Data handover on a peer-to-peer system

Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou

To cite this version:

Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou. Data handover on a peer-to-peer system. [Research Report] RR-8690, Inria Nancy - Grand Est (Villers-lès-Nancy, France); INRIA. 2015, pp.37. hal-01120837v2

HAL Id: hal-01120837
https://inria.hal.science/hal-01120837v2

Submitted on 3 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Data handover on a peer-to-peer system

Soumeya Leila Hernane   Jens Gustedt   Mohamed Benyettou

RESEARCH REPORT
N° 8690
February 2015
Project-Team Camus

ISSN 0249-6399   ISRN INRIA/RR--8690--FR+ENG

Data handover on a peer-to-peer system

Soumeya Leila Hernane[*†] Jens Gustedt[‡†]
Mohamed Benyettou[*]

Project-Team Camus

Research Report n° 8690 — version 2 — initial version February 2015 — revised version December 2015 — 37 pages

Abstract: This paper presents the *Data Handover* API and its integration into a *peer-to-peer* Grid architecture. It provides an efficient management of critical data resources in an extensible distributed setting consisting of a set of *peers* that may join or leave the system. Locking and *mapping* of such a resource are handled transparently for users: they may access them through simple function calls. On the lowest level of the proposed architecture the Exclusive Locks for Mobile Processes **ELMP** algorithm ensures data consistency and guarantees the logical order of requests. All operations in our architecture have an amortized cost of $O(\log n)$. An experimental assessment validates the practicality of our proposal.

Key-words: data consistency, peer-to-peer, data access, request ordering, distributed locks, resource mapping

[*] University of Science and Technology, Oran, Algeria
[†] ICube, Univ. of Strasbourg, France
[‡] INRIA Nancy – Grand Est, France

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Passage de données dans un système pair-à-pair

Résumé : Ce papier presente l'interface applicatif *Data Handover* et son intégration à une architecture pair-à-pair. Il fournit une gestion efficace de ressources de données critiques dans un cadre repartie extensible, composé de pairs qui peuvent joindre ou quitter le système. Verrouillage et mappage d'une telle ressource sont gérés de façon transparente pour les utilisateurs : ils peuvent les accéder par de simples appels à fonction. Au niveau le plus bas de cette architecture l'algorithme *Exclusive Locks for Mobile Processes*, **ELMP**, assure la cohérence et garantie l'ordre logique des requêtes. Tous les opérations de notre architecture ont un coût amortie de $O(\log n)$. Une évaluation expérimentale valide les aspects pratiques de notre proposition.

Mots-clés : cohérence de données, pair-à-pair, accès aux données, ordre logique de requêtes, mappage de ressources

1 Introduction and Overview

Large-scale distributed computing systems are serving a growing number of research communities and industries.

So-called *Grid Computing* aggregates large scale clusters to provide seamless and scalable access to wide-area distributed resources such as CPU cycles and storage of computers. Research has focused on different issues such as resource location and management, utilizing idle compute cycles, distributed scheduling, remote work processing, security issues and fault-tolerance. While Grid environments bring the advantages of an economy of scale, their heterogeneous structure limits the efficient use of system resources.

Peer-to-peer systems are viewed as overlay networks organized on top of a physical network. File sharing, process sharing and collaborative environments are some examples of applications in that domain. They are able to handle complex queries, but do not guarantee to respond to requests in bounded time. To that aim, they would need an explicit self-organization policy.

Combining Grid and peer-to-peer technologies provides the opportunity to do computations while sharing resources on peer-to-peer networks. Peer-to-peer Grid computing is interesting for applications that pursue a wide number of computational tasks and that access data resources concurrently. Challenges and open-issues raised by the research community are cost reduction, improved scalability and reliability of nodes that hold data, autonomy and anonymity.

SETI@HOME Anderson et al. [2002] is the most famous implementation of peer-to-peer Grid computing. The success of the project came from the aggregation of otherwise unused CPU time. The XtremWeb Fedak et al. [2001] peer-to-peer Grid computing project deploys and executes parallel and distributed applications on public resource infrastructures. It relies on a scheduling algorithm that assigns job requests to *peers* according to predetermined rules.

The goal of the present paper is to extend peer-to-peer Grids from only aggregating CPU cycles to a more general resource sharing model for parallel distributed computing.

JXTA Antoniu et al. [2005] provides a collaborative platform for a wide range distributed computing applications. It provides a network programming and computing infrastructure and allows the easy incorporation of new protocols and services, which can be operated by a wide range of users.

JXTA attaches a unique ID (generated by the users) to each *peer* in the group and does not guarantee uniqueness across all *peers* or across multiple groups Milojicic et al. [2002]. This results in a lack of scalability. The authors of Antoniu et al. [2007] have experimentally evaluated the scalability of JXTA protocols, but did not include the volatility of peers into their study.

In our approach, we provide the possibility to use and share *data* resources be-

tween a group of *peers* belonging to the same application. We target computation-intensive applications that require remote data resources. Simultaneously we aim to hide all underlying system complexities from users.

Possible application areas include the energy and financial industries, or semantic web services.

There has been very little research in the area of distributed computation with pure *peer-to-peer* networks. Most schemes are based on federating organizational domains rather than using the Internet.

We aim at users who are familiar with distributed and parallel libraries. We want to facilitate their task of claiming remote resources by allowing them to simply insert function calls in their existing code.

Data utilization in parallel and distributed systems relates to models and paradigms that originate from two separate classes of architectures: shared and distributed memory. Shared memory based programming environments are commonly *thread* based openmp, while distributed memory architectures use message passing interfaces such as *MPI* Arquet [2001].

In grid and *peer-to-peer* environments, applications may access data resources without prior knowledge of whether or not these are located on the same host or on a distant one. Therefore, both classical paradigms are not sufficient for such environments.

In Caniou et al. [2014], the authors present a basic implementation of the OGF standard GridRPC Data Management API Caniou et al. [2012] and its integration in two different middlewares. The API takes into account both synchronous and asynchronous calls, while the Data Management standard provides a modular architecture that ensures immediate portability and interoperability between the API and the middlewares. There is no theoretical study of the model provided. Our proposal is complementary to this approach by providing transparent data access and making our API completely independent of the chosen middleware. Moreover, both theoretical and experimental studies of scalability are given.

1.1 Motivations

We aim to deploy an easy-to-use API that calls and handles resources on our proposed Grid computing environment. The user only needs to know identifiers of requested resources. If the request has been issued, the system provides it within a finite time. The supporting library must be able to cope with all requirements for large scale distributed systems such as grids, in particular heterogeneity, scalability and mobility of *peers*.

In our model, applications evolve on top of a *peer-to-peer* Grid architecture with completely decentralized *peers*. The distributed infrastructure we provide should address following issues:

Robustness: Peers that comprise the system rely on the flexible nature of *peer-to-peer* networks. They enjoy a certain degree of autonomy. Even if they host resources, they may unsubscribe.

Scalability: Individual peers never deal with the whole set of peers that comprise the system, but only with a small subset of peers that are related to the particular usage of the resource that a peer has. In case of the need for more compute power or storage capacity, the application may launch new *peers* that join the system.

Consistency: Users should have a consistent view of the resource, despite of the dynamic features of the underlying system. Once the data in acquired locally by a given *peer*, it should then be updated transparently, before making it available again to other *peers*.

Cooperation: Peers can act both as *consumers* and as *providers* of resources. Moreover, *peers* cooperate to satisfy various requests from users.

Information hiding: The cooperation model we propose is based on a full separation between users, *peers* and data location. No user that wants to use the system has to know names and addresses of all *peers*, especially not the one currently holding the resource.

Transparency: The *peer-to-peer* network may evolve for different reasons and users should not be concerned by changes that are induced by this evolution of the underlying system. Their only role is to handle computational tasks by inserting the set of proposed functions in existing applications.

1.2 Contributions

We propose several contributions that are situated on two different levels:

1. At the applicant level, we present the *Data Handover* interface (**DHO**) interface for claiming and acquiring resources. It proposes an abstract view of these resources and is implemented as the top level of a three-level architecture. **DHO** had previously been introduced in Gustedt [2006], Hernane et al. [2011] but had only been realized with a centralized architecture.[1]

   DHO uses a mediator process that satisfies requests launched by users, and includes an abstraction level between resource and memory through a *handle*.

[1] In contrast to the original DHO proposal, here we present it solely for a write exclusive mode.

A user that claims the resource locally by using **DHO** is unaware of the complexity underneath, even if there are lots of *peers* in the system. All technical information such as resources location and physical characteristics of the network are hidden and encapsulated in a data structure called *handle*.

We replaced the *Client-Server* paradigm that had been used for **DHO** in previous work by a *peer-to-peer* paradigm. We implemented a mediator process within each *peer* that replaces a separate, fixed, server entity, which had been used before.

2. On the lower level, we provide a *peer-to-peer* Grid architecture that meets the challenges cited above.

   For that aim, we propose a grid service that is modeled by a three-level architecture. It guarantees responses of all data requests claimed by users on the higher level (the third level), through **DHO** routines. The grid service transparently achieves the desired requirements from above.

   Based on our previous work, Hernane et al. [2012], we propose the *Exclusive Locks with Mobile Processes* algorithm, **ELMP**, which is an extension of the distributed mutual exclusion algorithm of Naimi and Tréhel [1988]. Together with a data structure that provides scalability of processes it ensures consistency when accessing a **critical section** of the application. **ELMP** acts on the lower level of the grid service and provides **consistency**, **scalability** and **cooperation**.

   The **ELMP** algorithm also address the potential flooding caused by too many new arrivals in a **DHO** system. The shape of the ***Parent*** tree is constantly monitored during and after each atomic operation. Such a requirement prevents the flooding of the root by new insertions or the unproportional increase in height of the tree. All operations involved in maintaining a balanced tree-structure are still bound to a logarithmic scale.

The rest of this paper is organized as follows: after describing the basics of the Naimi and Tréhel algorithm in Sections 2 and 3 we introduce the algorithm *Exclusive Locks with Mobile Processes*, **ELMP**, and the data structure in detail. Then, in Section 6, we present the **DHO** library and our *peer-to-peer* Grid system. Section 7 reports the results of the experimental evaluation of our proposal before concluding in Section 8.

2 The Naimi and Tréhel Algorithm

Several algorithms have been proposed over the years to solve mutual exclusion problems within distributed systems. They can be either permission-based (Lam-

port [1978], Maekawa [1985], Ricart and Agrawala [1981]) or token-based (Naimi and Tréhel [1988], Raymond [1989]). The token based algorithms restrict the entrance into the critical section to the possession of a token, which is passed between two nodes. This group of algorithms is tree-based and many of them exhibit a $O(\log n)$ complexity in terms of the number of messages per request.

Our work focuses on this class of algorithms for the sake of this message complexity; the distributed algorithm of Naimi and Tréhel [1988] based on path reversal is *the* benchmark for mutual exclusion in this class. Many other extensions of this algorithm have already been proposed in the literature. A Fault tolerant token based mutual exclusion algorithm using a dynamic tree was presented by Sopena et al. [2005]. It improves over Naimi and Tréhel [1988] by ensuring a lower cost in terms of messages in the presence of failures. In Hernane et al. [2012], we have proposed a dynamic distributed algorithm for read/write locks that ensures *Safety* and *Liveness* properties, and a logarithmic complexity. Before presenting our mutual exclusion algorithm, we first describe that of Naimi and Trehel.

The Naimi and Tréhel algorithm is based on a distributed queue along which a token circulates, representing the protected resource. Queries are handled through a second structure, a distributed tree. The query tree is rooted at the tail of the queue to allow to append new requests to the queue at any moment.

2.1 The basics

The basics of this algorithm are summarized following Naimi, Tréhel, and Arnold [1996]:

1. There is a logical dynamic tree structure such that the *root* of the tree is always the last process that requested the token. In that tree, each process points towards a ***Parent***. Requests are propagated along the tree until the *root* is reached. Initially, all processes point to the same ***Parent*** which is the *root* which initially holds the token.

2. There is a distributed FIFO queue which keeps requests that have not yet been satisfied. Hence, each process ρ that requested the token points to the **Next** requester of the token. This identifies the process for which access permission is to be forwarded after process ρ leaves its **critical section**.

3. As soon as a process ρ wants to enter the **critical section**, it sends a request to its ***Parent***, waits for the token and becomes the new *root* of the tree. If it is not the current *root*, the ρ's ***Parent***, σ, forwards the request to it's ***Parent*** and then updates its ***Parent***'s variable to ρ. If σ is the *root* of the tree and not inside the **critical section**, it releases the token to ρ. If it is inside or still waits for the token, it points its **Next** to ρ.

Each process maintains local variables that it updates while the algorithm evolves:

Token_present: A Boolean set to *true* if the process owns the token, *false* otherwise.

***Requesting_cs*:** A Boolean set to *true* if the process has claimed the **critical section**.

Next: The **Next** process that will hold the token, *null* otherwise. Initially set to *null*. This might only be set while the process has claimed the token and a non-satisfied request has to be served after the own request.

***Parent*:** Initially the same for all processes except for the initial root itself.

Processes send two kind of messages:

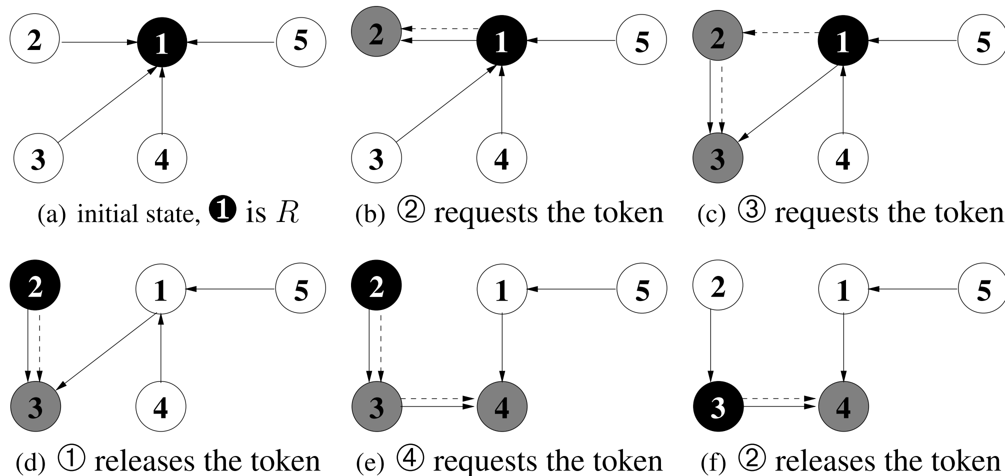
Request(ρ): sent by the process ρ. to its ***Parent***.

Token: sent by a process ρ to its **Next**.

we have the following invariant:

Invariant 1 *At the end of request processing, the root of the* ***Parent*** *tree is the tail of the* **Next** *chain.*

The Naimi and Tréhel algorithm provides a distributed model that guarantees the uniqueness of the token while ensuring properties of *safety* and *liveness*.



(a) initial state, ❶ is R (b) ② requests the token (c) ③ requests the token (d) ① releases the token (e) ④ requests the token (f) ② releases the token

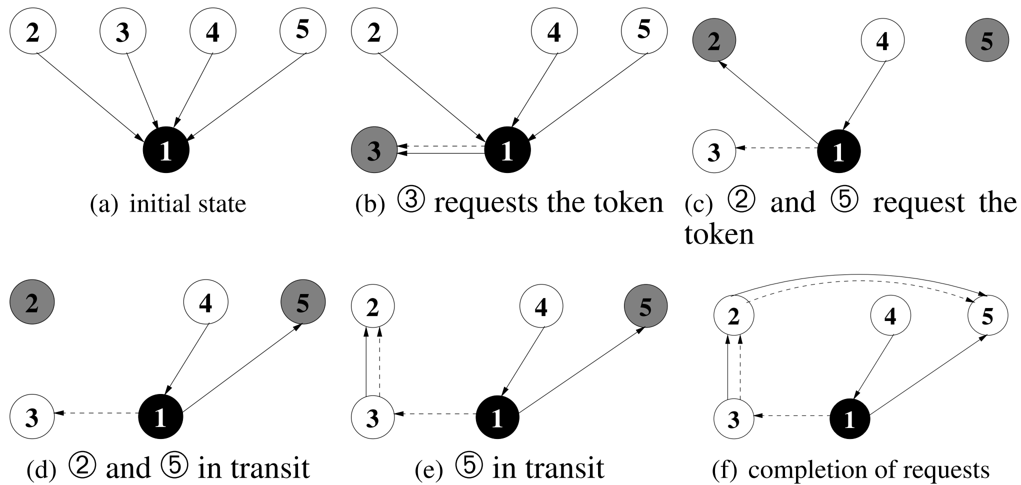
Figure 1: Example of the execution of Naimi and Tréhel's Algorithm

An example of the execution of the algorithm is shown in Fig. 1. Gray circles denote processes with requests, while the unique black circle is the one that holds

the token. Initially, process ❶ holds the token, Fig. 1(a). It is the ***Parent*** of the remaining processes and the *root*, R of the ***Parent*** tree. process ② asks the token from its ***Parent***, Fig. 1(b). Thus, ❶ points towards ② and updates its **Next** variable to the same process. Afterwards, ③ requests the token, Fig. 1(c). ❶ then forwards the request to its new ***Parent***, process ② which updates in turn its variables, ***Parent*** and **Next** to ③. In Fig. 1(d), ① releases the lock, while ❷ gets it and the ④ in turn, requests the **critical section**, Fig. 1(e). Thus, processes ① and ③ point their ***Parent*** variables to ④. Obviously, the latter updates its **Next** to process ④. Finally ❸ gets the lock, Fig. 1(f).

2.2 Concurrent requests

Within the Naimi and Tréhel algorithm, a given ***Parent*** can be queried simultaneously by different processes. We refer to Fig. 2 to explain consecutive access requests. This example is taken from Naimi and Tréhel [1988] where it is presented in the context of node failures.



(a) initial state (b) ③ requests the token (c) ② and ⑤ request the token

(d) ② and ⑤ in transit (e) ⑤ in transit (f) completion of requests

Figure 2: Example of concurrent requests in Naimi and Tréhel's Algorithm

Initially, Fig. 2(a), process ❶ holds the token and ③ claims the **critical section** by sending a request to its ***Parent***. In turn, ❶ updates its ***Parent*** and its **Next** to ③, Fig. 2(b). Then, processes ② and ⑤ claim the **critical section**. They send request to ❶ and set forthwith their ***Parent*** to *null*. So, ❶ points towards ② and forwards the request to ③, Fig. 2(c). Meanwhile, process ⑤ waits and is disconnected from the tree. Once ❶ sent the request of ② to ③, it switches to ⑤'s request. Thus, it forwards the request to ② and sets its ***Parent*** to ⑤. Meanwhile, ② is cut from the tree, Fig. 2(d). In Fig. 2(e), request of process ② is achieved and that of ⑤ ends in

Fig. 2(f). We notice that, processes set their ***Parent*** variable to *null* as soon as they forward the request. Within a system of n processes, *n-1* processes may request the token concurrently and this will generate n disjoint components. The ***Parent*** relation then is not a tree but only a forest.

3 A balanced tree structure

As we have seen in the discussion above, in the original version of the Naimi and Tréhel algorithm, the ***Parent*** relation becomes disconnected as soon as a process ρ has requested the token. The connectivity information remains implicit in the network, namely through the fact that ρ's request for the token eventually gets registered in the process r (by updating its **Next** pointer) that will receive the token just before ρ.

This lack of explicit connectivity information makes it difficult for a process to leave the group, if it is not interested in the particular token that is represented by the group. It is difficult for any process to determine, if its help is still needed to guarantee connectivity of the remainder of the group or not. Furthermore, the structure provided by the original algorithm lacks flexibility, it no longer meets current needs of large-scale dynamic systems. In our proposal, processes handle one request at a time.

In this section, we provide a new structure, which addresses following issues:

1. The maintenance of the connectivity such that any node will always be able to leave the group within a "reasonable" time-frame; "reasonable" here being the time needed to forward information to the other processes.

2. The possibility for new nodes to join the system whenever possible.

3. The control of the shape of the tree in order to meet the balancing requirement, such that all operations belonging to the system are bounded by a complexity of $O(\log n)$.

We assume that initially, processes are arranged in a balanced tree-structure wherein all arrows point towards the direction of the root holding the token. The balanced shape of the tree is maintained according to well known strategies. We report possible choices for these strategies in Section 4.5.

Beside the variables defined in the original Naimi and Tréhel algorithm, each process σ additionally handles the following:

ID We introduce a new variable *ID* that holds a number that will be used as a tie breaker during departure (see Section 4.3). The current root of the tree will maintain a global value that is the maximum of all these *ID*. Since new

processes must first reach the root, r, such a value can easily be maintained by that root and propagated along if the root changes.

Predecessor: Each process knows who will hold the token before it. It is easily updated simultaneously as **Next**. Instead of a distributed queue, the **Next** and the **Predecessor** form a doubly linked list. Once a process r passes the token to its **Next** σ, r's **Next** and σ's **Predecessor** are set to NULL.

With the aim of maintaining the connectivity of parental structure as well as of the linked list, two variables are added and associated to each process σ.

***children*:** A list of *Child* processes.

***blocked*:** A list of processes that are *Blocked* (by σ). The *Blocked* list guarantees an atomicity on the path borrowed by a task undertaken by σ.

The control of the shape of the tree is done through the additional following variables (see Section 4.5):

***height(σ)*:** The height of the subtree rooted at a process σ, which is the longest distance in terms of edges from σ to some leaf.

***weight(σ)*:** The weight of σ, i.e, the number of leaves belonging to the subtree of σ.

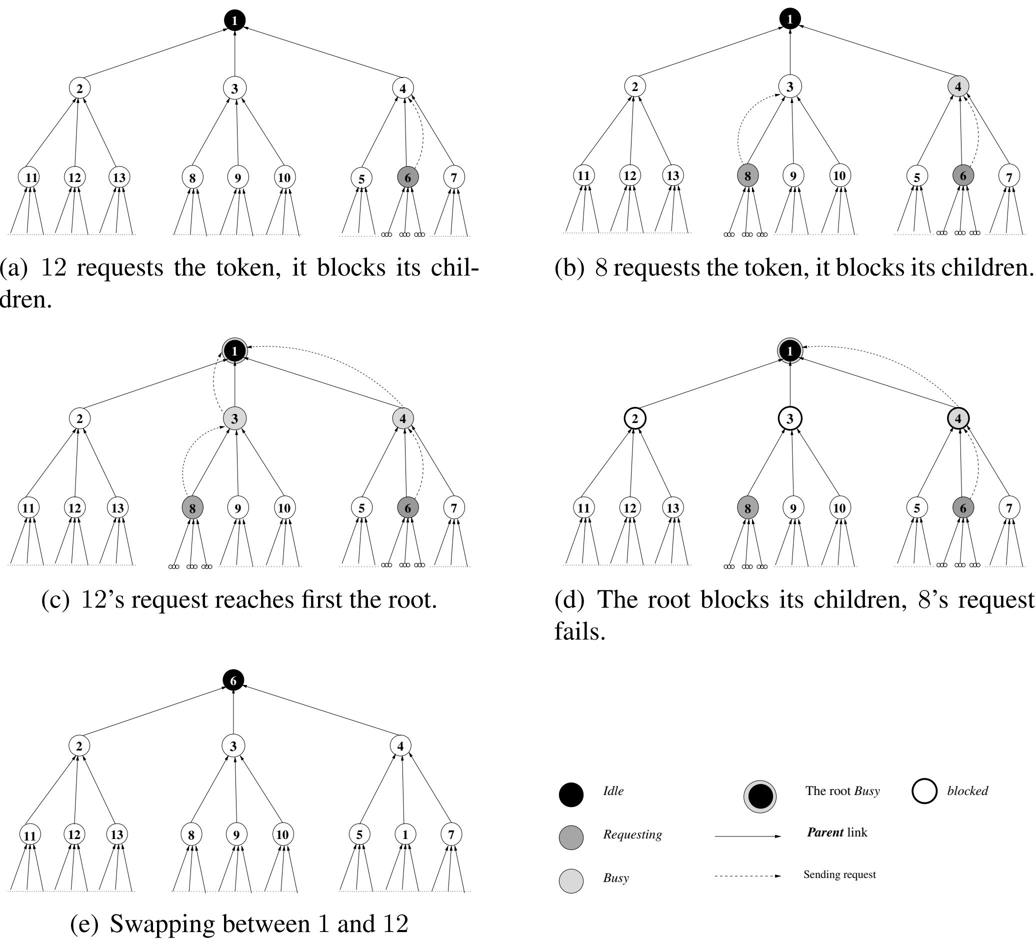
These variables are used for decisions that concern the restructuring of the tree, for example to find a position for a new arrival. They are updated whenever necessary. Note that such operations require no more than $O(\log n)$ messages, since the tree is consistently kept balanced.

4 Exclusive Locks for Mobile Processes (ELMP) algorithm

The **ELMP** algorithm is characterized by a set of atomic operations that a process σ may use. Such operations are always either fully completed or fully canceled. To describe our algorithm, we introduce a “State” variable for each process. It has different values to indicate the specific task, which the process is currently completing. Fig. 3 exemplifies our approach and illustrates the state concept. In this case, ❶, which is initially the root of the tree, holds the token.

The white circles denote *Idle* processes, which are not involved in any operations (for themselves or for others) and may directly switch to any other state.

The following section describes the activities that are defined by the **ELMP** algorithm involving σ, a given process in the ***Parent*** tree.



(a) 12 requests the token, it blocks its children.

(b) 8 requests the token, it blocks its children.

(c) 12's request reaches first the root.

(d) The root blocks its children, 8's request fails.

(e) Swapping between 1 and 12

Figure 3: Handling concurrent requests in the **ELMP** algorithm

4.1 Requesting the token

If σ wants to access the **critical section**, it first needs to be *Idle*. It then starts a request for getting the token, switches to the state *Requesting* and waits for a response. Such a response can be positive or negative. Dark gray circles in the figure denote *Requesting* processes. σ remains in this state and is unable to receive other requests until the first request has been completed (such as ⑥ in Figs. 3(a) and 3(b)). The *blocked* list of σ will include all corresponding children (bold circles).

Algorithm 1 summarizes the various steps that a peers σ takes to request the token:

1. Initially, σ walks up the tree and notifies processes $p_1, \ldots, p_n$ on the path about the insertion operation (lines 5 to 10 of Algorithm 1). In this case p_1 is the ***Parent*** and p_n is the current root. These processes will all switch to the *Busy* state (see Section 4.2), but will not change their ***Parent*** pointer. Note that, processes on the path to the root form a ***Parent*** branch of σ (as processes ④ and ❶). If at least one process is not available, σ tracks back and switches the Processes on the path back to the *Idle* state. σ then starts over (lines 4 and 17 of Algorithm 1). This is the case of process ⑧ which tries to send a request to its ***Parent*** (process ③). The request of ⑧ fails because the request of ⑥ reached the root first, see Figs. 3(b) and 3(c).

   Once the root is *Idle*, it includes its children in the *blocked* list, in this case consisting of ②, ③ and ④, see Fig. 3(d).

2. σ and the current root r exchange their positions. Their children are still blocked (lines 15 and 16 of Algorithm 1). Process r now ceases to be the root and its children update their ***Parent*** before returning to the *Idle* state. In our example, this is the case for ②, ③ and ④ as well as for children of process ⑥, see Fig. 3(d).

3. The new root σ, see ⑥ in Fig. 3(e), sends acknowledgments to the processes $p_1, \ldots, p_{n-1}$ on the path, changes its ***Parent*** to `empty` and its **Predecessor** to r. σ may be involved in re-balancing the tree before returning to the *Idle* state, see Section 4.6.

4.2 Handling an incoming request

If σ is *Idle*, it can deal an incoming request and then, may become *Busy* (light gray circles of Fig. 3).

σ then handles an insertion request for another process ρ, ⑥ for example. This is the case for ④, ③ and the current root ❶. σ is not ready to forward other

Algorithm 1: Requesting the token, process must be in *Idle* state

```
1  i ← 1;
2  success ← false;
3  State ← Requesting;
4  do
5      while (i <= n ∧ success) do
6          sends request message to Parent = p_i;
7          Wait for state(p_i);
8          if state(p_i) ≠ Idle then
9              success ← false;
10         i++;
11     if (i = n + 1 ∧ success) then
12         for item ∈ children_list do
13             Add item to Blocked_list;
14             Remove item from children_list;
15         Send children_list to p_n;
16         Wait for children_list of p_n;
17         Update children_list;
18 while ¬ success ;
19 Parent ← null;
20 State ← Idle;
```

insertion requests, to request the token for itself, or to be involved for the departure of another process.

If the ***Parent*** of σ is **empty** (the case of the actual root), it exchanges its position with ρ (the new root). It also sets its **Next** variable to ρ and updates its list of children (line 17 of Algorithm 1). In Fig. 2(e), ⑥ and ❶ exchange their positions at the end of the requesting process of ⑥. Then, upon receiving an acknowledgment from ρ, σ switches back to the *Idle* state. It may request the token for itself, as it may be called again for further operations.

4.3 Disconnecting

The *Exiting* state denotes the disconnecting activity for σ. When in that state, σ negotiates with some other processes (see below) and must wait in case these are, for example, in the middle of requesting the token (*Requesting*) or themselves *Busy* or leaving the system, or involved in the tree-restructuring.

In fact, a process σ does not disconnect from the ***Parent*** tree and from the linked list **Next-Predecessor** without precaution. It has to satisfy a number of constraints such that the disconnection will never compromise the consistency of the algorithm as a whole, nor the connectivity of the ***Parent*** tree and in particular not that of the linked list **Next-Predecessor**. Before σ can disconnect it must find replacements for all the roles that it plays in the structure.

From an application point of view it makes no sense for a process to leave the system while it is inside the **critical section**, or while it attempts to enter it. Therefore we can require that to be able to switch to the *Exiting* state, σ must be *Idle* and must not have requested the token.

The fact that σ must not have requested the token does not mean, that it cannot actually *possess* it. σ may well be the last process that had accessed the **critical section**. The following lemma is immediate:

Lemma 1 *Let σ be a process that is* Idle *and that has not requested the token.*

1. *σ possesses the token iff it is the root of the **Parent** tree.*
2. *If σ is the root of the **Parent** tree, no other process has successfully inserted a token request.*

Algorithm 2 summarizes the effective departure of σ from the system. During this departure, σ will contact all its *neighbors* in the ***Parent*** tree. These are its children and its ***Parent***, if it has one.

Previously being *Idle*, σ switches to the *Exiting* state. For all its neighbors consisting of ***Parent*** and `children`, σ initializes a handshake with a process η:

Algorithm 2: Disconnecting, require *Idle* state

```
1  Neighbors_list ← {Parent ∪ children};
2  Blocked_list ← ∅;
3  success ← false;
4  do
5      while (¬ success ∧ Neighbors_list ≠ ∅) do
6          State ← Exiting;
7          if item (∈ Neighbors_list) is Idle ∨ (item is Exiting ∧ ID <
           ID_item) then
8              Add item to Blocked_list;
9              Remove item from Neighbors_list;
10         else
11             for item ∈ Blocked_list do
12                 Add item to Neighbors_list;
13                 Remove item from Blocked_list;
14             success ← false;
15             State ← Idle;
16     if Neighbors_list = ∅ then
17         success ← true;
18 while ¬ success ;
19 if Parent ≠ ∅ then
20     sends Blocked_list to Parent;
21 else
22     elects ρ = new_parent among Blocked_list;
23     sends Blocked_list - {ρ} to ρ;
24     sends token to new_parent;
25 for item ∈ Blocked_list do
26     sends acknowledgments;
```

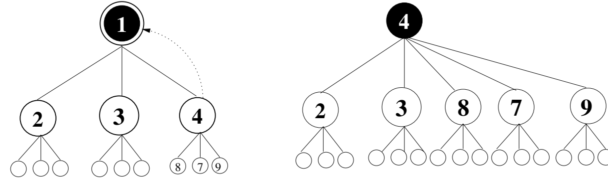


Figure 4: Disconnection of 1 and electing 4 as new root

If η is *Idle*, it switches to *Blocked* (by σ). σ moves η from its *children* to its *blocked* list.

If σ and η are both exiting, the one of them with lower ID has a priority for that request. The one with the higher ID switches to *Blocked* (by the other), and updates its lists analogous to the previous point (lines 7 to 9 of algorithm 2). If σ fails to move all its neighbors to the *Blocked* list it switches back to *Idle* and starts over.

Now all neighbors of σ are *Blocked* (by σ) and thus its *children* list is empty and all neighbors are listed in *blocked*. If σ is not the root of the tree, it chooses $\rho =$ ***Parent***, otherwise it is in the situation of Lemma 1 and chooses ρ among its children, as in Fig. 4, where ❶ chooses ④ among its children. Note that If σ has neither ***Parent*** nor children, the system consists only of σ.

σ notifies its neighbors of the choice of ρ. ρ itself will discover by that message that it has been chosen and that it will be the new root of the tree (lines 22 to 24 of algorithm 2). σ sends its list *blocked* (that are ② and ③ in Fig. 4) (excluding ρ) to ρ, ④. σ waits for an acknowledgment from ρ that it has integrated the list into the list of its children.

Finally, σ informs all its neighbors that it has completed the departure process.

4.4 Blocking mechanism

The Blocking state denotes a specific imposed situation. In fact, σ becomes *Blocked* if another process ρ succeeds in switching it to that state. The *Blocked* state is closely tied to *Exiting*, see above. There are two possible scenarios:

- σ is part of an unbalanced branch of the tree. Thus, it will be blocked for a subsequent repositioning in the tree. In Section 4.6.1 below, we present possible strategies for maintaining the tree as balanced in case of departure.
- An other process ρ that is disconnecting successfully (as described above) blocked σ.

  In fact, ρ (① in Fig. 4) will promote its children and its ***Parent*** to the *Blocked* state, such that they delay any requests that might be pending in their sub-trees (as processes ②, ③ and ④ in Fig. 4). Among these blocked neighbors,

ρ will chose another process η (that may be the same as σ) that will inherit all information that ρ held for the system. This is the case of ④. Namely, the children of ρ will become children of η and if ρ held the token previously to its departure, η will do so thereafter.

The blocking mechanism will also be applied to processes that are candidates for displacement for the balancing strategies, see Section 4.6.1.

Note that σ may switch to the *Blocked* state if it is *Idle* or if it is *Exiting* and if its ID is greater than that of another process ρ that attempts to exit simultaneously.

The actions that a neighbor σ of a disconnecting process ρ has to perform during the *Blocked* state are summarized as follows:

1. σ waits to receive the name η of the process chosen by ρ as ***Parent*** and sets η as its new ***Parent***.

2. If σ is choosen as ***Parent***, then it waits to receive the list of children from ρ, and updates its list of children accordingly (as ❹ in Fig. 4).

   If ρ is the same as ***Parent***, we have that $\sigma = \eta$ and we are in the situation of Lemma 1. σ becomes the new root of the tree. It sets its ***Parent*** accordingly and notes that it possesses the token.

3. After finishing its update, σ sends an acknowledgment to ρ. Then, σ waits that ρ signals the completion of its departure and finally switches back to its previous state (*Idle* or *Exiting*).

4.5 Balancing strategies

Many approaches have been proposed in the literature in order to achieve efficient maintenance, mainly for binary trees, with the challenge of finding a balance criteria that ensures a logarithmic height of the tree. We outline two approaches.

The first one is to restrict the shape of the tree such that it has order m. In Jagadish et al. [2005, 2006], the authors proposed a balanced tree structure overlay on a peer-to-peer network. It is based on a binary balanced tree (BATON) and on m-order trees (BATON*). Therein joining and departures of nodes are handled well and take no more than $O(\log n)$ steps. Thereby, new nodes will be inserted into positions that are close to the leafs and a balanced growth of the tree is guaranteed. For departures, they replace *non-leaf* nodes by *leaf-nodes*.

Additionally, links between siblings and adjacent nodes are maintained. This allows to jump in the tree, and to reach the root rapidly. This is particularly interesting for new processes that try to get their *ID* from the root. The cost of all

atomic operations handled by the structure will then be significantly reduced since the height of the tree is controlled.

Thus, if we opt for this schema, we should review the progress of events that make changes in the shape of the ***Parent*** tree (see below).

The second strategy is a "lazy" mode. Balancing of the tree is delayed until it is really needed. In this approach, no shape restriction is given as long as the height of the tree does not exceed some value defined by a balance criteria Andersson [1999], Galperin and Rivest [1993]. In Andersson [1999], the author defines α $log|weight(\sigma)|$, where α is some constant > 1. *weight(σ)* refers to the weight of any process σ as defined in Section 3.

Thus, as long as the tree is not too high, nothing is done. Otherwise, we walk back up the tree, following a process insertion for example, until a node σ (usually called a *scapegoat*) where $height(\sigma) > \alpha log|weight(\sigma)|$, is observed. Thus, a partial rebuild of the subtree starting from the scapegoat node is made. Many partial rebuilding techniques can be found in the literature as in Galperin and Rivest [1993].

Based on these balancing policies, in the following we describe how to keep our ***Parent*** tree balanced after the achievement of atomic operations handled by processes in the ELMP algorithm (see Section 3).

4.5.1 Balancing following new insertions

Our model channels new insertions such that flooding the root by new processes is avoided, since this could inhibit the handling of other requests. The following steps that are carried out by a new process σ summarizes the processing of insertion into the system.

1. σ first has to know some ρ, one of the other participants. With that information, it searches bottom up in the ***Parent*** tree to find the actual root r. Note that the root can be reached fast if we add adjacent links as in the BATON structure Jagadish et al. [2005, 2006].

2. σ tries to include η, a process on the path to its *blocked* list.

3. If η is in a *non-Idle* state, σ restarts with a certain delay at Step 1 and requests the same process ρ or another for the insertion issue. Note that η allows a limited amount of insertion requests per unit of time.

4. Once σ reaches the current root r that is *Idle*, r moves to another state, *Busy* for example and then:

   (a) It assigns an *ID* to σ with the highest value. This *ID* is used in case of a tie with another process, as in the case for departure.

(b) If we make a shape restriction of the tree, r tries to find a ***Parent*** for σ, probably down the tree, at a second-last node, that has less than m children Jagadish et al. [2006].

In case of lazy mode, instead of finding a second-last node, σ simply (after receiving the *ID*) inserts itself into ρ, the found process.

Afterwards, we back up along the path until a possible scapegoat node is found. Note that this is can easily be done since *height* and *weight* variables give appropriate information (for σ that is on the top) of the subtree. If this is the case, a partial rebuild is issued. Note that processes on the path of σ remain *blocked* until the subtree is known to be balanced.

4.6 Balancing following a token request

The requesting processing we have presented in Section 4.1 does not affect the shape of the tree. Indeed, at the end of a sending request, two processes (σ and the old root) exchange their positions. Thus, the structure of the tree remains unchanged.

4.6.1 Balancing following departure

The Exit strategy presented in Section 4.3 has to be slightly modified if we want to keep the tree at an order m. σ that is *Exiting* will simply find another leaf process as replacement that inherit all needed information, rather than making connection between ***Parent*** and children, neither a new ***Parent*** among the list of children.

In case of lazy mode, assume σ that has not yet completed its departure becomes on the path of a partial balanced restructuring. Based on this information, the ***Parent*** and subtrees on the top compute again their *height* and *weight* variables and seek again a possible scapegoat process.

5 The proof of the ELMP algorithm

In this section, we will give formal proofs of the *liveness* and *Safety* properties. The notion of *liveness* asserts that the **critical section** remains accessible for any *Requesting* process. *Safety* ensures the exclusivity of access to the **critical section**.

Lemma 2 *As soon as a request of σ reaches a tree-node ρ no other request of a process below ρ can overtake it.*

Proof: The ***Parent*** branch switch to *Busy* state upon being informed of the request of σ. Thus, even if a request of another process σ' that is launched after that of σ

may move up in the tree, it will meet a *Busy* process. Thus, such a request cannot progress before the one of σ is finished, it is the new root of the tree and *Idle*. □

Lemma 3 *The* **Next** *and* **Predecessor** *variables form a doubly linked list.*

Proof: The **Next** and **Predecessor** pointers are only set to a non-NULL value during the handshake between the actual root r and the the inserting node σ, and then point to each other. **Next** is only set to NULL when the process hands token to its **Next**; **Predecessor** is only set to NULL when the process receives the token. □

Lemma 4 (departure) *A process σ that wants to leave the system can do so within a finite time.*

Proof: First consider a departing node σ that is not the root of the tree and that is the only process in the system that is departing. Any child η of $sigma$ will either be *Idle* (and switch to *Blocked*) or be requesting the token for itself or some descendant process. For the later, at the end of processing the request η's ***Parent*** will point to the actual root of the tree, and thus not be a child of σ anymore. A similar argument holds for σ's ***Parent***: it may be in an non-*Idle* state for some time, but at latest as it has processed token request from all its children, it will become *Idle* again. Thus, after a finite time, all neighbors of σ will be *Blocked*, and σ may leave the system.

Now suppose in addition, that there are other departing processes. A neighbor η_0 could eventually be *Blocked* (by η_1), η_1 *Blocked* (by η_2), etc, but since our system is finite, such a blocking chain leads to an unblocked vertex η_k that is departing and that has no departing neighbors. Thus, the departure of η_k will eventually be performed, and so the departures of all $\eta_{k-1}, \ldots, \eta_1$. Thus η_0 will eventually return to *Idle* and then either leave itself or be switched to *Blocked* (by σ).

Observe that if η_0 is the ***Parent*** of σ and has an *ID* that is lower than the one of σ, it will leave the system before σ and σ may eventually become root.

Now, if σ also is the root, we have three possibilities:

- Another process requests the token eventually and σ will cease to be root.
- Another process ρ inserts itself to the system. σ will cease to be root.
- Any child η of σ in *children* will either depart from the system or will eventually become *Idle*. Then σ will be able to enter in a handshake with η and switch it to *Blocked*. Since *children* is finite and no new processes are added to it, eventually all children of σ will be *Blocked* and listed in *blocked*.

Finally observe that only a finite number processes can have an *ID* that is smaller that the one of σ. Thus σ while waiting for its departure, it can become root at most $ID - 1$ times □

Lemma 5 *The **Parent** tree as well as the doubly linked list are never disconnected.*

Proof: As long as there are no disconnections from the system (*blocked* state and Lemma 3), the ***Parent*** tree and the doubly linked list remain connected in the **ELMP** algorithm.

In case of departure of a given process σ, the ***Parent*** tree remains also connected since during the effective departure of that process all neighbors are *Blocked* until they receive a new ***Parent***. □

Lemma 6 *No other request of a process σ' that arrived later at the root can be completed before the request for σ.*

Proof: Since no other request of a given process σ' can be completed before the previous inserted request of σ (Lemma 2), we show that it is also the case during:

1. The departure of the ***Parent*** of a process σ.
2. Insertion of new processes to the actual root.
3. The tree restructuring process.

For (1), the ***Parent*** of a given process σ can not leave the system unless it is in the *Idle* state. Since it is *Idle* before switching to *Exiting*, no request of another process σ' on the path of σ can be in progress.

Furthermore, the *Blocked* state of the neighbors ensures that all necessary information related to any operation triggered by processes of this branch of the tree is conserved. Finally, on completion of the departure process, there is always a process that inherits all necessary information held by the leaving process.

For (2), new insertions never compromise the order of requests at the root, whatever the number of processes. Indeed, a process σ that wants to inserts itself to the system first checks the availability of processes on its path to the root (Section 4.5.1). Otherwise, σ tracks back. Thus, there is no request in progress which can not be achieved.

For (3), whichever strategy is adopted for the tree-balancing, the *blocked* state avoids any overlap between operations handled by the **ELMP** algorithm. □

Theorem 1 (Liveness) *If a given process σ claims the **critical section** it may access it within a finite time.*

Proof: This now follows directly from the following facts:

- The waiting time for the completion of a request is finite.
- The ***Parent*** tree as well as the doubly linked list are never disconnected (Lemma 5)
- **ELMP** guarantees that requests are treated as the same chronological order as their reception at the current root, even in case of departure or new insertions (Lemma 6).

□

Theorem 2 (Safety) *At any given time t, there is exactly one token in the system. In other words, the system guarantees that at most one process carries out the **critical section**, and additionally, that the token never gets lost.*

Proof: Initially, there is one token in the system, it is held by the *root*, p_0. As the **ELMP** algorithm evolves, the token is passed from one process to another across the linked list (**Next**, **Predecessor**). Whenever no process has claimed the token, it remains at the current root of the ***Parent*** tree.

A process σ is only leaving the system if it has not requested the token. If it holds the token without having requested it, we are in the situation of Lemma 1, that is σ is the root of the tree and no other process has requested the token. In such a case, the token is passed to the new root of the tree. □

6 Transparent distributed data management

The *peer-to-peer* GRID system we present in this section provides and manages concurrent access to critical remote resources. It ensures consistency and availability of that resources distributed over *peers* that may appear and disappear. To achieve these objectives, we use an API called **DHO** Gustedt [2006]. We propose to implement that API over a multi-level architecture that includes **ELMP** algorithm. With a set of functions, **DHO** introduces an interface that mediates between the abstract concept of a data resource and its concrete realization in a process' address space.

The main idea is to allow users to claim and manage remote resources in their local memories, simply by inserting some functions in their existing code. Then, an application process (*peer*) may gain access to a specific data without knowing if that resource is already present locally or on a remote machine. For example, the function `dho_ew_request(DHO_t* h)` takes only an internal structure (called

handle) as an argument. This then encapsulates all the necessary information about the data request.

The important function to call prior to any request is:

```
dho_create(DHO_t* h, char const* name)
```

This function initiates the *handle* for a resource named `name`. All remaining functions then only use that *handle* to specify the linked resource. Applications using **DHO** routines need at least one *handle* per resource

6.1 Cooperation model with DHO

Two asynchronous processes are assigned in the local space of the same *peer*, the *resource manager* and the *lock manager*.

They interact locally with each other but may act separately by interacting with other processes, commonly with those of the neighborhood. Both processes toggle between different states according the progress of local requests and of the distributed environment.

As a reaction to any of the **DHO** function calls, the *resource manager* is contacted, first. It manages the data resource for the local *peer*. It maps that resource into the local address space and transfers the current version of it to or from another *peer*.

However, to obtain exclusivity on the resource and to know its location, the *resource manager* forwards the request to the second process, the *lock manager*. The *lock manager* remotely negotiates the *locking* of the data with other *lock managers*. As a whole, the *lock managers* of all *peers* ensure the overall consistency of the data according to the ELMP algorithm (Section 3).

During negotiation phases, the *peer*, the *resource manager* and the *lock manager* keep inherent information about their current activities by means of assigned states that are saved in the *handle*. Fig. 6 outlines states of the *resource manager* making up the **DHO** life cycle. The *locking/mapping* is effectively done when the *lock manager* acquires the token.

Figure5 illustrates our architecture with two *peers*, with **DHO** sample code that claims the same data.

In the following, we describe the path of a request through different processes inside and outside the same *peer* and finally, how resources are mapped into local memory.

6.2 DHO life cycle

Now, we explore the progress of a given request from the call of `dho_ew_request` function up to the *locking* phase. Since we will be interested in the performance

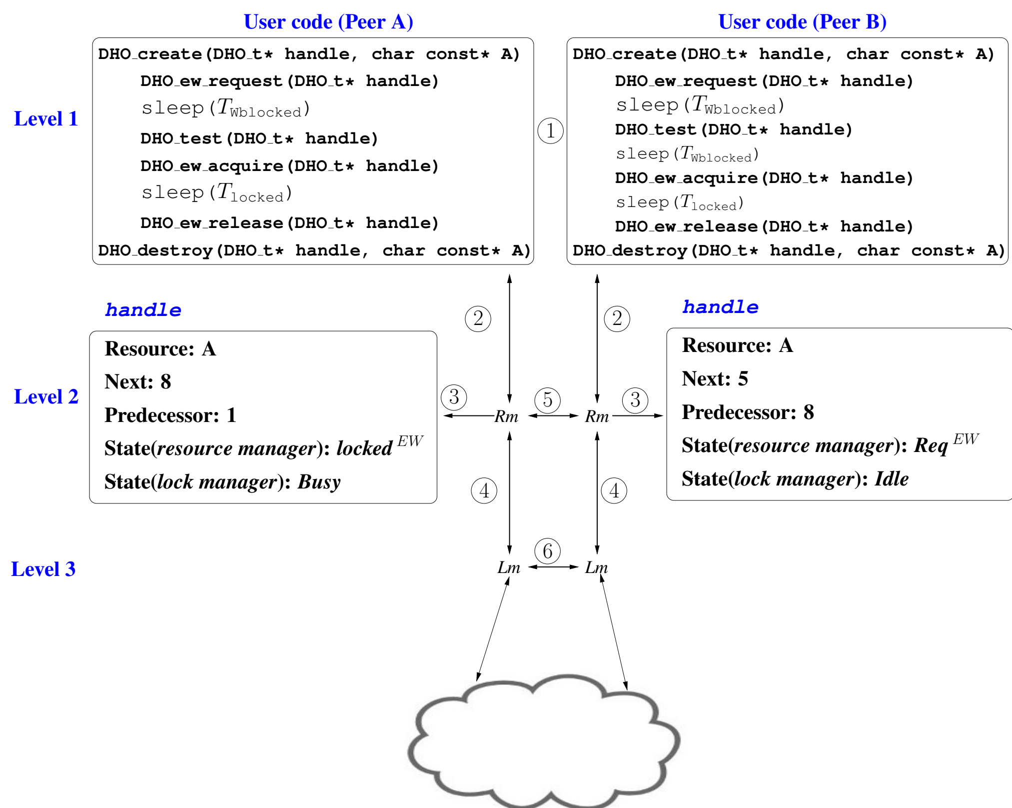


Figure 5: Handling requests between two neighboring *peers*. *Rm* and *Lm* that are respectively the *resource manager* and the *lock manager* are abbreviated for readability

of our approach, using notation with "T_{Name}" during that description we will also name some of the delays of these phases.

The main phases of **DHO** and states through which a *resource manager* and a *lock manager* go are designed to form a cycle. We emphasizes various steps and interactions between processes by the corresponding numbered circles of Fig. 5. The life cycle of a **DHO** *locking* and *mapping* request is described as follows:

Let σ be a given *peer* that is initially *Idle*. The corresponding *resource manager* will be then *valid* just following the call of **DHO** `create` ①.

1. Following the call of `dho_ew_request`, the *peer* becomes *Req* and the *resource manager* takes control ②. Below, the corresponding *resource manager* sets the *handle* to the *req*$^{\text{ew}}$ state ③ and forwards the request to the *lock manager* of the same *peer* ④. The latter asks its ***Parent*** for the token and becomes *Requesting*. The *lock manager* goes back into the *Idle* state upon completing the request and moves to the root position in the ***Parent*** tree (Section 4.1).

2. The *lock manager* may expect any requests from its *Children*. It can then become *Busy* or *blocked* by another *peer*.

3. Upon completing the routing request, the *peer* is placed at the head of the linked list **Next-Predecessor**. The *lock manager* expects the token from its **Predecessor**. Meanwhile, the application itself may continue while *resource manager* is (T_{Wblocked}) regardless if the resource has been acquired. T_{Wblocked} refers the to computation of the application that may occur regardless of the fact that a resource has been claimed. **This sentence is bizarre, here:** According to the **DHO** life cycle. The *peer* calls `dho_test`, ①. The corresponding *resource manager* ② queries the *lock manager* ④ if it already has got the token. Now, two cases may occur (see Fig. 6):

   (a) If the lock is held, the *resource manager* switches to *grant*$^{\text{ew}}$ state and updates the *handle* ③. By $T_{\text{WaitGrant}}$, we will denote the time to achieve this state.

   (b) Otherwise, the *peer* calls the `dho_ew_acquire` function and then, regains the *Blocked* state. Likewise, the *resource manager* updates the *handle* for the *blocked*$^{\text{ew}}$ value ③. The time that the *peer* waits until that call will be denoted T_{Wblocked}.

4. After the **Predecessor** has released the resource, it forwards the token to its **Next** that is *Requesting*, *Idle* or *blocked* ⑥. Once the token is acquired, the *lock manager* immediately informs ④ the *resource manager* that amends the state ③ for the *grant*$^{\text{ew}}$ value.

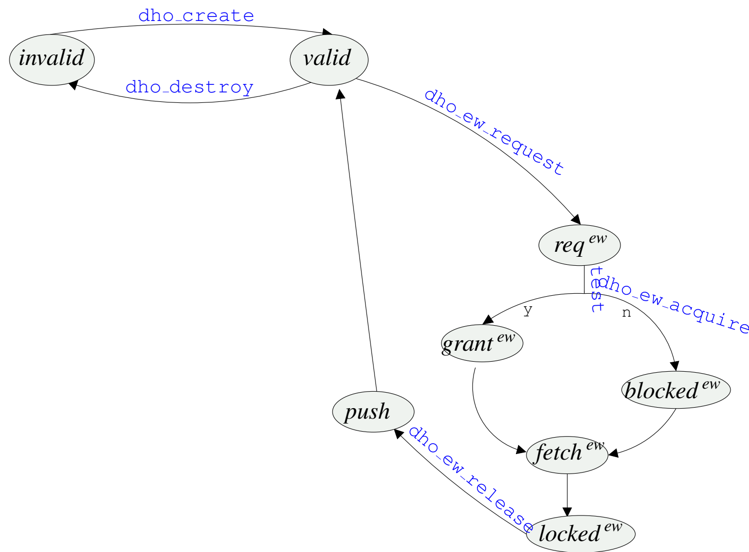


Figure 6: State diagram of *resource manager*

5. At that point, the *resource manager* enters the intermediate $fetch^{ew}$ (T_{fetch}) state to fetch the data from its **Predecessor** ⑥ and then to *map* ② the data into to the address space of the *handle*.

6. Once the *mapping* is done, the *resource manager* and the *peer* become respectively $locked^{ew}$ and *Locked*.

7. The cycle is completed by a call to `dho_ew_release`. The *peer* becomes *Idle*, again. Now, the *resource manager* is in the *Unlock* state. This is an intermediate state during which the *peer* has already released the resource, although the corresponding *lock manager* still holds the token and is ready to forward it to a possible **Next**. After that, the *resource manager* will be *valid* , again.

In an abstract way, we can say that the *peer* (represented by **DHO** functions), the *resource manager* and the *lock manager* form a three level hierarchical architecture, where the *lock manager* carries out instructions of the **ELMP** algorithm at lowest level. The access is granted according to a FIFO *access control policy* and the data is then presented to the application inside its local address space.

Table 1 presents the hierarchical order of possible states caused by successive events triggered from the three levels.

The overall architecture is constructed to allow an overlap of the activities of the different processes. For example, by the set of states $\{Req, fetch^{ew}, Busy\}$, we can easily deduce that there are several activities that are simultaneous handled by the same *peer*. Hence, that *peer* has issued a request that is currently in *mapping*

p	*Idle*				*Req*						*Blocked*		*Locked*	
h	*valid*		*Unlock*		*req*^{ew}		*grant*^{ew}		*fetch*^{ew}		*blocked*^{ew}		*locked*^{ew}	
lh	*blocked*	*Busy*	*blocked*	*Busy*	*Requesting*	*Busy*	*blocked*	*Busy*	*blocked*	*Busy*	*blocked*	*Busy*	*blocked*	*Busy*

Table 1: List of combined states. For readability, the *Idle* state of the *lock manager* is not represented. *p*: *peer*, *h*: *handle*, *lh*: *lock manager*

Listing 1: An example of an out of order DHO cycle

```
DHO_create(handle , A);
DHO_ew_request(handle);
sleep(T1);
DHO_ew_request(handle); // The previous request is dropped
sleep(T2);
DHO_ew_release(handle); // Ignored
DHO_ew_release(handle); // Ignored
DHO_create(handle , A);
```

phase. Meanwhile, the associated *lock manager* may deal with another request that it received.

6.3 Mobility of *peers*

The `dho_destroy` function implements a voluntary departure of the hosting *peer* regardless of the currently assigned state. Note that all **DHO** functions that follow after `dho_destroy` are ignored since the *resource manager* is *invalid*.

As a general policy, an application may chose not to respect the logical order of the calls to **DHO** functions as presented in level 1 of Fig. 5. The consistency of the locks is always guaranteed even if a **DHO** cycle is broken or canceled. The concerned *peer* (as in Listing 1) will just loose its acquired FIFO position in the queue of requests.

Once the *resource manager* receives the departure request from the corresponding *peer*, it informs the corresponding *lock manager*, and this *lock manager* carries out the departure part of the **ELMP** algorithm (Section 4.3). First, the *lock manager* switches to *Exiting*. Then, it forwards the token to its **Next** or to one of its neighbors, while the corresponding *resource manager* invites that neighbor to *mapping* the data. At the end of the disconnecting process, the *resource manager* destroys the *handle* by assigning an *invalid* state. Finally, the *peer* enters the *Exit* state for the resource.

The `dho_create` function makes the *handle valid*, again. The *peer* is connected to a given ***Parent*** according to the **ELMP** algorithm and according to the

Listing 2: Benchmark with DHO

```
char const* name;
dho_t *a;
double T_Idle, T_WBlocked, T_Lock;
dho_create(name, &a);
do {
  sleep(T_Idle);
  dho_ew_request(a);
  sleep(T_WBlocked);
  dho_test(a);
  dho_ew_acquire(a);
  sleep(T_Lock);
  dho_ew_release(a);
} while(time < 100);
dho_destroy(&a);
```

adopted balanced strategy (Section 4.5.1). From that point onward, the *resource manager* will be able to handle exclusive requests submitted by the user, whilst the *lock manager* will be ready to deal with those coming from *children* in the ***Parent*** tree.

7 Performance evaluation

An experimental study of the **DHO** library according to a client/server pattern was already reported Hernane et al. [2011]. We use the same experimental environment, a socket based library belonging to the SIMGRID toolkit, see Velho and Legrand [2009] and the same benchmark (Listing 2). As we can see, the code is written independently of the underlying structure.

In this benchmark, *peers* carry out 100 **DHO** cycles by requesting the data resource 100 times. Results refer to average values.

Three durations, T_{Idle}, T_{Wblocked} and T_{Locked} have been varied. These are application dependent and may refer to computational periods in real applications. T_{Idle} denotes the delay between two calls, T_{Wblocked} is the time that the *peer* waits until the call of `dho_ew_acquire`, while T_{Locked} is the *locking* time, that is the time that the application spends inside the **critical section**.

We performed experiments simulating a realistic platform (GRID'5000).

Besides the **DHO** cycle duration, Equation (3), we will analyze T_{Wait}, Equation (1), and T_{Blocking}, Equation (2). T_{Wait} is the waiting time of a request, *i.e* the time between the call to request and the completion of the data fetching opera-

tion. T_{Blocking} is the time a *peer* is blocked before acquiring the resource, *i.e.* the time between the call to acquire and, again, the completion of fetching. They are expressed by the following equalities, see Section 6.2 above for the different times:

$$T_{\text{Wait}} = T_{\text{WaitGrant}} \mid T_{\text{Wblocked}} + T_{\text{grant}} \mid T_{\text{blocked}} + T_{\text{fetch}} \quad (1)$$

$$T_{\text{Blocking}} = T_{\text{blocked}} + T_{\text{fetch}} \quad (2)$$

$$T_{\text{Dho}} = T_{\text{Idle}} + T_{\text{Wait}} + T_{\text{Locked}} \quad (3)$$

7.1 DHO cycle evaluation with synchronous locks

We aim to asses the ability of our *peer-to-peer* GRID system to deal with competing requests for the same resource that are issued simultaneously. Since this is the setting that stresses the platform the most, for the remainder of our discussion we will set T_{Idle} to 0. That is, *peers* insert a new request as soon as the previous cycle is finished. We also have verified that our system is in no way disturbed when dealing with different values for T_{Idle}. Doing so, simply extends the overall application time by the added T_{Idle} value. So we will not go into details for scenarios that vary T_{Idle}.

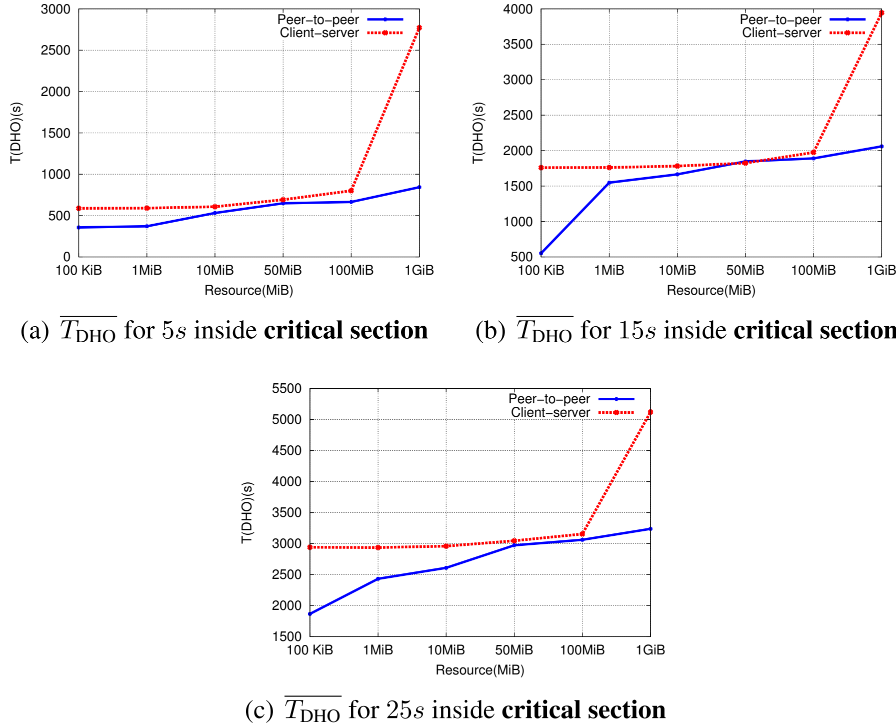
First, we are looking into the dependency between T_{Locked} and the size of the data resource. First, we focus on synchronous *locks* with $T_{\text{Wblocked}} = 0$, that is where the *peer* and the *handle* are blocked as soon as the request is issued. This corresponds to a classical lock sequence that does not separate lock request and acquisition.

Results in comparison to the previous client-server based approach are shown in Fig. 7. We observe that the *new peer-to-peer* algorithm behaves much more regularly, in particular for the extremes of the resource size. By doing some regression of the values we find that the cycle time behaves as:

$$\overline{T_{\text{DHO}}} \simeq [\overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \alpha \cdot \eta](\overline{q} + 1) \quad (4)$$

Where:

- $\alpha = 10.4$ is a constant value that denotes a correction factor in the communication model of SimGrid, see Velho and Legrand [2009].
- $\eta \simeq 430\mu s$ is an observed value that represents the cost produced by the various exchanges between managers, namely between neighbors.
- q denotes the length of the request FIFO, represented by the doubly linked list (**Next**, **Predecessor**).



(a) $\overline{T_{\mathrm{DHO}}}$ for $5s$ inside **critical section**

(b) $\overline{T_{\mathrm{DHO}}}$ for $15s$ inside **critical section**

(c) $\overline{T_{\mathrm{DHO}}}$ for $25s$ inside **critical section**

Figure 7: Cycle time, $\overline{T_{\mathrm{DHO}}}$ in the *Client-server* paradigm and the *peer-to-peer* GRID system.

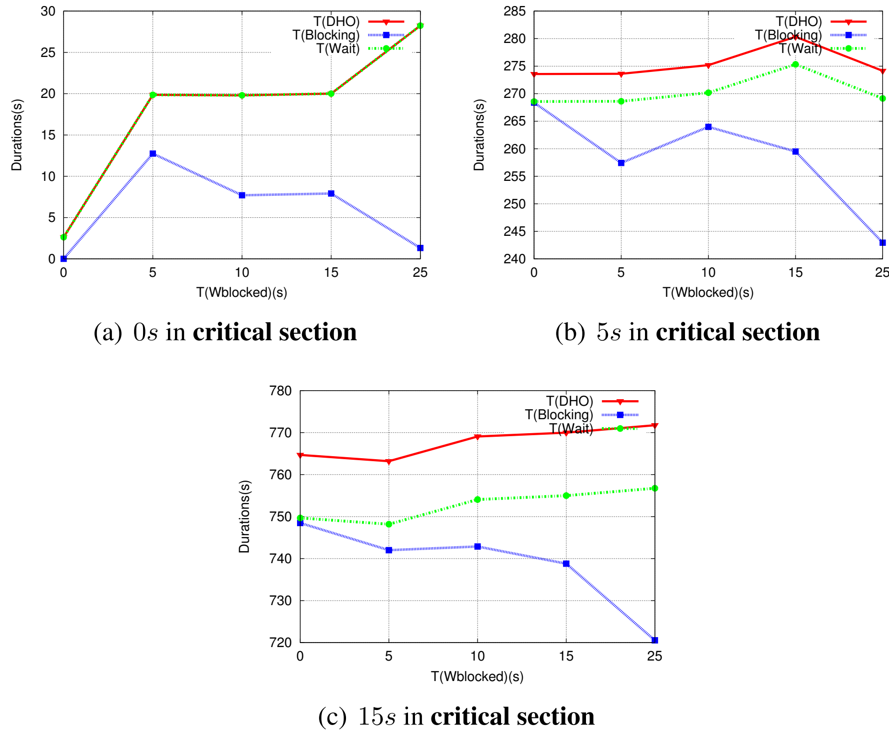
η has been approximately $70\mu s$ in the *client-server* paradigm. Indeed, an extra delay arises for the management of the *peer-to-peer* GRID system, specially for the partial conversion of the tree structure that requires a great number of messages between members of a given neighborhood.

Conversely, we observed that $\overline{q}$ was significantly reduced compared to the *client-server* paradigm where the length was the number of pending requests. Indeed, our distributed system involves various processes that operate simultaneously during the query processing such that the bottleneck of a centralized server is avoided (see Table 1). The involved processes are mainly those from the neighborhood of the *peer* that has inserted a request. For example, the *mapping* phase involves *resource managers* related to the *peer* and its **Predecessor**. Meanwhile, nothing prevents the associated *lock managers* to deal with possible arrived requests. If the *fetch*$^{\mathrm{ew}}$ state is time consuming in cases of large data sizes, the corresponding *peer* may simultaneously engage in other related operations of the *peer-to-peer* GRID system.

7.2 DHO cycle evaluation with asynchronous locks

A second series of experiments now concerns a setting that uses non-blocking locks, that is they distinguish a resource request and resource acquisition. Applications as above with an expected T_{Wblocked} time of 0 are strongly dependent of the resource, whilst those with a significant value of T_{Wblocked} may make progress a while acquiring the resource asynchronously.

Here, after an application dependent time T_{Wblocked}, the `dho_test` function returns the state of the *handle*. If *grant*$^{\text{ew}}$, then `dho_ew_acquire` just acts as an intermediate phase for the *fetch*$^{\text{ew}}$ state before then switching to that of *locked*$^{\text{ew}}$ (Fig. 6). This series of benchmark is conducted with 50 *peers*. The data resource



(a) $0s$ in **critical section**

(b) $5s$ in **critical section**

(c) $15s$ in **critical section**

Figure 8: Average duration of $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ and $\overline{T_{\text{Blocked}}}$ by varying T_{Wblocked}.

size is fixed to $50MiB$. Fig. 8 shows the observed delays (T_{DHO}, T_{Wait} and T_{Blocking}) with a set of experiences that fixes T_{locked} and vary T_{Wblocked}. With $T_{\text{locked}} = 0$ and $T_{\text{Wblocked}} = 0$, (Fig. 8(a)), *peers* request then the resource once the mapping is completed. In this case, it is clear that T_{DHO} corresponds to T_{Wait}, so the lines are superimposed.

Also, we note that T_{Wait} slightly increases in case of non-zero values of T_{Wblocked}, Figs. 8(b) and 8(c), but this is not due to an extra latency for receiving the token.

In fact, the *resource manager* assigns the *granted*^{ew} state to the *handle* right after being informed by the *lock manager* that the token has been acquired. The growth of T_{Wait} rather reflects that the grant is taken a bit later (T_{grant}) because of the increased application delay T_{Wblocked}.

From Figs. 8(b) and 8(c), we can conclude that if $5s$ is taken for T_{Wblocked}, a good overlapping is provided for the application, specially between computation and data transfer.

7.3 DHO cycle evaluation with mobility of peers

The last series of benchmarks concerns the mobility of peers. We aim to measure the overhead that is produced by removing and joining *peers* from and to the remaining system.

We divide the set of *peers* into two parts:

1. *peers* in the first subset perform a complete cycle, as that of Listing 2.
2. In the second one, *peers* perform an out-of-order cycle as given in Listing 1.

Note that the use of `dho_destroy` and `dho_create` functions imply mobility of the half of the system. Once `dho_destroy` is issued, the *resource manager* destroys the *handle* and becomes *invalid*. Then, all following **DHO** functions are ignored. Thus, the *lock manager* performs the **Exit strategy** and the **Insertion process** (see section 4.5.1) of the **EMPL** algorithm.

We only show the duration of uninterrupted DHO cycles that form the first class, namely for the cases that 25%, 33% and 50% belong to the second class, respectively.

The overhead is approximately the same for both sizes (Table 2) and the additional latencies introduced by the departure of *peers* are negligible. Indeed, the adopted structure and balancing strategies guarantee logarithmic complexity of messages in all operations. We would expect, however, a relative increase of these values with an even higher mobility frequency of *peers*.

Disconnection	50 *peers*	120 *peers*
25%	$2.27s \pm 0.842\%$	$4.27s \pm 0.725\%$
33%	$2.65s \pm 0.98\%$	$6.46s \pm 1.05\%$
50%	$4.29s \pm 1.56\%$	$10.34s \pm 1.68\%$

Table 2: The overhead caused by a group of disconnecting *peers* on **DHO** cycle durations of the remaining *peers*.

8 Conclusion and Futur Work

We have presented a *peer-to-peer* Grid system and API for the design of parallel and distributed applications that integrates seamless handling of data resources. The **DHO** API aims to ease development of resoure-intensive applications, while the architecture upon which the system is based on guarantees transparency, robustness, flexibility and scalability. The big challenge we have been addressing is to achieve all these properties along with a consistent data access between users who are spread over a large distributed environment that evolves in time.

The **ELMP** algorithm ensures the required consistency. It also guarantees dynamicity and extensibility of the basic structure. *Liveness* and *Safety* properties of the corresponding algorithm have been demonstrated. The balanced tree structure guarantees scalable performance. All operations within the **ELMP** algorithm have a logarithmic cost, accounted in the number of messages that are issued per operation.

The proposed *peer-to-peer* Grid system interfaces **DHO** routines and **ELMP** algorithm. The *resource manager* and the *lock manager* introduce independent levels in the architecture. Still, they cooperate closely together in response to various concurrent requests that are made by users. Non-blocking functions provided by **DHO** allow to overlap computations and data transfer. Additionally, a second type of overlap may occur between the different task that the two managers perform.

Indeed, even though they interact together to ensure consistency of the application, they may carry out some tasks independently. The experimental study has show an interesting property of this schema, namely the possible load sharing between the two managers.

We have presented the first evaluation of our design. However, future works are heading towards different directions. The API and library could be enhanced by further: by functions that enable safety locking part of the same data resource (mainly of large size) by different processes.

In the original **DHO** interface Gustedt [2006] there is a `DHO_duplicate` function to achieve this goal. It duplicates an existing *handle* but takes additional arguments: `offset` and `length` which specify the desired subrange of the data. To integrate this concept, an extended amended version of the entire design is required. It will be interesting to address fault-tolerance issues. The current approach deals voluntary departures of *peers* and not unexpected ones. Finally, we plan to test the present approach with real world applications.

References

D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, Nov. 2002. ISSN 0001-0782. doi: 10.1145/581571.581573. URL http://doi.acm.org/10.1145/581571.581573.

A. Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, 1999. doi: 10.1006/jagm.1998.0967. URL http://dx.doi.org/10.1006/jagm.1998.0967.

G. Antoniu, L. Bougé, and M. Jan. Juxmem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3), 2005. URL http://www.scpe.org/index.php/scpe/article/view/336.

G. Antoniu, L. Cudennec, M. Jan, and M. Duigou. Performance scalability of the JXTA P2P framework. In *Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10. IEEE, March 2007. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4203121.

P. Arquet. Introduction à MPI – Message Passing Interface, 2001. www2.lifl.fr/west/courses/cshp/mpi.pdf.

Y. Caniou, E. Caron, G. Le Mahec, and H. Nakada. Transparent Collaboration of GridRPC Middleware using the OGF Standardized GridRPC Data Management API. In *The International Symposium on Grids and Clouds (ISGC)*, page 12p. Proceedings of Science, February 26 - March 2 2012.

Y. Caniou, H. Croubois, and G. L. Mahec. Standardized multi-protocol data management for grid and cloud gridrpc frameworks. In *Globe'14*, pages 61–72, 2014.

G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: a generic global computing system. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*. IEEE Computer Society, 2001.

I. Galperin and R. L. Rivest. Scapegoat trees. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 165–174, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics. ISBN 0-89871-313-7. URL http://dl.acm.org/citation.cfm?id=313559.313676.

J. Gustedt. Data Handover: Reconciling message passing and shared memory. In J. L. Fiadeiro, U. Montanari, and M. Wirsing, editors, *Foundations of Global Computing*, number 05081 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. URL http://drops.dagstuhl.de/opus/volltexte/2006/297.

S. L. Hernane, J. Gustedt, and M. Benyettou. Modeling and experimental validation of the data handover API. In J. Riekki, M. Ylianttila, and M. Guo, editors, *GPC*, volume 6646 of *Lecture Notes in Computer Science*, pages 117–126. Springer, 2011. ISBN 978-3-642-20753-2.

S. L. Hernane, J. Gustedt, and M. Benyettou. A dynamic distributed algorithm for read write locks. In *PDP*, pages 180–184, 2012.

H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *In VLDB*, pages 661–672, 2005.

H. V. Jagadish, B. C. Ooi, K.-L. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 1–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: 10.1145/1142473.1142475. URL http://doi.acm.org/10.1145/1142473.1142475.

L. Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978. ISSN 0001-0782.

M. Maekawa. An algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3:145–159, May 1985. ISSN 0734-2071.

D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. 2002.

M. Naimi and M. Tréhel. How to detect a failure and regenerate the token in the $\log(N)$ distributed algorithm for mutual exclusion. In *Proceedings of the 2nd International Workshop on Distributed Algorithms*, pages 155–166, London, UK, 1988. Springer-Verlag. ISBN 3-540-19366-9.

M. Naimi, M. Tréhel, and A. Arnold. A $\log(N)$ distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34:1–13, April 1996. ISSN 0743-7315.

openmp. The OpenMP API specification for parallel programming. www.openmp.org.

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

G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24:9–17, January 1981. ISSN 0001-0782.

J. Sopena, L. B. Arantes, M. Bertier, and P. Sens. A fault-tolerant token-based mutual exclusion algorithm using a dynamic tree. In *Euro-Par'05*, pages 654–663, 2005.

P. Velho and A. Legrand. Accuracy study and improvement of network simulation in the SimGrid framework. In *Simutools '09*, pages 1–10, Brussels, Belgium, 2009. ICST. ISBN 978-963-9799-45-5.

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique

CS20101

54603 Villers-lès-Nancy Cedex

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

ISSN 0249-6399