Building Effective Mutual Exclusion Services for Grids

Julien Sopena, Luciana Arantes, Fabrice Legond-Aubry, 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,luciana.arantes,fabrice.legond-aubry,pierre.sens]@lip6.fr

Abstract

Taking into account the intrinsic heterogeneity of communication latency of Grid environments, we propose in this article a composition approach that enables to build mutual exclusion services for Grids. By using our approach, different intra and inter cluster token-based mutual exclusion algorithms can be combined easily. Performance evaluation tests were conducted on the French national grid testbed called Grid'5000, whose results show that our approach is effective and that the choice of the most suitable inter cluster algorithm depends on the behavior of the application.

Keywords: Grid, distributed mutual exclusion algorithm, mutual exclusion service, composition, performance evaluation, Grid'5000.

1 Introduction

Computing grids federate geographically distributed computational resources in order to make them cooperate and act a single massive computer. Such a platform is extremely interesting for distributed and/or parallel applications that require a lot of computational power, data storage or access to resources that are not available locally. A Grid is usually composed of a large number of machines gathered into small groups called clusters. As such clusters are usually spread out over different sites, cities or even countries, communication in a Grid environment is intrinsically heterogeneous. Nodes within one cluster are linked by local networks (LAN) whereas clusters are connected by wide area network (WAN) links. Therefore, Grids present a hierarchy of communication delays where the cost of sending a message between nodes of different clusters is much higher than sending the same message between nodes within the same cluster.

As Grid resources can be shared, applications that run on top of a Grid usually require that their processes get exclusive access to one or more of these shared resources by executing a segment of code called the critical section (*CS*). Thus, a Grid service that provides mutual exclusion is extremely important for such applications. Mutual exclusion is in fact one of the basic blocks for building distributed systems which ensures that exactly one process can execute the critical section at any given time (*safety* property) and that all critical section requests will eventually be satisfied (*liveness* property). It is worth also mentioning that the performance of a mutual exclusion service can have a major impact on the overall performance of Grid applications.

Mutual exclusion algorithms can be divided into two families: *permission-based* (e.g. Lamport [8], Ricart-Agrawala [16], Maekawa [10]) and *token-based* (Suzuki-Kazami [21], Raymond [15], Naimi-Tréhel [13], Martin [11]). 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 [10]). In the second group of algorithms, a system-wide unique token is shared among all nodes, and its possession gives a node the exclusive right to execute a critical section. Token-based algorithms present different solutions for the transmission and control of critical section requests of processes. Each solution is usually expressed by a logical topology that defines the paths followed by critical section request messages which might be completely different from the physical network topology.

With regard to the number of nodes, token-based mutual exclusion algorithms present an average message traffic which is lower than that of permission-based ones. Thus, they are more suitable for controlling concurrent access to shared resources of Grids whose number of nodes is often very large. However, existing token-based algorithms still have intrinsic limits and do not take into account the above-mentioned hierarchy of communication latencies. We propose in this article a generic composition approach which enables the combination of any two token-based mutual exclusion algorithms: one at *intra-cluster* level and a second one at *inter-cluster*. By using our composition mechanism, we can provide effective mutual exclusion Grid services that take into account communication latency heterogeneity and which can be easily deployed by just "plugging in" token-based algorithms on each levels of the hierarchy. Furthermore, the extensive performance evaluation tests that we have conducted on Grid'5000 [1] show 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 for the shared resource. In other words, the parallelism degree of the application has an impact on the choice of *inter-cluster* algorithms. Using our compositional approach and relying on both the application behavior and our performance study results, we can provide an efficient token-based mutual exclusion service dedicated to specific type of Grid applications.

The remainder of this paper is organized as follows. In section 2, we describe our compositional approach for mutual exclusion algorithms and an example of execution. Performance evaluation results are presented in section 3. Related work is studied in section 4. The last section concludes our work.

2 Composition approach to mutual exclusion algorithms

Similarly to a classical mutual exclusion algorithm, a mutual exclusion Grid service should offer two operations: *CS_Request()*, which allows a process to request exclusive access to a shared resource, and *CS_Release()* called by the same process when it wants to release the resource. However, in order to be effective, such a service must tolerate heterogeneous network latencies. Our approach does it by using a hierarchy of mutual exclusion algorithms: a per cluster token-based mutual exclusion algorithm instance that controls critical section requests for processes within the same cluster and another algorithm instance that controls *inter-cluster* requests for the token. The former is called the *intra* algorithm while the latter is called the *inter* algorithm and their executions are clearly separated. Furthermore, an *intra* algorithm instance of a cluster runs independently from a second *intra* algorithm instance. Thus, a process obtains access to the shared resource and later releases it by calling the above two operations which belong to the *intra* algorithm instance of its cluster. Another important advantage of our approach is that the chosen algorithms of both layers do not need to be modified. Hence, it is very simple to offer different implementations of a mutual exclusion service by just assembling multiple mutual exclusion algorithms.

Without loss of generality, we will consider in the rest of this paper that there is just one

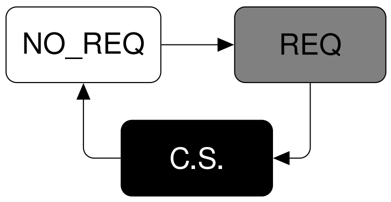
[1]Grid'5000 is an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https:// www.grid5000.fr)

application which is composed of a set of processes. There is one process per node and we call it an *application* process.

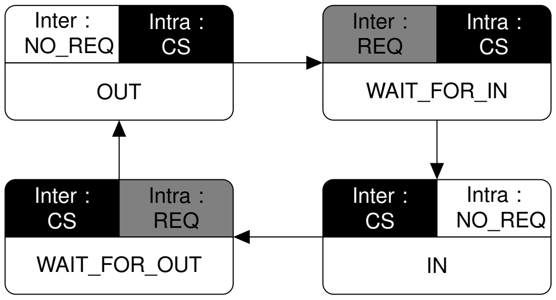
When an *application* process wants to access the shared resource, it calls the operation *CS_Request()* of the *intra* algorithm. Upon getting the *intra* token, the process executes the critical section. After executing it, the process calls the operation *CS_Release()* of the same *intra* algorithm to release it. However, since one *intra* algorithm instance runs on each cluster, several processes could simultaneously access the critical section which would violate the safety property. In order to overcome this problem, we have introduced a special node within each cluster, called the *coordinator*, which ensures the safety property at a Grid wide level. The *inter* algorithm runs on top of the *coordinators* and allows a *coordinator* to request access to the shared resource on behalf of an *application* node within its 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. Nevertheless, even if the *intra* algorithm sees a *coordinator* as an *application* process, the *coordinator* neither takes part in the application's execution nor requests access to the shared resource for itself.

The *coordinator* also uses the *CS_Request()* and the *CS_Release()* operations offered by the *inter* algorithm. However, a *coordinator* being in critical section for the *inter* algorithm instance means that one of the *application* processes of its cluster can access the resource. It is considered to be in the critical section by the other *coordinators*.

Each *intra* algorithm instance controls an *intra* token while the *inter* algorithm instances controls an *inter* token. For every shared resource, there is one *intra* token per cluster but a single *inter* token for the whole system of which only the *coordinators* are aware. Therefore, holding the *intra* token of its cluster is sufficient and necessary for an *application* process to enter the CS since the local *intra* algorithm instance ensures that no other local *application* node of the cluster has the *intra* token. Besides, considering the hierarchical composition of algorithms, our solution must also guarantee that no other *application* process of the other clusters are also in the critical section by holding the *intra* token of their own instance (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 letting one of its *application* processes to execute the CS. This property can be asserted by the possession of the *inter* token by a *coordinator*.



(a) Classical mutual exclusion



(b) Coordinator

Figure 1: Mutual exclusion automatas

2.1 Coordinator Processes

The guiding principle of our approach is represented by the automata of figure 1.(b), which describes the behavior of a coordinator process. 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). A *coordinator* process can also find itself in one of these three states likewise the classical mutual exclusion, but 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 instance and *inter* algorithm instance respectively. Moreover, a *coordinator* has additional states with respect to the *global state* of the composition, which can be one of the following: OUT, IN, $WAIT\_FOR\_OUT$, $WAIT\_FOR\_IN$.

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*) but it does not hold the *inter* token (*Inter = NO_REQ*). When an *application* process wants to enter the critical section, it sends a request to the processes of its cluster by calling the *CS_Request()* operation of the *intra* algorithm. The *coordinator* of the cluster, which is the current holder of the *intra* token, will also receive such a request. Upon receiving it, the *coordinator* holder changes its global state to $WAIT\_FOR\_IN$. However, a *coordinator* can only grant the *intra* token to a requesting *application* process of its cluster if it holds the *inter* token too. To this end, it calls the *CS_Request()* operation of the *inter* algorithm in order to request the *inter* token. Therefore, when the *coordinator* finds itself in the $WAIT\_FOR\_IN$ global state, there are one or more pending *intra* algorithm requests, the *coordinator* still holds the local *intra* token (*Intra = CS*) but it is waiting for the *inter* token delivery (*Inter = REQ*).

The coordinator state changes to the IN global state when it receives the *inter* token. It then grants the *intra* token to the requesting *application* process by calling the *CS_Release()* operation of the *intra* algorithm. Thus, the coordinator holds the *inter* token (*Inter = CS*) but has given the *intra* algorithm token (*Intra = NO_REQ*) to one of the *application* processes of its cluster.

A *coordinator* which holds the *inter* token must also treat *inter* token requests received from the other coordinators. However, it can only grant the *inter* token if it also holds its local *intra* token. Holding this token ensures that there is no *application* process within its cluster in the critical section. When another *coordinator* requests the *inter* token, but the current holder of it does not keep both tokens, the latter sends a request to its *intra* algorithm asking for the *intra* token. Its global state triggers to $WAIT\_FOR\_OUT$ *coordinator*, i.e., the *coordinator* still holds the *inter* token (*Inter = CS*) but it is waiting for the *intra* token (*Intra = REQ*) in order to be able to satisfy the *inter* algorithm pending requests. Upon obtaining the *intra* token, the *coordinator* can grant the *inter* token to the requesting *coordinator* by calling the *CS_Release()* operation of the *inter* algorithm. It returns to the OUT state where it holds the *intra* token (*Intra = CS*) but not the *inter* token (*Inter = NO_REQ*).

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

Figure 2 shows the algorithm of the *coordinator* corresponding to the automata which we have just described. The *mutexState* variable corresponds to both the *Inter* and *Intra* states of the automata and the variable *state* refers to the *global state* of the automata. Notice that the body of *CS_Request* (line 20) and *CS_Release* (line 25) functions depends on the chosen token-based algorithm, i.e., how the token is requested and granted respectively. The *pendingRequest( )* is a trap callback function inserted in the code of both *intra* and *inter* mutual exclusion algorithms which indicates if there are token requests of the respective level waiting to be satisfied.

```
Coordinator Algorithm ()
    intra.CS_Request()
    /* Holds intra-token CS */
    while TRUE do
        if ¬ intra.PendingRequest() then
            state ← OUT
            Wait for intra.PendingRequest()
        state ← WAIT_FOR_IN
        inter.CS_Request()
        /* Holds inter-token. CS */
        intra.CS_Release()
        if ¬ inter.PendingRequest() then
            state ← IN
            Wait for inter.PendingRequest()
        state ← WAIT_FOR_OUT
        intra.CS_Request()
        /* Holds intra-token CS */
        inter.CS_Release()

CS_Request ()
    ...
    mutexState ← REQ
    Wait for Token
    mutexState ← CS

CS_Release ()
    ...
    mutexState ← NO_REQ

pendingRequest ()
    return $\begin{cases} TRUE & \text{if } \exists \text{ pending request} \\ FALSE & \text{otherwise} \end{cases}$
```

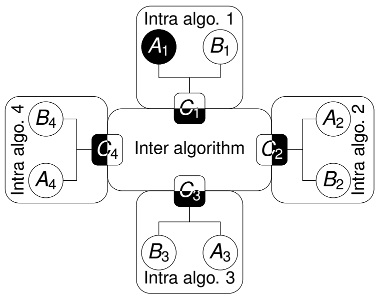
Figure 2: Coordinator algorithm

2.2 Example of Execution

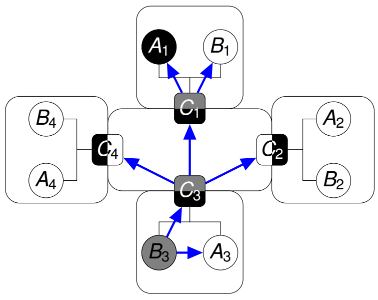
Figure 3 shows an execution of our hierarchical mutual exclusion approach. It consists of a Grid composed of four three-node clusters. Each cluster i comprises two *application* nodes, A_i and B_i, as well as the coordinator C_i. We consider a token-based mutual exclusion algorithm based on broadcast of requests, such as Suzuki and Kasami's [21] algorithms, for both the *intra* and *inter* algorithms. In Suzuki and Kasami's algorithm, a node that wants to execute a critical section broadcasts a request to all the participants of the algorithm (*CS_Request ()*). Upon receiving the token, the node can execute the critical section. When it leaves the CS (*CS_Release()*), it grants the token to the first node from which it had received a request.

As we can observe in Figure 3(a), at the beginning of the execution, node A_1 keeps the *intra* token of cluster 1 and the coordinator C_1 keeps the *inter* token. The *intra* tokens of clusters 2, 3, and 4 are held by their respective coordinators C_2, C_3, and C_4.

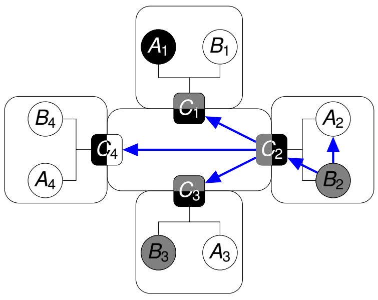
Suppose that node B_3, which is in the *OUT* state, wants to execute the CS. It then calls the *CS_Request* (see Figure 3(b)). Thus, a request is broadcast to all the nodes of its own cluster. Upon receiving the request, the coordinator C_3 calls the *CS_Request ()* of the *inter* algorithm since it does not hold the *inter* token which broadcasts a request to all coordinators. However, only the holder of the *inter* token, the *coordinator* C_1, will forward such a request to the *application* nodes of its cluster by calling the *CS_Request ()* operation of its *intra* algorithm. The global state of C_3 and C_1 change to $WAIT\_FOR\_IN$ and $WAIT\_FOR\_OUT$ respectively. We can already observe at this point the advantage of our hierarchical approach in terms of number of messages when compared to the original algorithm. The hierarchy structure of our approach has limited the number of *inter* cluster request messages to three. Furthermore, the request was not forwarded to the nodes of clusters 2 and 4, which is coherent with the automata: since coordinators C_2 and C_4 are both in the *OUT* state, no *application* node of these clusters finds itself in critical section and it is worthless to send them the request.



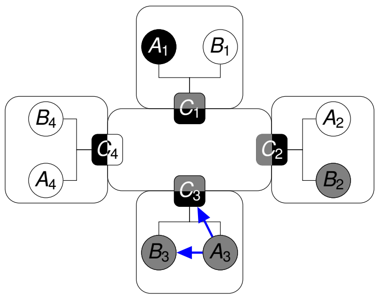
(a) Initial state: A_1 is in CS



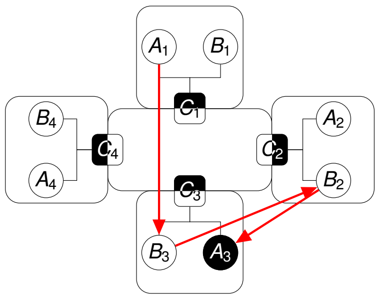
(b) Node B_3 requests the CS



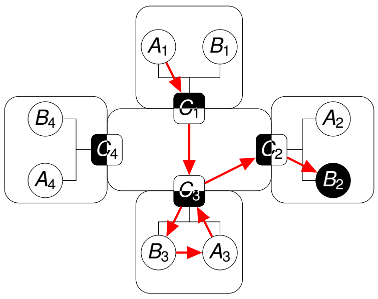
(c) Node B_2 requests the CS



(d) Node A_3 requests the CS



(e) Granting the token in the order of requests



(f) Natural reordering of requests

Figure 3: Example of execution

Before node B_3 gets the *intra* token, suppose that node B_2 calls the *CS_Request()* operation (figure 3(c)). A request message will be sent to all nodes of its cluster. Upon reception of this request, C_2 will broadcast it to all coordinators. However, the request will not be broadcast inside those clusters where there is no node executing the critical, i.e., clusters C_3 and C_4, and neither inside cluster C_1 since it is already in the $WAIT\_FOR\_OUT$ global state.

Finally, suppose that a second *application* node of cluster 3, A_3, also decides to call the *CS_Request ()* operation. We observe in Figure 3(d) that the request will not be broadcast at the *inter* level of our composition approach. This happens because the coordinator C_3 is in the $WAIT\_FOR\_IN$ global state which means that it has sent a previous *inter* request that has not been satisfied yet. Therefore, our composition approach naturally aggregates requests inside the same cluster which reduces the number of *inter* cluster messages.

Three critical section requests are now pending in the system: requests from B_3, B_2, and A_3. In a classical fair token-based mutual exclusion, the requests would be satisfied in the order of their receptions, i.e., first B_3's request, then B_2's request, and finally A_3's request. However, such an order forces the token to cross the clusters (figure 3(e)): from cluster 1 to cluster 3 when A_1 grants the token to B_3, from cluster 3 to cluster 2 when B_3 grants the token to B_2, and back to cluster 3 when B_2 grants the token to A_3. The round trip time for the token's travel between cluster 2 and 3 considerably increases the time for a process to get access to the critical section. Contrarily to the delay of a token request which can be overlapped by the duration of a critical section execution, no process can execute a critical section during a token transfer. Hence, reducing the delay of a token transfer has a direct impact on the overall performance of the algorithm since the time during which a node waits for the token will decrease as well.

In our composition approach, see Figure 3(f), the hierarchy of algorithms may naturally reorder the requests, giving priority to local requests over remote ones. When node A_1 ends the critical section, it calls *CS_Release()* and grants the *intra* token to C_1 which in its turn sends the *inter* token to C_3 by calling the *CS_Release()* of the *inter* algorithm. The coordinator C_3 then grants the *intra* token of cluster 3 to node B_3 (*CS_Release()* of the *intra* algorithm) and changes its global state from $WAIT\_FOR\_IN$ to IN. But at the same time, C_3 knows that there is a pending request from C_2, as explained above. Therefore, by calling the *CS_Request()* operation of the *intra* algorithm, C_3 broadcasts a request to all nodes of its cluster 3 in order to get the *intra* token. C_3 then triggers to $WAIT\_FOR\_OUT$ global state. Notice that when node B_3 concludes the execution of the critical section, there are two pending requests: one from A_3 and another from C_3. By keeping the order of reception of requests, B_3 grants the *intra* token to A_3 since it received A_3's requests before C_3's request. This means that without changing the original *intra* algorithm, the request of A_3 will be satisfied before the request of B_2 thus avoiding the mentioned round trip travel time of the *inter* token between clusters 2 and 3. Upon ending the execution of the critical section, A_3 grants the *intra* token to C_3, since requests are satisfied in order and C_3's is the next one. Having the *intra* token, C_3 can grant the *inter* token to C_2 by calling the *CS_Release()* operation of the *inter* algorithm. Such a token will allow the nodes of cluster 2 to execute the critical section, i.e., B_2 in this example.

3 Performance Evaluation

This section presents some performance evaluation results conducted on the French large-scale Grid'5000 [4]. Some preliminary results have been presented in [20] and [19].

Our performance tests aim at comparing the efficiency of some mutual exclusion algorithm compositions considering applications with different degrees of parallelism. The basic algorithms that we have chosen are **Martin**'s [11], **Naimi-Trehel**'s [13], and **Suzuki-Kasami**'s [21] which are well-known token-based algorithms found in the literature. They are briefly described below.

3.1 The chosen token-based algorithms

Martin's, **Naimi-Trehel**'s, and **Suzuki-Kasami**'s respectively consider a *ring*, a *tree*, and a *complete logical connection graph* for forwarding critical section requests. As they present distinct solutions for both transmitting requests and controlling the algorithm's liveness, they present different message complexity with regard to the number of nodes.

3.1.1 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 keep 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 sends the token to its predecessor. Each node between k and i will do the same. Therefore, the token will eventually reach i, which then can 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 plus $(x+1)$ 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.

3.1.2 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. This tree is called 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. This queue is called 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.

Upon 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. In opposition, 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 messages 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 the token granting delay T_{token} takes T.

3.1.3 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.

3.2 Experiment Parameters

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

Considering N as the total number of *application* processes (180 in our experiment), the three degrees 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* that request the CS.

- **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 with regard to the *inter* algorithm.

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 interval 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 T_{token}. Thus, the three metrics that we considered are: the **obtaining time** i.e., 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-Tréhel's algorithm is used as the *intra* algorithm of every cluster and Martin's algorithm as the *inter* algorithm.

3.3 Performance Results : Composition Study

In this section we present evaluation performance results by composing the three algorithms described in section 3.1 with different application behaviors.

The evaluation performance experiments were conducted on Grid'5000, a French large-scale grid experimental testbed. It 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 4 presents the average latency between the clusters.

to / from	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 4: Grid'5000 RTT Latencies (average ms)

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 3.3.1 and 3.3.2 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 3.3.4.

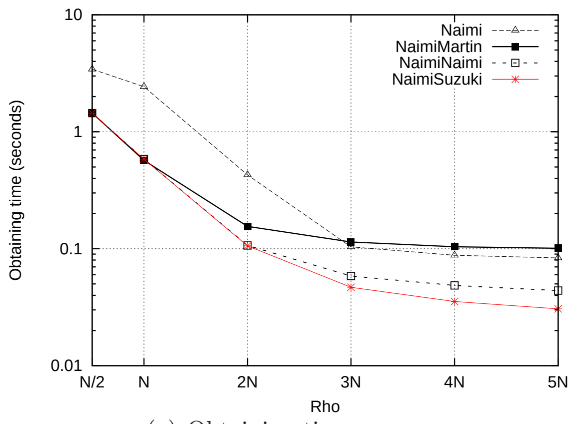
3.3.1 Obtaining time of application processes

We consider the following notations:

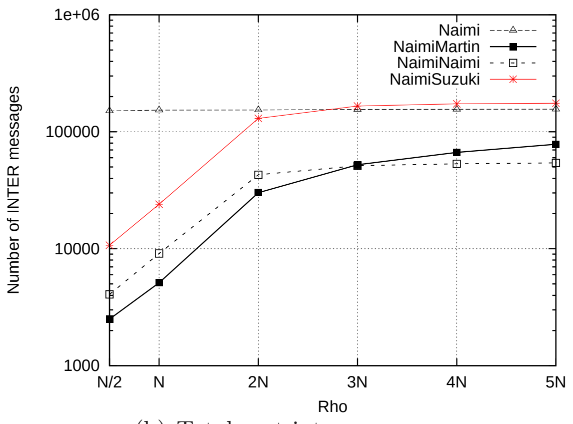
- T : average message delay for transmitting a message between two coordinators;
- T_{req} : average message delay for transmitting an *inter* token request message from a coordinator to the coordinator that will grant it the token.
- T_{token} : average message delay for granting the token between the 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 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 5.(a). Such a benefit depends on ρ .

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



(a) Obtaining time average



(b) Total sent inter-messages

Figure 5: Composition evaluation

the other pending CS requests to be satisfied before getting the token. This delay, which we called T_{pendCS}, is usually higher than the one 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 5.(a).

Low parallel application: We did not observe any significant difference with respect to the average *obtaining time* of all three algorithms of figure 5.(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 parallel applications, the *obtaining time* does not vary much. Such a behavior will be explained in section 3.3.3, 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 5.(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 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 5.(a) for $\rho \geq 3N$,

We can summarize our study about the *obtaining time* by the table of figure 3.3.1.

Parallelism / Composition	Low	Intermediate	High
Naimi-Suzuki	$T_{pendCS} + T$	$T_{pendCS} + T$	$T + T$
Naimi-Martin	$T_{pendCS} + T$	$T_{pendCS} + K * T$	$(N/2) * T + (N/2) * T$
Naimi-Naimi	$T_{pendCS} + T$	$T_{pendCS} + T$	$log(N) * T + T$

Figure 6: Average token obtaining time per composition

3.3.2 Number of inter-cluster sent messages

In figure 5.(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.

When ρ is small, there is a lot of concurrent CS requests from *application* processes of the same cluster which 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. But 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.

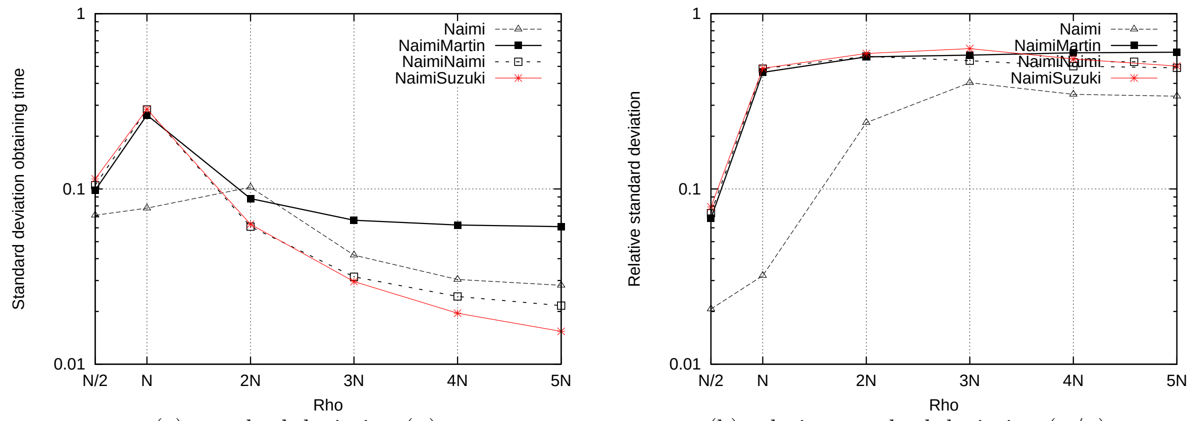
In the case of Suzuki's and Naimi's inter algorithms, the number of sent messages per *inter* token request of a coordinator consists of one message for the grant of the *inter* token and respectively N messages and $\mathcal{O}(log(N))$ messages for *inter* token request. Hence, in terms of number of *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 5.(b). However, in the case of Martin's algorithm, that number depends on ρ. For low 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 3.1.1. 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 5.(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.

The number of *inter* cluster messages can be summarized by the table of Figure 3.3.2.

Parallelism / Composition	Low	Intermediate	High
Naimi-Suzuki	$N+1$	$N+1$	$N+1$
Naimi-Martin	2	$2 < K < N$	$(N/2)+(N/2)$
Naimi-Naimi	$log(N)+1$	$log(N)+1$	$log(N)+1$

Figure 7: Average number of Inter Cluster message per composition

3.3.3 Standard deviation



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

Figure 8: Obtaining time standard deviation

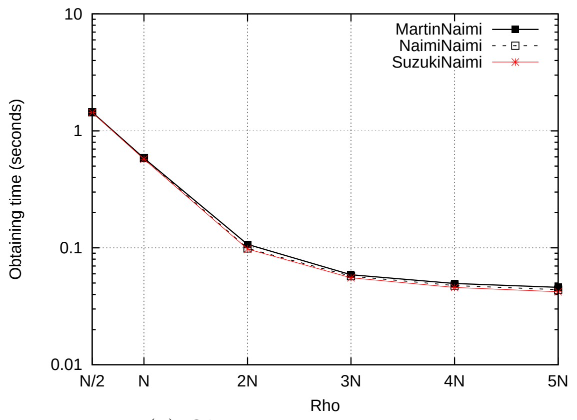
In order to analyse more precisely the variation of the *obtaining time*, its standard deviation σ has been measured, as shown in figure 8.(a). A first remark when observing this figure is that σ is in fact quite significant for all ρ values compared to the average CS time. This is due to the communication 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 4.

To measure the importance of σ and to evaluate 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 average *obtaining time* $\bar{x}$ - see figure 8.(b). 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 independent of the actual token position. However, in our approach, a request can have one of the following two delays: a very short one when the token is already in the cluster of the requesting node, and a long one when the token is not in the same cluster.

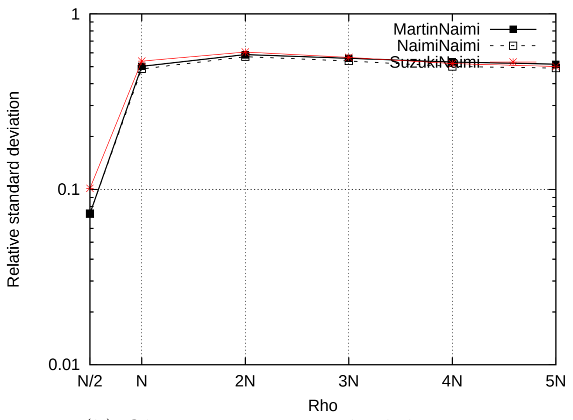
All curves of the figure 8.(b) have the same form: a significant growth for the lower values of ρ and then a stabilizing phase. This growth of σ_r can be explained by two phenomena : the overlapping of the requesting trip time (T_{req}) by the process time of the requesting queue and the sequential ordering 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 degree ($N < \rho \leq 3N$), 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 8.(a).

3.3.4 Intra algorithm choice



(a) Obtaining time average



(b) Obtaining time standard deviation

Figure 9: 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 not to load Figure 9, 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 *intra* 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 9.(a)), all algorithms present almost the same curve, independently of ρ with a slight advantage for Suzuki-Naimi. Still, the latter has a weaker regularity (figure 9.(b)) than Naimi-Naimi. 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, the regularity and performance of Naimi's algorithm justify choosing it as the intra algorithm in the experiments of the previous sections.

3.4 Performance Results: The Impact of the Grid Architecture

We are interested now in studying the influence of the Grid architecture in the performance of both the mutual exclusion algorithms and our composition approach. To this end, the number of nodes of the Grid was fixed to 120 but the number of clusters varied: 2, 3, 4, 6, 8, 12, 20, 30, 40, 60, and 120. However, since the number of clusters of Grid'5000 is limited to 17, the current experiments were conducted on a dedicated cluster of twenty-four machines Bi-Xeon 2.8 Ghz with 2GB of RAM where a Grid environment with 120 virtual nodes was emulated. For those configurations where the number of virtual clusters is greater than the number of available machines, nodes of the same virtual cluster run on the same machine. This approach prevents side effects of *intra* cluster communication.

The network latencies between clusters were emulated by using DUMMYNET [17]. DUMMYNET is a flexible tool which emulates bandwidth limitations, delays, and packet losses. Based on addresses and ports of both destination and source nodes, DUMMYNET intercepts packets, passing them through one or more queues and pipes, which simulate different message transmission configurations. Hence, for emulating several virtual clusters, every message exchanged between two virtual clusters goes through a dedicated machine, a P4 machine 3Ghz, which runs DUMMYNET. The mean of *Intra* cluster communication latency is equal to 0.5ms while *inter* cluster latency is equal to 20ms.

All the machines are connected by a 140 Gbits/s Ethernet switch. Notice that in order to validate our emulation platform, some of the tests described in the previous sections on top of GRID'5000 were conducted on it. Results were very close.

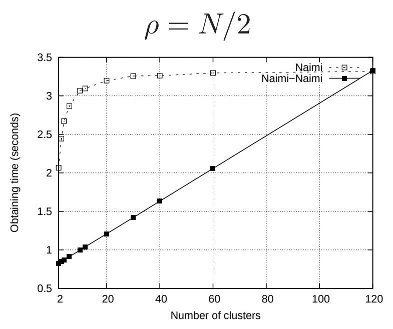
We have fixed the composition to *Naimi-Naimi*. Our choice can be explained based on the results of the previous sections since such a composition presents the best *obtaining time* for applications with low and intermediate parallelism and a reasonable one for applications with a high parallelism. Furthermore, its low complexity in terms of number of messages helps us to evaluate the particular case where each cluster is composed of one machine per cluster. For such a study, we have considered applications with different degrees of parallelism.

Figures 10(a) and 10(d) correspond to a low degree parallel application ($\rho = N/2$); Figures 10(b) and 10(e) correspond to a medium degree parallel application ($\rho = 2N$); Figures 10(c) and 10(f) correspond to a high degree parallel application ($\rho = 5N$). For each experiment, we have measured the *obtaining time* (Figures 10(a), 10(b), and 10(c)) and the number of *inter* cluster messages (Figures 10(d), 10(e), and 10(f)) for both the original Naimi-Tréhel algorithm and the *Naimi-Naimi* composition.

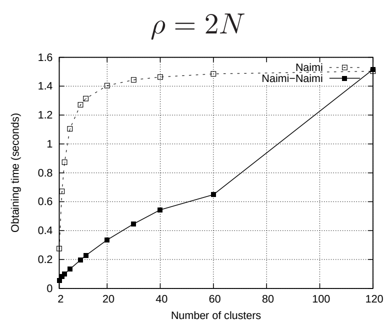
3.5 Impact of the number of clusters on the algorithm without composition

We start by studying the impact of the number of clusters of the Grid on both the *obtaining time* and the number of *inter* cluster messages in the original Naimi-Tréhel algorithm. We can observe in Figure 10, that the curves related to Naimi-Tréhel algorithm have a quite similar form. Independently of ρ, all these curves present a hyperbolic form: a significant growth when the number of clusters varies from 2 to 12. This growth then strongly flattens, becoming almost null, when the number of clusters is greater than 40.

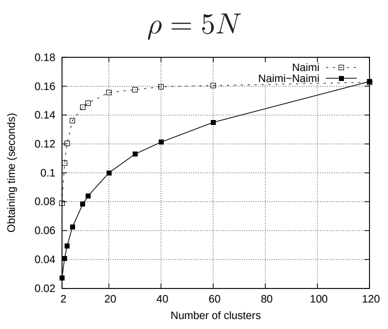
In order to better explain the form of such curves, we have decided to theoretically study the frequency with which a mutual exclusion algorithm sends an *inter* cluster message, i.e., the probability $\mathcal{P}$ that the destination node of a message does not belong to the same cluster of the sender of the message. To this end, we consider a Grid architecture composed of N nodes uniformly distributed over c clusters. Without loss of generality, we also suppose that a node can send a message to itself. This assumption models two successive accesses to the critical section by the



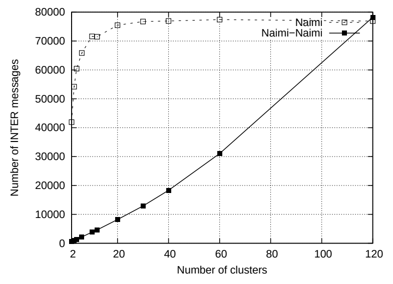
(a) Average Obtaining time



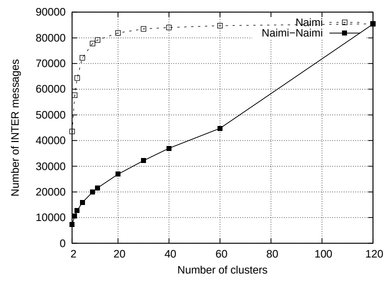
(b) Average Obtaining time



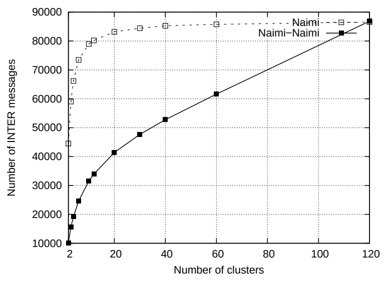
(c) Average Obtaining time



(d) Number of *inter* messages



(e) Number of *inter* messages



(f) Number of *inter* messages

Figure 10: Impact of the number of clusters

same node. We then get the following probability $\mathcal{P}$:

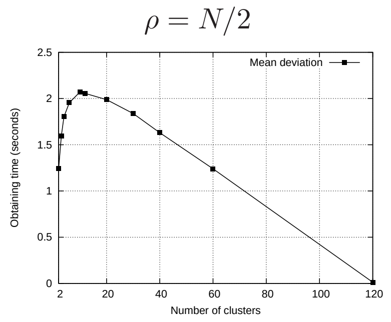
$$\mathcal{P} = \frac{N - \frac{N}{c}}{N} = 1 - \frac{1}{c}$$

This equation is totally in accordance with the form of the curves of Figure 10 for Naimi-Tréhel algorithm. It also shows that such a probability does not depend on the number of nodes N whenever they are uniformly distributed over the Grid. A last important conclusion from this equation is that the clustering effect due to the communication latency heterogeneity of a Grid has a negligible impact on the algorithm. One could always argue that such a heterogeneity might change the order of priority of the requests in such a way that request from closer nodes would be satisfied before distant ones. However, in the above equation, any node can be chosen among N with the same probability, i.e., independently of the topology of the Grid. In addition, if theoretical curves were produced from the equation, they would be similar to the ones of Figure 10. Thus, we can deduce that the assumption of equiprobability is reasonable and that the impact of the clustering effect on Naimi-Tréhel's algorithm is not significant.

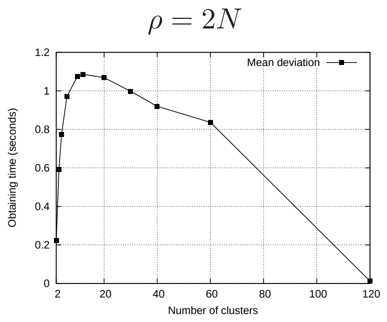
Let's come back to the curves in order to now study the impact of the number of clusters in function of the application behavior. The results of Figures 10(a), 10(b), and 10(c) confirm those studied in section 3.3.1 on top of Grid'5000, i.e., the degree of parallelism of an application has an impact on the *obtaining time*. However, since the forms of the curves are quite similar, this impact does not depend on the architecture of the Grid. Furthermore, the curves of Figures 10(d), 10(e), and 10(f) show that the parallelism degree of an application has no influence in the number of *inter* cluster messages even if we observe a small reduction of this number for low parallel applications.

3.6 Impact of the number of clusters on the composition approach

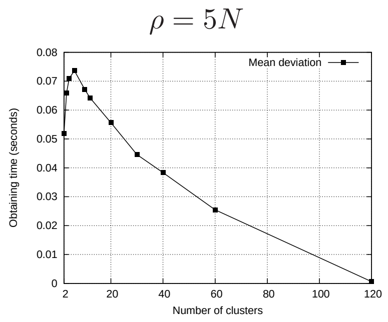
We are going now to study the impact of the Grid architecture on our composition approach. Similarly to what we have observed in the previous section for the original algorithm of Naimi-



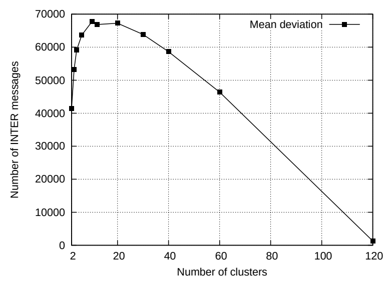
(a) Average obtaining time



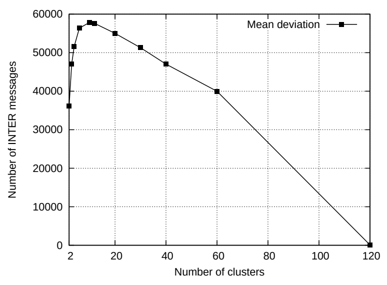
(b) Average obtaining time



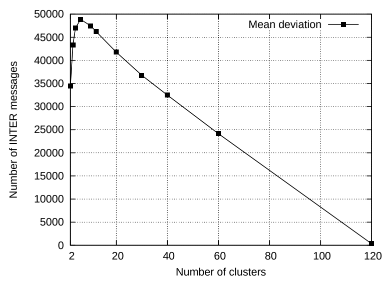
(c) Average obtaining time



(d) Number of *inter* messages



(e) Number of *inter* messages



(f) Number of *inter* messages

Figure 11: Mean deviation between the composition approach and the original algorithm

Tréhel, the number of clusters has an influence in the *obtaining time* as well as in the number of *inter* cluster messages which increase with the number of clusters. However, if we exclude the configuration with one node per cluster where there is in fact no hierarchy of communication at all, our approach always presents a smaller *obtaining time* and number of *inter* cluster messages when compared to the original Naimi-Tréhel algorithm. Notice that the benefit of using our composition approach is considerable even for a Grid composed of 60 two-node clusters.

Since the topology of the Grid has not the same impact on our composition approach as on the original plain algorithm, it would be interesting to study the mean deviation between our composition approach's curves and the original algorithm's curves for both the *obtaining time* and the number of *inter* cluster messages. Thus, based on the curves of Figure 10, Figure 11 shows such mean deviations.

We can remark in Figure 11 that the gain of our composition approach increases when the number of clusters varies from 2 to 12. This is in accordance with the curves of Figures 10 where both the obtaining time and the number of *inter* cluster messages increase fast for the original algorithm but smoothly for our composition approach. Such a different behavior explains why the maximum mean deviation between the two curves is reached with 12 clusters. Beyond this threshold value, the clustering effect does not have an influence on the *obtaining time* neither on the number of *inter* cluster messages since in our composition approach the curves progressively increase and Naimi-Tréhel's curves remain linear. Thus, the respective mean deviations inversely decrease till it becomes null for the configuration where each node represents a cluster (120 clusters).

We would like to theoretically evaluate the above threshold in a Grid composed of N nodes uniformly divided into c clusters. Hence, similarly to what we do in section 3.5 for Naimi-Tréhel algorithm, we need to calculate the probability $\mathcal{P}$ that a node sends an *inter* cluster message for our own composition approach considering a aforementioned Grid. Without loss of generality, we consider the case where the cluster locality is maximum, i.e., every time a coordinator of a cluster gets the *inter* token, all the N/c nodes of this cluster executes a critical section. So, the probability

$\mathcal{P}$ is equal to the probability of executing the last of the N/c critical section executions:

$$\mathcal{P} = \frac{1}{\frac{N}{c}} = \frac{c}{N}$$

Therefore, the mean deviation $E(c)$ between our composition approach and the original algorithm in function of the number of clusters c is equal to :

$$E(c) = 1 - \frac{1}{c} - \frac{c}{N}$$

and the mentioned threshold $c_{threshold}$ is equal to:

$$\begin{aligned} E'(c) &= \frac{1}{c^2} - \frac{1}{N} \\ E'(c) = 0 &\Rightarrow c_{threshold} = \sqrt{N} \end{aligned}$$

Such an equation shows that the maximum benefit when using our composition approach is reached for a Grid architecture composed of $\sqrt{N}$ nodes. This result can be verified by the curves of Figure 11 since $\sqrt{120} = 10.95$. Consequently, for $\rho = N/2$, the maximum mean deviation is reached between 8 and 12 clusters. It is also worth remarking that for low parallel applications ($\rho = 5N$), the Grid architecture corresponding to the peak benefit shifts to a 6 clusters Grid. Lastly, we can observe on the curves of Figure 10 that our approach becomes less effective when the degree of parallelism increases: it does not present a linear behavior anymore when the number of clusters increases contrarly to low parallel applications.

4 Related work

Several studies have propose to adapt existing mutual exclusion algorithms to a hierarchical scheme. In Mueller [12], 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.

Several authors have propose hierarchical approaches for combining different mutual exclusion algorithms. Housni et al. [7] and Chang et al. [5]'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. [7], sites with the same priority are gathered at the same group. Raymond's tree-based token algorithm [15] is used inside a group, while Ricart-Agrawala [16] diffusion-based algorithm is used between groups. Chang et al.'s [5] hybrid algorithm applies diffusion-based algorithms at both levels: Singhal's algorithm [18] locally, and Maekawa's algorithm [10] between groups. The former uses a dynamic information structure while the latter is based on a voting approach. Similarly, Omara et al. [14]'s solution is a hybrid of Maekawa's algorithm and Singhal's modified algorithm which provides fairness. In Madhuram et al. [9], 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 [6] 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.

5 Conclusions

In this paper, we have proposed a new approach for composing mutual exclusion algorithms in order to offer mutual exclusion service 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 both the real French wide Grid Grid'5000 and an emulation platform 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 forwarding 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 CS 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 approach provides a framework for easily choosing the best two algorithms combination for composing mutual exclusion services.

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. *Journal of Parallel and Distributed Computing*, 66:128–144, 2006.

[4] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jegou, S. Lanteri, N. Melab, R. Namyst, P. Vicat-Blanc Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet. Grid5000: a nation wide experimental grid testbed. *International Journal on High Performance Computing Applications*, Vol. 20(No. 4):481–494, 2006.

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

[6] K. Erciyes. Distributed mutual exclusion algorithms on a ring of clusters. In *International Conference on Computational Science and Its Applications*, volume 3045 of *LNCS*, pages 518–527, 2004.

[7] A. Housni and M. Tréhel. Distributed mutual exclusion by groups based on token and permission. In *International Conference on Computational Science and Its Applications*, pages 26–29, June 2001.

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

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

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

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

[12] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, March 1998.

[13] 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.

[14] 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.

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

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

[17] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

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

[19] J. Sopena, L. Arantes, F. Legond, and P. Sens. The impact of clustering on token-based mutual exclusion algorithms. In *Euro-Par 2008 Processing, Las Palmas , SPAIN*. Springer Berlin, August 2008.

[20] J. Sopena, F. Legond, L. Arantes, , and P. Sens. A composition approach to mutual exclusion algorithms for grid applications. In *The 36th International Conference on Parallel Processing*, pages 65–75, September 2007.

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