Data handover on a peer-to-peer system

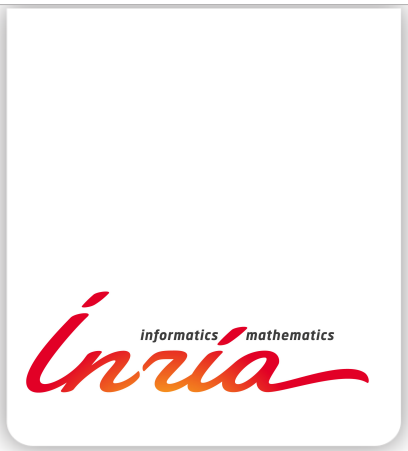
Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou

Data handover on a peer-to-peer system

Soumeya Leila Hernane, Jens Gustedt, Mohamed Benyettou

RESEARCH REPORT
N° 8690
February 2015
Project-Team AlGorille

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 AlGorille

Research Report n° 8690 — February 2015 — 30 pages

Abstract: We propose a *Data Handover* API for an efficient management for *locking* and *mapping* data in extensible distributed settings, on the top of a three level structure. At the lowest level, the *lock manager* ensures consistency in the logical order of requests, although involved processes in the request processing may leave the system. This is what we describe in our proposal **ELMP** algorithm. We also deal with the flooding issue of insertion of new processes. All operations have an amortized cost of $O(\log n)$. In the middle of the structure, the *resource manager* carries out requests sent at the applicant level through **DHO** routines, interacts with the *lock manager* and finally maps the resource in the local memory. An experimental assessment validates the practicality of our proposal.

Key-words: Consistency data access peer-to-peer *resource manager lock manager locking mapping*

RESEARCH CENTRE
NANCY – GRAND EST

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

Data handover on a peer-to-peer system

Résumé : Nous proposons dans ce papier une extension de l'algorithme d'exclusion mutuelle, à base de jeton de Naimi-Tréhel. Notre contribution se présente sous deux angles différents.

Premièrement, notre algorithme progresse dans un environnement inconstant. Les processus peuvent joindre et quitter le système. En même temps, l'arbre ***Parent*** subit des transformations au fur et à mesure que les requêtes d'accès à la ***section critique*** sont insérées. Les modifications portent aussi bien sur l'algorithme que sur la structure arborescente ***Parent*** ainsi que sur la chaîne **Next**.

De ce fait, Nous imposons de nouvelles règles et de nouvelles variables aux structures de départ, de sorte que la connexité de l'arbre ***Parent*** ainsi que celle de la chaîne **Next** soient maintenues.

Deuxièmement, nous rendons possible l'agencement du jeton partagé à l'exclusif. Ainsi, la ***section critique*** devient accessible, soit par plusieurs lecteurs concurrents soit par un seul écrivain. Dans la chaîne **Next**, le est introduit pour assurer l'entrée en ***section critique*** de tous les lecteurs successifs. De même, le garde le jeton partagé tant qu'au moins un lecteur est en ***section critique***.

Dans tous les cas de figure, l'implantation de notre approche garantit une complexité logarithmique de l'ordre de *O(log(n))* messages par requête.

Mots-clés : Cohérence d'accès peer-to-peer Gestionnaire de ressource Gestionnaire de verrou Verrouillage *mapping*

1 Introduction and Overview

Large-scale distributed computing systems are serving a growing number of scientists from all domains. While these environments bring the advantages of an economy of scale, their heterogeneous structure limits the efficient use of system resources. Moreover, efficient data utilization in parallel and distributed systems relates to models and paradigms that emanate from two classes of architectures: shared and distributed memory. Shared memory based programming environments are commonly *thread* based, while distributed memory architectures use message passing interfaces such as *MPI*. In grid and peer to peer environments, applications may consume resources and access data for reading or writing without prior knowledge where these are hosted. Therefore, paradigms mentioned aren't sufficient for such environments.

1.1 Motivations

We aim to provide an easy-to-use interface that handles data for exclusive or shared locks on large-scale distributed environments. We propose an advanced version of the *Data Handover* interface (**DHO**) that previously has been introduced in Gustedt [2006], Hernane et al. [2011].[1] **DHO** uses a mediator process that satisfies requests launched by users, and includes an abstraction level between resource and memory through a *handle*.

With a *handle*, users issue non-blocking request to lock a resource and regain local access to its data. The owner of that resource moderates such requests with respect to a strict FIFO strategy. Once the lock is acquired, access to the resource is realized by *mapping* it into the address space of the requester.

We aim to provide an architecture such that applications that use **DHO**, may progress in a scalable dynamic environment. Therefore, one of the goals of this paper is to replace the *Client-Server* paradigm that had been used for **DHO** in previous work by a *peer-to-peer* paradigm. In fact, we want to implement the mediator process within each client instead of realizing it in a separate, fixed, server entity. We seek the following properties from such a system:

The system should ease the use of the *Data Handover* API. A user that claims the resource locally by **DHO** functions should not know the complexity underneath even there are many different *peers* in the system. All technical information such as resources location and physical characteristics of the network must be hidden.

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

The user only needs to know identifiers of requested resources. If the request has been issued, the system should provide it within a finite time.

The library must be able to cope with all requirements for large scale distributed systems such as grids, e.g heterogeneity, scalability or mobility of *peers*.

peers may leave or join the system seamlessly for users. However, their movement should never compromise the correctness or the progress of applications.

1.2 Mutual exclusion algorithms

As **DHO** provides safe access to critical resources, it must use a consistent and efficient strategy for mutual exclusion. Several algorithms have been proposed over the years to solve mutual exclusion problems within distributed systems. They can be either permission-based (Lamport [1978], Maekawa [1985], Ricart and Agrawala [1981]) or token-based (Naimi and Tréhel [1988], Raymond [1989]). The second ones condition the entrance into the **critical section** by the possession of a token which is passed between 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. In continuation of this work, we propose another version of that algorithm.

1.3 contributions

The contributions of this paper are as follows:

1. We present the **DHO** interface for claiming and acquiring resources in an abstract way for users, on the top of a three level architecture, based on a *resource manager* and a *lock manager* at the lowest level.

2. Based on our previous work, Hernane et al. [2012], we propose the *Exclusive Locks with Mobile Processes* algorithm, **ELMP**, together with a

data structure that provide extensive scalability of processes, whilst ensuring consistency when accessing the **critical section**. **ELMP** algorithm is handled by the *lock manager* in the underlying structure.

3. 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 is constantly monitored during and after each atomic operation. Such a requirement avoids that the root is flooded by new insertions or that the the height of the tree increases unproportionally. All operations involved in maintaining a balanced tree-structure are still bound to a logarithmic scale.

1.4 Overview

The rest of this paper is organized as follows: after describing the basic of the Naimi and Tréhel algorithm in Section 2, in Section 3, we introduce the algorithm *Exclusive Locks with Mobile Processes*, **ELMP**, and the data structure in detail. Then, in Section 4, we present the **DHO** library and our proposal *peer-to-peer* system. Section 5 reports the results of experimental evaluation of our proposal before concluding in Section 6.

2 The Naimi and Tréhel Algorithm

The Naimi and Tréhel algorithm is based on a distributed queue along which a token circulates which represents 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 resumed 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, it is the same for all processes but for the initial root, it is set to *null*.

Processes send two kind of messages:

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

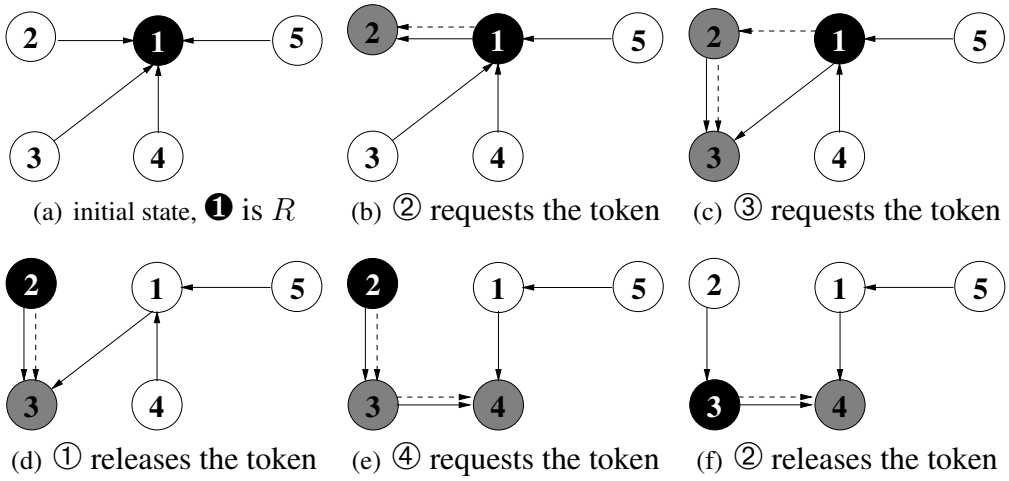
Token: sent by a process ρ to its **Next**.

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*.

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



(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

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.

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 isn't a tree but only a forest.

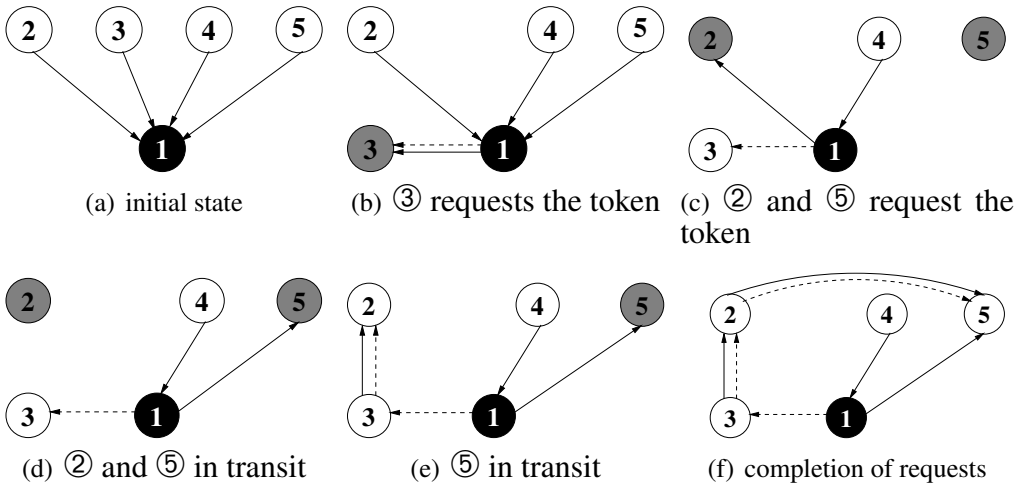


Figure 2: Example of concurrent requests in Naimi and Tréhel's Algorithm

3 An Algorithm for Exclusive Locks for Mobile Processes based on 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 basically being the time that is needed to forward information to the other processes.

2. Allow new nodes to join the system whenever possible.

3. Keep 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 that holds the token. The balanced shape of the tree is maintained according well known strategies. We report such possibilities in Section 3.2.

Beside of 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 3.1). 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 him. 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 3.2):

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

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

These variables are used for decision concerning the restructuring of the tree, for example to find a positions 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.

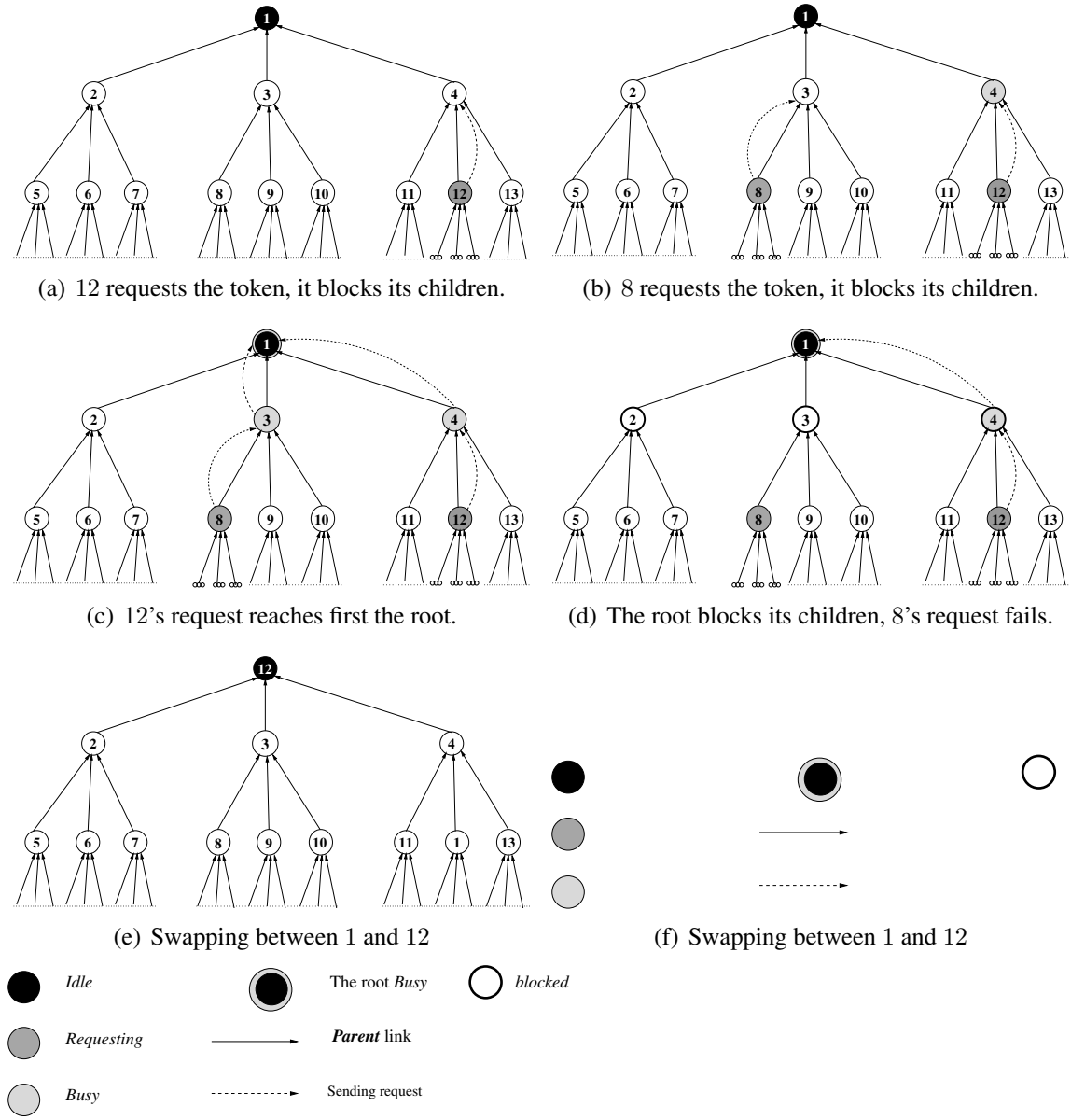


Figure 3: Handling concurrent requests in the **ELMP** algorithm

3.1 Exclusive Locks for Mobile Processes (ELMP) algorithm

In order to highlight the atomicity of operations in our system, we introduce a **State** variable for each process. It may have different values that indicate the specific task the process is currently completing. Fig. 3 exemplifies our approach and emphasizes on the state concept.

***Idle*:** In this state, a process σ is not involved in any operation, whether for himself or for others. However, it can hold the token. Its corresponding list *blocked* is *empty*. From *Idle* state, σ can switch to any other state. Besides process 1 that holds the token, in Fig. 3 all white circles denote *Idle* processes.

***Requesting*:** The process σ has started an insertion into the queue for requesting the token but that request is not yet completed (such as process 12 in Figs. 3(a) and 3(b)). It is not ready to receive any request as long as in this state. Dark gray circles denote the *Requesting* processes. *blocked* list will include all corresponding children (bold circles). Once in this state, σ operates in phases:

1. First it walks up the tree, if available, (*Idle* state) notifies processes $p_2, \ldots, p_n$ on the path about the insertion operation. They will all switch to the *Busy* state, but will not change their ***Parent*** pointer. Note that, that processes form a ***Parent*** branch of σ (as processes 4 and 1). and identifies $r = p_n$, the current root. If at least one process is not available, σ tracks back so that that processes will regain the *Idle* state, unblocks its children and starts over. This is the case of process 8 which in turn, tries to send a request to its ***Parent*** (process 3). It fails because it was preceded by that of process 12 at the root (Figs. 3(b) and 3(c)).

   Once the root is *Idle*, it in turn includes its children in the *blocked* list (processes 2, 3 and 4 (Fig. 3(d))).

   same path, but in modified order $r, p_2, \ldots, p_{n-1}$.
2. σ and the current root r exchange their positions and then their children that are still blocked. Thus, r, that is no longer root, and its children update their ***Parent***, before returning to the *Idle* state. This is the case of processes 2, 3, 4 as of children of process 12 (Fig. 3(d)).
3. σ the new root (process 12 in Fig. 3(e)) sends acknowledgments to the processes $p_2, \ldots, p_{n-1}$ on the path. It changes its ***Parent*** to `empty` and its **Predecessor** to r. Instead of returning to the *Idle* state, σ will rather be implied by the audit of the balancing of the tree (Section 3.2.2).

Busy*:** ρ handles an insertion request for another process σ (light circles as processes 4, 3 and the current root 1 in Fig. 3). It is not ready to forward other insertion requests yet, neither to be involved for the departure of another process. If ***Parent is **empty** (ρ is the actual root), it exchanges its position with σ (the new root). It also sets its **Next** variable to that process and updates its list of children (process 1).

From there, ρ switches back to the *Idle* state. It may request the token for itself, as it may be called again for further operations.

Lemma 1 *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 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: As soon as σ succeeds to notifying all processes on the path to the root, its ***Parent*** and all parental structure switch to the *Busy* state. Thus, even if a request of another process σ' that is launched after that of σ may go up some path in the tree, it will meet a *Busy* process. The request of σ' can't be undertaken before σ, the new root be *Idle*. □

We continue with the remaining states and corresponding atomic operations of our algorithm.

***Exiting*:** 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 in the middle of requesting the token or themselves leaving the system, or involved in the tree-restructuring for example.

***blocked*:** Closely tied to *Exiting* is the *Blocked* state. There are two possible scenarios:

- An other process ρ that is disconnecting successfully blocked σ. In fact, ρ will promote its children and its ***Parent*** to the *Blocked* state, such that they delay any requests that might be pending in their subtrees, as processes 12 and 1 in the example (Fig. 3). Among these

blocked neighbors, σ will chose another process ρ that will inherit all information that σ held for the system. Namely, the children of σ will become children of ρ and if σ held the token previously to its departure, ρ will do so thereafter. However, we also suggest the replacement method in case of maintaining the tree at order m (see Section 3.2.3).

- σ is part of an unbalanced branch of the tree. Thus, it will be blocked for a further repositioning in the tree. In Section 3.2.3, we present possible strategies for maintaining the tree as balanced in case of departure.

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 that of the linked list (**Next**, **Predecessor**), in particular. σ can not break away from its ***Parent*** neither from its children suddenly. It should rater find a successor.

Maintaining the fact that σ chooses another ***Parent*** would be more difficult. Therefore we adopt a "lazy" deletion property: any list item that is accessed by process ρ will first be checked for its validity. If the process σ in question is still accessible and its ***Parent*** points to ρ, σ is still a child of ρ. Otherwise, the list entry is invalid and dropped from the list.

To be able to switch to the *Exiting* state:

- The process σ must be *Idle*.
- It must not have requested the token.

The fact that σ must not have requested the token doesn't mean, that it can't actually *possess* it. The following lemma is immediate.

Lemma 3 *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.*

The following operations that are chronologically carried out by σ, summarize the effective departure form the system. During this departure σ will contact all its *neighbors* in the ***Parent*** tree, that is its children and its ***Parent***, if it has any.

1. σ verifies that it is *Idle* and that it has not requested the token.

2. σ switches from the *Idle* to the *Exiting* state.

3. For all its neighbors η, ***Parent*** last and children, σ initializes a handshake with process η:

   - If η is already *Blocked* (by σ), we encountered a duplicate entry in the *children* list. η is already in *blocked* and the entry in *children* is simply discarded.
   - If η is *Idle*, it switches to *Blocked* (by σ). σ moves η from its *children* to its *blocked* list.
   - If η is not *Idle*, σ waits until it is contacted by η at the same point of its exit procedure.

     Once σ and η meet on their departure request, 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.
   - In all other cases, σ switches back to *Idle* and restarts at 1.

4. Now all neighbors of σ are *Blocked* (by σ) and thus its *children* 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 3 and chooses ρ among its children.[2]

5. σ sends ρ to all its neighbors. ρ itself will discover by that message, that it has been chosen and if it will be the new root of the tree.

6. σ sends its list *blocked* (excluding ρ) to ρ.

7. σ waits for an acknowledgment from ρ that it has integrated the list into the list of its children.

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

The actions that the neighbors η of a disconnecting process σ have to perform are summarized as follows:

1. η is in the *Idle* or *Exiting* state and notes that state in an auxiliary variable s_0.

2. η switches to the *Blocked* (by σ) state.

[2] If it has neither ***Parent*** nor children, the system consists only of σ.

3. η waits to receive the name ρ of the process chosen by σ.
4. If $\rho = \eta$:
   (a) $\rho = \eta$ waits to receive the list of children from σ, and updates its list of children accordingly
   (b) If σ is the same as ***Parent***, $\rho = \eta$ and we are in the situation of Lemma 3. $\rho = \eta$ is the new root of the tree. It sets its ***Parent*** accordingly and notes that it possesses the token.
   (c) η signals σ that it has finished the update of its data structures.

   Otherwise η updates its ***Parent*** to ρ.
5. η waits that σ signals its departure and then switches back to state s_0.

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

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 ***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 1), 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*** (Steps 3 and 4 of the Exit atomic operation 3.1. □

3.2 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 make a shape restriction of the tree that should always be $m - order$. In Jagadish et al. [2005, 2006], authors proposed a balanced tree structure overlay on a peer-to-peer network, based on, firstly a binary balanced tree (BATON) and secondly on m-order trees (BATON*), wherein joining and departures of nodes are well computed and take no more than $O(\log n)$ steps. In that way, for making a balanced growth of the tree, new nodes will inevitably fit into those prior to leafs ones. For departures, authors propose replacement of *non-leaf* nodes by *leaf-nodes*.

What is interesting in this schema, is the additional links between siblings and adjacent nodes. They enable to jump in the tree, and then to reach the root rapidly. This is particularly interesting for new processes that attempt to get their *ID* from the root. The cost of all atomic operations handled by our 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 one is a "lazy" mode. The balancing processing is not made 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 $\alpha\ log|weight(\sigma)|$, 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).

3.2.1 Balancing following new insertions

Our model channels new insertions in a way to avoid the root to be flooded by new processes, which could inhibit handling 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 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. It can be used if conflict arises with another process, as in the case of 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. 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 rebuilding is made as in scapegoat trees. Note that processes on the path of σ remain *blocked* until the subtree is stated as balanced.

3.2.2 Balancing following a token request

The requesting processing we have presented in Section 3.1 doesn't 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 tree remains unchanged.

3.2.3 Balancing following departure

The Exit strategy presented in Section 3.1 will 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.

Lemma 6 *No other request of a process σ' arrived later (at the root) can be completed before the request for σ, whatever circumstances.*

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. Restructuring processing.

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

Furthermore, the *Blocked* state assigned to the neighbors ensures keeping all needed information related to any operation triggered by processes of this branch of the tree. Moreover, on completion of departure, there is always a process that inherits all necessary information held by the outbound process (Step 4) of Section 3.1, Lemma 3 and Section 3.2.3.

For (2), new insertions never compromise the order of requests at the root, whatever the number of processes. Indeed, a process that wants to inserts itself to the system first checks the availability of processes on its path to the root (Section 3.2.1). Otherwise, σ backtracks. 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. □

3.3 The proof of the ELMP algorithm

In this section we prove *safety* and *liveness* properties.

Theorem 1 (Liveness) *If a given process σ claims the **critical section** then, it gets it within a finite time whatever the circumstances.*

Proof: Knowing that the the waiting time for the completion of a request is finite, since the ***Parent*** tree as well as the doubly linked list are never disconnected (Lemma 5) and the **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) then, the **critical section** is obtained within a finite time whatever circumstances. □

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 hasn't requested the token. If it holds the token without having requested it, we are in the situation of Lemma 3, 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. □

4 *Data Handover* on a peer-to-peer system

The following section addresses basic concepts and definitions of the proposed interface, and all phases through which a request sent by the user passes across the whole *peer-to-peer* system we propose.

4.1 The basic model of DHO

With a set of functions, **DHO** introduces an interface that mediates between the abstract concept of a data resource and its concrete realization memory. With **DHO** functions, an application process (*peer*) attempts to gain access to a specific data resource, without knowing if that resource is already present locally or on a remote machine. For example, the function `dho_ew_request(DHO_t* h)` is used to claim a data resource for **e**xclusive **w**rite. The argument `h`, a *handle*, comprises all the necessary information about the remote access to data resource.

For a given *peer*, two asynchronous processes are assigned to the same data resource, the *resource manager* and the *lock manager*: once a **DHO** request is inserted by the user, the *resource manager*, a local process, manages the control of the data resource for the local peer. It is responsible of *mapping* that resource in the local address space, and to transfer the actual version of the data to or from another *peer*. To obtain exclusivity on the resource and location information for the resource it forwards a lock request to the second asynchronous process, the *lock manager*. The *lock manager* negotiates the *locking* of the data with others *lock managers* remotely by sending messages. As a whole the *lock managers* of all peers ensure the overall consistency of the data according to the ELMP algorithm (Section 3).

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.

A request for a resource triggers events at the *resource manager* and crosses various states from request insertion until resource release. The left part of Fig. 4 shows the different states of knowledge about an acquisition (or not) that a *peer* has. We use capitalized names for such states such as *Idle* . States of the *resource manager* itself (that might be hidden to the *peer*) are shown on the right and are denoted in all minuscules like *invalid* . The first operation on a *resource manager* has always to be the `dho_create` function and waits for a reply from the *peer* that holds the resource. After the reply, the *resource manager* switches to the *valid* state. Note that all **DHO** routines take DHO *handle* as argument, such as

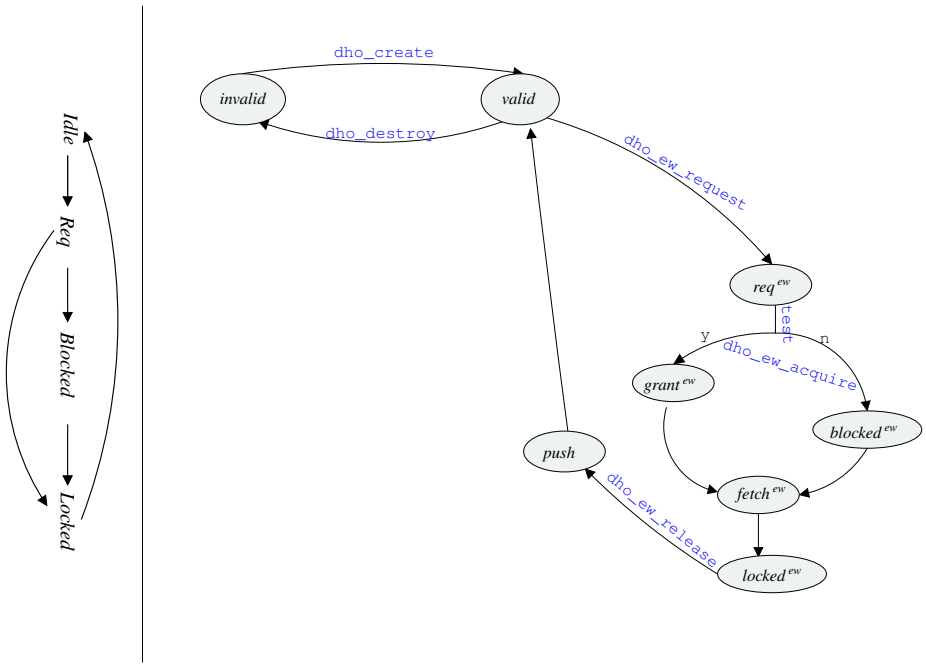


Figure 4: State diagram of **DHO** *peer* |*resource manager* for exclusive locks

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

4.2 DHO life cycle

The main phases of **DHO** and states through which a *resource manager* and a *lock manager* go are designed to form a cycle. The life cycle of a **DHO** *locking* and *mapping* request is described as follow:

When a *peer* claims a given resource, the *resource manager* and the *lock manager* interact locally with each other. Furthermore, the corresponding *lock manager* negotiates remotely the token with other *lock managers* distributed in the network, commonly with those of its ***Parent*** and its **Predecessor** (see Section 3.1). We note that information related to the neighborhood is saved in the internal structure of the *handle*.

The *resource manager* is contacted first when the *peer* requests a given resource. It is also responsible of the *locking*/*mapping* of that resource. However, the *lock manager* solely deals with received requests from other *peers*.

During negotiation phases, the *peer*, the *handle* and the *lock manager* keep inherent information through assigned states. The *locking*/*mapping* is effectively done when the *lock manager* gets the token.

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

Let ρ be a given *peer* that is initially *Idle*. The corresponding *handle* is then *valid*. The **DHO** life cycle for an exclusive write access follows the above phases:

1. Following the call of `dho_ew_request`, the *peer* becomes *Req*. Below, the corresponding *resource manager* sets the *handle* to the *req* $^{\text{ew}}$ state and forwards the request to the *lock manager* of the same *peer*. 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 3.1).

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.

   According to the **DHO** life cycle, two cases may occur:

   (a) At the application level, the *peer* calls the `dho_test` function. The corresponding *resource manager* asks the *lock manager* if it has already got the token. In that case, the *resource manager* assigns the *grant* $^{\text{ew}}$ state to the *handle*. 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. 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*. Once the token is acquired, the *lock manager* immediately informs the *resource manager* that amends the *handle*'s state. It will then enter the *grant* $^{\text{ew}}$ state.

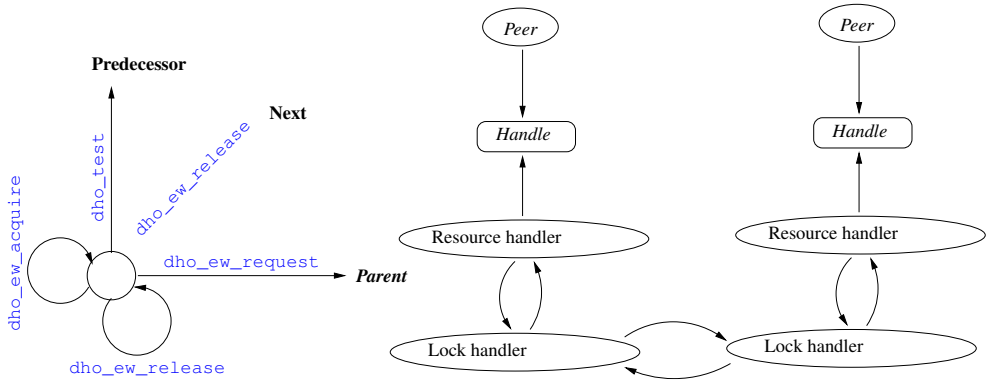
5. At that point, the *resource manager* fetches the data from its **Predecessor** (the intermediate *fetch* $^{\text{ew}}$ (T_{fetch}) state) and then *maps* the data into to the address space of the *handle*.

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*

6. Once the *mapping* is done, the *handle* and the *peer* become respectively *locked* ew, *Locked*.

7. The cycle is completed through a call to `dho_ew_release`. The *peer* becomes *Idle*, again. Now, the *resource manager* assigns the *Unlock* state to the *handle*. 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 *handle* will be *valid* , again.



(a) *peers* involved in *Data Handover* functions

(b) Hierarchical order of processes

Figure 5: *peers* and processes within the *peer-to-peer* system

Table 1 presents the hierarchical order of possible states caused by successive events triggered from the three levels. Fig. 5, emphasizes on the transmission channel between processes of a given *peer* (the *resource manager* and the *lock manager*), as well as the relationship between different *peers*. As shown in Fig. 4.2, the *peer* is dealing with three separate *peers* when it claims the resource, achieves the *locking*/*mapping* or grants access.

The first positive conclusion we can draw about the whole is the overlapping provided by involved processes activities. For example, by the set of states $\{Req, fetch^{ew}, Busy\}$, we can easily deduct activities handled by the same *peer*.

Hence, that *peer* has issued a request that is currently in *mapping* phase. Meanwhile, the associated *lock manager* deals with another received request.

4.3 Mobility of *peers*

The `dho_destroy` function reflects a voluntary departure of the hosting *peer* regardless of the currently assigned state. Note that all functions that follow after `dho_destroy` are ignored since the *handle* is *invalid*.
As a general policy, an application may chose not to respect the logical order of the calls to **DHO** functions as presented above without jeopardizing the consistency of the locks. If a **DHO** cycle is broken or canceled the concerned *peer* 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 3.1). 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 adopted balanced strategy (Section 3.2.1). 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.

5 Performance evaluation

The 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]. We performed experiences simulating a realistic platform (GRID'5000).

Two times, T_{Wblocked} and T_{Locked}, that are application dependent will be varied for our experiments. 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 in the **critical section**.

Besides of the **DHO** cycle time, we will analyze T_{Wait} and T_{Blocking} delays. T_{Wait} is the waiting time of a request, *i.e* the time between the call to request and

the return from fetching into state *locked* . T_{Blocking} is the time a *peer* is blocking before acquiring the resource, *i.e.* the time between the call to acquire and the return from the fetching into state *locked* . They are respectively expressed by the following equalities, see Section 4.2 above for the different times:

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

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

We assume that T_{Idle} is zero, so the *peers* insert a new request as soon as the previous cycle is achieved. *peers* carry out 100 cycles. Results refer to average values.

5.1 DHO cycle evaluation with synchronous locks

In our experiments we varied 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 at application level. This corresponds to a classical lock sequence that doesn't separate lock request and acquisition.

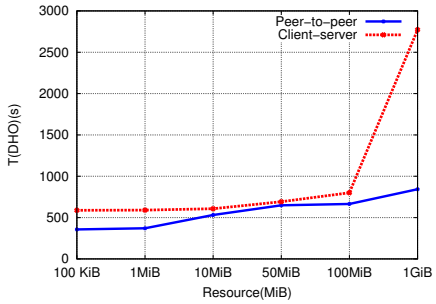
Results in comparison to the previous client-server based approach are shown in Fig. 6. We observe that the new peer-to-peer algorithm behaves much more regularly, in particular for the extremes of the resource size. By doing some regression of the values we find that the cycle time behaves as:

$$(3) \qquad \overline{T_{\text{DHO}}} \simeq [\overline{T_{\text{locked}}} + \overline{T_{\text{fetch}}} + \alpha \cdot \eta](\overline{q} + 1)$$

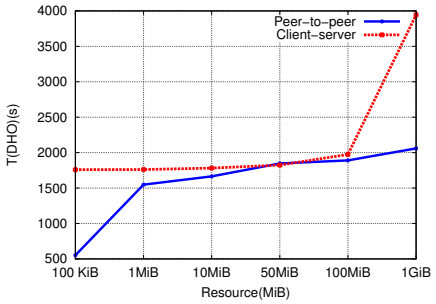
Where:

- $\alpha = 10.4$ 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**).

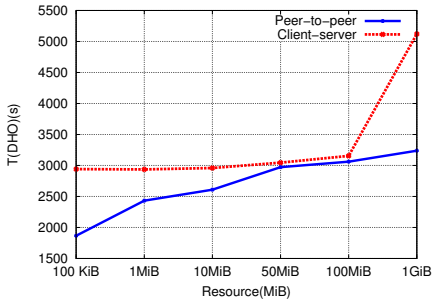
η has been approximately $70\mu s$ in the *client-server* paradigm. Indeed, an extra delay arises for the management of the *peer-to-peer* system, specially for partial conversion of the tree structure that requires a great number of messages between members of a given neighborhood.



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



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



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

Figure 6: Cycle time, $\overline{T_{\text{DHO}}}$ in the *Client-server* paradigm and the *peer-to-peer* system.

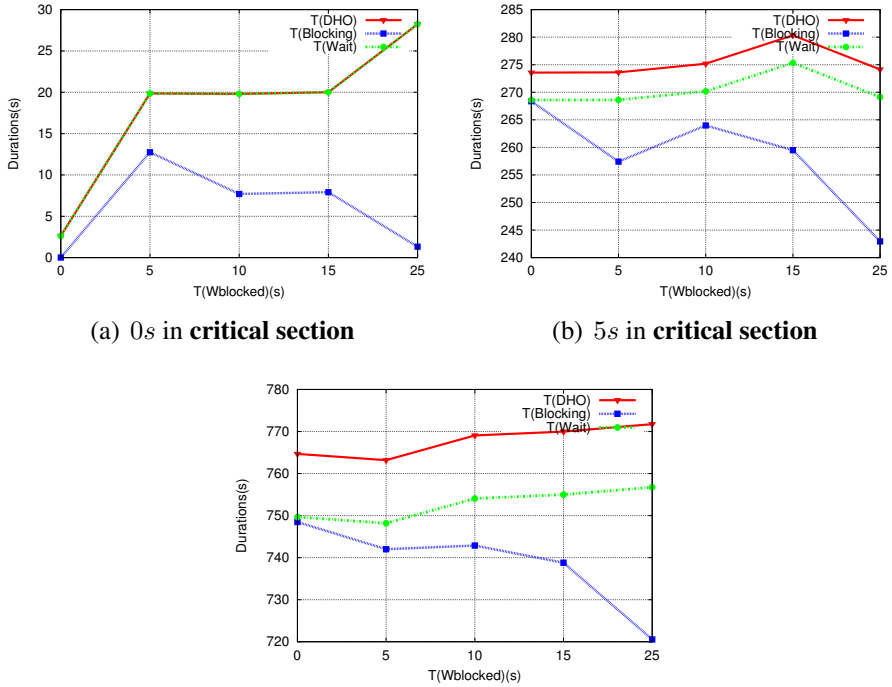
Conversely, we observed that $\overline{q}$ was significantly reduced compared to the *client-server* paradigm where the length was the number of pending requests. Indeed, our distributed system involves various processes that operate simultaneously during the query processing such that the bottleneck 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*$^{\text{ew}}$ state is time consuming in cases of large data sizes, the corresponding *peer* may simultaneous engage in other related operations of the *peer-to-peer* system.

5.2 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. 4). This series of benchmark is conducted with 50 *peers*. The data



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

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

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

Figure 7: Average duration of $\overline{T_{\text{DHO}}}$, $\overline{T_{\text{Wait}}}$ and $\overline{T_{\text{Blocked}}}$ by varying T_{Wblocked}.

resource size is fixed to $50MiB$. Fig. 7 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. 7(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. 7(b) and 7(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. 7(b) and 7(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.

5.3 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 *peers* from the remaining system. Once `dho_destroy` is issued, the *resource manager* destroys the *handle* that becomes *invalid* and then, all following **DHO** functions are ignored. Thus, the *lock manager* performs the **Exit strategy** of the **EMPL** algorithm.

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

- Peers in the first subset perform a complete cycle.
- In the second one, *peers* interrupt their cycle by calling the `dho_destroy` function.

We only note the duration of uninterrupted DHO cycles for the first class, for the case 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. However, we would expect an increase of these values with an even higher mobility frequency of *peers*.

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

Table 2: The overhead caused by subsets of *peers* on complete cycle times of the remaining *peers*.

6 Conclusion

In this paper we proposed a distributed mutual exclusion algorithm, **ELMP**, based on a hierarchical tree structure, that provides scalability and flexibility and allows for insertion and departure of peers. We studied two methods to keep the tree structure balanced, which is necessary after a set of conversions of the tree structure have been triggered. We have shown that all operations within the **ELMP**

algorithm guarantee a logarithmic cost, accounted in the number of messages that are issued per operation. The *liveness* and *safety* properties have been demonstrated.

Another contribution concerns the proposal of a three level architecture, to implement **DHO** in a peer to peer systems on top of **ELMP**. It comprises **DHO** routines, the *resource manager* and the *lock manager*. Our simulation experiments show an interesting property of this schema, namely the load sharing between the two managers. Even though they interact together to ensure consistency of the application, they may carry out some tasks independently.

In the future, it will be interesting to extend **ELMP** to manage exclusive and inclusive accesses as e.g for POSIX read-write locks.

References

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.

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.

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.

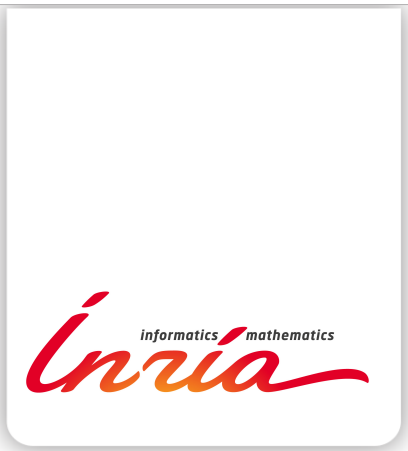
M. Naimi, M. Tréhel, and A. Arnold. A $\log(N)$ distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34:1–13, April 1996. ISSN 0743-7315.

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