying our composition approach, the number of inter cluster sent messages is not constant but increases with ρ. Such a behavior is explained in section 4.4.

It is worth noting that if we consider the path of a token request in terms of clusters only, the original algorithm can present cycles while our approach does not, which also explains the reduced number of inter cluster sent messages.

In terms of *obtaining time*, a first remark is that for all curves the obtaining latency decreases with the decreasing of concurency, i.e. the reduction of the waiting queue size. The clustering of *intra* token requests has also an advantageous impact on the *obtaining time* when compared to the original algorithm, as we can observe in figure 4.(a). Such a benefit depends on ρ and it is analyzed in the next section.

4.3 Composition approach: obtaining time of application processes

We consider the following notation:

- T : average message delay for transmitting a message between two coordinators;
- T_{req} : average message delay for transmitting an *inter* token request message from a coordinator to the coordinator that will grant it the token.
- T_{token} : average message delay for granting the token between the current coordinator token holder and the requesting coordinator;
- T_{pendCS}: average delay for satisfying all the current pending *inter* token requests before satisfying the studied *inter* token request.

In highly parallel applications where there is almost no concurrency among accesses to the shared resource ($\rho \geq 3N$), the *obtaining time* of a coordinator comprises the request message delay T_{req} plus the token message delay T_{token}. However, in applications with high concurrency for accessing a shared resource, as in low parallel applications ($\rho \leq N$), a coordinator must wait for all the other pending CS requests to be satisfied before getting the token. This delay, which we called T_{pendCS}, is usually higher than that for sending the request T_{req} and completely overlaps

T_{req}. Therefore, the *obtaining time* of a coordinator consists of T_{pendCS} plus the token message delay T_{token}. This explains why the *obtaining time* tends to be higher when $\rho \leq N$, since in this case there are always many application processes in the *requesting* state, and quite short when $\rho \geq 3N$, since the number of waiting coordinators for the token is small. Such a behavior can be observed in figure 4.(a).

Low parallel application : We did not observe any significant difference with respect to the average *obtaining time* of all three algorithms of figure 4.(a) for $\rho \leq N$. As explained above, in this case, the *obtaining time* of a coordinator is equal to T_{pendCS} plus T_{token}. T_{pendCS} is the same for all three *inter* algorithms while T_{token} is reduced to T in the case of Naimi's (a send to the *next* node) and Suzuki's (a send to the first node of Q) algorithms. In Martin's algorithm, the current token holder grants the token to its predecessor in the ring. However, as this node has a very high probability of having requested the token too, the token granting delay also takes one message ($Ttoken = T$), as in the other two algorithms.

As concurrency among accesses to the shared resource is quite high in low level parallel applications, the *obtaining time* does not vary much. Such a behavior will be explained in section 4.5, where the standard deviation of the *obtaining time* is discussed.

Intermediate parallel application : A first remark is that Naimi-Naimi's *obtaining time* is comparable to Naimi-Suzuki's (cf. figure 4.(a) for $N < \rho \leq 3N$) whereas Naimi-Martin's is slightly higher. This is explained by the fact that when using Martin's as the *inter* algorithm, there are some coordinators waiting for the *inter* token which implies that their T_{req} can still be covered up by their T_{pendCS}. Thus, similarly to low level parallel applications, the main factor for the *obtaining time* is T_{token}. Suzuki's and Naimi's *inter* algorithm invariably need only one message, whose delay is T while Martin's needs more than one message in average. For Martin's, the smaller is the number of pending requests, the lower is the probability that a second coordinator has also requested the token and the higher is the probability that T_{token} increases. Therefore, Martin's algorithm is not suitable as the *inter* algorithm for this type of application.

Highly parallel application : In the case of applications with high degree of parallelism, CS requests from *application* processes are quite sparse. As explained above, in such applications, the

obtaining time of a coordinator comprises the requesting message delay T_{req} plus the *inter* token message delay T_{token}. As the application does not present much concurrency, T_{token} is equal to T to both Naimi's and Suzuki's algorithms while for Martin's it is equal to $N/2 * T$.

In terms of T_{req}, the most effective *inter* algorithm is Suzuki's, since a CS requesting is performed by a single message sent in parallel to each coordinator, taking just T. As Naimi's uses a tree to route requests, the average delay for a request travel is $log(N) * T$ between coordinator nodes. The less suitable algorithm is Martin's. Since the number of requesting coordinator tends to zero, a CS request tends to travel along the ring an average of $N/2$ successive hops, which implies a T_{req} of $N/2 * T$. Hence, the impact of T_{req} in the *obtaining* time of the three algorithms explains why Suzuki's presents the lowest *obtaining time* and Martin's the highest one as observed in figure 4.(a) for $\rho \geq 3N$,

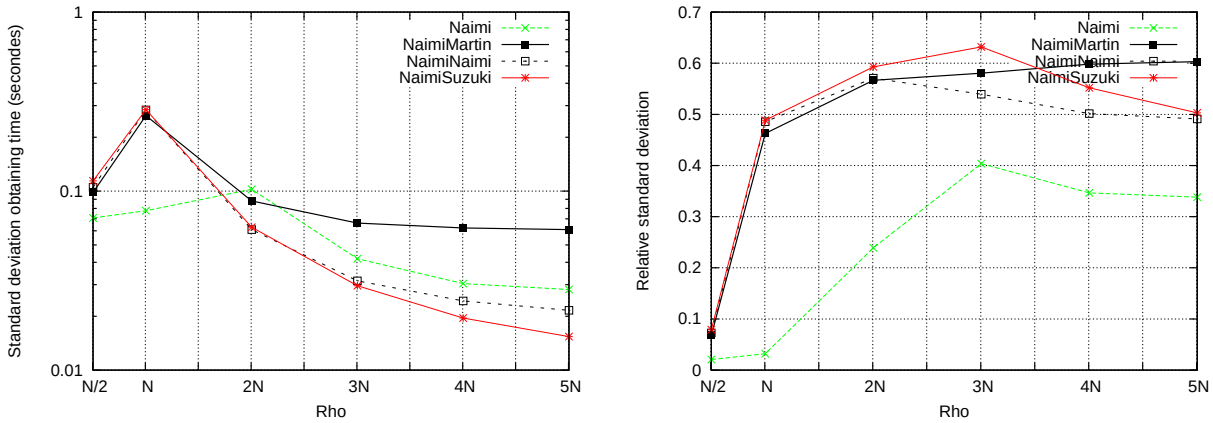
4.4 Composition approach : number of inter-cluster sent messages

As said in section 4.2, our composition approach in general reduces the number of inter cluster sent messages when compared to the original algorithm. When ρ is small, there is a lot of concurrent CS requests from *application* processes of the same cluster which will result in a single *inter* token request by the coordinator of the cluster in question. In this particular case, we should emphasize the advantage of using the Naimi-Naimi's algorithm composition compared to the original one. On the other hand, when concurrency for the CS decreases, the gathering of *intra* CS requests by a coordinator decreases as well which implies in more *inter* cluster requests.

In the case of Suzuki's and Naimi's inter algorithms, the number of sent messages per *inter* token request of a coordinator consists of one message for the grant of the *inter* token and respectively N messages and $\mathcal{O}(log(N))$ messages for *inter* token request. Hence, in terms of number of inter cluster sent messages, Naimi's is more efficient than Suzuki's, which can be observed in the curves Naimi-Naimi and Naimi-Suzuki of figure 4.(b). However, in the case of Martin's algorithm, that number depends on ρ. For low level parallel applications ($\rho \leq N$), the probability of having all coordinators requesting the *inter* token at a given time is high. Therefore, the grant of the *inter* token takes just one message as well as a coordinator request since a second coordinator which is in a *requesting* state does not forward a request, as explained in 2.1. When the parallelism of the

application increases, the number of inter-cluster sent messages per *inter* token request increases as well. This growth can be explained since the probability that some coordinator requests the *inter* token decreases. Thus, the number of hops of a request message increases proportionally, which generates more messages. In a highly parallel application, a token request in Martin's generates $N/2$ messages and the grant of the token generates $N/2$ messages. By comparing Naimi-Martin and Naimi-Naimi curves of figure 4.(b), we can observe that for highly parallel applications ($\rho \geq 3N$), the number of inter cluster messages sent by Martin's is slightly higher than Naimi's.

4.5 Standard deviation



(a) standard deviation (σ) (b) relative standard deviation ($\sigma/\bar{x}$)

Figure 5: Obtaining time standard deviation

In order to analyse more precisely the variation of the obtaining time, its standard deviation σ has been measured, as shown in figure 5.(a). A first remark when observing this figure is that σ is in fact quite significant for all ρ values compared to the average CS time. This is due to the communication heterogeneity of the Grid platform: inter cluster latencies are much higher than intra cluster ones and the former are not uniform with regard to two different clusters, as described in figure 3.

To measure the importance of σ and to void the side effects of the average *obtaining time* variations, we choose to study the relative deviation time $\sigma_r = (\sigma/\bar{x})$, which is the ratio of the standard deviation σ to the *obtaining time* average $\bar{x}$ - see figure 5.(b).

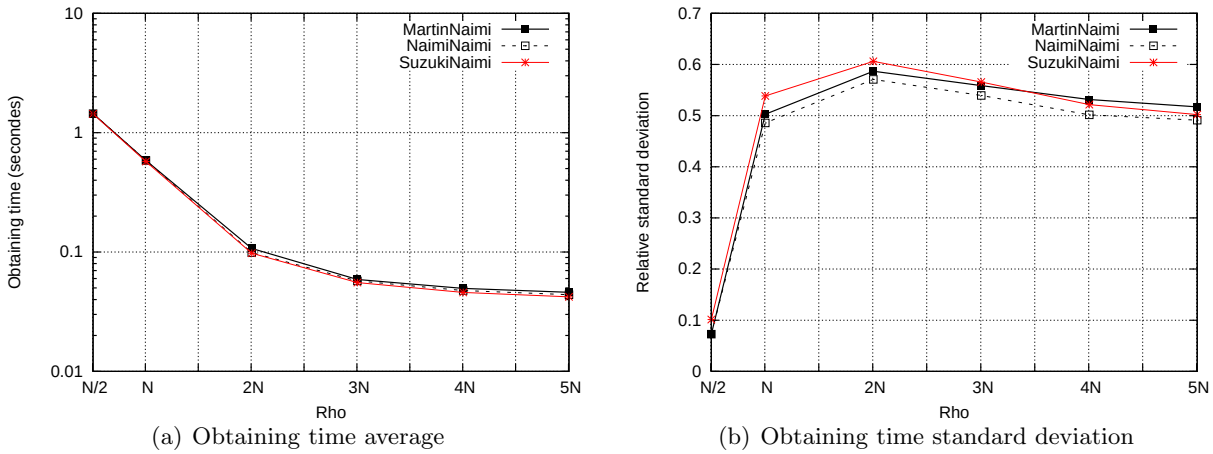
A first note is that the original Naimi's algorithm relative deviation σ_r is always smaller than that of any composition of algorithms. This happens because in the case of Naimi's, the path

covered by the token is independant of the actual token position, as explained in the section 4.2. However, in our approach, a request can have one of the following two delays: a very short one when then token is already in the cluster of the requesting node, and a long one when the token is not in the same cluster.

All curves of the figure 5.(b) have the same form : a significant increase for the lower values of ρ and then a stability phase. This growth of σ_r can be explained by two phenomena : the overlapping of the requesting trip time (T_{req}) by the process time of the requesting queue and the sequential ordering due to the extreme number of requests (for $\rho = N/2$).

With respect to the difference between the compositions curves, we can note that they are equivalent for lower values of ρ. For the intermediate parallel level ($N < \rho \leq 3N$), Naimi-Martin's has the worst absolute standard deviation due to its logical ring structure. While, Naimi-Suzuki's and Naimi-Naimi's present a better absolute standard deviation. However Naimi-Suzuki exhibits a better relative standard deviation. For $\rho > 3N$, Naimi-Suzuki has the smallest σ as show in figure 5.(a).

4.6 Intra algorithm choice



(a) Obtaining time average

(b) Obtaining time standard deviation

Figure 6: Intra Algorithm

We have carried out several experiments aiming at choosing the best *intra algorithm* with respect to the behavior of the applications. In order to make clear figure 6, we just show the curves when the *inter* algorithm is fixed to Naimi's. Experiments with the other two algorithms have presented the same behavior.

In terms of the number of inter cluster messages, all algorithms have an acceptable local overhead. One could argue that since Suzuki's algorithm sends a much higher number of request messages per critical section than the other two algorithms, it might be not chosen as the *intra* algorithm. However, as nodes within a cluster are linked by a LAN, a multicast primitive could be used to diffuse the request which will significantly reduce the number of sent messages.

Concerning the *obtaining time* (figure 6.(a)), all algorithms present almost the same curve, independently of ρ with a slight advantage for Suzuki-Naimi's. On the other hand, the latter has a weaker regularity (figure 6.(b)) than Naimi-Naimi's. This difference is due to the lack of fairness of Suzuki's algorithm when appending nodes to the token queue Q since it does not consider the arrival time of the requests.

Therefore, Naimi's algorithm has been chosen in the expermients as the intra algorithm in the experiments of the previous sections because of its regularity and performance.

4.7 Choosing the best composition of mutual exclusion algorithms

Based on the evaluation performance results presented in the previous sections, we can pointed out that the choice of a good *inter* mutual exclusion algorithm depends on the application behavior and the trade-off between the *obtaining time* and the total *number of sent messages*.

For low parallel applications ($\rho < N$), Martin's *inter* algorithm presents the same *obtaining time* as Suzuki's and Naimi's but it sends far less inter cluster messages than the other two. Hence, when almost all *application* processes request the CS, Martin's is the most effective.

In the case of intermediate parallel applications ($N \leq \rho < 3N$), Naimi's and Suzuki's are equivalent in terms of the *obtaining time* and regularity but Suzuki's presents a higher message overhead. Therefore, Naimi's is the *inter* algorithm best choice in this case.

Finally, for highly parallel applications ($\rho \geq 3N$), Suzuki's algorithm generates much more *inter* cluster messages than the other two algorithms due to the broadcast of request. However, since the *obtaining time* of Suzuki's is much smaller than the *obtaining time* of the other two algorithms, Suzuki's is a good choice when the application is massively parallel.

It is worth remarking that another feature of our composition approach is to be more scalable than the original algorithm. For instance, considering the total number of nodes N of a Grid, the

original Suzuki's algorithm presents poor scalability since it requires N message per critical section and the size of messages exchanged is proportional to N. However, the "Suzuki-Suzuki" algorithm composition scales much better since the number of messages required per critical section in the *inter* algorithm and *intra* algorithm is respectively the number of clusters and the number of nodes within a cluster. As explained in section 4.4, the composition "Naimi-Naimi" also presents better scalability than the original Naimi's.

5 Related work

Several studies have propose to adapt existing mutual exclusion algorithms to a hierarchical scheme. In Mueller [11], the author presents an extension to Naimi-Tréhel's algorithm, introducing the concept of priority in it. A token request is associated with a priority and the algorithm first satisfies the requests with higher priority. Bertier et al. [3] adopt a similar strategy based on the Naimi-Tréhel's algorithm which treats intra-cluster requests before inter-cluster ones.

Some approaches have adapted the mutual exclusion mechanism of a DSM system to the latency hierarchy of an interconnection of clusters. In [1] or [2], the authors propose a solution based on a centralized token-based mutual exclusion protocol.

Finally, several authors have propose hierarchical approaches for combining different mutual exclusion algorithms. Housni et al. [6] and Chang et al. [4]'s mutual exclusion algorithms gather nodes into groups. Both articles basically consider hybrid approaches where the algorithm for intra-group requests is different from the inter-group one. In Housni et al. [6], sites with the same priority are gathered at the same group. Raymond's tree-based token algorithm [14] is used inside a group, while Ricart-Agrawala [15] diffusion-based algorithm is used between groups. Chang et al.'s [4] hybrid algorithm applies diffusion-based algorithms at both levels: Singhal's algorithm [16] locally, and Maekawa's algorithm [9] between groups. The former uses a dynamic information structure while the latter is based on a voting approach. Similarly, Omara et al. [13]'s solution is a hybrid of Maekawa's algorithm and Singhal's modified algorithm which provides fairness. In Madhuram et al. [8], the authors also present a two level algorithm where the centralized approach is used at lower level and Ricard-Agrawala at the higher level. Erciyes [5] proposes an approach close to ours based on a ring of clusters. Each node in the ring represents a cluster of nodes. The

author then adapts Ricart-Agrawal to this architecture.

Our work is close to these hybrid algorithms when gathering machines into groups (clusters in our case) which has in influence in the conception of the algorithm. However, such algorithms do not consider differences in communication latency as the main reason for grouping machines. Furthermore, our approach is more generic as it tries to chose the good combination of algorithms according to the application's behavior comparing different mutual exclusion algorithm compositions on top of Grid.

6 Conclusions

In this paper, we have proposed a new approach for the composition of mutual exclusion algorithms for Grid environments where application processes are spread over several clusters interconnected by long distance links. Such a composition is totally transparent to the application and any classical token-based algorithm can be chosen as both inter and intra algorithms. Our two-level approach is scalable and can be easily extended to multiple levels of algorithm hierarchy which render it extremely suitable for large-scale systems.

Performance evaluation results from experiments conducted on a real national wide Grid show that the degree of parallelism of an application has an impact on the choice of the *inter* algorithm. Such a choice depends on the logical topology that the algorithm takes into account for transmitting the token request. To this end, Martin's, Naimi-Tréhel's and Suzuki-Kasami's algorithms which respectively consider a ring, a tree and a complete graph topology, where used as the *inter* algorithm in our tests. When the system is stressed (the rate of CS request is high and there are requests in all clusters), a ring topology is the most effective; when the rate is lower (ie., the application exhibits a higher degree of parallelism) both the tree and the complete graph configurations are more efficient since they reduce the number of hops of CS request messages. Such results prove that our composition approach provides a framework for easily choosing the best two algorithms combination. Therefore, we propose as a future work, a dynamic and adaptive composition scheme where the the *inter* algorithm will be replaced according to the application behavior.

References

[1] G. Antoniu, L. Bouge, and S. Lacour. Making a DSM consistency protocol hierarchy-aware: an efficient synchronization scheme. In *Proceedings of the Workshop on Distributed Shared Memory on Clusters*, pages 516–521, May 2003.

[2] L. Arantes, P. Sens, and B. Folliot. An effective logical cache for a clustered LRC-based DSM system. *Cluster Computing*, 5(1):19–31, 2002.

[3] M. Bertier, L. Arantes, and P. Sens. Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. *JPDC: Journal of Parallel and Distributed Computing*, 66:128–144, 2006.

[4] I. Chang, M. Singhal, and M. Liu. A hybrid approach to mutual exclusion for distributed system. In *Proc. of the 14th IEEE Annual International Computer Software and Applications Conference*, pages 289–294, 1990.

[5] K. Erciyes. Distributed mutual exclusion algorithms on a ring of clusters. In *Computational Science and Its Applications 2004, International Conference, Italy, May 14-17, 2004, Proceedings, Part III*, volume 3045 of *LNCS*, pages 518–527. Springer, 2004.

[6] A. Housni and M. Tréhel. Distributed mutual exclusion by groups based on token and permission. In *Proceedings of the ACM/IEEE International Conference on Computer Systems and Applications*, pages 26–29, June 2001.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[8] Madhuram and Kumar. A hybrid approach for mutual exclusion in distributed computing systems. In *SPDP: 6th IEEE Symposium on Parallel and Distributed Processing*, 1994.

[9] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[10] Alain J. Martin. Distributed mutual exclusion on a ring of processes. *Sci. Comput. Program.*, 5(3):265–276, 1985.

[11] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *Proceedings of 12th Int. Parallel Proc. Symposium & 9th Symp. on Parallel and Distr. Processing*, pages 791–795, March 1998.

[12] M. Naimi, M. Trehel, and A. Arnold. A log (N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.

[13] F. Omara and M. Nabil. A new hybrid algorithm for the mutual exclusion problem in the distributed systems. *International Journal of Intelligent Computing and Information Sciences*, 2(2):94–105, 2002.

[14] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *Trans. on Computer Systems*, 7(1):61–77, 1989.

[15] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM (CACM)*, 24, 1981.

[16] M. Singhal. A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel Distributed Systems*, 3(1):121–125, 1992.

[17] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *Trans. on Computer Systems*, 3(4):344–349, 1985.